K. C. Bowler, R. D. Kenway, G. S. Pawley, D. Roweth

An Introduction to OCCAM 2 Programming

Chartwell-Bratt
Studentlitteratur

Chartwell-Bratt (Publishing and Training) Ltd
ISBN 0-86238-137-1

Printed in Sweden
Studentlitteratur, Lund

ISBN 91-44-27151-4 1 2 3 4 5 6 7 8 9 10 | 1991 90 89 88 87

Preface

This introduction to occam 2 was originally produced as a detailed set of lecture notes for an intensive course, giving practical hands-on experience in an emerging computer technology. As there is a real need for a publication in this field we have made some minor modifications to the notes to satisfy a wider readership. Although the occam language is still in development we have chosen to present only that part of the putative language that is actually implemented on our hardware, fully aware that later additions will be necessary. Our aim has therefore been to produce a low-cost booklet for practical use. There are other publications concerning occam 2, notably that by Pountain, and these undoubtedly will contain the language constructs which are most likely to be included in future language definitions.

The hardware on which all the examples have been tested is the Meiko Computing Surface (CS), our version having 42 INMOS T414 transputers. This has been supported by the Department of Trade and Industry and the Computer Board, and was delivered in April 1986. Considerable progress has been made during the last year by our Theory and Computational Physics Group at Edinburgh, but it was thought not appropriate to include much of this which is specific to the CS. Nevertheless we have included a number of details of the Computing Surface.

Our lecture notes are understandably not in the usual format because occam is not yet a language with a long history. It should be easy to read through the first chapter and get an overall view of the language, but then some further help is needed to use this information in practice. The final, summary chapter, is designed for practical usage, giving cross-references to the other chapters whenever fuller details are required. This chapter contains a statement of the constructs of occam 2, but it also includes some of the necessary information about OPS, the occam operating system, for otherwise it would not be a full working tool.

The second chapter gives some simple programs ideal for the beginner, and there is the added advantage that it is possible to implement these programs on an IBM PC (with a B004 board). Chapter 3 follows with a specific case study, showing how to use a number of processors on one problem. This is then followed by a chapter on the use of parallel algorithms; many problems have to be rethought in order to exploit the enormous possible gains that massive parallelism offers.

A chapter surveying some of the various parallel computers is intended to put the transputer and the Computing Surface in perspective. This leads on to a more thorough chapter on the Computing Surface itself, and although there are some details which are specific to our own version of the computer, the reader should get a good indication of the versatility of this MIMD computer.

The history of the development of the scientific computer owes much to the seemingly unreasonable demands of the physicist, but we do have a shrewd idea as to where the physical limitations on computer technology lie. To get yet another factor of a thousand it is clear to us that massive parallelism is the only

way. We have been delighted to be able to discuss these matters with the designers in Meiko, and are especially indebted to Miles Chesney.

The last few years has been a very exciting time for us in Edinburgh. We started work on the ICL DAP in 1980, and this led to our acquisition of two DAPs, a growth of our commitment to large scale computation and a deepening of our understanding of the issues of parallelism. In all of this work, and in our future hopes, Professor David Wallace FRS has played a major and energetic role, for which we are all profoundly grateful. For my part I would like to take this opportunity to thank all my colleagues for such genuine and fruitful collaboration, especially to the other authors of this book who did by far the major part of the work.

Stuart Pawley

Professor of Computational Physics

Department of Physics

University of Edinburgh

April, 1987

Contents

1. Introduction to Occam 2

[1] 1. Some basic ideas

This chapter will provide an elementary introduction to concurrent programming using the occam language, which was developed by C. Hoare of Oxford University and D. May of INMOS Ltd. Although occam may be used on conventional computers, it has a special relationship with the INMOS Transputer, a high-performance VLSI microprocessor which was designed with on-chip communications to facilitate the construction of parallel processing systems of arbitrary size. Occam may be thought of as the assembly language of the transputer, although it can be regarded as a language in its own right.

Occam is based on the process model of computing. A process is an independent computation with its own program and data, which can communicate with other processes which are executing at the same time. A process can be thought of as a black box with inputs and outputs, that can communicate by message passing using explicitly defined channels. Processes can be connected together by such channels to build more complex concurrent systems. Each channel provides a one-way connection between two concurrent processes. Communication is synchronised; if a given channel is used for input in one process and output in a second process, communication takes place only when both processes are ready.

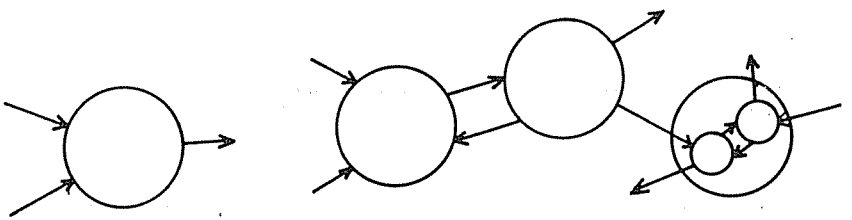

Fig. 1: (a) A process (b) Processes connected together by channels

Occam enables a system to be described as a collection of concurrent processes which communicate with each other through channels. An occam program may be executed by an array of transputers. However, it is important to realise that the same program may be executed almost unchanged on a smaller array, or even on a single transputer. An occam channel describes communication in the abstract and does not depend upon a particular hardware implementation. Thus an occam program which uses channels may be written and tested without deciding where particular processes will be executed.

[1] 2. Primitive processes

Occam programs are constructed from a small number of simple building blocks called primitive processes which we will now describe.

[1] 2.1 Assignment process

An assignment process simply changes the value of a variable, in just the same way as in most other programming languages. The special symbol used for assignment in occam is :=. Thus for example

```
var := 6
```

assigns the value 6 to the variable var. The value assigned to a variable could be an expression, which may contain other variables:

```
var := 6 + var2
```

Note that = and := are not the same. In occam = means a test for equality, not an assignment.

[1] 2.2 Input process

An input process inputs a value from a channel into a variable. The symbol for input in occam is ?. For example,

```
chan1 ? xvar
```

sets the variable xvar to the value input from the channel chan1.

Input processes can only input values to variables, not to constants or to expressions. An input process will wait until a corresponding output process on the same channel is ready.

[1] 2.3 Output process

An output process outputs a value to a channel. The symbol for output in occam is !. Thus,

```
chan2 ! yvar
```

outputs the value of the variable yvar to the channel chan2. An output process cannot proceed until a corresponding input process on the same channel is ready.

Communication is thus synchronised; at any time a process may be ready and waiting to communicate on one or more of its channels. When both an input and an output process are ready to communicate on the same channel, the value to be output is copied from the output process to the input process.

[1] 2.4 SKIP and STOP

Occam also has two special processes called SKIP and STOP. SKIP is a process which starts, does nothing and then finishes. This may seem bizarre, but we will see later that there are instances where the syntax of

occam requires a process to be present even though nothing is required to happen. It might also be used in a partially completed program to represent a process which has still to be written.

STOP is a process which starts but never proceeds or finishes ! It can be thought of as representing a process which doesn't work. For example, it could be used instead of a process for handling errors, when a program is under development.

We should be more careful about what is meant by finishing. A process which completes all its actions is said to **terminate**. Normally a process starts, proceeds and terminates. A process which starts but cannot proceed is said to be **stopped**. A stopped process never terminates. For example, a process might be waiting for an input which will never happen because of a programming error; such a process is said to be **deadlocked**, the curse of concurrent programming !

[1] 3. Constructs

Constructs are used to combine primitive processes into larger processes which may, in their turn, be combined into larger processes still. Constructs start with an occam keyword which states how the component processes are to be combined.

[1] 3.1 SEQ

The sequential construct SEQ causes the component processes to be executed one after another, terminating when the last component terminates. For example

```
SEQ
  chan1 ? var1
  var2 := var1 + 1
  chan2 ! var2
```

which inputs from chan1 to var1, assigns var1 + 1 to var2 and outputs var2 to chan2. Thus we see that a SEQ process is just like a program in more conventional programming languages, terminating when the last component process terminates. Note that the component processes which make up the SEQ are all indented with respect to the word SEQ by two characters, which is how occam knows which processes are part of the SEQ construct. Note also that SEQ is compulsory in occam whenever two or more processes are to run in sequence, unlike more conventional languages, where the sequential execution of consecutive statements is usually taken for granted.

[1] 3.2 Replicated SEQ

Occam allows us to create replicas of processes using a replicator index akin to an array index. We shall postpone discussion of arrays until later, but here is an example of a replicated SEQ:

```
INT var :
SEQ index = 0 FOR 5
  channel.out ! var + index
```

The effect of this is equivalent to writing

```
INT var :
SEQ
  channel.out ! var
  channel.out ! var + 1
  channel.out ! var + 2
  channel.out ! var + 3
  channel.out ! var + 4
```

that is, we create 5 replicas of the input process and execute them in sequence. Another new feature is the declaration of the variable **var** to be of type INT, that is, integer. We will specify the types in occam a little later. The general form of a replicated SEQ is

```
SEQ index = base FOR count
  process
```

Individual processes in a replicated construct can be referred to using the replicator index. Examples will be given very shortly. Note that if *count* is zero, the replicated SEQ will act like SKIP.

[1] 3.3 Naming a process

At this point it is useful to point out that a process may be named by means of the keyword PROC and a name, followed by the body of the process or procedure. PROC takes zero or more formal parameters, which will be described later. For example

```
PROC add.one (CHAN chan1,chan2)
  INT var1, var2 :
  SEQ
    chan1 ? var1
    var2 := var1 + 1
    chan2 ! var2
:
```

Here we have packaged up the previous example code into a procedure named **add.one** which takes the names of the two channels **chan1** and **chan2** as parameters. The procedure is terminated by the **:** on a line by itself at the same level of indentation as the PROC keyword.

[1] 3.4 Scope

In occam, variables, channels and other named objects are local to the process which immediately follows their specification. Declarations, like INT, introduce variables for the process that follows at the **same level of indentation**. Indentation defines the scope of a construct.

[1] 3.5 WHILE

We may wish to have a process executed repeatedly, until some condition is satisfied. One way to achieve this is to use

```
WHILE expression
```

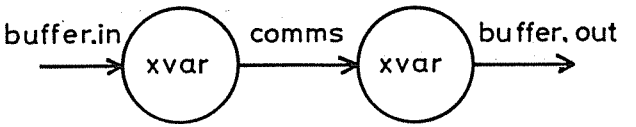

Fig. 3: A two-stage buffer

Here it is important to note that the written order of the component processes is irrelevant, as they are performed concurrently. Each process awaits input on a channel and, upon receipt, outputs a value. In this example, the concurrent processes communicate using the channel comms. In general **note** the following:

- concurrent processes may only communicate using channels
- processes within a PAR construct must be independent
- each concurrent process operates on its own variables
- communication is synchronised.
- only two component processes of a PAR may use any particular channel, one as sender and the other as receiver.

Here is a second example, admittedly artificial, which does some arithmetic on the input values before passing them on:

```
CHAN comms :
PAR
  INT var1 :
  SEQ
    channel.in ? var1
    comms ! var1*var1
  INT var2 :
  SEQ
    comms ? var2
    channel.out ! var2 + 1
```

We have here assumed that the channels channel.in and channel.out have been defined elsewhere in the program. Of course it would be much simpler in this instance to perform the square and add in a single SEQ process or in a single expression, but in more complicated applications, such as the construction of an arithmetic pipeline, this kind of construct is very useful.

[1] 3.7 Replicated PAR

As for the SEQ construct, PAR may be replicated to build an array of parallel processes, any of which can be referred to by the replicator index. The general form is

```
PAR index = base FOR count
  process
```

As an example, we consider the 'bucket brigade' or pipeline, in which data is passed from one process to another in a chain, using an array, or vector of channels, which is declared in the first line. We will discuss arrays in detail at a later point, but the minimal use made here should be self-explanatory:

```
[10]CHAN link :
PAR bucket = 0 FOR 9
  WHILE TRUE
    INT water :
    SEQ
      link[bucket] ? water
      link[bucket+1] ! water
```

The replicator sets up 9 parallel processes, each of which continually transfers values between one node in the pipeline and its neighbour. Of course this example is not self-contained, because as it stands, the first 'bucket' has nowhere to input its 'water' from, nor has the final process anywhere to output its 'water' to. However, we can imagine embedding this code in a larger program which supplies data to `link[0]` and extracts data from `link[9]`.

[1] 4. Types & specifications

Unlike the first version of occam, described in the Occam Programming Manual, published by Prentice Hall International in 1984, occam 2 requires that objects used by a program should have a type, which must be specified before using that object in a process. We have also been a little cavalier with names for variables and channels in the examples given so far, so let us now be more precise.

[1] 4.1 Names

Names must begin with a letter of the alphabet, may include letters, digits and the dot character and can be of any length. Upper and lower case are treated as distinct by occam. Occam keywords such as **SEQ, PAR, CHAN** and **PROC** are always in upper case and are reserved. Examples of valid names are:

```
x  X  var  var1  VarOne  var.one  very.long.name.for.variable
```

[1] 4.2 Data types

The types which are available for variables in occam are:

```
INT       -- an integer
BYTE      -- an integer restricted to the range 0 to 255
          -- can be used to represent characters
BOOL      -- logical; either TRUE or FALSE
```

In addition to these generally available types, some implementations of occam permit some or all of the following types:

```
INT16       -- 16 bit integer
INT32       -- 32 bit integer
INT64       -- 64 bit integer
REAL32      -- 32 bit real
REAL64      -- 64 bit real
```

Note the use of occam comments, preceded by --

[1] 4.3 Channels

Occam channels are all of the type **CHAN** in the current release of occam2, although this may change in future versions.

[1] 4.4 Constants

In occam a name may be given to a constant value by using the specification

VAL *type name* IS *value* :

so it is possible to write, for example

```
VAL INT hours IS 24, minutes IS 60 :
```

The type is normally omitted when it is obvious from the value. Possible ambiguities over **BYTE** and **INT**, or **REAL32** and **REAL64** are resolved by explicitly specifying the type of the value. Examples:

```
VAL empty IS 0 (BYTE) :

VAL pi IS 3.14159 (REAL32) :
```

The first of these is equivalent to

```
VAL BYTE empty IS 0 :
```

The colon which ends a specification effectively joins that specification to the process which follows it, an idea which is emphasised by indenting the specifications to the same level as the process, as we noted earlier in our brief discussion of scope. Thus the scope of a specification is restricted to the following process. This is illustrated by the squarer/adder example which we discussed under **PAR**. The variables **var1** and **var2** are local to their respective **SEQ** processes.

[1] 4.5 Initialisation

In occam the value of a variable is unassigned until it has either input a value or has been assigned a value. Furthermore, the value of a variable has meaning only during the execution of the process for which it has been declared. Once the process has terminated the variable no longer has a well-defined value. If the process is to execute again it is important that the variable has again been assigned a value. If one wants a variable to keep its value from one execution of a process to another, it can be declared in an outer scope, that is, before a process which contains the process which is to be executed repeatedly.

[1] 5. Operators

So far we have not specified the arithmetic and logical operators which are available in occam. Let us now be more precise.

[1] 5.1 Arithmetic operators

The elementary arithmetic operations in occam are as follows:

```
a + b       -- add b to a
a - b       -- subtract b from a
a * b       -- multiply  a by b
a / b       -- divide a by b
a REM b     -- remainder when a is divided by b
a \ b       -- alternative form of REM
```

In occam there is no priority (precedence) for arithmetic operators so that brackets must be used to remove all ambiguities. For example:

```
(2+3)*(4+5)     -- result is 45
2+(3*(4+5))     -- result is 29
(2+(3*4))+5     -- result is 19
2+3*4+5         -- illegal
(2+(3*4)+5)     -- illegal
```

[1] 5.2 Modulo arithmetic operators

For integers only, a further set of operators is provided which permits modulo arithmetic. The modulo operators **PLUS**, **MINUS** and **TIMES** permit addition, subtraction and multiplication modulo **N/2** where **N** = 2 to the power **n**, with **n** the number of bits in an **INT**), in unchecked 2's complement arithmetic. In general

(i PLUS j) is **(i+j) + (k*N)**

where **k** is the unique integer for which

(i+j) + (k*N) >= -(N/2) and **(i+j) + (k*N) < (N/2)**

Similarly

(i MINUS j) is **(i-j) + (k*N)**
(i TIMES j) is **(i*j) + (k*N)**

As a simple, rather artificial example, consider the case of 3 bit **INT** modulo arithmetic, where **N** is 8:

(3 PLUS 3) is **6 + k*N,** with **k = -1** and hence yields **-2**

[1] 5.3 Boolean operators

A boolean values is produced as the result of a test performed by a comparison operator. The following operators are available in occam:

```
=       -- equal to
<>      -- not equal to
>       -- greater than
<       -- less than
>=      -- greater than or equal to
<=      -- less than or equal to
```

Note that again brackets are needed to remove ambiguities whenever more complicated tests are made by combining two or more comparisons.

Occam also provides the boolean constants TRUE and FALSE which may be used anywhere that a test could be used. We have already seen an example.

Occam provides the standard boolean operators AND, OR and NOT, which are defined by

```
NOT TRUE = FALSE          NOT FALSE = TRUE
TRUE AND log = log        FALSE AND log = FALSE
TRUE OR log = TRUE        FALSE OR log = log
```

where log is either TRUE or FALSE.

[1] 5.4 Bit operators

More sophisticated applications than those we have discussed so far may require operations on individual bits in a word. Occam provides the following bit operators:

```
/\      -- bitwise and
\/      -- bitwise or
><      -- bitwise exclusive or
~       -- bitwise not
<<      -- left shift
>>      -- right shift
```

[1] 6. More on constructs

[1] 6.1 IF

Occam provides a form of conditional choice by means of the construct IF, which takes any number of processes, each preceded by a test, and builds from them a single process. The conditions are tested **sequentially** and the first one which is TRUE is executed. Note that **only** that process is executed. As an example, consider

```
IF
  var = 1
    chan1 ! x
  var = 2
    chan2 ! x
  var = 3
    chan3 ! x
```

The effect of this piece of code is to output the value of the variable x on chan1, chan2 or chan3 depending on whether the value of var is 1, 2 or 3. Note that if the value of var is anything other than 1, 2 or 3 the effect of this IF process is equivalent to STOP. The program can only proceed if one of the choices is executed. One way of avoiding this difficulty is as follows;

```
IF
  var = 1
    chan1 ! x
  var = 2
    chan2 ! x
  var = 3
    chan3 ! x
  TRUE
    SKIP
```

We see that if var fails the first three tests, it is bound to satisfy the fourth, namely TRUE, and hence the program can proceed via the SKIP process.

More complicated tests can be performed by nesting IF's:

```
IF
  var = 1
    chan1 ! x
  var = 2
    IF
      x = 6
        chan2 ! x
      TRUE
        chan3 ! x
  TRUE
    SKIP
```

If the value of var is 2 the effect of this code fragment is to output the value of x on chan3 if x has any value other than 6.

[1] 6.2 ALT

Whereas the IF construct enables us to choose different processes according to the values of variables in the program, the alternative construct, ALT, allows us to make choices which depend on the state of channels. The component parts of ALT, called **alternatives**, are combined by ALT into a single construct, but in ways which can be difficult to grasp at first. The simplest kind of ALT is where each alternative consists of an input process followed by an executable process of some sort. Thus

```
CHAN chan1, chan2, chan3 :
INT var :
ALT
  chan1 ? var
    process 1
  chan2 ? var
    process 2
  chan3 ? var
    process 3
```

The ALT watches all the input processes, known as **guards**, and executes the process associated with the first input to become ready. We can think of the ALT construct as a first-past-the-post race between a set of channels, with only the winner's process being executed.

As a second example we consider again the simple buffer. ALT may be used to provide for an interrupt

```
BOOL running :
SEQ
  running := TRUE
  WHILE running
    INT xvar, any :
    ALT
      buffer.in ? xvar
        buffer.out ! xvar
      interrupt ? any
        running := FALSE
```

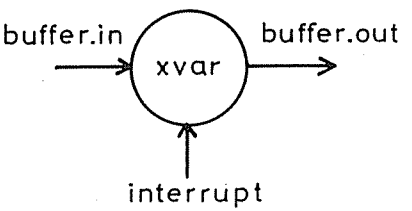

Fig. 4: A simple buffer with interrupt

Here the channel **interrupt** permits the termination of the continuous **WHILE** loop.

In general note the following features:

- each component process has a guard which is an input, with an optional condition. Permissible guards are:

  channel ? *variable*

  (*boolean*) **& channel ?** *variable*

  SKIP

- a process which is guarded by an input is not executed unless the process at the other end of the channel is ready to output.

- the earliest process which is ready to be executed is chosen. The guard is executed, followed by the guarded process.

- if several alternative guards are ready, an arbitrary one is chosen.

- a process guarded by **SKIP** is always ready.

Here is an example of **ALT** with a test in addition to an input as guard:

```
CHAN chan1, chan2, chan3 :
INT any :
ALT
  (ext.var < 0) & chan1 ? any
    process1
  (ext.var = 0) & chan2 ? any
    process2
  (ext.var > 0) & chan3 ? any
    process3
```

As with the **IF** construct, **ALT** may be nested inside an outer **ALT**.

[1] 7. A longer example

We now illustrate the use of these more difficult constructs with a longer and somewhat more complex example. Suppose that we want to write a program to control a heating system via two digital press-buttons labelled warmer and cooler. Pressing warmer increases the heat output by one unit whereas pressing cooler decreases the output by one unit.

Suppose that we have two occam channels called **warmer** and **cooler** which produce an input whenever the appropriate button on the control panel is pressed, and a third channel called **heater** whose function is to transmit a value to the regulator mechanism on the heat source.

Without worrying about declarations at this stage, we first construct simple processes to increase or decrease the heat produced:

```
SEQ
  heat := heat + 1
  heater ! heat
```

and

```
SEQ
  heat := heat - 1
  heater ! heat
```

The program can poll the two channels **warmer** and **cooler** to determine which button has been pressed by making use of the ALT construct:

```
INT heat, any :
SEQ
  heat := 0
  heater ! heat
  WHILE TRUE
    ALT
      warmer ? any
        SEQ
          heat := heat + 1
          heater ! heat
      cooler ? any
        SEQ
          heat := heat - 1
          heater ! heat
```

The use of WHILE TRUE means that the control panel buttons are interrogated repeatedly and the process never terminates. We can ensure that the program terminates in a sensible fashion by adding an extra channel to communicate the status of the OFF button on the control panel to the program, and by replacing the boolean constant TRUE by a boolean variable which represents the ON/OFF status:

```
BOOL running :
INT heat, any :
SEQ
  running := TRUE
  heat := 0
  heater ! heat
  WHILE running
    ALT
      warmer ? any
        SEQ
          heat := heat + 1
          heater ! heat
      cooler ? any
        SEQ
          heat := heat - 1
          heater ! heat
      off ? any
        running := FALSE
```

We might also wish to limit the values which are transmitted to the heater to some fixed range which corresponds with the physical limitations on the minimum and maximum heat output of the device. Let us suppose that these values are 0 and 10 respectively. We declare these minimum and maximum values at the beginning of the program using named constants so that if we ever need to revise the values, when, for instance a new model of the heater is installed, we need only change the values in a single place in the program.

```
VAL min.heat IS 0, max.heat IS 10 :
BOOL running :
INT heat, any :
SEQ
  running := TRUE
  heat := min.heat
  heater ! heat
  WHILE running
    ALT
      (heat < max.heat) & warmer ? any
        SEQ
          heat := heat + 1
          heater ! heat
      (heat > min.heat) & cooler ? any
        SEQ
          heat := heat - 1
          heater ! heat
      off ? any
        running := FALSE
```

Note that the channels **heater**, **warmer**, **cooler** and **off** have not been declared in the program because they connect to pieces of hardware, rather than to other occam processes.

[1] 8. Abbreviations

One of the most powerful and useful features of occam is the abbreviation. Abbreviations may be used to give a name to any expression in occam. We have already seen an example when we discussed named constants, employing **VAL**. Here is an expression abbreviation

```
VAL seconds IS 60*((60*hours)+mins) :
```

which defines **seconds** to be shorthand for the value of the expression on the right. The scope of the abbreviation is the process which follows it, to which it is attached by the terminating colon, as usual.

If an abbreviation, such as this example, contains variables on its right hand side, then in general those variables should remain constant throughout its scope. The full form of an expression abbreviation contains a type specifier before the name

```
VAL INT name IS expression :
```

but as before, the type can be omitted, leaving occam to deduce the type from the type of the right hand side. Occam assumes that integers less than 256 are of type **INT** unless otherwise instructed, which can be done:

```
VAL soh IS 1 (BYTE) :
```

In the next section we shall see how abbreviations can be used to name arrays and parts of arrays.

[1] 9. Arrays

Unlike earlier versions of occam, occam 2 supports multidimensional arrays. Quite generally, an array is a set of elements of the same type. An array may have one or more subscripts; the number of subscripts is referred to as the dimension of the array. An element of an array is specified by giving the value of each of the arrray subscripts, and is usually known as a component of the array.

[1] 9.1 Array types

Array variables in occam are declared in the same way as single variables of any type, but with the number of components in each dimension prefixing the type declaration. Examples are:

```
[4] INT vector :              -- a vector of 4 integers
[8][8] BYTE chessboard :      -- a 2 dimensional byte array
[5][5][5] INT cube.sites :    -- a 3 dimensional integer array
[6]REAL32 values :            -- a vector of 6 reals
[3]CHAN ttyMux :              -- a vector of 3 channels
```

An array can be referenced by name in order to transmit it to another process, for example.

```
PAR
  comms ! chessboard
  other processes
  comms ? chequer
```

sends the entire array **chessboard** to another process, provided that **chequer** has also been declared to be of type [8][8]BYTE.

Note that the sizes of arrays in occam must be fixed at compile time and cannot be assigned or altered during execution.

The convention for referring to particular components of arrays in occam is to specify the array name followed by a suffix or suffices giving the particular values of the array subscripts in brackets. For example

```
vector[0]                     -- the first element of vector
chessboard[0][1]              -- the white knight's square!
ttyMux[1]                     -- the second channel
```

A segment can itself be treated as an array. Thus defining the array [20]INT store we may treat the segment

```
[store FROM 8 FOR 6]
```

as an array of 6 components, starting with store[8] and ending with store[13].

Array segments can be input, output or assigned to in the same manner as arrays, provided always that the expression which is assigned is an array of the same type and size as the segment. For example

```
[store FROM 10 FOR 5] := [store FROM 0 FOR 5]

[store FROM 10 FOR 5] := cache
```

where cache has been declared as [5]INT.

[1] 10. More about abbreviations

We have already met the use of abbreviations for constants and constant expressions. The description of array segments can be much simplified by using abbreviations. For example

```
st IS [store FROM 10 FOR 5] :
```

enables us to consider st to be an array of size 5, with subscripts running from 0 to 4, where, for example, st[1] is identified with store[11] and so on.

An important point to note here is that VAL is not used; we are not merely naming a value. The abbreviation may thus be used instead of the full name of the object when we wish to change the value of that object by assignment or by input. We could write the previous example more succinctly as

```
st := cache
```

Note that it is **not** legal, when using an abbreviation for an array component or segment, to change which component or components are referred to. For example, if we declare an abbreviation

```
pocket IS store[i] :
```

we should not change the value of the subscript i within the scope of pocket.

Abbreviations may also reward the programmer with performance benefits, as well as improving the conciseness of code. If instead of subscripting an array, we use an abbreviation for an array component within a loop

- the compiler recognises that the subscripts are constant and so does not compile run-time range checks

- the address of the array component becomes local to the loop process rather than global, and occam processes handle local data faster.

An abbreviation may also be used to set up an array constant:

```
VAL days IS [1,2,3,4,5,6,7] :
```

The components of days can be accessed in the usual way, so days[3] is 4.

We finish this section on arrays with two examples to illustrate some of the features just introduced. Here is a procedure which can be used to compute the mean and variance of a set of data, as that data is accumulated.

```
PROC time.average ([]REAL32 statistics, REAL32 data)

  sample.size IS statistics[0] :
  sum.x IS statistics[1] :
  sum.x.sq IS statistics[2] :
  mean IS statistics[3] :
  variance IS statistics[4] :
  SEQ
    sample.size := sample.size + 1.0(REAL32)
    sum.x := sum.x + data
    sum.x.sq := sum.x.sq + (data*data)
    mean := sum.x/sample.size
    variance := (sum.x.sq/sample.size) - (mean*mean)
:
```

We see the use of an array as a formal parameter of a procedure, and abbreviations for array components. Note that in using this procedure, it must be ensured that the actual array used when the procedure is called has been properly initialised. For instance

```
VAL zero IS 0.0(REAL32) :
VAL zero.stats IS [zero,zero,zero,zero,zero] :
REAL32 pulse.height :
[5]REAL32 pulse.height.stats :
SEQ
  -- initialisation
  pulse.height.stats := zero.stats
  WHILE TRUE
    SEQ
      -- read event and update statistics
      data.chan ? pulse.height
      time.average (pulse.height.stats, pulse.height)
```

In the second example we use a formal parameter which is an array of constants, rather than variables, in a procedure which computes the scalar product of two fixed vectors:

```
PROC scalar.product (VAL []REAL32 a, b, REAL32 a.b)
  SEQ
    a.b := 0.0 (REAL32)
    SEQ i = 0 FOR (SIZE a)
      a.b := a.b + (a[i]*b[i])
:
```

Here **a.b** is initialised to zero and then used to accumulate the products of the corresponding components of the vectors **a** and **b**.

[1] 11. More on replicators

We saw earlier that replicators could be used with the SEQ and PAR constructs to useful effect. Now that we are armed with more details of the properties of arrays in occam we could proceed to construct more complex examples of their use, but instead we will introduce two more kinds of replicated construct, IF and ALT.

[1] 11.1 Replicated IF

The general form is

```
IF index = base FOR count
  choice
```

where *choice* consists of a condition followed by a process.

Here is a very simple example:

```
IF component = 0 FOR 5
  store[component] = 0
    store[component] := 1
```

whose effect is simply to test the first 5 elements of the array **store** and to replace the first one found to be **0** by **1**. Note, however, that if no **0** is found, the program will be stopped. The replicated construct does not admit (sensibly) a concluding **TRUE SKIP**, and hence the usual use of a replicated **IF** involves nesting within an outer **IF**. For instance

```
IF
  IF component = 0 FOR 5
    store[component] = 0
      store[component] := 1
  TRUE
    SKIP
```

will now **SKIP** if no zeroes are found in the array, and the program can proceed. This construct can be used for searching for the first occurrence of a given character in a string. If all such occurrences are needed a replicated **SEQ** may be used instead.

[1] 11.2 Replicated ALT

The general form of the replicated ALT is similar to the previous construct:

```
ALT index = base FOR count
  alternative
```

where *alternative* is a guard followed by a process.

The effect is to monitor an array of channels and it may therefore be used to construct a multiplexer, for example:

```
PROC multiplex ([]CHAN inputs, CHAN output, interrupt)

  INT any, signal :
  BOOL running :
  SEQ
    running := TRUE
    WHILE running
      ALT
        ALT i = 0 FOR (SIZE inputs)
          inputs[i] ? signal
            output ! signal
        interrupt ? any
          running := FALSE
:
```

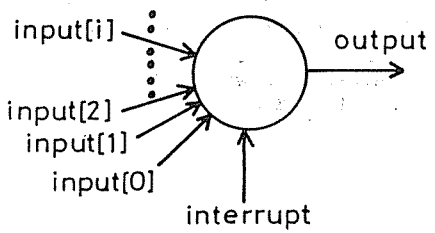

Fig. 5: A multiplexer

[1] 12. Priority

It is sometimes useful, particularly in real-time programming, to be able to assign priorities to processes in a well-defined way. We saw earlier in the case of the ALT construct that when the inputs guarding two alternative processes become ready simultaneously, occam makes an arbitrary choice between the two. Occam allows us to assign priorities for the ALT and the PAR constructs by preceding the keyword by PRI. In either case the component processes are assigned a priority which corresponds to the textual order in which they appear in the program.

[1] 12.1 PRI ALT

As already indicated, when two processes become ready simultaneously, the process with the higher priority will be executed. An example of the use of this construct is when it is essential to guarantee that a particularly important channel is examined:

```
WHILE running
  INT any :
  PRI ALT
    emergency.halt ? any
      running := FALSE
    TRUE & SKIP
      ... main cycle
```

Here the PRI ALT forces the program to check the channel emergency.halt because of its higher priority. Without the priority, the alternative TRUE & SKIP, which is always ready, could be taken at every cycle of the WHILE loop.

[1] 12.2 PRI PAR

Assigning priorities to the component processes means that processes with lower priority can proceed only if no higher priority process is able to proceed. Consider

```
PRI PAR
  SEQ
    chan1.in ? var1
    chan1.out ! var1
  SEQ
    chan2.in ? var2
    chan2.out ! var2
```

where the second SEQ cannot proceed, even when ready, unless the first is waiting for its input or output.

As an example, it might be the case that it is more important to relay any messages to an external device than to continue computation in a program. PRI PAR can be used to ensure that computation only proceeds when there is no message waiting to be input or output:

```
PRI PAR
  WHILE TRUE
    [message.length]BYTE message :
    SEQ
      message.in ? message
      SEQ i = 0 FOR message.length
        message.out ! message[i]
  ... main computation
```

We have assumed for illustrative purposes that the external device on the other end of channel **message.out** requires serving with data one byte at a time.

When running a high priority process of this kind it is generally a good rule to buffer the communications to other processes so that data can be sent without delay. The size of buffer needed depends in practice on the timings of the various processes involved.

```
PRI PAR
  CHAN buffer.chan :
  PAR
    WHILE TRUE
      [message.length]BYTE message :
      SEQ
        message.in ? message
        buffer.chan ! message
    WHILE TRUE
      [message.length]BYTE message :
      SEQ
        buffer.chan ? message
        SEQ i =0 FOR message.length
          message.out ! message[i]
  ... main computation
```

As a general rule, **PRI PAR** should only be used when it is essential to impose explicit priority. Since priority does not impinge upon the logical structure of a program, it is really a configuration issue rather than an occam programming issue and thus should be left until last, when the overall program logic has been established and the program works.

[1] 13. More on procedures

We have already seen several examples of procedures and their uses in earlier sections, but we have not discussed in a systematic way the rules for their use or the conventions for parameter passing. Let us now remedy this deficiency.

[1] 13.1 Scope

Procedures obey the same scope rules as other occam objects such as names and variables; the procedure is only known throughout the process which immediately follows it, to which it is linked by the final **:**. The body of the **PROC** is executed whenever its name is found in that process; such an occurrence of the name is called an **instance** of the procedure. Whenever an instance of the **PROC** is encountered, it is executed exactly as if the body of the procedure had been substituted for the name.

[1] 13.2 Parameters

Earlier examples of PROCs introduced the idea of **formal parameters**, which are the means by which different values may be passed to and from the body of the procedure at different instances. Formal parameters may be of any type, including, as we have seen, CHAN. When the body of the PROC is substituted for the instance of the procedure name in a process, the formal parameter names are replaced by the actual parameters, which may be values, variables or expressions.

We are allowed by occam to have any number of formal parameters, which must be separated by commas in the heading of the PROC definition. The actual parameters of an instance are similarly separated by commas, and must correspond in number, position and type with the formal parameters of the PROC.

[1] 13.3 Passing conventions

When we pass a variable as an actual parameter **to** a procedure in occam, that variable effectively replaces the formal parameter throughout the procedure. Anything which the procedure does to the formal parameter is also done to the variable, which may in consequence have its value changed. This is in **contrast** to the call-by-value convention commonly used in other programming languages in which the actual parameter is used as the initial value of the formal parameter, which then behaves as a local variable of the procedure. To give a very simple example, consider the procedure :

```
PROC double (INT number)
  number := 2 * number
:
```

When an instance of this procedure is encountered, such as **double (n)**, the value of **n** will be twice its previous value when **double (n)** terminates, so that caution may be required. For example

```
INT n, m :
SEQ
  n := 1
  m := 2
  double (n)
  m := m + n
```

assigns a final value of 4, not 3, to m.

We sometimes require that a procedure should not alter the value of a variable which has been passed to it as a parameter. One way to achieve this is of course to assign the value of the parameter to a local variable within the PROC and to perform any operations on this local copy. If we need only the original value of a variable in the body of a procedure, that is, the formal parameter is never altered by assignment or by input, then we can say explicitly that only the value is to be passed, using VAL. We saw an example of this in the code for the scalar product of two vectors, given earlier. In this case, the formal parameter can be thought of as a constant throughout the body of the procedure. The compiler may exploit this to produce more efficient code.

so that if the value of the boolean **running** is TRUE, then INT **running** is 1.

The following does conversion between INT and REAL32, and *vice versa*, with either rounding or truncation to IEEE standards:

REAL32 ROUND x converts x of type INT to type REAL32

INT ROUND x converts x of type REAL32 to type INT

REAL32 TRUNC x converts x of type INT to type REAL32

INT TRUNC x converts x of type REAL32 to type INT

[1] 14.2 RETYPES

Occam 2 provides a potentially powerful way of performing type conversion in the form of an operator RETYPES whose use is of the general form

specifier name RETYPES *element*

or

VAL *specifier name* RETYPES *expression*

Warning: the use of retyping conversion will usually result in implementation dependent processes, as the representation of variables will vary from one implementation to another.

Here are some examples:

```
VAL REAL32 x IS 1.0 (REAL32) :
VAL INT xint RETYPES x :

INT yint :
REAL32 y RETYPES yint :

VAL volume IS N*(N*N) :
[N][N][N]INT cube.sites :
[volume]INT vec.sites RETYPES cube.sites :
```

The last example here shows how a three-dimensional array may be retyped as a vector, or one-dimensional array.

[1] 15. Miscellany

We finish this introductory chapter with some miscellaneous features which are probably of lesser importance at a first reading.

[1] 15.1 Tables

One feature that was not included in the discussion of arrays and their properties was the idea of a table in occam. Tables are a means of generating an array value. Suppose that v is an INT variable, with current value 1. The table

```
[v, v+1, v+2]
```

generates an array of type [3]INT with component values 1, 2, 3. The array so generated may be referenced by subscript in the usual way:

```
[v, v+1, v+2][2]
```

or assigned to a variable:

```
INT v :
[3]INT vector :
SEQ
  ...
  vector := [v, v+1, v+2]
```

or abbreviated to a name for later use:

```
INT v, w :
...
VAL vector IS [v, v+1, v+2] :
SEQ
  ...
  w := vector[2]
```

[1] 15.2 Continuations

When writing occam code we may sometimes create expressions or lists which occupy more than one line of text. There is no explicit way in occam of indicating that one line is a continuation of the previous line. However, the general rule is that in occam we may continue from one line to another at the same level of indentation, provided that the syntax makes it clear that a continuation is intended. In other words, a line which is to be continued had better not terminate with a character which would allow the compiler to suppose that the line is complete ! Examples:

```
very.long.variable.name := very.long.variable.name +
                           even.longer.variable.name
```

is fine, but

```
very.long.variable.name := very.long.variable.name
                           +even.longer.variable.name
```

is not.

[1] 16. Postscript

In this chapter we have tried to give an overview of the occam programming language in enough detail, and with sufficient examples, to enable the reader to start writing occam programs. No attempt has been made to be comprehensive or definitive; for this reference must be made to the INMOS product definition for occam 2, which describes features that have not been covered here and, in some instances, have not yet been implemented in the currently available compiler releases. Highly recommended also is the INMOS document 'A Tutorial Introduction to Occam Programming', written by Dick Pountain, but as yet not generally available. This was an invaluable aid in the preparation of the lectures which gave rise to the present book, and the inspiration for many useful examples.

2. Implementation of simple programs

[2] 1. Transputer Hardware

The **transputer** is the 'computer on a chip' (processor, memory and communications) built by INMOS Ltd; hence the name, which is a composite of 'transistor' and 'computer'. It is a **programmable building block for concurrent systems**, spanning a range of sizes from microcomputer to supercomputer. The transputer implements the **process model of computation** embodied in its native language **occam**. Although transputers may be programmed in other languages, occam, being equivalent to assembler, is the most efficient. The transputer architecture is **wordlength independent** so that transputers of different wordlengths may be interconnected and programmed as a single system. Since all **memory is local**, the memory bandwidth grows in proportion to the number of transputers. Each transputer has an external memory interface which extends the address space into off-chip memory (it can be arranged for frequently-accessed data to be stored on chip).

Transputers use **point-to-point communication links**. Every member of the transputer family has one or more standard links which may be connected to links on other transputers to build networks of various sizes and topologies. Hence, the **communications bandwidth does not saturate as more transputers are added**. Each link provides **synchronous bidirectional communication** corresponding to two occam channels, one in each direction. **Communication via any link may occur concurrently with communication on all other links and with program execution**. An occam program is the same regardless of whether it involves communication between processes executing on different transputers or on a single transputer. More generally, a program intended for a network of transputers, may be compiled and executed on a single transputer, which shares its time between the concurrent processes. A process which is waiting for communication or timeout does not consume any processor time.

A **message** is transmitted as a sequence of **bytes**, each sandwiched between two 'start bits' and a 'stop bit', as in fig. 1.

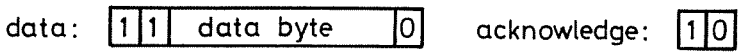

Fig.1: Communication packets

After transmitting a data byte, the sending link controller waits until an acknowledge (see fig. 1) has been received. The receiving link controller can transmit an acknowledge as soon as it starts to receive a data byte, so

transmission can be continuous (in practice, early transputers do not send the acknowledge until after the byte has been received). This protocol synchronises communication of each byte, ensuring that slow and fast transputers, and transputers of different wordlength, can communicate reliably.

All transputers support 10Mbit/sec links, some support 20Mbit/sec. INMOS supplies **link adaptors** which interface transputers to non-transputer devices. A low frequency clock (5MHz) is used irrespective of the performance of the transputers. Each transputer increments a timer which may be read in occam and used, for example in real-time systems, to determine the activity of a process. Communication depends on frequency not phase, so transputers with independent clocks can communicate reliably.

After reset, a transputer waits for the first message to be received on a link, and interprets this as a program to be loaded and executed. This provides the standard mechanism for bootstrapping a network of transputers. It is also possible to bootstrap from external ROM.

Fig. 2 is a schematic diagram of a transputer. The following is a brief specification of the hardware.

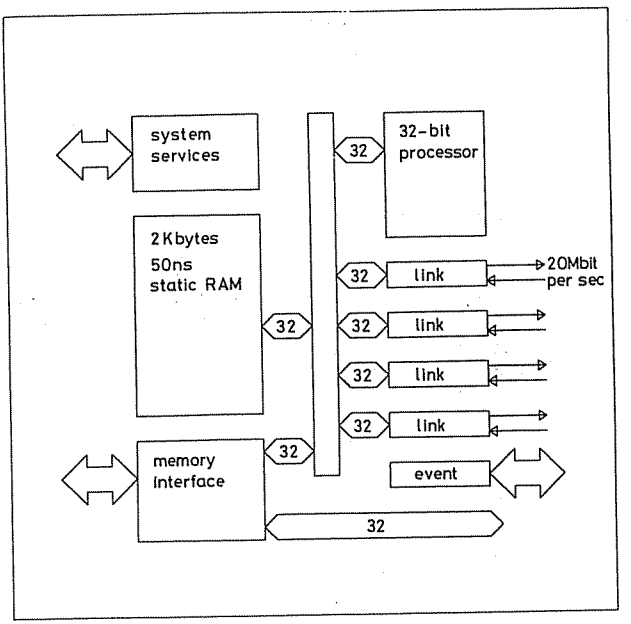

Fig. 2: Schematic diagram of a transputer

T414: 32 bit, 10 MIPS processor;
2Kbyte static memory (80Mbyte/sec access);
direct address space for up to 4Gbyte off-chip memory (25Mbyte/sec access);
typical floating point operation ≃ 240 cycles;

T212: 16 bit, 10 MIPS processor;
2Kbyte static memory (40Mbyte/sec access);
direct address space for up to 64Kbyte off-chip memory (20Mbyte/sec access);
typical floating point operation ≃ 550 cycles;

Both: single CMOS chip (1.5 micron);
4 INMOS standard, full duplex, serial links (20Mbit/sec each);
reduced instruction set, providing direct implementation of occam model;
various processor speeds e.g. T414-20 executes 20 cycles per µsec;
timer: high priority 1 tick = 1 µsec
low priority 1 tick = 64 µsec;
compilers for C, Fortran 77 and Pascal;

T800: design in preparation via ESPRIT project, see section [6] 1, page 80 for more details.

[2] 2. Configuration

Configuration is what happens at the topmost level of an occam program to determine **how the program is mounted on particular hardware**. Configuration associates specific processes with real processors, and specific occam channels with real hard links. **It does not affect the logical behaviour of the program.**

The rules for configuration depend on the particular occam system in use, and the manual will need to be consulted for details.

[2] 2.1 IBM PC + B004 Board

In the simplest case, this is configured as follows. The complete program should be packaged as an occam procedure with the channel **keyboard** and **screen** as parameters (analogous to an SC PROC). The corresponding fold should be filed and labelled EXE. It may then be compiled, linked and run, inputting from the keyboard and outputting to the screen.
e.g.

```
{{{F ascii.tsr
PROC ascii (CHAN keyboard, screen)
  ...F defs.tsr
  ...F streams.tsr
  INT char :
  SEQ
    Writes(screen, "Hello *C*N")
    char := 0
    WHILE char <> 32
      SEQ
        keyboard ? char
        Writen(screen, char)
        newline()
:
}}}
```

It is instructive to discuss parts of this program in detail.

(a) The **PROC Writes** outputs a string of characters to the screen. It is to be found in the filed fold **streams.tsr**. It is

```
PROC Writes (CHAN out, VAL []BYTE string)
  SEQ i = 0 FOR SIZE string
    out ! tt.out.int; INT string[i]
:
```

This sends the characters in the **BYTE** array **string** sequentially down the channel **out**. **SIZE string** gives the number of elements in the array **string**. The channel communication **implements a particular protocol appropriate for the PC+B004**, in which each character is converted to type **INT** and preceeded by a tag **tt.out.int** (whose value is previously defined in the filed fold **defs.tsr**). Note that if several values are to be output to the same channel sequentially, this may be written as a single line of occam in which the values are separated by semicolons, i.e. the above output is equivalent to

```
SEQ
  out ! tt.out.int
  out ! INT string[i]
```

The particular **instance** of **Writes** in **ascii** replaces the **formal parameter out** by the **actual parameter screen**, and similarly replaces **string** by **"Hello *C*N"**. Notice that the 'newline' (***N**) and 'carriage return' (***C**) characters must follow the text of the message. This is because the screen channel is buffered, and otherwise would not display the message until the buffer is full. The effect is to write **Hello** on the screen and place the cursor at the start of a new line.

(b) **keyboard ? char** inputs the code for a single character typed at the keyboard into the variable **char**. Note that this will usually be the **ASCII code** for the key typed, so that if the 1 key is pressed **char** receives the value 49, which is the ASCII code for the numeral 1. The same applies in sending characters to the screen; sending 49 will result in a 1 appearing on the screen. In occam **'x'** is the ASCII code for character **x** and is of type **BYTE**. The ASCII codes are given at the end of this chapter.

(c) The program **ascii** outputs the ASCII code for any key that is pressed, terminating when it receives the ASCII code for 'spacebar' **(32)**. Outputting a number to the screen is performed by the procedure **Writen (CHAN out, VAL INT n)**, which is too long to describe here. It has the effect of outputting the ASCII codes for successive digits of the integer **n** to the channel **out**, each preceeded by the appropriate protocol.

Clearly, if it is required to do arithmetic with the numbers input from the keyboard, they will have to be adjusted by subtracting **48** or **INT '0'**. Similarly, adding **INT '0'** converts a number of type **INT** to its ASCII code of type **INT**.

For the more general case including file I/O, the manual for the Transputer Development System on the IBM PC must be consulted.

As a simple example, consider the following program consisting of two processes running in parallel on two processors (which is appropriate for a **standalone** program on the Computing Surface; see [3] 4.1 and chapter 6).

```
... SC host.proc
... SC master.proc

{{{ declarations

{{{ hard channel placement values
VAL link0out IS 0 :
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in  IS 4 :
VAL link1in  IS 5 :
VAL link2in  IS 6 :
VAL link3in  IS 7 :
}}}

{{{ channel declarations
CHAN host.to.master, master.to.host :
}}}
... other declarations

}}}

PLACED PAR
  PROCESSOR 0 T4                    -- the host processor
    {{{ placement for host
    PLACE host.to.master AT link1out :
    PLACE master.to.host AT link1in :
    }}}
    host.proc (host.to.master, master.to.host)
  PROCESSOR 1 T4                    -- the master processor
    {{{ placement for master
    PLACE master.to.host AT link0out :
    PLACE host.to.master AT link0in :
    }}}
    master.proc (master.to.host, host.to.master)
```

host.proc runs on the host transputer (in the Edinburgh system, only **system code** is allowed to run here), and **master.proc** runs on the master transputer (which is connected to the host and graphics transputers, as well as two slave transputers, in the Edinburgh configuration, see [3] 4.1). In this example, the user program is presumed to be in **master.proc**. More generally, the master plays a controlling role (for example, having the capability of file I/O) over some number of slave transputers on which other processes are running.

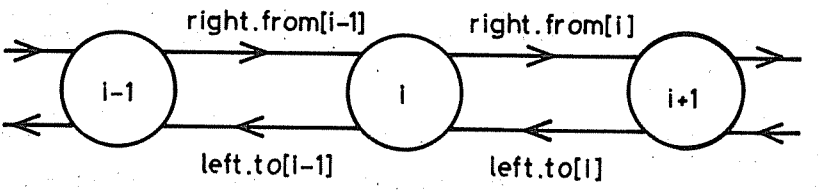

Fig. 3: Channel names in placement for singly-connected ring

Here, each bidirectional link is drawn as two lines. The 1-dimensional arrays of channels are defined in the declarations fold.

[2] 3. Timers

Objects of type TIMER behave like channels which can be input from but not output to; the value input is the current time, of type INT.
e.g.

```
TIMER clock :
INT time:
clock ? time
```

Typically,

$$1 \text{ tick} = (5 \times 64) / \text{input clock rate} = 64 \times 10^{-6} \text{ sec}$$

Whenever the value of time exceeds the maximum value that can be represented by an INT, it becomes maximum negative (2's complement) and continues counting towards zero (every 4.2 min in 16-bit, every 75 hr in 32-bit). Therefore, **time differences must be calculated in modulo arithmetic.**

It is possible to declare more than one timer (although they will all return the same value when running on the same processor). Several components of a PAR **may input from the same timer.**

[2] 3.1 Delays

A **delayed input** is an input from a timer which cannot proceed until the time has reached a certain value.
e.g.

```
PROC delay (INT interval)
  TIMER clock :
  INT now :
  SEQ
    clock ? now
    clock ? AFTER now PLUS interval
:
```

Note: no variable changes value in a delayed input; the value from `clock` is only **compared** with the value of the expression `now PLUS interval`.

`AFTER` is a comparison operator which returns a truth value.

`x AFTER y` is equivalent to `(x MINUS y) > 0`

Delayed input may be used in `ALT` to provide a real-time wait. `AFTER` may be used to check whether one time is later than another, but **care must be taken** unless the difference between the two times is known to be less than the largest (2's complement) integer.

[2] 4. Benchmark Programs

In this section, two example programs are given which make use of the occam constructs which have been covered so far. Both involve timing various activities of a single transputer. The reader should construct versions of these programs to run either on a workstation or on the Computing Surface. The actual codes written here will run on a workstation if they are put in a filed fold labelled `EXE`. On the Computing Surface in Edinburgh, a 'template' has been written for the 'master' transputer (i.e. the transputer connected directly to the local host), and in this case the `PROC` should be put in a fold marked `USER PROGRAM` in the template, and the protocol for outputting to the screen changed appropriately.

[2] 4.1 Floating Point Arithmetic

The program `flops` computes the number of additions, multiplications and divisions performed per second by a transputer. It does this by first determining the clock rate of the transputer, then counting the number of ticks of the clock between starting and ending some number of operations supplied by the user, and finally converts this into flops. The top level of the program is listed below.

```
{{{F flops.tsr
PROC flops (CHAN keyboard, screen)
  ...F defs.tsr
  ...F streams.tsr
  {{{ PROCs
  ... find.machine.type
  ... adds
  ... multiplies
  ... divides
  }}}
  INT any, num, ticks.per.sec :
  SEQ
    find.machine.type(ticks.per.sec)
    ... output machine type
    ... input number of operations
    adds(num, ticks.per.sec)
    multiplies(num, ticks.per.sec)
    divides(num, ticks.per.sec)
    keyboard ? any
:
}}}
```

The main features of this program are the following.

(a) The filed folds **defs.tsr** and **streams.tsr** contain system code and in particular the I/O utilities needed.

(b) **find.machine.type** is a **PROC** which measures the number of ticks per second of the clock on the transputer being used. It is contained in the fold **PROCs** and is:

```
{{{ find.machine.type
PROC find.machine.type (INT ticks.per.sec)
  TIMER time :
  INT start, end:
  SEQ
    time ? start
    SEQ i = 0 FOR 5
      SKIP
    time ? end
    end := end MINUS start
    IF
      end <= 1
        ticks.per.sec := 15625
      TRUE
        ticks.per.sec := 625000
:
}}}
```

This times the operation **SKIP** and, on the basis of the number of ticks, determines which of the two possible clock speeds the transputer has.

(c) The code for outputting the machine type and inputting the number of operations is contained in the following two folds:

```
{{{ output machine type
Writen(screen, ticks.per.sec)
Writes(screen, " ticks per sec*C*N")
}}}
{{{ input number of operations
Writes(screen, "How many operations ?*C*N")
Readn(keyboard, screen, num)
newline()
}}}
```

The utilities for writing to the screen were described in section [2] 2.1 and are further described in chapter 6. **Readn** is similar, except that it reads an integer, **num**, from the keyboard and echoes it to the screen. All the utilities are in **streams.tsr** and the user is encouraged to look at them.

(d) There follows three very similar procedures which compute the flops for different arithmetic operations. The first is

```
{{{ adds
PROC adds (VAL INT num, ticks.per.sec)
  INT start, end, ticks :
  REAL32 a, b, c, flops :
  TIMER clock :
  SEQ
    b := 1.2345(REAL32)
    c := 0.9876(REAL32)
    clock ? start
    SEQ i = 0 FOR num
      SEQ
        a := b + c
        b := c - a
        c := a + b
    clock ? end
    ticks := end MINUS start
    flops := ((REAL32 ROUND (3*num))*(REAL32 ROUND
             ticks.per.sec))/(REAL32 ROUND ticks)
    Writen(screen, INT ROUND flops)
    Writes(screen, " flops for addition*N*C")
:
}}}
```

The new features of this PROC are the assignment of REAL32 values to b and c and the **type conversion** required in the computation of flops to avoid overflow. REAL32 ROUND x converts x of type INT to type REAL32 and INT ROUND converts back again (rounding and applying IEEE standards). Note the use of parentheses to specify the order of the arithmetic operations; occam insists on this.

(e) Finally, keyboard ? any prevents the program from terminating before you get a chance to read the output on the screen!

Exercises

(i) Run the floating-point arithmetic benchmark on your system.

(ii) Modify the program to obtain benchmarks for integer arithmetic.

(iii) What happens if b and c are defined as VALs (and the assignments to b and c omitted, of course)?

[2] 4.2 Communications

This program computes the bandwidth (in byte/sec) for soft channel communication between parallel processes on a single transputer. (A **soft** channel implements communication between parallel processes via writing to and reading from memory, rather than via transfer along **hard** links.) A character is read from the keyboard and circulated around a ring of processes, each of which reads the character from its input channel and sends it down its output channel. The top level of the program is:

```
{{{F comms.tsr
PROC comms (CHAN keyboard, screen)
  ... master
  ... slave
  VAL INT n IS 99 :
  [n+1]CHAN ring :
  PAR
    master(ring[0], ring[n], n)
    PAR i = 0 FOR n
      slave(ring[i], ring[i+1])
:
}}}
```

This displays the overall structure, in which the **master** process sends a character around a ring of **slaves**. Channel **ring[0]** 'emerges' from the **master** process and 'enters' the first **slave** process, from which channel **ring[1]** 'emerges', etc., eventually joining back to the **master**. Details of those parts of the program which differ from **flops** are given below.

(a) The job of timing the transfers and computing the bandwidth is done by the **master** process:

```
{{{ master
PROC master (CHAN to.ring, from.ring, VAL INT n)
  ...F defs.tsr
  ...F streams.tsr
  ...  find.machine.type
  INT ticks.per.sec, char, start, end, ticks, any :
  REAL32 bytes.per.sec :
  TIMER clock :
  SEQ
    find.machine.type(ticks.per.sec)
    ... output machine type
    ... input character from keyboard
    ... time the passage around the ring
    ... compute bandwidth & output result
    keyboard ? any
:
}}}
```

This has several PROCs in common with **flops**. The two new ones, and the basic parts of this program are:

```
{{{ time the passage around the ring
clock ? start
to.ring ! char
from.ring ? char
clock ? end
ticks := end MINUS start
screen ! tt.out.int; char
newline()
}}}
```

and

```
{{{ compute bandwidth & output result
bytes.per.sec := ((REAL32 ROUND (4*(n+1)))*(REAL32 ROUND
                  ticks.per.sec))/(REAL32 ROUND ticks)
Writen(screen, INT ROUND bytes.per.sec)
Writes(screen, " bytes per sec*C*N")
}}}
```

These should be self-explanatory. As before, real arithmetic is used to compute the bandwidth in order to avoid overflow.

(b) The **slave** process, which is replicated **n** times and run in parallel with the **master** (why in parallel ? - because, otherwise the program would **deadlock** with the **master** trying to send and nobody else ready to receive) is very simple:

```
{{{ slave
PROC slave (CHAN in, out)
  INT char :
  SEQ
    in ? char
    out ! char
:
}}}
```

Exercise

Modify this program so that each **slave** sends the message **Hello from slave number**, followed by its own ID, around the ring and onto the screen.

character	decimal	character	decimal	character	decimal
line feed	10	A	65	a	97
carr/return	13	B	66	b	98
space	32	C	67	c	99
!	33	D	68	d	100
"	34	E	69	e	101
#	35	F	70	f	102
$	36	G	71	g	103
%	37	H	72	h	104
&	38	I	73	i	105
'	39	J	74	j	106
(	40	K	75	k	107
)	41	L	76	l	108
*	42	M	77	m	109
+	43	N	78	n	110
'	44	O	79	o	111
-	45	P	80	p	112
.	46	Q	81	q	113
/	47	R	82	r	114
0	48	S	83	s	115
1	49	T	84	t	116
2	50	U	85	u	117
3	51	V	86	v	118
4	52	W	87	w	119
5	53	X	88	x	120
6	54	Y	89	y	121
7	55	Z	90	z	122
8	56				
9	57	[	91	\	92

ASCII characters and the decimal code equivalent.

3. Case Study: Cellular Automata

[3] 1. Introduction

The purpose of this chapter is to investigate how to implement an occam program on a network of transputers. The example chosen is that of a **1-dimensional cellular automaton**. This is conceptually very simple, has a high degree of parallelism and produces interesting pictures on a simple VDU screen.

Cellular automata are discussed in some generality in chapter 4, where it is emphasised that they constitute a generic class of problems for parallel computers. Here, we will focus on the simplest of them as this exposes the structure of the program, and especially the communication harness, without the distraction of complicated calculations. You will see that parallel implementations of this type occur quite often in real applications.

The cellular automaton we are going to simulate consists of a linear chain of cells with periodic boundary conditions (i.e. a closed ring). Each cell may exist in one of two states, 0 or 1, represented by ' ' and '*', respectively, on the screen. The system starts off (at time 0) with any initial state input from the keyboard by the user. It subsequently evolves in discrete time steps in which every cell is updated simultaneously according to a simple rule, and the new state is drawn on the next line of the screen. The rule is that the state of each cell is given by the sum of the states of itself and its two nearest neighbours at the previous time step, modulo 2.

[3] 2. The Process for One Cell

The obvious way to map this problem onto a network of transputers is **geometrically** (see chapter 4 on Parallel Algorithms), i.e. put consecutive groups of cells on consecutive transputers connected in a closed ring, as shown in fig. 1.

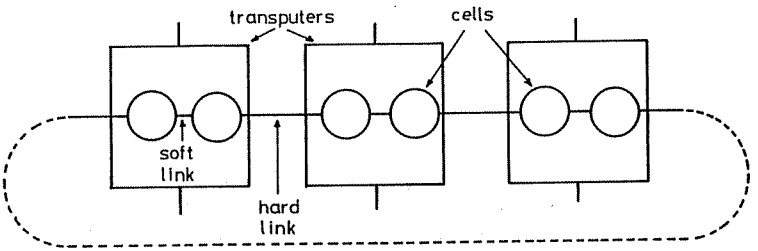

Fig. 1: Geometric decomposition of 1-dimensional cellular automaton

This uses up two of the links on each transputer. At least one transputer (the **master**) will have to take charge of I/O from the keyboard and to the screen, but either of the spare two links are available for this. On the Computing Surface configured as in chapter 6, the master should be connected directly to the **host**, but this will be considered further when we discuss placement.

Let us begin by considering the process which will reside on any one of the **slaves**. This will consist of processes for some number of cells, **running in parallel** (this is just the physical situation). All the cell processes are identical. Each cell process must first input its initial state, and then perform a sequence of updates (taking information about the states of neighbouring cells from neighbouring cell processes). After every update, each cell must send its current state to the master for outputting to the screen. Finally, the cell must be ready at all times to receive a message, originating from the master, telling it to terminate. Adopting a 'top-down' approach to this program, the cell process might look like this:

```
PROC cell (CHAN left.in, left.out, right.out, right.in)
  [[[ PROCs
  ... initialise
  ... update
  ... send.state
  ]]]
  INT state :
  BOOL running :
  SEQ
    initialise(left.in, right.out, state)
    running := TRUE
    WHILE running
      SEQ
        update(state)
        send.state(left.in, right.out, state, running)
:
```

Here the channels in and out of the cell are labelled as shown in fig. 2.

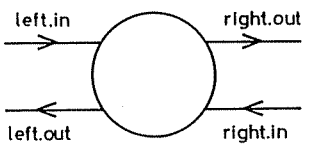

Fig. 2: Soft channels for a cell

The simplest process to describe is **update**, which broadcasts the current state of the cell (called **state** in the above program) to its neighbours **in parallel** with inputting the states of its neighbours. It is essential that these processes run in parallel to avoid **deadlock**. Once, these communications have taken place, the cell may update its state using the rule described in section [3] 1. The occam process for this is:

The code here allows for the possibility of an incomplete initial state being sent out by the master; cells which receive an 'end of initial state' command instead of a 1 or 0, set their initial state to 0.

Following the command structure outlined above, the cell process may be completed with

```
PROC send.state (CHAN in, out, VAL INT state, BOOL running)
  INT x :
  BOOL talking :
  SEQ
    talking := TRUE
    WHILE talking
      SEQ
        in ? x
        IF
          x > 1
            SEQ
              out ! state; x
              talking := FALSE
          x < 0
            SEQ
              out ! state; x
              talking := FALSE
              running := FALSE
          TRUE
            out ! x
:
```

This implements the final piece of the command structure, which will terminate the cell process on receipt of a negative integer, having previously output its final state followed by the negative integer in order to terminate the other cells further around the ring. The first process to terminate is the first in the ring, the last process to terminate is the master, after it has received the final state of the automaton and output it to the screen.

[3] 3. Connecting Processes Together & I/O

This is the major exercise for the reader. In the next section the placement for rings of 10 and 40 transputers will be discussed. These include the master in the configuration assumed as in fig. 3 (see [3] 4.1). Those that have a workstation for this exercise may choose a slightly different configuration.

The cell process is complete (unless you choose to change the control structure). You will need to write a master process which handles I/O with your terminal and sends appropriate commands around the ring (see [3] 4.4 for a solution). You might decide, initially, to restrict the freedom of the user to choose arbitrary initial conditions (e.g. setting up the initial state in the master, rather than inputting it from the keyboard) and to terminate the program at will (e.g. fixing the number of time steps). In some sense, the most elementary initial states are either a random sequence of 0s and 1s, or a single nonzero cell.

Also, depending on how many transputers you include in the ring, you may choose to have one, two, or more cell processes per transputer in order that the cellular automaton picture approximately fills the screen. Note that if you place more than one cell on each transputer, some of the channels `left.in`, `left.out`, `right.in` and `right.out` will be **soft channels** internal to one transputer, while others will be `PLACED AT` hard links. This in no way alters the logical structure of the program.

[3] 4. Ring Placement

[3] 4.1 Use of 10 Transputers

For the course for which these notes were prepared, the transputer array at Edinburgh was connected up as a 4 x 10 array, with periodic boundary connections (i.e. **2-dimensional torus**) and with the **host** and **graphics** transputers inserted on the links of the **master** transputer in the 'short' and 'long' directions respectively. This is shown in fig. 3. The placements given are based on the assumption that you are **booting the code from within OPS and hence placements for the host** are omitted (c.f. [2] 2.2, and see chapter 6 for details of this).

To begin with, recall that the placement for the `master` alone is:

```
PROCESSOR 1 T4
  master()
```

where the placement for the channels `to.host` and `from.host` are contained in `master`.

There are many ways of placing rings of various lengths on the 4 x 10 array. One of the simplest is the ring of 10 transputers, closed by the graphics transputer G, shown by the solid lines in fig. 3.

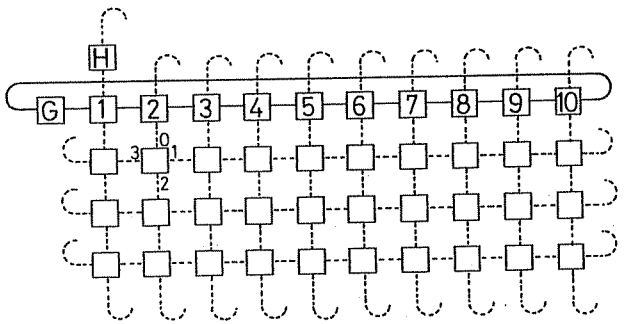

Fig. 3: Ring of 10 transputers embedded in the 4 x 10 array

The graphics transputer may play a completely passive role as `message.passer`:

```
PROC message.passer (CHAN left.in, left.out,
                             right.out, right.in)
  PAR
   WHILE TRUE
     INT any :
     SEQ
       left.in ? any
       right.out ! any
   WHILE TRUE
     INT any :
     SEQ
       right.in ? any
       left.out ! any
:
```

where the channels are named as in fig. 2.

In the 'Edinburgh configuration', the hard links are numbered as in fig. 3. Consequently, the following placement is appropriate:

```
... SC master
... SC slave
... SC message.passer
{{{ declarations
... hard channel placement values
{{{ channel declarations
[11]CHAN right.from, left.to :
}}}
... other declarations
}}}
PLACED PAR
  PROCESSOR 1 T4                        -- master processor
    PLACE right.from[0] AT link3in  :
    PLACE left.to[0]    AT link3out :
    PLACE right.from[1] AT link1out :
    PLACE left.to[1]    AT link1in  :
    master(right.from[0], left.to[0], right.from[1], left.to[1])
  PLACED PAR i = 2 FOR 9
    PROCESSOR i T4                      -- slave processors
      PLACE right.from[i-1] AT link3in  :
      PLACE left.to[i-1]    AT link3out :
      PLACE right.from[i]   AT link1out :
      PLACE left.to[i]      AT link1in  :
      slave(right.from[i-1], left.to[i-1], right.from[i], left.to[i])
  PROCESSOR 11 T4                       -- graphics processor
    PLACE right.from[10] AT link3in  :
    PLACE left.to[10]    AT link3out :
    PLACE right.from[0]  AT link1out :
    PLACE left.to[0]     AT link1in  :
    message.passer(right.from[10], left.to[10],
                   right.from[0], left.to[0])
```

The arrangement of soft channels is shown in fig. 3 of chapter 2, page 34.

[3] 4.2 Use of 40 Transputers

Placing a closed ring of 40 transputers, including the **master**, but not the **host** or **graphics**, is harder to visualise, but only slightly more difficult to program. There are, of course, several solutions to this problem, but perhaps the simplest is shown in fig. 4, where the hard links that are not needed in the ring have been omitted.

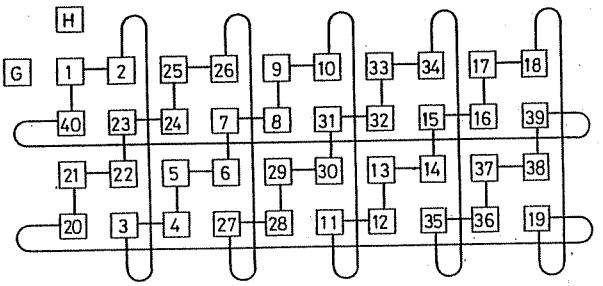

Fig. 4: A closed ring of 40 transputers on the 4 x 10 array

The advantage of this choice (apart from the neat property described in the next section) is that only two different types of placement are needed: one for **odd-numbered** processors and one for **even-numbered** processors, as shown in fig. 5.

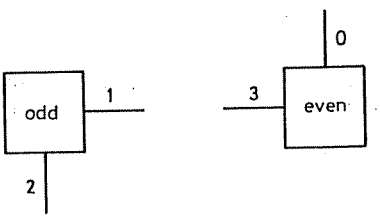

Fig. 5: The two types of node (and hard links) in fig. 4

The placement is as follows:

```
... SC master
... SC slave
{{{ declarations
... hard channel placement values
{{{ channel declarations
[40]CHAN right.from, left.to :
}}}
... other declarations
}}}
PLACED PAR
  PROCESSOR 1 T4                          -- master processor
    PLACE right.from[0] AT link2in  :
    PLACE left.to[0]    AT link2out :
    PLACE right.from[1] AT link1out :
    PLACE left.to[1]    AT link1in  :
    master(right.from[0], left.to[0], right.from[1], left.to[1])
  PLACED PAR i = 1 FOR 20
    VAL id IS 2*i :
    PROCESSOR id T4                       -- even slave processors
      PLACE right.from[id-1]  AT link3in  :
      PLACE left.to[id-1]     AT link3out :
      PLACE right.from[id\40] AT link0out :
      PLACE left.to[id\40]    AT link0in  :
      slave(right.from[id-1], left.to[id-1],
                    right.from[id\40], left.to[id\40])
  PLACED PAR i = 1 FOR 19
    VAL id IS (2*i)+1 :
    PROCESSOR id T4                       -- odd slave processors
      PLACE right.from[id-1] AT link2in  :
      PLACE left.to[id-1]    AT link2out :
      PLACE right.from[id]   AT link1out :
      PLACE left.to[id]      AT link1in  :
      slave(right.from[id-1], left.to[id-1],
                    right.from[id], left.to[id])
```

[3] 4.3 Two Complete Rings ?

The second nice feature of the placement in fig. 4, alluded to above, is that **the links which have not been used form a completely separate closed ring**, which includes both the host and graphics. Thus, our 4 x 10 array admits two disjoint closed rings. (As an exercise, you might like to prove the result for all rectangular arrays.)

A slight rearrangement of the placement permits one ring to include the graphics processor and the other the host, as shown in fig. 6.

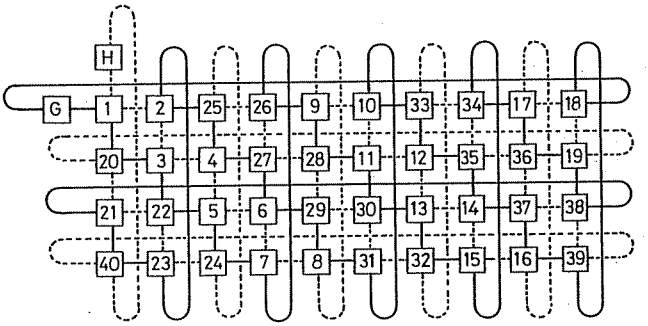

Fig. 6: Two closed disjoint rings

This placement is appropriate for applications making use of real-time graphics display. Geometric decomposition of a problem may take place on the **host** ring, in which neighbouring transputers on this ring contain neighbouring data (see chapter 4). The **graphics** ring, being completely disjoint from the **host** ring, may be used to output graphics data without interfering with the computation going on in the **host** ring. Note, however, that neighbouring transputers on the **host** ring are **not neighbours** on the **graphics** ring. Data arriving at the graphics processor will be jumbled up, and so it is necessary for each data packet to carry an identifying label.

[3] 4.4 The Master Process for the Cellular Automaton

The **master** process for the cellular automaton requires access to the channels **screen** and **keyboard**, as well as to the ring connecting up the cells. In the code for the **master** given below, these are included as formal channel names in the procedure definition. In practice, **master** should be run in parallel with system code, and it is best for it to be packaged in some sort of template (see chapter 6).

The toplevel of **master** is

```
PROC master ( CHAN left.in, left.out, right.out, right.in,
                   keyboard, screen)
  ... PROC send.state
  INT flag:
  SEQ
    ...  input state
    ...  run until signalled to terminate
:
```

The **input state** fold inputs characters from **keyboard** until a character other than ' ' or '*' is encountered. The process **initialise**, running on each cell, handles the case when too few initial states are input. If too many are received, these are absorbed by the code in the fold called **check for excess characters**.

```
{{{  input state
BOOL inputting:
SEQ
  inputting := TRUE
  WHILE inputting
    INT char:
    SEQ
      keyboard ? char
      IF
        char = (INT ' ')
          right.out ! 0
        char = (INT '**')
          right.out ! 1
        TRUE
          SEQ
            right.out ! 2
            inputting := FALSE
  {{{  check for excess characters
  BOOL checking:
  SEQ
    checking := TRUE
    WHILE checking
      SEQ
        left.in ? flag
        IF
          flag = 2
            checking := FALSE
          TRUE
            SKIP
  }}}
}}}
```

The **run until signalled to terminate** fold is opened below. The interrupt from **keyboard** is placed at high priority to ensure that it gets noticed. A cell process has not been included in **master**, although it might have been, so the master must act as a message-passer between the cells on its left and right.

```
{{{  run until signalled to terminate
BOOL running:
SEQ
  running := TRUE
  WHILE running
    INT any:
    PRI ALT
      keyboard ? any
        SEQ
          right.out ! -1
          running := FALSE
          send.state (left.in, screen)
      TRUE & SKIP
        INT right.state, left.state:
        SEQ
          right.in ? right.state
          left.out ! right.state
          left.in ? left.state
          right.out ! left.state
          right.out ! flag
          send.state (left.in, screen)
}}}
```

Finally, the process **send.state** is used to output the latest state of the automaton to **screen**. Note that it is not the same as the process of the same name running on each cell.

```
{{{ PROC send.state
PROC send.state (CHAN in, out)
  BOOL running:
  SEQ
    running := TRUE
    WHILE running
      INT x:
      SEQ
        in ? x
        IF
          x = 0
            out ! INT ' '
          x = 1
            out ! INT '**'
          TRUE
            SEQ
              out ! INT '*N'
              running := FALSE
:
}}}
```

4. Parallel Algorithms

[4] 1. Introduction

Parallel architectures of the **MIMD** (Multiple Instruction Multiple Data) type, and particularly distributed memory systems like the transputer, offer a modular approach to the construction of computers which may be tailored to suit individual applications. [For the purpose of this chapter, SIMD (Single Instruction Multiple Data) machines like the DAP and Connection Machine will be ignored.] Parallel processing offers a speed-up beyond the technological limitations on single-processor systems which, at the high-performance end, may be an order of magnitude more cost-effective than vector supercomputers.

Ideally, a program runs N times faster on N processors than on a single processor, although the actual speed-up may be much less. The design of algorithms to achieve this sort of speed-up is an active area of research. Since the algorithm, programming language and hardware are intimately connected, this exercise is difficult to carry out in general. Occam and the transputer constitute a world in which these questions can be addressed and, in a growing number of cases, answered. Unfortunately, parallel computers are not very forgiving; the difference between the performance of a good and a bad program is much greater than for a serial computer. The crux of the matter is not in the writing of a program, but in the way in which an application is mapped onto the architecture. To do this efficiently the user must 'think parallel'.

In addition to the normal considerations of numerical analysis, the user must now take account of

(a) how data is to be distributed in memory;
(b) how computations are distributed among processors;
(c) inter-processor communications;
(d) inter-processor connections, if reconfigurable.

The aim is to match the parallelism of the algorithm to the parallelism of the computer in such a way as to minimise the execution time of the program. **At any stage within an algorithm, the parallelism of the algorithm is the number of operations that are independent and can therefore be performed concurrently.** This may vary from stage to stage. The **natural hardware parallelism is the number of processors that may run concurrently**, including both arithmetic and link processors on the transputer. In devising parallel algorithms we are concerned with maximising

$$\text{efficiency} = t_1 / N \times t_N$$

t_1 = time taken by program on one processor

t_N = time taken by program on N processors

[4] 2. Independent Tasks

[4] 2.1 The Task Farm

One of the simplest, and often the most efficient, ways of exploiting parallel processing is to distribute independent tasks to each of the processors. Such a configuration of the system may be called a **task farm**, and in general it will consist of a **'master' processor**, whose job it is to distribute the tasks and collect the results, and some number of **'slave' processors**, which actually do the work. Many different organisations of processors are possible, e.g. a tree with the master as a root, or a continuous chain beginning with the master, as shown in fig. 1.

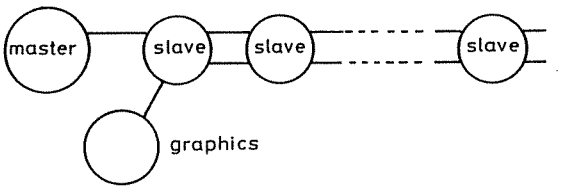

Fig. 1: A transputer pipeline.

Of course, other processors may be added for example to handle external I/O, graphics, or to perform a statistical analysis of the results from each of the tasks.

In the simplest situation, each slave processor executes the **same serial program** on its own data set. The assignment of tasks by the master processor becomes part of the operating system and can be made transparent to the user. All that the user needs to supply is a serial program and some number of data sets requiring processing. In order to make efficient use of a large processor array in this way, the number of data sets should be very large. The major difficulty is in achieving sufficiently fast data-transfer rates between magnetic tapes, or disks, and the slave processors, so that none of them is held-up waiting for others to read or write data. This may be solved using distributed mass storage devices, each associated with a small group of slave processors and controlled by some peripheral processor. In the situation in which the I/O bandwidth is high enough, such a task farm can achieve a speed-up relative to one processor equal to the number of processors.

A rather more sophisticated operating system would permit more than one user program to run on a task farm at any one time, each user being allocated a portion of the farm. It would be important, under such circumstances to be able to trap wayward programs to prevent them corrupting others.

[4] 2.2 Ray Tracing

Another variant on the task-farm approach, is when a single computation can be divided up into many independent sub-tasks which can be farmed out amongst the slaves. A successful implementation of this is ray tracing, which is a way of displaying 3-dimensional pictures on a 2-dimensional screen (Dettmer, 1986). This involves setting up the 3-dimensional 'world', i.e. the objects which are to be viewed through the screen, and then devising a mapping of this onto the screen.

The basic idea behind ray tracing is to reproduce what happens in a pinhole camera. The image on the screen is built up from rays of light, coming from the objects, which pass through the pinhole. These rays may be identified by starting at the screen and tracing back, through the pinhole, onto a surface in the 3-dimensional world. Each ray is then reflected backwards to determine whether it comes from another surface or from a light source. This backward tracing continues until the ray ends at a light source or passes out of the world. Once the source of a ray has been identified, the path of the ray is retraced from the source to determine the colour and brightness of the corresponding pixel on the screen. The complete picture is built up by tracing one ray for each pixel. The paths of the reflected rays and the levels of illumination depend on the nature of the light source (e.g. ambient light, which is uniform in all directions, or point sources) and on the type of reflecting surface (e.g. matt, from which reflection is diffuse and in all directions, smooth, which reflects an incident ray as a cone of light, or mirrored, in which the light is reflected as a single ray); other optical effects, such as refraction by transparent objects, can also be included.

The ray tracing algorithm involves a great deal of computation. For each ray, the first task is to determine the point at which it strikes a surface in the 3-dimensional world. This involves solving a system of linear equations, given the equations of all the objects. The properties of the surface determine how the ray is reflected, and the procedure is repeated. This can become very complicated in the case of multiple reflections from smooth surfaces. Although computationally intensive, the algorithm is highly parallel as all of the rays are completely independent.

The arrangement of transputers in fig. 1 is appropriate for this. Portions of the screen are distributed by the master transputer down the pipeline. Each slave takes a job from this stream, computes the image corresponding to that portion and sends the result back down the chain to the graphics processor.

[4] 2.3 The Mandelbrot Set

Much the same technique may be used to compute the mathematical structure known as the Mandelbrot set (Mandelbrot 1982). This has become established as a benchmark for MIMD computers, because the algorithm is easy to state and computationally intensive, involving many independent computations of varying length. Consequently, it is susceptible to the same parallel attack as ray tracing. The set has a very intricate, fractal structure, which can provide hours of artistic pleasure, provided portions of it can be computed quickly.

The set was discovered in 1980 by Mandelbrot, using computers at IBM. Fig. 2 is a photograph of part of this set in the final stages of computation by the Computing Surface at Edinburgh.

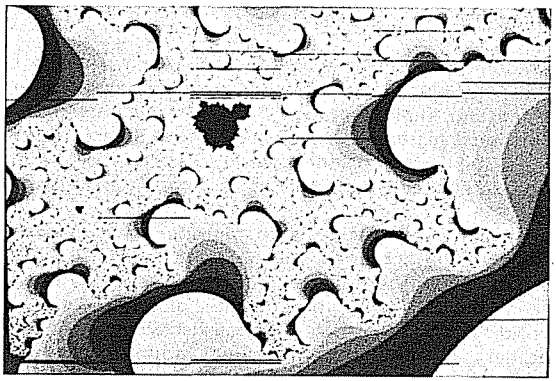

Fig. 2: Photograph of partially completed image of the Mandelbrot set.

The picture represents part of the complex plane of the variable $c = p + iq$ (through coordinates p and q: $-2.25 < p < 0.75$, $-1.5 < q < 1.5$). For each pixel, which defines the value of c, the sequence of complex numbers z_n is generated via the feedback loop:

$$z_{n+1} = z_n^2 + c$$

Either the z's are attracted into a closed cycle, or they tend to infinity. The points that cycle correspond to values of c in the Mandelbrot set. So for each value of n, $r = |z_n^2|$ is computed. If $r > 100$ then choose colour n for that pixel and move on to the next pixel; if n = maximum number of colours, set the pixel to black and go on to the next pixel; otherwise calculate a new z (see Peitgen 1986 for a fuller description of the algorithm).

Clearly, the number of times the feedback loop is executed varies from pixel to pixel. Each slave transputer in fig. 1 may accept portions of the screen for computation, as distributed by the master. Provided the screen is divided up into many more small tasks than there are transputers in the chain, the algorithm naturally load-balances. Transputers given black areas take longer to complete than others, but no-one is held up waiting for their result. On the Edinburgh Computing Surface, with 39 slave transputers, a 576 x 768 screen image is typically completed in a few tens of seconds. Because of the high degree of parallelism, a linear speed-up by a factor of 10 may be achieved using 10 times as many transputers.

[4] 3. Geometric Parallelism

[4] 3.1 Introduction

Geometry is based on the concept of distance, and the **geometric decomposition** of a problem divides the data up into subsets such that the data points in any one subset are in some sense closer to each other than to the data points in any other subset. The problem possesses **geometric parallelism** if, in addition, the algorithm involves only **local operations** i.e. connecting data points that are close together.

As an example, consider the computer simulation of fluid flow. The region of space occupied by the fluid may be divided up into subregions, equal in number to the number of available processors. Each processor is given the responsibility of handling the fluid in one subregion. Because the behaviour of a tiny fluid element is determined only by the fluid elements immediately surrounding it (which exert forces on it), the evolution of the fluid in the interior of each region is determined entirely by data which is present in that processor's local memory. However, fluid may flow across the boundary of one subregion into the neighbouring subregion. The data corresponding to those fluid elements that flow across the boundary must be transferred between the memories of the processors handling the two subregions. Typically then, there must be a transfer of surface data, while interior (or bulk) data remains local to a single processor. The amount of data which must be **transferred** relative to the amount which can be **processed internally** goes like the **ratio of the surface area to the volume**. This is directly related to the **balance of computation vs communication**. It is therefore important to pick subregions which have as small a surface area as possible for fixed volume. This maximises the likelihood that it will be possible to overlap communication of boundary data by computation using only bulk data.

The communication time is also proportional to the distance between processors. So, in order to keep down the communication time, the processors handling neighbouring subregions of the fluid should themselves be close together; ideally they should be neighbours. This completes the geometric decomposition i.e. **the processor array should have, as far as possible, the same geometry as the system being simulated.**

At this point two general questions may be raised:

(i) The need for the processor array to have the same geometrical configuration as the system being simulated comes about because, at the present level of technology, **communication is expensive**. However, communication chips with transfer rates comparable to current memory-access times are being developed. These may be available by 1990. The geometric decomposition onto arrays incorporating this technology will be much less restrictive. The aim is for such arrays to appear to the user as shared-memory machines, in which every processor can access every other processor's memory at negligible cost. If you can afford to wait until 1990, then the problems of geometric decomposition need not concern you greatly! However, this approach is bound to be expensive.

(ii) Transputers only have four links, and hence at most four nearest neighbours, yet the number of neighbours of a subregion in 3-dimensional space (discretised on a regular cubic lattice) is six. It is clear that we cannot simply map subregions onto individual transputers, as implied above. The solution is to build **supernodes**, such as in fig. 3. These have six links and could be used as building blocks for the processor array, with subregions being mapped onto supernodes. This introduces the extra complication of distributing the data in one subregion over the transputers within one supernode, and of handling internal communications. But there is little choice until a 6-link transputer appears on the market! (A fast local switch might also solve the problem, but could be expensive.)

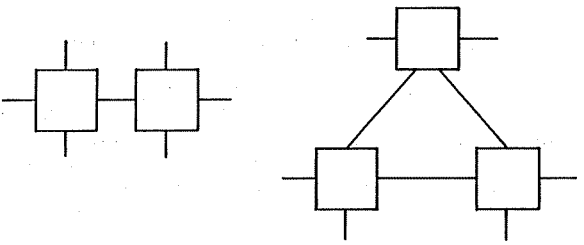

Fig. 3: Examples of 6-link supernodes.

Thus, in geometric parallelism a physical system is simulated on a **homogeneous** array of supernodes. The same program runs on each supernode, operating on local data and transferring boundary data to neighbouring supernodes as necessary. Of course, there will be a few other transputers running special tasks such as external I/O, monitoring and data analysis. Each of the supernodes is brought into synchronisation with its neighbour when boundary data is transferred. Since each supernode is doing approximately the same amount of work, this effectively brings the supernodes into something akin to lockstep. This is how an MIMD array operates in **SIMD mode**. It can be a highly efficient mode of operation for big problems, and is relatively simple to program because of the homogeneity. There may be difficulties if the algorithm requires global information at any stage, but this will be dealt with later.

[4] 3.2 Cellular Automata

Many problems in the computer simulation of systems in science and engineering can be tackled using geometric decomposition e.g. weather, wind tunnels, oil reservoirs, finite element analyses of structures, semiconductors, molecules, image processing, the interactions of elementary particles etc.. The simplest of all, and arguably the most fundamental, are **cellular automata** (CA) (see Wolfram 1986 for a collection of modern papers on the subject).

CAs were invented by von Neumann around 1950 during his search for a self-replicating machine, i.e. a machine capable of constructing exact copies of itself, given an appropriate supply of material. Ulam suggested he restrict his attention to **uniform cellular space** i.e. space filled with cells, each of which may exist in a **finite number of states** including the empty state, and evolving by discrete time steps in lockstep, via **transition rules which depend only on the states of nearby cells**. Von Neumann was then able to prove that, if each cell could exist in one of 29 states and had 4 orthogonally adjacent neighbours, then there is a configuration of some 200K cells that contains a universal computer (Turing machine) and hence is a universal constructor. Subsequently, many CA 'games' have been devised, in 1-, 2- and 3-dimensions, of which perhaps the most famous is Conway's 'Life' (Gardner 1970).

There seems no limit to the application of CA ideas: from self-replicating moving automata resulting from complex transition rules in the primordial soup of amino acids, to board games like chess, image processing techniques, self-learning machines and even the universe itself. From our point of view, it is important to note that CAs model parallel computers, such as DAP and CLIP (Cellular Logic Image Processor). It is not surprising then that CA systems map very naturally onto parallel architectures and that the natural mapping is a geometric one.

The simplest CA consists of a line of cells a_i, i=1,...,n, with periodic boundary conditions ($a_{n+1} = a_1$), each of which can exist in one of two states $a_i = 0$ or 1. This system evolves in time according to a deterministic rule, applied simultaneously to every cell, whereby a_i at the next time step depends only on the present state of itself and its nearest neighbours i.e., in general,

$$a_i(t+1) = f(4a_{i-1}(t) + 2a_i(t) + a_{i+1}(t) + 1)$$

It is easy to see that there are 2^8 possible rules, labelled by rule numbers r = 0 to 255, such that f(x) is defined to be the x^{th} digit in the binary representation of r (reading from the least significant digit). Examples of the patterns generated from a single nonzero seed are shown in fig. 4; clearly, there is a wide diversity of behaviour, even in such a simple system as this. [For drawing coloured pictures one must consider cells with a number of states equal to the number of colours and generalise the above rule accordingly (see Wolfram 1986).]

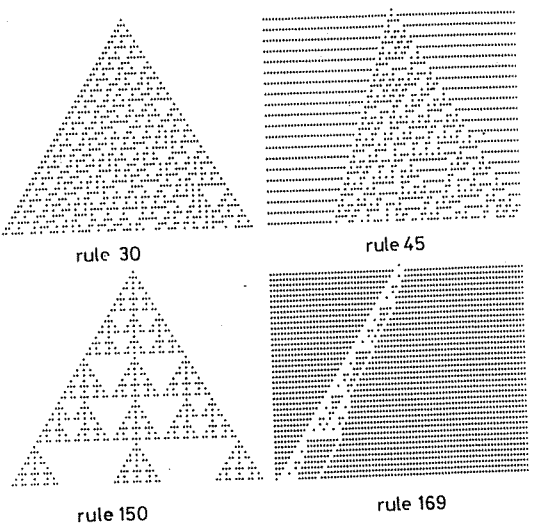

Fig. 4: Examples of 1-dimensional CA patterns.

The simplest implementation of this algorithm is on a chain of transputers, as shown in fig. 1. An occam procedure which updates a cell may run in parallel with the updating of every other cell, communicating with left and right neighbours through channels. In the simplest mapping, one such procedure is placed on each transputer and the channels are mapped onto transputer links. Bigger CA systems may be simulated by placing more than one cell on each transputer, in which case some of the channels are internal soft channels (i.e. memory locations). Since there are two more spare links per transputer, these may be conveniently used to form a second chain containing the graphics processor, down which display information is sent. If the same links are used to pass both graphics and nearest-neighbour information, then these data packets must be preceeded by protocols to identify them and their destination. Clearly, it is easier to overlap communication by computation as the number of CAs per transputer increases.

Anyone spending time exploring the different CA rules will quickly discover situations in which most of the cells are dead and any interesting activity is confined to a relatively small region. The geometric decomposition of the problem then results in most of the transputers doing no useful work. This situation becomes even more pronounced in higher dimensions e.g. in Life. The rule for Life is the following. Each cell in a 2-dimensional square array may be alive or dead. Its transition to the next generation is determined by the states of the surrounding 8 cells: a live cell with 2 or 3 live neighbours survives into the next generation, otherwise it dies (of loneliness, or overcrowding); a dead cell surrounded by exactly 3 live neighbours gives birth and is alive in the next generation, otherwise it remains barren. Conway was seeking a model of bounded growth. However, the AI group at MIT soon found a 'glider gun' which emits a continuous stream of 'gliders' (groups of 5 live cells which travel with constant velocity), and, shortly after, a 'breeder' which endlessly produces 'glider guns'. An 'acorn' starting configuration has been found, which grows for 5206 cycles into a stable 'oak', so a rich variety of behaviour is possible. It would be difficult to devise any decomposition other than the obvious geometric one which could efficiently compute all these different behaviours, particularly because any sophisticated algorithm is likely to make use of global information.

2-dimensional CA-like models cover some important applications, apart from these games. For example, much of image processing is to do with applying local cellular logic rules to picture elements. An application of potential importance for engineering has come from recent work on the Connection Machine which shows that CAs can be used to model the Navier-Stokes equations of fluid flow (see Wolfram 1986). In the simplest case, 2-dimensional flows are represented by CAs consisting of particles moving on a triangular lattice. The system evolves in discrete time-steps in which first each particle moves from one lattice site to the next in the direction of its velocity vector, then it collides with any other particles which arrive at the same site, conserving particle number and momentum. Hydrodynamic variables are computed by averaging over the particles in subregions of, say, 64 x 64 sites. Since the occupancy of any site is restricted to no more than one particle moving in each direction, the simulation can be efficiently coded in bits, or short integers, which together with the obvious geometric parallelism, makes it ideal for SIMD arrays like the DAP and Connection Machine.

Models of magnetism give rise to similar simulations where probabilistic transition rules are needed to represent thermal fluctuations. The simplest is the Ising model which restricts the atomic spin at each site of a regular lattice to the values 0 or 1. These spins interact with their nearest neighbours only, tending to align with them. The thermodynamics is generated by a Monte Carlo algorithm.

The spin at each site flips between the two allowed values according to a probabilistic rule which depends on the energy change and the temperature.

In both the above examples, the configuration of the system is represented by a small number of bits at each site of a regular lattice and the variables evolve in parallel (though there is a subtlety in the thermodynamic case) in discrete time-steps according to local rules. The geometric decomposition consists of mapping subregions of the lattice onto a 2-dimensional array of transputers (neighbouring subregions onto neighbouring transputers) in the natural connectivity of a 'computing surface'. The actual shape of the transputer array may depend on the shape of the system being simulated e.g. an elongated rectangle, possibly even a linear chain, for fluid flow down a channel (with non-slip boundary conditions implemented using special CA rules at the edges), or a square array with periodic boundary conditions (obtained by joining links on opposite edges) for the Ising model. As regards the efficiency of these implementations, the general discussion in section 3.1 applies, and computation will completely overlap communication provided the subregions are large enough. Similar types of simulations in 3 or more dimensions may be performed using supernodes with an appropriate number of external links.

The advantage of supernodes in this context is that extra links may be built in for getting data on and off the array. This is particularly important if the simulation happens to be feeding a real-time graphics display. However, in many situations, data needs to be moved only at the beginning and end of a run. This then neither interferes with the main part of the program, nor has to be particularly efficient.

[4] 3.3 Partial Differential Equations

The situation when solving partial differential equations (PDE) by the method of **finite differences** is almost identical to that described above, except that floating point arithmetic is used instead of bit-manipulation. The same sort of geometrical decomposition applies onto a supernode array with the same geometrical configuration as the region of space (and time) in which the PDE is to be solved, and iterative solutions typically consist of the application of local transition rules.

As an example, consider the solution of Poisson's equation,

$$\nabla^2 u = f$$

for u = u(x,y) in a rectangle, given the values of f = f(x,y) everywhere and of u on the boundary. The first step is to replace space by a rectangular grid of points and the spatial derivatives by finite differences. The simplest approximation is to use the 5-point star stencil in fig. 5. If the variables on the grid points are u(i,j) etc., this means

$$\nabla^2 u \rightarrow u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) - 4u(i,j).$$

A relaxation algorithm for the solution rewrites the discretised Poisson equation as

$$u(i,j) = [\ u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) - f(i,j)\ ] / 4$$

in which it is assumed that the right hand side is evaluated and used to overwrite the array element u(i,j) corresponding to the **interior point** (i,j). If one of the

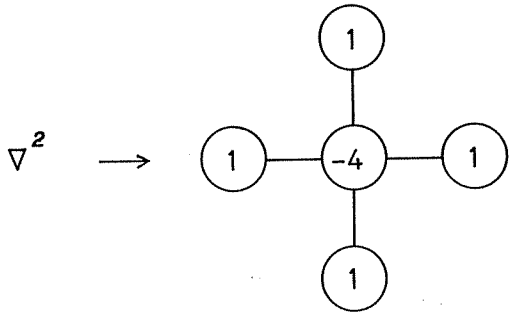

Fig. 5: 5-point star stencil approximation to ∇^2.

points in this expression resides on the boundary of the region then its **constant** boundary value is used. This is a particularly simple example of a local update, or transition rule. Assuming this algorithm (Gauss-Seidel) converges fast enough to be practical, the parallel implementation is exactly the same as for the CAs.

Sometimes, because of poor convergence of relaxation methods, more sophisticated algorithms are required. These often make use of **global information**, such as the sum of the squares of variables like u(i,j) at the grid points (e.g. the Conjugate Gradients algorithm for solving large sparse systems of linear equations, such as result from discretising the Poisson equation). Local sums may be computed by each processor, but then these values must circulate around the array in order to accumulate and distribute the global sum.

For an $L_1 x L_2 x L_3$ array connected as a 3-dimensional torus (i.e. periodic in each direction), an efficient way to do this is as follows. First, everyone sends their value in the 1-direction, adding values they receive to their partial sums and passing the values they receive on around the chain. After L_1-1 transfers, everyone in the same chain has the sum of all the local values in that chain. Next, the same thing happens in the 2-direction, with everyone circulating their local 1-direction partial sums, until everyone has the partial sum for the (1,2) plane in which they lie. This requires a further L_2-1 transfers. Finally, these partial sums are circulated in the 3-direction, to accumulate the global sum on every node. In total $L_1+L_2+L_3-3$ successive transfers are required (although each of these corresponds to $L_1L_2L_3$ transfers in parallel, which are completely overlapped because they are of identical size).

It is worth mentioning here that a distinction is sometimes made between **fine-grained** and **coarse-grained** parallelism. The former is appropriate for homogeneous arrays with, say, 1000 (simple) processors such as the Computing Surface, the latter for a few 10s of (perhaps very sophisticated) nodes such as the T-Series. Geometric decomposition works for both, and in this chapter it has been implied that the larger the subregion placed on a node, the greater the ratio of bulk computation to surface communication, and hence the higher the efficiency. However, the efficiency depends critically on the **balance between the computational power on each node and the communications bandwidth between nodes**. If a coarse-grained array and a fine-grained array are to simulate the same system, in the same real time, the coarse-grained array will have to transfer proportionately more data between nodes than the fine-grained system.

If both rely on a single transputer link between neighbouring nodes, the coarse-grained array will be more severely communications bound. This is in fact the major drawback with the combination of hypercube architecture and powerful vector nodes, being marketed by FPS and Intel; in order to realise the high performance of these machines much faster internode communications is required.

[4] 4. Algorithmic Parallelism

[4] 4.1 The General Situation

This approach is to construct a **network of transputers**, each with its own special role to play, **through which all the data flows**, as in a factory production line. Typically, all the data is stored in the memory space of one transputer, which then naturally acts as controller for the rest of the array. It feeds data through the network of slave transputers, which need have only limited storage capacity and, in particular, may have no off-chip memory at all (hence lowering the cost of the system). Having constructed such a system, it may be replicated geometrically, as in the previous section, but now each node contains more than one transputer i.e. is a **supernode**.

There are a number of difficulties with algorithmic parallelism. One is that at different stages during the computation, different algorithms may apply, and a configuration of transputers optimised for implementing one algorithm is unlikely to be appropriate for another. For example, one algorithm may be used to generate a set of data and a different one used to analyse it. This problem is, of course, much reduced if the transputers are connected by a **dynamically reconfigurable switch** (as they will be in the ESPRIT, ITEM and Meiko arrays, for example). Otherwise, the data analysis must be done on a different group of transputers from those generating the data. Since analysis cannot typically proceed in parallel with data generation, either a duplicate of the data must be stored or data generation must be suspended during analysis. The latter is undesirable as it reduces efficiency.

Another difficulty to be solved is **how to get control data to each of the slave processors**, for example in order to initialise them at the start of the computation. Since each slave has a different job to do, it will expect to receive 'personalised' instructions. One way of accomplishing this is via a **package-routing** network which uses the same link configuration as during operation. A **packet** has a header indicating its destination and a data length. The data content of most of the packets originating from the master controller consists of an instruction code followed by some parameters. Similar packets may be used by the slaves to convey their status back to the master.

Finally, it may happen that one process dominates the execution time. If this process cannot be divided up amongst more than one processor, it alone will determine the throughput and constitute a bottleneck.

[4] 4.2 Long-range Interactions

The applications considered up to now have involved only short-range interactions. This is typical of the finite difference method for solving partial differential equations, where updating the approximation to the field variable at each grid point requires only information about the field values at neighbouring grid points. Then a straightforward geometric decomposition in which each processor handles a sub-region of space (and time) provides an efficient

implementation on a regular processor array. The algorithm for each processor proceeds in much the same way as for a serial computer until boundary information is required, at which point the processor must communicate with a neighbouring processor.

Problems with **long-range interactions** may also be implemented efficiently in parallel (Fox et al. 1984). As an example, consider the time evolution of the solar system. This is a 10-body gravitational problem in which the force between any two particles is given by Newton's law,

$$F_{ij} = G\, m_i m_j / r_{ij}^2$$

and we wish to evolve the 10 equations of motion for some period of time T. There are several new difficulties here. First, the large parameter is T and this cannot be divided up amongst a large number of processors. The best that can be done is to put one particle on each processor, unless the results from several different sets of initial conditions are required, in which case decomposition into particles and initial conditions permits efficient use of a larger processor array. The second difficulty is due to the fact that every particle interacts with every other.

In the direct method for evolving this system forward in time, the total force on each particle, due to all the other particles, is computed, then each particle is moved forward one time step and the process repeated. The computation is dominated by the calculation of the forces, because this is proportional to the square of the number of particles, whereas the timestepping grows only linearly with the number of particles. Since every particle interacts with every other, a geometric decomposition of the problem, in which the particles in different sub-regions of space are associated with different processors, is no use. Instead, each processor is given the job of following the time evolution of one subset of the particles, which may be at widely scattered locations. In order to achieve load balancing, each processor is given the same number of particles to look after. This is illustrated in fig. 6. The algorithm may be mapped onto any network of processors which incorporates a closed ring. The first step then is for each processor to pick one of its particles and send its mass and coordinates to the next processor around the ring. Next, each processor computes the force on its particles due to the incoming 'travelling' particle, and then sends the information about the travelling particle on to the next processor around the ring.

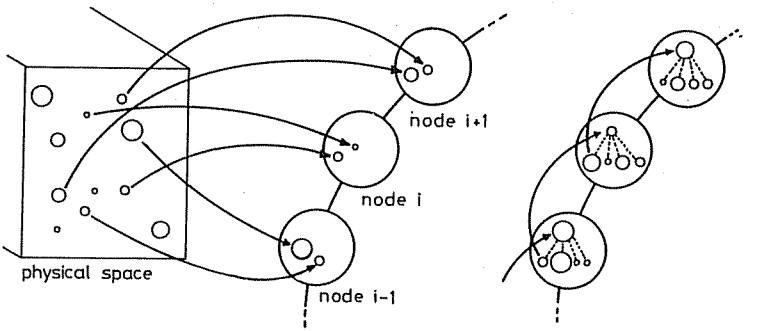

Fig. 6: Algorithm for system with long-range forces.

5. Survey of Parallel Architectures

[5] 1. Introduction

For many years the development of computers and the change in their architecture has been of relatively little importance to the user. This is understandable if the software produced by the user is readily transferred between different computers, especially between the user's current machine and a new machine. Software compatibility over this extended period has led to the development of large packages, which are the result of considerable investment. The challenge that faces us today is that posed by the VLSI technology revolution. The outstanding progress made in miniaturisation and micro-fabrication is initiating radical changes in computer design, and to capitalise on the increase in cost-effectiveness that this delivers, we are forced to reconsider our whole software strategy.

It has been recognised for a long time that computation speed can be vastly improved by using parallelism, indeed this was recognised by Babbage well before the electronic computer was conceived. The earliest of these computers worked bit-serially, using a single central processor. Thus if two numbers were to be added together, the result was formed bit by bit. This mode of operation soon gave way to a form of parallelism, in which parts of the representative words were treated together, though the serial time sequence of computational instructions was still operative. The user observed an increase in power from this innovation, compounded with the increase due to the advance of technology; users were not generally aware of the reasons for the improved performance, but were satisfied as the old code ran quicker.

To sustain the advance in cost-effectiveness of computing, other aspects of parallelism were incorporated, leading to the development of the pipelined computers. In the execution of any arithmetic process there are a number of distinct stages, and these various stages can best be done using different specialised hardware. Therefore after the first stage of the arithmetic process has been completed the hardware concerned can be made available to start another independent process on new data. The various stages of the overall arithmetic processes thus overlap, better use is made of the hardware and a greater speed is attained on the problem being done. This progress has been achieved in the vector supercomputers without the need for new user software, but users have had to make certain minor changes, and the art of **vectorising** code is now widely practised. To be able to vectorise code successfully one has to know what tasks can be done independently, and therefore in parallel, but at this level there is not very much difficulty and all difficulties can be left unsolved by leaving the old code alone.

We have now reached the stage where any further real advance will entail rethinking our software. Parallelism must be exploited on a much more extensive scale in order to make use of the benefit of massive cheap replication of complex

circuit chips. The density of components that can be integrated on to a chip is now so great that a single chip can perform all the electronic functions of a computer. The development costs of a successful chip are very large as compared with the cost of bulk production, and therefore any computer whose design consists of a massive number of identical chips will clearly be the cheapest to produce and to maintain, and if the design facilitates the building of machines of varying sizes, a most viable product will result. Computers are now being constructed with thousands of similar chips, interconnected in some way, and it is a considerable intellectual challenge to find the best way of making the interconnections and of using the resulting computer. Architecture now impinges on the avid user as it is the key to success with VLSI technology.

For the last seven years we have been aware of this impending technology revolution, and in 1979 the three avenues which seemed to us the most promising for large-scale computation were the Motorola 68000, the FPS 164 and the ICL DAP. The 68000 was arguably in essence the first 'computer on a chip' produced in bulk. At present the front-runner in this category is the transputer, its position of eminence depending in good measure on its communications ability. Multiply-connected FPS machines are the basis of IBM's &CAP, the logical progression being to FPS's T-series. Other companies have decided on concurrent use of much more powerful processors, such as the CRAY X-MP and successors, the HEP (company now out of business), and ETA's GF-10. Hypercube connectivity (see later) now enjoys much attention but the requisite software is in a state of flux. The architecture of the DAP has its restrictions, and an attempt to overcome them has led to the construction of the Connection Machine.

For a relative assessment of the available computers a number of aspects need to be considered:

(a) overall cost of the computer,
(b) maintenance cost,
(c) running cost, including personnel,
(d) availablity of system software,
(e) ease of programming,
(f) possibility of importing code.

We have recently engaged in benchmarking some front-runners as it is very difficult to get a reliable estimate of performance from manufacturer's specifications. Many manufacturers measure performance in MIPS (million instructions per second), a debatable measure of ability even for logic; Mflops (million floating point operations per second) is a standard measure for scientific computations, but may be misleading as an unattainable peak performance is usually quoted. Nevertheless we have learnt that a peak performance is **that which in no circumstances can be exceeded**.

[5] 2. Basic Architectures

[5] 2.1 SIMD parallelism

A very simple way to construct a massively parallel computer is to connect processing elements (PEs) in a two-dimensional array, where each PE has four neighbouring PEs. The ICL DAP (see later) is the best example of this construction available today. The mode of operation is SIMD (single instruction, multiple

data-stream), in which the instructions of the program are broadcast to all the PEs simultaneously, whereupon each PE executes the instruction using the data set which is stored locally. Communications between the PEs are east/west or north/south on the array, and in the DAP the array can be taken to have either fixed or cyclic boundary geometry. On a DAP with 64x64 PEs, the longest communications path is 32 steps east/west followed by 32 steps north/south, involving 61 PEs which have no interest in the data received from one neighbour which it has to pass on to another neighbour.

The machine architecture of the DAP is often described as having its PEs on a torus. This can be understood from fig. 1, drawn for a 16x16 DAP. The lines in this diagram can be thought of representing the connections between PEs, or alternatively the areas on the torus can be thought of as the PEs. It is clear from this figure that the longest path between processors does become rather large.

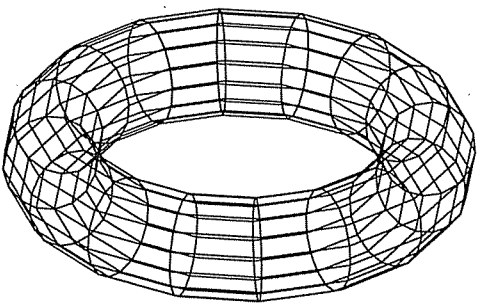

Fig.1: 16x16 processor torus

Some SIMD machines have eight neighbour connections, and others have a hypercube geometry. A fuller description is given later.

[5] 2.2 Hypercube geometry

In an effort to reduce the number of processors which have to pass on information in which they have no direct interest, computer architects have turned their attention back to the binary hypercube construction as an interconnection scheme for multi-processor machines. For a machine of $N=2^n$ processors, each processor is connected to n neighbours. Thus processor i has communication paths to all processors j such that the binary representations of i and j differ by 1 bit. The maximum data path length is n, a considerable saving over the two-dimensional mesh geometry. A diagrammatic representation of any hypercube beyond four dimensions becomes rather confusing, so a 4-dimensional example is chosen for fig. 2. This is not in one of the forms often presented, but has been prepared by the same plotting program which produced fig. 1. This is possible because the 4-dimensional hypercube is topologically equivalent to the 4x4 DAP, a coincidence that does not occur for higher dimensions.

A modified form of hypercube geometry is used in the SIMD Connection Machine (see later), but most of the machines with a true hypercube geometry are MIMD.

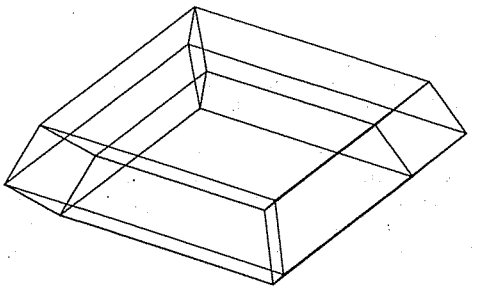

Fig.2: 4-D hypercube or 4x4 torus

One aspect which is very important to consider in a large computer is **redundancy**. If one small part of the computer fails, does this mean that the whole computer should be inoperable? For the DAP or hypercube architecture this would seem to be the case (but see AMT DAP later), but it is possible to build in some redundancy to give a more robust machine. This is possible by including an extra column of processors (actually 4 columns in the case of the MPP, see later) into the array, so that when one PE fails, say the black one in fig. 3, the whole column can be short-circuited in software.

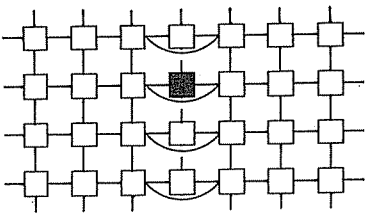

Fig.3: Short-circuit of a column of PEs

[5] 2.3 MIMD concurrency

The Multiple Instruction, Multiple Data-stream machine is the most general possible. It has become a feasible proposition over the last few years due to the cheapness of microprocessor systems. Numerous microprocessors can be linked together in a loosely-bound network in which they all have their own independent memory, or they can be combined as a tightly-bound multiprocessor in which each processor can access **any** memory. The program which runs on a MIMD machine is obeyed by all the processors, but each processor will be at a different place in the program at any one moment. In order to contrast with the parallelism of SIMD, the term **concurrency** is used in this context.

The range of MIMD computers can be roughly divided into those which are constructed out of (a) large processors each of which could be the basis of a powerful computer, (b) processors which occupy a board and which are equivalent to a full mini-computer, and (c) single-chip processors which can perform the full range of requisite functions. Our main interest is in category (c) where the single chip is the transputer, but the present chapter aims to touch upon the full range. The possibility of a category (d) in which there are many processors on a single chip is only viable at present with SIMD processing elements.

The architecture of computers in category (a) is often complicated in detail but simple in essence. The number of large, powerful units in any machine is never enormous, and so it is possible to have connectivity between any two units which need communications. For category (b) this is not the case, and a design decision must be made as to where to place the memory in relation to the processors. One common solution is typified by the BBN Butterfly (see later) in which there are memory units shared by the processors through a switch. This switch cannot be a crossbar as a crossbar gives direct communication between each processor and each memory unit. The switch shown in fig. 4 has two rows of 4x4 crossbars and connects 16 processors one one side to 16 memory units on the other. This can be extended to a switch between 64 processors and 64 memory units by using a third row of 4x4 crossbars.

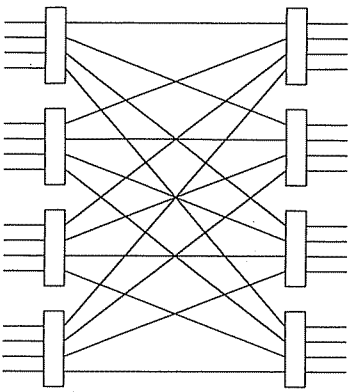

Fig.4: Butterfly switch

Although switch technology is improving to the point where the number of processors and memory units so connected can be in thousands, the fundamental architecture is not optimum for category (c). As minitaurisation continues to give a higher device density it becomes more practical to locate the memory much more closely to the processors. Of course this is an option for category (b) and is implemented in hypercube machines. The optimum machine architecture depends very often on the problem in hand, and it is for this reason that it is very attractive to have a machine where the topology can be designed by the user, as in the case of the Computing Surface. This freedom may well be the key to the successful exploitation of massively parallel computers.

[5] 2.4 Local or Global Memory?

The question as to whether the memory in a highly parallel computer should be local to the processors or globally available is fundamental for the modern computer designer. The BBN Butterfly design is one where the processors access global memory through the switch, but each processor does have a cache of local memory otherwise it would be using the switch repetitively for basic data.

The ICL DAP has its memory local with each processing element, but there is an efficient way for each processor to access global memory when needed through the broadcasting mechanism.

Thus there is no one clear answer; each computer is a compromise in this respect, although it is usually quite clear as to which category the particular computer is in.

[5] 3. Some Specific Computers

The following gives some information about a selection of machines in various categories. The list is by no means exhaustive.

[5] 3.1 SISD - pipeline processors

These are the computers most favoured for central facilities, such as the CRAY-XMP. In these computers, mathematical operations are pipelined and performed on data which flows through the processor in a stream. In the computers here described the data stream is a single vector stream so the operation mode is SISD. A great advantage of these machines is that old serial code can be run without modification, users then being able to improve performance by 'vectorising' the costly sections of code. This is what makes them ideal for a large community, especially as they are more cost-effective when vectorised than the ubiquitous VAX which we do not consider here. The running costs of these computers is high as the power consumption is a large fraction of a Mwatt, much of this being used in cooling. The complexity of such a facility is demanding in maintenance staff, and an extensive operation staff is needed as for any general purpose computer.

There are computers which compare very favourably with those presented in the following three short sections, and have the advantages that go with smaller size. The most notable is the FPS 264. Although omitted here, it should not be passed over when searching for the best general purpose computer of this architecture.

[5] 3.1.1 CRAY X-MP, CRAY-2, CRAY-3

The X-MP (eXperimental Multi-Processor), introduced in 1982, is available with 1, 2 or 4 processors. Each processor has a 9.5ns clock and has a peak performance of 400 Mflops. The 4 processor version has 64 Mbytes of memory. Also available is a solid-state storage device (SSD) with memory sizes up to 8 Gbytes, and a new disk drive (DD-49) which has 1.2 Gbytes capacity and 10 Mbyte/s bandwidth. Transfer to and from the SSD is via one or two 1 Gbyte/s channels. The prototype X-MP/48 can achieve a 3.8 speed-up and so reach the Gflops range. The cost of a 4 processor X-MP is around $20M which, at 50% peak, should deliver 0.8 Gflops.

The multi-processor CRAY-2 has a new vector register architecture and uses liquid-immersion to obtain a 4ns clock. It is about as powerful as the X-MP but has a much larger memory. The CRAY-3 is a GaAs version of the CRAY-2, being developed for the 1990s. It will be one cubic foot in size, have 16 processors with 2ns clock and memory twice as fast as that in the CRAY-2, giving a speed-up over this computer by a factor of 8.

[5] 3.1.2 CYBER-205, ETA GF-10

The CDC CYBER-205 evolved from the STAR-100. It is available with 2 or 4 general purpose pipelines, a 20ns clock and up to 128 Mbytes memory with 80ns access time. It is highly competitive with the CRAY-1, being faster for large vectors but slower for small vectors. A re-engineered 2-pipe CYBER-205 is the processor for the GF-10, which is to be constructed with 8 such processors by the CDC-funded Engineering Technology Associates company. Each processor should run about 3 times faster than the CYBER-205, will be made from VLSI CMOS chips, and have a 5ns clock and 256 Mbytes local memory. The whole computer, which should occupy a 5 foot cube cooled by liquid nitrogen, will have a further 2 Gbytes of shared memory. The company aim is availability of GF-10 in 1987 with a peak performance of 10 Gflops.

[5] 3.1.3 Facom VP, Hitac S-810, NEC SX and MITI's plans

The three largest Japanese computer companies have been developing computers comparable with the CRAY and CYBER series. A detailed comparison is out of place here. Fujitsu's Facom VP-100 and VP-200 peak at 250 and 500 Mflops, Hitachi's Hitac S-810/10 and /20 are marginally faster at 315 and 630 Mflops, the most powerful being NEC's SX-1 and SX-2 at 570 and 1300 Mflops. These three companies, in collaboration with Oki, Toshiba and Mitsubishi, are currently involved in MITI's (Japanese Ministry of International Trade and Industry) National Super-Speed Computer Project. This is a five year programme of R&D aimed at producing a 10 Gflops computer system with 1 Gbyte of semiconductor memory and 100 Gbytes of disk storage by March 1990.

[5] 3.2 SIMD - processor arrays

These computers are constructed as described above arrays of identical processing elements (PEs) operating in lockstep performing the same operation on different data.

[5] 3.2.1 Distributed Array Processor, DAP

The DAP, made by International Computers Limited, was begun in 1972 and the first production machine was installed in 1980 at Queen Mary College, London. It comprises a 64x64 array of PEs, each with 4 Kbits of associated memory (16 Kbits now at QMC), and is a memory module of a 2900 ICL mainframe which acts as a host. The PEs are two-dimensionally connected to the four nearest neighbours on a torus and access highways to them, along the two dimensions, contribute greatly to the DAP's performance. Each PE can only perform bit-serial arithmetic, so arithmetic operations must be done in software offering a word-length flexibility unavailable in conventional computers. The first generation DAP is SSI/MSI with

[5] 3.2.4 Massively Parallel Processor, MPP

Made in VLSI by Goodyear Aerospace Corporation, this SIMD computer has 132x128 bit-serial PEs (132=128+4, the 4 giving built-in redundancy) each with 1 Kbit memory. The high level language chosen for this machine is parallel Pascal. The power is projected to be 200 Mflops, but we have not been able to test this. The MPP was designed for NASA, and a machine is currently in operation.

[5] 3.2.5 Cellular Logic Image Processor, CLIP

The pilot model (CLIP3) was built in University College, London, in 1973 and then the CLIP4 system in 1979. This is a 96x96 array of PEs each with 32bits of memory and connections to the eight nearest neighbours. The four-phase clock cycle is 400ns. It is used for image processing and pattern recognition. It has been redesigned with custom chips as CLIP7, a 4x512 array of PEs, each with 32 Kbits of memory, running with a 200ns clock. The programming language is an extension of C. This is now a company product of Stonefield Ltd. of Swindon, UK.

[5] 3.2.6 Adaptive Array Processor

The Japanese Nippon Telegraph and Telephone Public Corporation (NTT) is building an LSI Adaptive Array Processor. The prototype will use 128x128 2μ Si-gate p-well CMOS LSI chips, each chip containing an 8x8 PE array with peripheral circuits and each PE having a 96bit memory. There will be an eight-nearest-neighbour PE connection network with hierarchical bypasses, allowing various types of two-dimensional data processing. The clock cycle will be 55ns, and power consumption 1.1W per chip.

[5] 3.2.7 The GEC GRID

The General Electric Company (UK) is building the GEC Rectangular Image and Data processor. It is to be a 64x64 PE array with each PE having 64bits of on-chip (register) memory and 64 Kbits of off-chip (main) memory, and being connected to its eight nearest neighbours. This arrangement probably gives the best of both worlds: a simple PE with a small fast 'cache' memory (as in CLIP and AAP) backed up by a large amount of slower memory (as in DAP, CM and MPP). The 64x64 array is to be made up of 16 boards each containing 2x4 GRID chips and their off-chip memory, each chip containing an 8x4 array of PEs and their associated on-chip memory, an edge control register, a histogram counter and the peripheral circuitry for instruction decoding and zero detection. The on-chip edge control register defines the PEs which form the edge of the array, thereby allowing the array to be connected in a variety of topologies: linear, cylindrical or toroidal.

The GRID system will be hosted by a minicomputer, running the UNIX operating system. Thus the GRID will be programmed in C, extended (initially by adding a preprocessor) to support parallel instructions. These are implemented, as for the DAP, by means of subroutine calls. Responsibility for the project has now been transferred to Marconi.

[5] 3.3 MIMD - multi-processors

[5] 3.3.1 BBN's Butterfly and Monarch

Bolt, Beranek and Newman (BBN) have developed the Butterfly Parallel Processor which is an MIMD tightly-coupled, shared-memory machine consisting of up to 256 68000 microprocessors, each with 1 to 4 Mbytes of memory, interconnected via a Butterfly switch. The single-board processors can be upgraded to 68020/68881 which yield 0.5 Mflops each. The Butterfly switch, which uses packet switching (see fig. 4), is made up from 4x4 switching elements implemented as custom VLSI chips with 8 on a board forming a 16x16 switch. Machines with more than 16 processors have redundant paths. Interprocessor data transfers occur at 32 Mbyte/s and remote access is claimed to take less than 4 microseconds.

The Monarch Multiprocessor, BBN's future MIMD shared-memory machine is designed to have 8192 processors, 1024 memories (4 to 512 Mbytes) and a 16384 port butterfly switch with bandwidth of 2 Gbyte/s. The target performance is 3 Gflops, to be achieved by the end of 1987. Many of the developments taking place with the Monarch are currently being used by BBN to upgrade the Butterfly.

[5] 3.3.2 The Heterogeneous Element Processor

Denelcor of Denver, Colorado manufactured the MIMD Heterogeneous Element Processor which consists of up to 16 process execution modules (PEMs), each with its own data memory bank, connected to a switch. Each PEM can access its own data memory bank directly, but access to the other memories is through the switch. A PEM may run up to 64 processes concurrently. The switch is a high-speed, packet-switched network. Process synchronisation is achieved in hardware by the use of so-called asynchronous variables; they cannot be read from until they are full or written to until they are empty. Although Denelcor are now out of business, HEPs are still available for purchase.

[5] 3.3.3 The IBM &CAP

The Loosely Coupled Array of Processors has been built by Professor Enrico Clementi at IBM Kingston. This distributed system consists of 10 FPS-164 attached array processors hosted by an IBM 3081 with an expected peak performance of 110 Mflops, to be increased to 550 Mflops by the addition of 2 FPS MAX boards to each processor. IBM have expressed a willingness to market this product, &CAP-1, and have installed a similar system at IBM Rome. &CAP-2 now exists at Kingston and has 10 FPS-264s hosted by an IBM 3084, peak performance being 330 Mflops.

[5] 3.3.4 The Ultracomputer and the RP3

New York University's Ultracomputer is a MIMD shared-memory machine in which the processors and memories are separate, but are interconnected through an Omega switching network. This switch is such that each processor is at the top of a binary tree whose leaves are the memories and vice versa. It is pipelined and packet-switched.

An 8 processor prototype was built and a 4096 68000 processor version was planned to yield a 10 Gflops machine, but the Ultracomputer project has now become integrated with the IBM RP3 (Research Parallel Processing Project) and the first machine is to have 64 IBM 32bit microprocessors, 2 Gbytes of memory and a VLSI omega switch giving a possible 100 Mflops in 1987. This is to be followed by a 512 node version. IBM now state that this is not a commercial product though it clearly has potential as a rival to the Butterfly.

[5] 3.3.5 Myrias 4000

The Myrias Research Corporation's Myrias 4000 distributed memory MIMD system will offer configurations of up to 64K processors (measured as 64 Krates where 1 Krate = 1024 processors), giving an average performance of around 1.6 Gflops (32bit arithmetic). The minimum primary memory is 8 Gbytes and can be upgraded to 32 Gbytes. The processors each consist of a 68000, 128 Kbytes 150ns dRAM and a high speed DMA interface. 8 processors together form a board, 16 boards populate a cage, 8 cages are housed in 1 Krate, 4 Krates form the minimum configuration - a 64x64 array with 512 Mbytes of memory. They also have 1 Gbyte disk storage and I/O bandwidth of 80 Mbyte/s per Krate. System is expandable in 1 Krate increments. An optical communications system is a feature of this machine, but it is not clear whether this will overcome the communications problem better than standard electronic links. Processing power will increase (by 5) when the 68881 is added and memory will increase (by 4) when 256K dRAM becomes available.

[5] 3.3.6 Caltech hypercube

Geoffrey Fox and Charles Seitz of the California Institute of Technology designed the Caltech Hypercube comprising an array of microprocessors with local independent memories. Nearest-neighbour connections are implemented by means of mailbox communications which operate as follows: a processor waits on a mailbox until the data is put in by the neighbour, and a processor waits on a mailbox until it becomes empty if it wishes to deliver data to it. The processors are configured as a binary hypercube.

The first machine (after a four node prototype), which was completed in October 1983 and called the Mark I Hypercube, has 64 nodes, each of which is an Intel 8086/87 microprocessor with 128 Kbytes of memory on a single board, giving a total of 3.2 Mflops. The Mark II Hypercube (completed in September 1984) is as Mark I but with 256 Kbytes of memory. It exists as one 128-node or four 32-node machines, and has an Intel 310 workstation as intermediate host. The present Mark III Hypercube is to be firstly 32 then 256 and finally up to 1024 nodes, each of which contains a Motorola 68020/68881 as main processor, another 68020 as I/O processor plus Weitek scalar floating point chips (1064 multiplier and 1065 ALU - 12 Mflops for 32bit operation) and 4 Mbytes of memory. DMA will be used as the communication method for message passing.

[5] 3.3.7 The FPS T-series

The T-series are homogeneous parallel computers with a modular structure based on hypercube geometry, so that a wide range of system sizes is potentially possible. The basic unit is the module, or T-10, consisting of eight node boards

connected together as a 3-dimensional cube, a system board and a disk. The system board is connected to each of the node boards via a linear chain, to the disk, and to a front end computer which is a MicroVAX running VMS. This system network is distinct from the hypercube interconnections between nodes. Additional communication links on the system and node boards allow modules to be connected together; the system boards are connected in a ring, the node boards are connected into an n-dimensional cube (hypercube) where $n < 14$. The most common configuration at present is the T-20, which forms a single cabinet, consisting of two modules in which the 16 nodes are connected as a 4-dimensional cube; the T-100 consists of four cabinets in which the 64 nodes are connected as a 6-dimensional cube (T-numbers represent the number of nodes in octal).

Each node board is a powerful vector computer in its own right, with a peak 64-bit performance of 16 Mflops. The original node design involved a Weitek coprocessor chip set fed from 1Mbyte of video RAM controlled by a transputer, the transputer being also responsible for communications. Potential (peak) performance ranged from 256Mflops to 60 Gflops or more, depending on the number of nodes. It seems likely that this design will evolve to a more balanced unit with a wider communications band-width and more memory.

[5] 3.3.8 Intel's personal supercomputers, iPSC & iPSC-VX

The iPSC/d5 (d6 or d7) system contains 32, (64 or 128) nodes hosted by an Intel 310 workstation. Each node consists of a 80286/80287 microprocessor with 512 Kbytes of memory and 7 bidirectional channels (each giving 10 Mbit/s data transfer). The performance/price ratio of the d7 is 16 Mflops/$M.

The iPSC-VX is Intel's machine in competition with the FPS T-series. It is a hypercube built from their 80286 and 80287 processors but with vector processors at each node. The ALU is pipelined with a 100ns clock and 1 Mbyte of 250ns dynamic RAM plus 16 Kbytes of 100ns static RAM. For 32bit arithmetic each node is theoretically capable of 20 Mflops. These computers should be being shipped in 1987.

[5] 3.3.9 Ametek System 14

A hypercube is formed of from 16 to 256 nodes and is connected to any VAX (running Unix or VMS) as a host via a 1 Mbyte/s parallel interface. Each node consists of a 80286/80287 microprocessor and 1 Mbyte of memory with 8 bidirectional channels (each 3 Mbyte/s). It runs the Ametek Hypernet Operating System (HOS), as will the Caltech Mark III Hypercube. First machines were projected for early 1986 costing about $100K for 32 nodes.

[5] 3.3.10 NCUBE

One to four boards, each containing 4 processors, form the 8 Mflops NCUBE/four which has an IBM PC-AT bus interface; 1 to 16 boards, each containing 64 processors, form the 500 Mflops NCUBE/ten. Each processor consists of an NCUBE custom VLSI chip which contains a 32bit processor, a 16bit error correcting memory interface (to 128 Kbytes of memory) and 22 independent DMA communication links (11 in and 11 out) with a claimed data transfer rate of

1 Mbyte/s. One pair of links is used for system I/O. There are eight 90 Mbyte/s system I/O channels - one for each set of 128 nodes connected via I/O boards. One of the I/O boards is a host board containing an 80286/80287 with 4 Mbytes of memory which runs a Unix style operating system called Axis, supporting 8 users and facilitating allocation of subsets of the hypercube. (Each node runs a simple operating system called Vertex.) The other I/O boards may do other things such as graphics.

[5] 3.3.11 Sequent BALANCE and Alliant FX-series

A number of multi-processor systems have recently appeared which aim to provide improved performance through concurrency, in a way which is essentially transparent to the user. This is typically achieved by tightly-coupled processors sharing common memory. Examples are the Sequent BALANCE and the Alliant FX-series. As yet such machines have a modest number of processors, roughly up to 20, and as yet do not achieve very high performance, but future upgrades may meet this challenge.

[5] 3.4 Data-flow and systolic architectures

Computers of these architectures do not appear to be competitive for large-scale scientific computing. Nevertheless this field of development must be monitored especially as Japan's Electrotechnical Laboratory (ETL) is developing a large dataflow computer called SIGMA-1 which contains 256 PEs and should achieve 100 Mflops.

[5] 3.5 Transputer-based systems

The INMOS transputer is a 250,000-component (1.5µ CMOS) VLSI chip which contains a 10 MIPS RISC processor, 50ns static-RAM memory and four 20 Mbit/s bidirectional communication links. It can address up to 4 Gbytes off-chip memory with a bandwidth of 25 Mbyte/s.

On October 1st 1985, the INMOS T414 transputer went on sale for about $ 500. It has a 32bit processor and 2 Kbytes memory. There have been considerable developments since this time, and fuller details of the extended range of transputers is given in chapter 6.

The transputer is programmed in occam which is the native language of this device, a language specifically designed to facilitate communications and implement concurrency.

[5] 3.5.1 INMOS Transputer Evaluation Module

INMOS are constructing a range of boards, one of which has 4 Transputers hard-wired in a square. When coupled to an IBM PC the result is an evaluation tool which is very useful for familiarisation and occam program development and constitutes a powerful computing device.

[5] 3.5.2 The Esprit Project

The development of the T800 floating point transputer has been partially funded through an Esprit project which involves RSRE, Thorn-EMI and Southampton University in the UK in addition to Inmos and European collaborators. This is to form the basis of a machine comprising 256 or more transputers, where the architecture, we understand, is based on a multi-transputer supernode.

[5] 3.5.3 ALICE

The Applicative Language Idealised Computing Engine is a highly parallel computer being designed and built at Imperial College out of 16 or 64 transputers. It is primarily for the parallel evaluation of declarative languages by graph reduction - hence it is termed a reduction machine. Declarative languages are languages with no assignment, so no side-effects, and no state (program history). They come in two varieties: functional (or applicative) and relational (or logic). Functional language programs comprise a set of data declarations, a set of functions to be performed on the data and a top-level expression whose evaluation produces the result of executing the program. Relational language programs comprise a set of assertions (facts), a set of rules and a query whose satisfaction is the result of the program.

[5] 3.5.4 The Meiko Computing Surface

This computer, the subject of the next chapter, is designed to make use of all the facilities of the transputer. A great advantage of this machine lies in its ability to be electronically reconfigured, so that at one moment it can mimic a hypercube and the next moment it can mimic a DAP or MPP.

6. The Computing Surface

[6] 1. The Transputer

The INMOS T414 transputer is a 1.5μ CMOS VLSI chip which contains a 10 MIPS, 32-bit RISC processor, 2 Kbytes of on-chip 50ns static RAM and four 20 Mbit/s bidirectional communication links. It can address up to 4 Gbytes of off-chip memory with a bandwidth of 25 Mbyte/s. The RISC architecture allows context switching to be very fast, supporting efficient implementation of concurrent program execution on a single transputer. The INMOS links support communications between programs running on separate transputers. The serial protocol requires only two wires per link in each direction, the respective pins of one transputer being directly connected to those of another. The point-to-point communication links allow transputer networks of arbitrary size to be constructed, with the advantage that the communications bandwidth does not saturate as the size of the system increases. Each transputer in a system uses its own local memory. Overall memory bandwidth is proportional to the number of transputers in the system, in contrast to a large global memory, where the use of additional processors tends to degrade memory bandwidth.

The transputer is designed to implement the process model of concurrency, expressed through the occam programming language, which was developed in parallel with the hardware. Occam is a high-level language based on communicating sequential processes which, although themselves sequential, may be run in parallel with other such processes. Communication between parallel processes is effected by uni-directional channels, which may connect processes on the same processor or on different processors. Each INMOS link implements two such channels, one in each direction. We stress that the transputer can be programmed in other high level languages such as Fortran, but that if concurrency is to be exploited, occam should be used as a harness to link modules written in the selected language.

The T212 Transputer is a 16-bit transputer providing up to 10 MIPS processing power with 2 Kbytes of on-chip RAM and four 20 Mbit/s links. The memory interface provides 64 Kbytes of direct address space with a maximum data rate of 20 Mbyte/s.

The M212 Transputer is a disk controller. It contains a 16-bit processor, two standard links, 4 Kbytes of ROM and 1 Kbyte of RAM, an interface to external RAM and two disk interfaces.

The most interesting member of the transputer family from the viewpoint of large-scale scientific simulations is the floating-point version of the transputer, the T800, with an integral hardware 64-bit floating point unit capable of a sustained performance of 1.5 Mflops, and pin-compatible with the T414. The T800 was announced in November 1986 and silicon now exists in engineering sample quantities. It features 4 Kbytes of on-chip RAM for 80 Mbyte/s data rate, four 20 Mbit/s INMOS links, and an external memory interface with bandwidth of 26.6 Mbyte/s. The floating point unit has been designed to operate on both

single-length (32 bit) and double-length (64 bit) floating point numbers to the ANSI-IEEE 754-1985 floating point standard. The T800-20 (20MHz part) is claimed to achieve more than five times the performance of the Motorola MC68020/MC68881 combination on the Whetstone benchmark.

[6] 2. The Computing Surface

The Computing Surface is a computer designed by Meiko Ltd. in order to exploit the power of the transputer on compute-intensive applications. The transputer itself was inspired by the conclusion that the best way to achieve highly concurrent processing is through communicating sequential processes, as proposed by Professor C.A.R. Hoare at Oxford. The goal is to achieve a balance of computation and communication without jeopardising the integrity of the result of the computation. Various topologies were considered for the construction of a large computer, such as rings, grids, hypercubes, cylinders, toroids, but all were faulted for the same reason. All such topologies are compromises, and all restrict the software engineer in some way. Thus it is advantageous to have a system where the optimum topology can be implemented at will, and it is this flexibility which is the goal of the design of the Computing Surface.

Meiko is a start-up company founded by former members of the INMOS team responsible for the design and implementation of the transputer itself. The Meiko Computing Surface is a modular, reconfigurable transputer array. Physically, a Computing Surface is contained in one or more modules or cabinets. Modules may be populated with a mixture of boards (elements) chosen according to the requirements of the user. The M40 module has 40 slots, at least one of which is occupied by a local host board (see fig. 1) carrying a T414 (**T**) transputer providing a network interface (**N I**), 3 Mbytes of error-checked RAM and IEEE488 or SCSI interface, plus dual RS232 interface. The local host performs house-keeping tasks such as monitoring for hardware errors, control of the electronic routing network, hardware reset and hosting the interactive program development environment, which supports file access via the IEEE488 or SCSI interface, compilers, topology configuration software and the Computing Surface loader/bootstrapper.

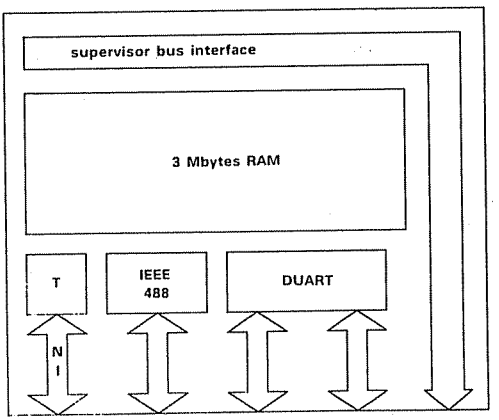

Fig.1: Local host and interface board

Other slots may be occupied by standard boards, shown schematically in figs. 2, 3 and 4. The quad computing element carries 4 transputers, currently T414s, each with 256 Kbytes of error checked RAM, as in fig.2.

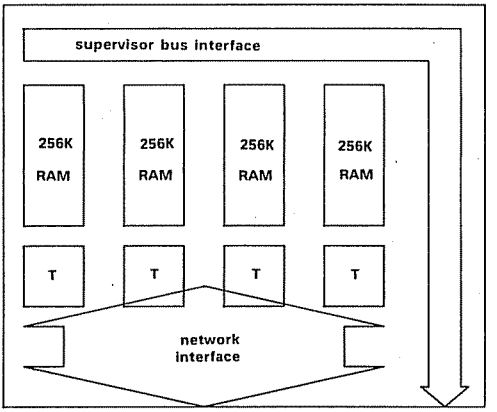

Fig.2: Quad computing element

The graphics display board, fig. 3, contains a single T414 plus 1.5 Mbytes of dual-ported video memory and is specially designed to give high performance.

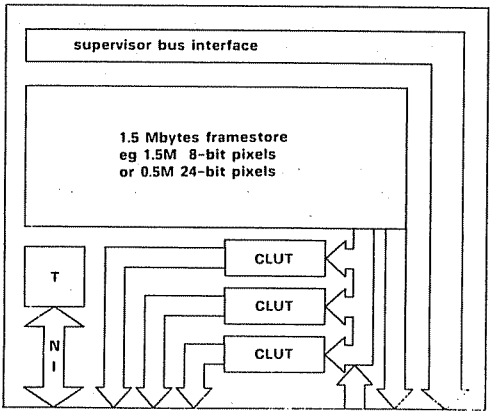

Fig.3: High performance display element

Mass storage is given by the mass store board, fig.4, which also features a single T414 but with 8 Mbytes of error checked RAM and a 2 Mbyte/s DMA controlled SCSI disk and peripheral interface.

[6] 3. The Edinburgh Computing Surface

The Computing Surface in Edinburgh University, which is supported by the Department of Trade and Industry and the national Computer Board, is built around 40 **worker** transputers. It is being used for research into parallel processing and as a demonstrator. It is hosted by a microVAX II running VMS, which provides file storage (650Mbyte hard disk), backup (6250 BPI tape drive), and networking services. The system services and I/O are controlled by the host board. Advanced graphics capability is provided by a display board, giving bit-mapped graphics at a maximum screen resolution of 512 x 1024 at 24 bits per pixel, and is currently configured as 3 banks each of size 576 x 768 at 8 bits per pixel.

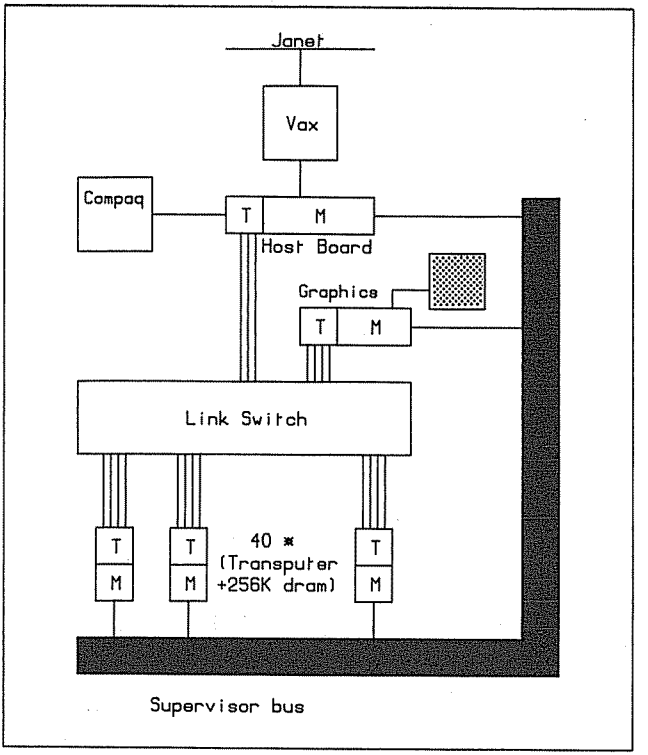

Fig.5: The Computing Surface.

Each of the 40 transputers has 2K of fast on-chip SRAM and 256K of off-chip DRAM (fig. 2); their links are all taken to the backplane for hard configuration. This is to be upgraded by the installation of an electronic switch, which is now in production. The inter-processor links are all rated at 10Mbits/sec.

Each of the worker transputers is connected via the supervisor bus to the host board, giving them low bandwidth external communications capability. The supervisory mechanism is independent of the transputer links; at present it is used for system services, error detection and forwarding debugging messages.

Occam 2 code is compiled on the CS host board, which is file-served by the microVAX. Fortran 77 is currently under test, and will be released in the near future.

The Computing Surface provides three unique facilities,

1) an environment for the development of parallel algorithms,

2) a high performance attached processor for the microVAX – the Computing Surface runs at up to 80 times its speed,

3) advanced graphics facilities in close proximity to real computational power, capable of pumping data at the graphics board at up to 5Mbytes/sec.

[6] 3.1 System Configuration

The immediate question to be addressed is how to build a single array processor from a box of transputers, given the flexibility afforded by the links.

In the absence of the electronic switch the configuration has to be hard-wired. All of the transputer links (4 per chip) are taken to the backplane, where they are connected together with short twisted pairs to create the desired configuration. In the early stages of our work, the machine was connected up first as a hypercube and a ring, and then as a 10x4 array. Standardising on the latter allowed us to develop software while permitting the users a wide variety of array architectures. When the electronic switch is installed the machine configuration will be matched to the PLACEMENT specification.

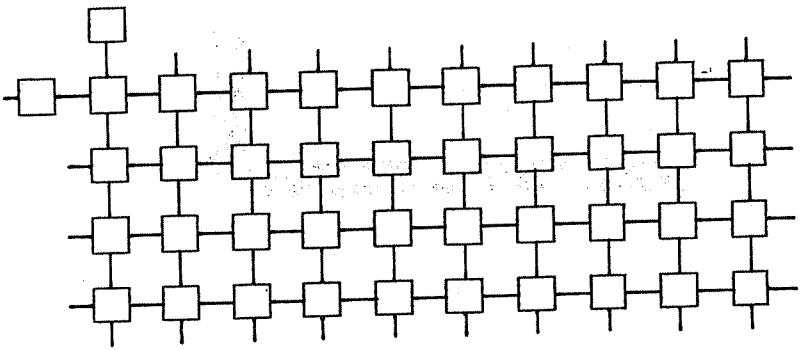

Fig.6: Configuration of the Computing Surface as a 10x4 array

The edges of the array have been wrapped around to form a torus. It is possible to map chain, ring, grid, box-girder and hypercube configurations onto this torus without changing any of the link connections.

[6] 3.2 The Occam Programming System (OPS)

The Occam Programming System is an environment from within which occam programs can be written, compiled and run. It runs on the CS host board, file-served by the microVAX, and also on the PC-based workstations. The editing facilities are also available on the microVAX alone, for up to 8 simultaneous users.

Occam allows the user to develop a heirarchy of processes that reflect the structure of the application, and OPS allows a heirarchical representation of a program via a mechanism known as folding. Folding is analogous to taking a document with headed paragraphs and then folding it so that the text is hidden, leaving only the headings.

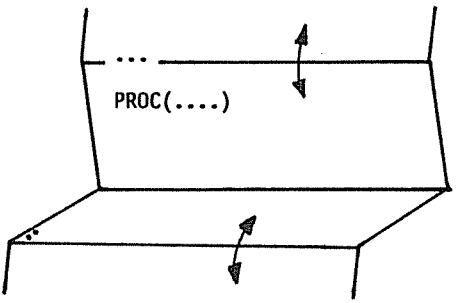

Fig.7: Folding up text.

Folds can be opened to reveal (in context) their contents, or closed, leaving only the header visible. Folds can also be **entered**, in which case only the contents of that fold are displayed, not its surroundings (context). Folds can be nested to any depth.

There are two approaches to the use of folds:

1) **Procedural.** Having written a procedure it can be checked and documented and then folded away, represented only by its header. This approach is suitable for a library of procedures.

2) **Top-Down Program development.** The folding editor encourages Top-Down development of code. A high level program description can be written using occam constructs and folds (which may be empty) representing distinct tasks and their inter-relationships. e.g.

```
{{{ test application

PROC job( CHAN in, out, ....,  VAL INT .... )

     ... system code
     ... libraries
     ... declaration of variables
     SEQ
       ... initialisation
       WHILE running
         SEQ
           ... task 1
           PAR
             ... task 2
             ... task 3
           ... task 4
       ... task 5

:
}}}
```

It should be noted that the editor marks closed folds by ... followed by the fold header. The start of an open fold is marked by {{{ and the header, and the end by }}} on a line of its own.

At this stage the software engineer can decide upon the channels needed for communication between processes running concurrently. Having written this specification, the details of each of the tasks can be filled in in the same way, representing the structure of each task in terms of constructs and subtasks. This process is repeated down to the lowest level of fundamental processes and library procedures.

[6] 3.2.1 Files and Folds

Files are created within OPS by folding up text and filing the fold. This creates a file whose name is the first word in the fold header, with .TSR as the default extension. To create a file with the name FRED.TXT you should type FRED.TXT on the fold line prior to filing.

Existing files can be attached to folds. To do this create an empty fold, type the filename and file the fold. The message "File attached to fold" appears when the process is complete. If the contents of an existing file are to be used as the basis for a new file then the old file should be attached to a fold, unfiled, the name changed on the header, and then refiled.

OPS makes extensive use of folding:

1) When you start up OPS you enter a toplevel fold (called TOPLEVEL.TOP), from which all work is accessed.

2) All help information and utility parameters are held in folds.

3) Most importantly, folds have attributes associated with them. Folds can be marked as SC for those which contain procedures for separate compilation, PROGRAM for those which contain compiled procedure definitions and placement information, EXE for those which contain procedures suitable for running within OPS, or COMMENT for those whose contents are to be ignored by the compilers.

[6] 3.2.2 Using OPS on the microVAX

The **OPS** environment is available on the microVAX, but only for writing and editing code. The compilation utilities all require the version of OPS that runs on transputer hardware which is either the CS host board or the PC+B004 workstation. To start up **OPS** type **VOPS TOPLEVEL.TOP** and this will enter the toplevel fold. This fold contains all the references necessary to use the Computing Surface - system code, libraries and demonstration programs.

[6] 3.3.3 Using the Computing Surface

A program for the Computing Surface consists of a set of procedure definitions, and a description of how these procedures are to be distributed over the transputers available. The code to run on each processor must be separately compiled, and referenced within the PROGRAM fold, together with the placement information This fold must be configured and extracted before the program can be loaded and run.

To do this you must log on to the Computing Surface microVAX, allocate the machine by typing **getmeiko** in the occam default directory, connect up the terminal by typing **reset** and then **term** and load **OPS2** onto the host-board. Do this by selecting the "load occam 2 code" option (press <RETURN> twice to do this) and when prompted for a file to load type **OPS2**. The screen will fill with dollar signs as the code is being loaded. When prompted for a terminal type, enter vt100 and press <RETURN>. The environment will now appear the same as VOPS. To compile and then configure the program follow the instructions in the next section.

[6] 4. Utility Packages

There are two packages of utilities available, the **Program Development Package** and the **Transputer Development System**. Their organisation and use is the same on the microVAX and the PC+B004 systems. This section describes how to use them to run an occam 2 program on the Computing Surface. The **Program Development Package** contains a checker, a compiler, a linker, and utilities for attaching attributes to folds and for searching files. The **Transputer Development System** contains the utilities for configuring programs for the Computing Surface, running them and locating user software errors.

To get a **utilities package** or a **user program** (such as the terminal emulator) point the cursor at it and type **FUNC g**. Only one package and one user program can be held at a time.

To run a user program (normally an **EXE**), get it (as above) and type **FUNC r**. Control will return to **OPS** when (and if) it terminates. You may find it useful to make **keyboard ? any** as the last line in such a **PROC**. This will leave the output from the program on the screen until something is typed; the output will disappear when control returns to **OPS**.

Many of the utilities have a **parameter fold** which specifies the options for the utility. The first time you invoke such a utility its parameter fold will 'pop up' with all the options set to their defaults. You can change options at this point. Subsequently, the parameter fold will not appear unless you explicitly ask for it. To change options at this point **ENTER** the parameter's fold by typing **FUNC s**, select the fold required, **ENTER** it and alter the options. Return to the text by typing **EXIT FOLD** twice. You can alter options in the parameter's fold at any time other than while a utility is running.

[6] 4.1 The Program Development Package

The most important members of this package are the compilers, but you will need to use several of the other utilities first, so they will be described first.

Search and Replace: FUNC 8 and FUNC 9

To search for a string, type **FUNC 8** and the search and replace parameter fold will open. Type in the search and replace strings and type **EXIT FOLD**. The cursor will move to the next occurrence of the search string in the fold currently open, or in any fold within it. To replace this occurrence type **FUNC 9**. There is no global search and replace; to do multiple search and replace you must type **FUNC 8 and then FUNC 9** over and over again. Like the other utilities, the search and replace parameter fold only 'pops up' the first time you use it. To change the parameters subsequently you have to enter the parameter fold (as described above). The search and replace utilities are also available in VOPS.

Make Foldset: FUNC 7

The **Make Foldset** utility, when applied to a source fold creates another fold around it with special attributes. You select (via the usual mechanism) whether you want to give the fold **SC** (separately compilable), **EXE** (**PROC** suitable for running within **OPS**) or **COMMENT** (to be ignored by the compiler) attributes. The utilities will create other filed folds within this outer fold.

Make Comment: FUNC 0

This comments out a fold so that it will be ignored by the compiler.

Check: FUNC 1

This checks a fold for syntax and semantic errors. The checker performs the same syntax checks as the compiler, but does not go on to generate code; it is much faster than the compiler at doing these checks. The checker can be applied to any filed source fold. It reports the first error, positioning the cursor where the error has occurred.

The checker and compiler share a parameter fold. It will appear the first time either is used. To make subsequent changes you must enter the parameters fold. These options include:

Usage Checking. Usage of channels, and the independence of concurrent processing is checked if this option is set **TRUE**. Unfortunately the current release has several serious problems with this option. It is set **FALSE** by default. It can however be used for small procedures under most circumstances.

Range Checking. Standard array bound checking; some programs will go faster if **Range Checks** is set **FALSE**, but don't do it until you know it works.

Extended Data Types. Set this **FALSE** if you aren't going to use the types **REAL32, REAL64, INT16, INT32** or **INT64**, as it will speed things up during compilation, otherwise leave it **TRUE**.

Protocol checking. Not Implemented. Leave it **FALSE**.

Locate Error. If set **TRUE** this will allow the **Locate Error** utility to find the point in the program responsible for a user software error, if one occurs.

Compile All. Compiles the current filed fold and all separately compiled folds referenced within it. Don't use it if you include references to system code, as it will try to compile them, and this will fail as you don't have write access to the directories in which such files live.

Force Pop Up. If set **TRUE** then the compilation options will be presented every time you invoke a compilation utility.

Compiler: FUNC 2 and **FUNC 4.**

To compile a filed source fold first mark it as separately compilable, **SC**, or **EXE** if it is to run within OPS. If this fold contains other uncompiled **SC**s then use the **'bottom up' compiler FUNC 2** as this will search down through the fold structure looking for anything that needs compiling. The **Unit Compiler FUNC 4** does not do this, it simply compiles the fold selected.

Both compilers create several separate new folds containing code, linking, and debugging information, all within the compilation fold. These folds are filed automatically; they are not readable. Use the **Compilation Info** utility **FUNC 6** to create a textual description of their contents. The compiler shares a parameter fold with the checker; see above for details.

If the source of a compiled fold is altered then the compilation fold is marked as invalid, and must be recompiled before it can be used again. Use the **Fold Info** key **FUNC f** to check on the status of a fold.

Linker: FUNC 3

Occam programs need to be linked before they can be run. The linking process involves including within the code, references to library routines. Programs for the Computing Surface are linked automatically during **Configuration**, as are **EXE**s that are compiled with the **'bottom up' compiler**. If an **EXE** is compiled with the **unit compiler** then it will have to be linked using **FUNC 3**.

[6] 5. The Transputer Development System Utilities

This section describes the utilities that enable you to configure the program for the Computing Surface, and then load the code onto it. You should get the terminal emulator at the same time as the **TDS** package.

Make Program: FUNC 7

Having separately compiled the code, and written (or copied) a suitable placement harness, you should bundle them up in a fold, file it and apply the **Make Program** utility. This will create an outer fold, marked **PROGRAM**, into which the TDS utilities will put their files.

Configure: FUNC 2

The configurer checks that a PROGRAM has as a consistent placement harness, that channels have two ends **PLACED** correctly and that the separately compiled procedures are referenced correctly. It also links in libraries for extended data types. It then builds a map of the target network, and checks that it is loadable from the host.

Config Information: FUNC 6

This utility analyses the information produced by the configurer and generates a memory map for each processor, a wiring diagram, and a description of the boot path. The boot path is the route which is used in the loading of the transputers. This information is put into a fold within the **PROGRAM** fold called **CONFIG INFO**.

Extract: FUNC 3

The extractor takes all the code from a fold set and puts it into a single fold within the **PROGRAM** fold together with all the routing and bootstrapping information necessary to load the Computing Surface.

Load Network: FUNC 4

The loader sends code to the Computing Surface. The **VAL link** in the loader parameter fold should be set to 1.

When this point is reached successfully, the message **Network loaded ok** will appear. By typing **FUNC r** the terminal emulator will start to run and the output from the program will be forwarded to **OPS**. Typing **<ctrl> Z** will return control to **OPS**.

There are listings of error messages and information on how to correct the the faults associated with them in the **TDS** User manual.

[6] 6. System Code

The system code for the MEIKO boards has been bundled into separately compiled procedures, and to include them in the code the user should attach the files to a fold, or simpler still, use one of the template procedures. There are 5 such system procedures, and 4 templates. This section describes these procedures, and the protocols for using their channels. There are templates for using the MK009 compute boards, and the MK015 graphics board, and a separately compiled procedure for the host board (this is only needed when running the system in 'stand-alone' mode.

Minimal System code for the Compute Boards.

The template **course:mk009_min.tsr** provides a basis for *worker* programs running on the compute boards. It includes the minimal system code **m_system:mk009.sys** for the MK009 boards. This procedure runs the system services on the board, and provides a channel **debug** that you can use to write debugging messages to the screen via the supervisor bus. Use the streams package to write such messages, and ensure that they are terminated with **linefeed** or **breakch**.

Minimal System code for a Master processor.

The transputer in the top left corner of the array in fig.2 is connected to both the host and the graphics board. It has access via the host board to the keyboard and screen. The template **course:mk009_full.tsr** gives access to these channels. The system code is in **m_system:mk009_full.sys**. Keyboard input is buffered until a return character is detected. Messages written to the channel **screen** go there directly, and messages written to **debug** go via the supervisor bus.

Full System code for a Master processor.

This procedure gives access to all of the facilities on the host board, keyboard, screen, files, and mouse. A template can be found in **course:mastercall.tsr** and **d_lib:mastersys.sys** contains the system code.

screen, keyboard and debug: As per minimal procedure above.

Screen Logging: A log of all messages sent to the channel **screen** can be activated by setting the boolean **log** to **TRUE** and data is then written to the file **screen.log** in the default directory.

Mouse: If the boolean **mouse** is set **TRUE** then **mastersys** handles all the low level calls necessary to run the mouse. A call to **mouse.coords** in the **user.process** procedure causes the mouse cursor to be switched on. When the buttons on the mouse are pressed the cursor disappears and **mouse.coords** sets the coordinates of the mouse cursor and the status (an integer between 1 and 7 depending upon which button combination was pressed). To use the mouse the process **gmux** must be run concurrently with **mastersys**, and the graphics board must be running a command orientated graphics program such as **course:commander.sys**.

Files: Mastersys gives access to up to 4 file streams which can be run concurrently. These streams should be used in conjunction with the procedures in **m_files:files.lib** or the file I/O routines. There are high level routines to write streams of characters to a file or to read them back, and to read/write unformatted fixed-length record data files to/from arrays.

which raises **input** to the power **power**. All three parameters are of type **REAL32**, and

```
RANP( result, seed )
```

which generates pseudo-random real numbers between 0 and 1 using a linear congruential algorithm. Here **result** is of type **REAL32** and **seed** is of type **INT**. **seed** should be given a large initial value.

Notes:

a) You must not try to call **RANP** with **seed** declared as **VAL** as **seed** is updated by the procedure.

b) Be careful to alter the initial value of **seed** from one run to the next unless you require the same sequence.

c) Don't call **RANP** with the same initial value of **seed** on two or more processors, as they will then each generate the same sequence of pseudo-random numbers.

d) **RANP** has passed 'run', 'maximum of 5' and 'spectral' tests of randomness.

A multi-processor linear random number generator is available in **lib_maths:random.lib** which uses a modified multiplier in the linear congruential algorithm to ensure that if **T** processors each generate **N** numbers then these **TxN** numbers are the same as would be generated by a single linear congruential random number generator called **TxN** times.

There is also a gaussian random number generator. It uses a *polar Box-Muller* algorithm to generate pairs of gaussian-distributed random numbers.

```
GaussRan( result1, result2, seed )
```

It can be found in **lib_maths:random.lib**. The type of **result1** and **result2** is **REAL32** and **seed** has type **INT**. This uses the multi-processor linear random number generator. It has been tested, and gives the correct mean and variance, and is free of processor-to-processor correlations; nothing else is guaranteed. The same care should be taken with the initial values of **seed**.

The names given above are those for 32-bit procedures. The 64-bit procedures all have names beginning with **D** followed by the name given above, **DCOSP(), DSINP()**, etc.. The gaussian random number generators are only available in single precision.

Details of the implementation of each of the procedures is given in the Occam Elementary Function Library Manual.

[6] 8. Input Output Procedures

We use a standard set of I/O routines. These write data to, or read data from channels as streams of ASCII characters, communicated as integers. They can be found in **m_streams:streams.lib** on the microVAX, or **streams.tsr** on the PC+B004 systems.

[6] 8.1 Output

These procedures write numbers or strings as streams down channels. It is important that you terminate all output with a **linefeed** or a **breakch** (see procedure MuxStream(...)).

`Writes( CHAN out, VAL []BYTE string )`

Writes( ) writes literal strings (e.g. "hello") or BYTE arrays as a stream of integers to the channel out, such a routine might typically be

```
PROC Writes( CHAN out, VAL []BYTE string )
  SEQ i=0 FOR SIZE string
    out ! INT string[i]
:
```

`Writen( CHAN out, VAL INT number )`

Writes the integer number as a stream of ASCII characters to the channel out.

`Writed( CHAN out, VAL INT number, columns )`

Performs as Writen, but formats the output so that columns digits are sent, the extra being leading spaces.

`REAL32write( CHAN out VAL REAL32 real, VAL INT before, after )`

Writes the 32 bit real number real to the channel, with before places before the decimal place and after places after. Total field width is before+after+2. Exponential notation is adopted where necessary.

`Writef( CHAN out, VAL []BYTE format, VAL INT p1, p2, p3, p4 )`

Writef does a formatted write of the string format. Up to four integer parameters can be written into the string. Mark the positions of the numbers by "%N". e.g.

```
VAL id IS 3:
VAL total IS 333:
SEQ
  Writef( out, "Processor %N, total is %N*N",id,total,0,0)
```

This will write the string "**Processor 3, total is 333**" to the channel out.

An output channel, for example the screen, can only be used by one process at a time. It cannot be used by (say) three parallel processes simultaneously. If you want to write output from parallel processes to one channel then you must declare an output channel for each, and join them with a multiplexer. The procedure MuxStream(...) does this.

```
MuxStream( VAL []BYTE name, []CHAN in, CHAN out )
```

Streams written to the channels **in** are joined and sent down the channel **out**. In order to ensure that the messages don't get jumbled up, the multiplexer locks on to one channel until it is sent a **linefeed** or a **breakch**. To do this add "*N" to the end of a string or ensure that **linefeed** is sent to the output channel (see below). A consequence of this is that they must be sent! If they are not, the multiplexers in the system code will hang. The messages are tagged by the string **name**, set it to "*#00" if the tags are not required. The procedure terminates when it receives an **endstreamch** from each input. The following example illustrates correct use of MuxStream(...).

```
SEQ
  [2]CHAN to.screen:
  PAR
    SEQ
      ...  task 1
      Writes( to.screen[0], " Task 1 complete*N")
      to.screen[0] ! endstreamch
    SEQ
      Writes( to.screen[0], " Task 2 starting*N")
      ...  task 2
      Writes( to.screen[1], " Task 2 complete*N")
      to.screen[1] ! endstreamch
    MuxStream( "joiner", to.screen, screen )

  Writes( screen, "All done*N")
```

It will write the messages

```
(joiner(1)) Task 2 starting
(joiner(0)) Task 1 complete
(joiner(1)) Task 2 complete
All done
```

to the channel **screen**, although the messages tagged by **joiner** will not necessarily arrive in this order.

[6] 8.2 Input

The following procedures take input in the form of a stream of ASCII characters and convert to strings or numbers as appropriate.

```
ReadLine( CHAN in, []BYTE buffer )
```

Reads characters from the channel **in** until either **buffer** is full, or a linefeed character is encountered, in which case **buffer** is padded with byte zeros ("*#00").

Readname(CHAN in, []BYTE name)

Performs as **ReadLine()** but also terminates by a space character. It writes the first word encountered into **name**.

Readn(CHAN in, INT number)

Converts a stream of ASCII characters terminated as above into an integer.

REAL32read(CHAN in, REAL32 real)

Builds a 32-bit real from a stream of appropriate characters; **real** will be set to "not a number" (**NaN**) if the stream is invalid.

7. Notes on Occam 2

These are brief notes on occam 2, its reserved strings and glyphs. Fuller details are given in the sections indicated, and cross-references at the end of those sections. Statements marked *[0]* are described sufficiently in one line.

[7] 1. Reserved Strings

Certain strings in capital letters are reserved, as follows.

[16]	AFTER	comparison operator, used to cause a delay
[15]	ALT	process after first activated channel executes
[8]	AND	boolean operator
[18]	AT	used with PLACE
[4]	BYTE	an integer between 0 and 255
[4]	BOOL	boolean type, can be TRUE or FALSE
[4]	CHAN	definition of a channel
[8]	FALSE	boolean constant
[14]	FOR	used in array segments and in replicators
[14]	FROM	used in array segments
[7]	IF	controls conditional execution of processes
[4]	INT	integer type definition
[10]	INT	as used in type conversions
[4]	INT**	** = 16, 32, 64, bits in an integer definition
[5]	IS	see VAL
[11]	MINUS	unchecked arithmetic operator
[8]	NOT	boolean operator
[8]	OR	boolean operator
[3]	PAR	the start of a parallel construct
[18]	PLACE	used for memory address placements
[17]	PLACED PAR	placement of process on a particular processor
[11]	PLUS	unchecked arithmetic operator
[15]	PRI	priority, used as PRI ALT or PRI PAR
[12]	PROC	procedure
[17]	PROCESSOR	declares a processor
[4]	REAL**	** = 32, 64, bits in real number type definition
[10]	REAL	as used in type conversions
[10]	REM	x REM y gives remainder when x is divided by y
[0]	RETYPES	generalised type conversion
[10]	ROUND	used for conversions between INT and REAL
[3]	SEQ	the start of a sequential construct
[13]	SIZE	gives the size of an array
[0]	SKIP	process starts, proceeds and terminates
[0]	STOP	process starts but does not proceed or terminate
[16]	TIMER	type TIMER used as clocks by processes
[11]	TIMES	unchecked arithmetic operator
[8]	TRUE	boolean constant
[10]	TRUNC	for truncation, not rounding
[5]	VAL	sets a value, eg VAL INT year IS 365:
[7]	WHILE	conditional execution loop
[18]	WORKSPACE	workspace allocation

[7] 2. Glyphs

[0]	.	valid character within a name
[1]	... {{{ }}} {{{F	for folds - occam programming system, OPS
[2]	--	comment
[0]	,	separator
[4]	:	terminator for any definition
[12]	:	terminator for a procedure
[6]	? ! ;	for channel input and output
[10]	:=	arithmetic assignment
[10]	+ - * /	arithmetic operations
[16]	\	same as REM
[5]	( )	for type declarations in VAL
[10]	( )	for arithmetic, there is no precedence
[12]	( ) ()	for procedure parameters
[13]	[ ] []	used for array variables
[8]	&	same as AND
[8]	= <> > < >= <=	boolean comparison operators
[9]	/\ \/ >< ~ << >>	bitwise operators
[0]	"	for byte strings, eg "hello"
[0]	'	byte ASCII code for a character, eg 'h'
[0]	#	for entering hexadecimal number, eg #A7
[0]	*C *N	reserved strings for formatting

[7] 3. More details about reserved strings and glyphs

[1] **... {{{ }}} {{{F**

An occam program is constructed in folds. In the occam programming system, a fold is indicated by

```
... name describing fold
```

the words following ... are comments, and are optional. For a filed fold, ... is replaced by ...F The first word following ...F is taken as the file name. Other types of fold are distinguished by ... SC (separately compilable), ... EXE (completed procedure) ... PROGRAM ... COMMENT An open fold then starts with the marker {{{ and ends with }}}, {{{F denotes a filed fold.

see page 87

[2] **--**

On any line of code, or a blank line, a comment is preceded by -- thus

```
-- this is a comment
```

see pages 8,88

[3] PAR SEQ

The indented processes which follow PAR are done in parallel, those which follow SEQ in sequence

```
PAR
  SEQ
    ... fold 1
    ... fold 2
  SEQ
    ... fold 3
    ... fold 4
```

see pages 3,5

[4] BYTE BOOL CHAN INT REAL :

BYTE, BOOL, INT, CHAN, REAL are types. Everything must be typed, and type declarations last for the scope of the following process.

```
INT i , j , BOOL b , BYTE c , REAL r :
CHAN chan1 , chan2 :
```

: terminates each declaration line and attaches them to the process that follows them.

Types must match in an expression, so type changes are needed as below with INT i and BYTE b set,

```
i := i + ((INT b) + (INT TRUE))
b := (BYTE FALSE)
```

see pages 3,4,7,8

[5] VAL IS ( )

VAL causes type definition and sets the value, and would set x below to INT by default unless otherwise directed as shown

```
VAL x IS 7 (BYTE) , VAL INT year IS 365 :
```

There seem to be flaws in occam's razor. There are two other ways of defining the value of year

```
VAL year IS 365 (INT) :
VAL year IS 365 :
```

Another use of VAL is as an abbreviation which involves variables on its right hand side which are used once at execution time to evaluate the expression, and then should not be changed within its scope. As an example

```
VAL example IS (a*(b+c)) :
```

see pages 8,15

[6] ? ! ;

Input on a channel is requested by ? output by !
In this example the messages are single integers.

```
INT i, j, k :
CHAN chan1, chan2 :
SEQ
  chan1 ? i
  chan2 ! j ; k
```

see page 2

[7] WHILE IF

Assuming x is previously defined the SEQ which is within its scope will happen repeatedly while a condition is fulfilled

```
WHILE x=0
  SEQ
    chan1 ? x
    chan2 ! x
```

IF selects the first boolean in textual order that is TRUE and starts only the associated process

```
IF
  bool.1
    ...one
  bool.2
    ...two
```

If neither bool.1 nor bool.2 is TRUE then the result is the same as STOP; this can be remedied with a third option

```
TRUE
  SKIP
```

which then starts SKIP preventing stoppage.

see pages 2,4,11

[8] TRUE FALSE AND OR NOT & = <> > < >= <=

The following are boolean comparators and must compare values of the same type

=	equal to
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

The boolean operators and constants may be used to construct boolean expressions as follows, where a, b and c are declared BOOL

```
NOT b = FALSE
a = ( b = c) AND ( a OR c )
```

TRUE is the same as BOOL 1 and
FALSE is the same as BOOL 0.
& is the same as AND.

see pages 5,10

[9] /\ \/ >< ~ << >>

These are bitwise operators, and operate on the individual bits in a word.

/\	bitwise and
\/	bitwise or
><	bitwise exclusive or
~	bitwise not
<<	left shift
>>	right shift

see page 10

[10] REM INT REAL ROUND TRUNC

:= + - * / ( ) \

As there is no precedence for the arithmetic operations parentheses must be used to specify operation order; x REM y gives remainder when x is divided by y. All variables must be of the same type, so if x, y and z are INT, r is REAL and b is BYTE, the following assignment

```
x := ((INT b)+(y/z))-((x REM y)*(INT ROUND r))
```

involves most of the operators. For converting an integer, say i, use REAL ROUND i.
TRUNC truncates, \ is the same as REM.

see pages 2,9,26

[11] PLUS MINUS TIMES

These are unchecked arithmetic operators for addition, subtraction and multiplication in 2's complement. The result lies in the range -n to n-1 where n is 2**(number of bits in an INT - 1). As these operators are unchecked they are much faster for small-number manipulation.

[12] PROC : ( ) ()

A procedure can take parameters which may be channels, variables or values. If the value of a variable parameter is altered within the body of the PROC, that variable retains its new value after the instance of the PROC. If the value of a variable parameter is not to be altered within the PROC, then it is more efficient to declare it as a VAL, e.g. VAL INT fix

```
PROC ext (INT i , VAL INT fix)
  SEQ
    i := i + fix
    ... more code
:
```

: indicates the end of the PROC and must be indented to the same level as PROC. Procedures without parameters must have () in their declaration and in their use. The example below shows the use of a free variable which is available to every PROC and process within its scope.

```
INT free.var :
PROC ext ()              -- procedure declaration
  INT i :
  SEQ
    i := free.var
    ... more code
:
SEQ
  ext ()                 -- procedure use
  ... more code
```

see pages 4,23

[13] SIZE [ ] []

Square brackets are used for arrays and their declarations; SIZE measures the array size. The following is a declaration of two arrays

```
[3]INT i , [3][3]INT j :
```

The following PROC reads single integers until the SIZE of the array whose name replaces k in the instance, is reached. The size of k is undefined in the declaration by the use of []

```
PROC read (CHAN in, []INT k)
  SEQ i = 0 FOR (SIZE k)
    in ? k[i]
:
```

see pages 16,26

[14] FROM FOR

These can be used to make array segments which are parts of bigger arrays.
Given the declaration `[10]INT x :` then

```
[x FROM 2 FOR 5]
```

is a `[5]INT`.

Smaller arrays of variables, channels or timers can be selected from arrays of the same type

```
[5]INT z :
SEQ
  z := [x FROM 4 FOR 5]
```

FOR is also used in replicators, e.g. where `...` eg is repeated sequentially for i=0,1,2,3

```
SEQ i = 0 FOR 4
  ... eg
```

see page 17

[15] ALT PRI ALT PRI PAR

ALT – The first guard that is ready causes its process to execute. A guard can be an input, or a construct such as **(boolean & input)**

```
INT tag :
ALT
  chan1 ? tag
    ... first process
  chan2 ? tag
    ... second process
```

Nesting or replication is allowed,

```
PROC join ( []CHAN in, CHAN out )
  INT x:
  ALT i = 0 FOR SIZE in
    SEQ
      in[i] ? x
      out ! x
```

[17] PLACED PAR PROCESSOR

Used to distribute processes over processors, e.g.

```
PLACED PAR
  PROCESSOR 0 T4          -- identifier is 0
    ... channel placement
    process.0 (chan1, chan2)
  PROCESSOR 1 T4          -- identifier is 1
    ... channel placement
    process.1 (chan1, chan2)
```

This declares two processors each with a unique identifier and a specification of the transputer type, T2 or T4. It then gives them a job to do.

see page 32

[18] PLACE AT WORKSPACE

Allocation of a variable, channel, timer or array e.g. computer, to an address, e.g. Edinburgh.

```
PLACE computer AT Edinburgh :
```

The allocation of workspace, say [9]INT work, for a specific process, say PROC specific, can be done

```
WORKSPACE work : specific
```

and then in conjunction with PLACE this workspace can be placed at a specific memory location, e.g. in on-chip RAM.

see page 32

Index