Communicating processes and occam

INMOS Technical Note 20

David May

February 1987
72-TCH-020-00

Contents

1 Introduction

The occam programming language [1] enables an application to be described as a collection of processes which operate concurrently and communicate through channels. In such a description, each occam process describes the behaviour of one component of the implementation, and each channel describes a connection between components.

The design of occam allows the components and their connections to be implemented in many different ways. This allows the choice of implementation technique to be chosen to suit available technology, to optimise performance, or to minimise cost.

Occam has proved useful in many application areas. It can be efficiently implemented on almost any computer and is being used for many purposes - real time systems, compilers and editors, hardware specification and simulation.

2 Architecture

Many programming languages and algorithms depend on the existence of the uniformly accessible memory provided by a conventional computer. Within the computer, memory addressing is implemented by a global communications system, such as a bus. The major disadvantage of such an approach is that speed of operation is reduced as the system size increases. The reduction in speed arises both from the increased capacitance of the bus which slows down every bus cycle, and from bus contention.

The aim of occam is to remove this difficulty; to allow arbitrarily large systems to be expressed in terms of localised processing and communication. The effective use of concurrency requires new algorithms designed to exploit this locality.

The main design objective of occam was therefore to provide a language which could be directly implemented by a network of processing elements, and could directly express concurrent algorithms. In many respects, occam is intended as an assembly language for such systems; there is a one-one relationship between occam processes and processing elements, and between occam channels and links between processing elements.

2.1 Locality

Almost every operation performed by a process involves access to a variable, and so it is desirable to provide each processing element with local memory in the same VLSI device.

The speed of communication between electronic devices is optimised by the use of one directional signal wires, each connecting only two devices. This provides local communication between pairs of devices.

Occam can express the locality of processing, in that each process has local variables; it can express locality of communication in the each channel connects only two processes.

2.2 Simulated and Real concurrency

Many concurrent languages have been designed to provide simulated concurrency. This is not surprising, since until recently it has not been economically feasible to build systems with a lot of real concurrency.

Unfortunately, almost anything can be simulated by a sequential computer, and there is no guarantee that a language designed in this way will be relevant to the needs of systems with real concurrency. The choice of features in such languages has been motivated largely by the need to share one computer between many independent tasks. In contrast, the choice of features in occam has been motivated by the need to use many communicating computers to perform one single task.

An important objective in the design of occam was to use the same concurrent programming techniques both for a single computer and for a network of computers. In practice, this meant that the choice of features in occam was partly determined by the need for an efficient distributed implementation. Once this had been achieved, only simple modifications were needed to ensure an efficient implementation of concurrency on a single sequential computer. This approach to the design of occam perhaps explains some of the differences between occam and other 'concurrent' languages.

3 The occam primitives

Occam programs are built from three primitive processes:

v := e	assign expression e to variable v
c ! e	output expression e to channel c
c ? v	input from channel c to variable v

The primitive processes are combined to form constructs:

SEQ	sequence
IF	conditional
PAR	parallel
ALT	alternative

A construct is itself a process, and may be used as a component of another construct.

Conventional sequential programs can be expressed with variables and assignments, combined in sequential and conditional constructs. The order of expression evaluation is unimportant, as there are no side effects and operators always yield a value.

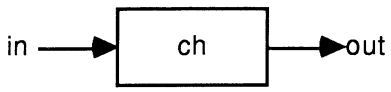
Conventional iterative programs can be written using a while loop. The absence of explicit transfers of control perhaps needs no justification in a modern programming language; in occam it also removes the need to prohibit, or define the effect of, transferring control out of a parallel component or procedure.

Concurrent programs make use of channels, inputs and outputs, combined using parallel and alternative constructs.

The definition and use of occam procedures follows ALGOL-like scope rules, with channel, variable and value parameters. The body of an occam procedure may be any process, sequential or parallel. To ensure that expression evaluation has no side effects and always terminates, occam does not include functions.

A very simple example of an occam program is the buffer process below.

```
WHILE TRUE
  VAR ch:
  SEQ
    in ? ch
    out ! ch
```



Indentation is used to indicate program structure. The buffer consists of an endless loop, first setting the variable `ch` to a value from the channel in, and then outputting the value of `ch` to the channel out. The variable `ch` is declared by `VAR ch`. The direct correspondence between the program text and the pictorial representation is important, as a picture of the processes (processors) and their connections is often a useful starting point in the design of an efficiently implementable concurrent algorithm.

4 The Parallel Construct

The components of a parallel construct may not share access to variables, and communicate only through channels. Each channel provides one way communication between two components; one component may only output to the channel and the other may only input from it. These rules are checked by the compiler.

The parallel construct specifies that the component processes are "executed together". This means that the primitive components may be interleaved in any order. More formally,

```
PAR                    SEQ
  SEQ          =         x := e
    x := a               PAR
    P                      P
  Q                        Q
```

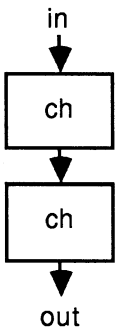
so that the initial assignments of two concurrent processes may be executed in sequence until both processes start with an input or output. If one process starts with an input on channel c, and the other an output on the same channel c, communication takes place:

```
PAR                    SEQ
  SEQ          =         x := e
    c ! e                PAR
    P                      P
  SEQ                      Q
    c ? x
    Q
```

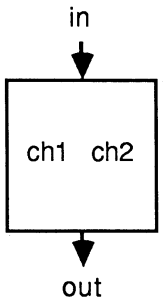
The above rule states that communication can be thought of as a distributed assignment.

Two examples of the parallel construct are shown below.

```
CHAN c:
PAR
  WHILE TRUE
    VAR ch:
    SEQ
      in ? ch
      c ! ch
  WHILE TRUE
    VAR ch:
    SEQ
      c ? ch
      out ! ch
```




```
VAR ch1:
VAR ch2:
SEQ
 in ? ch1
 WHILE TRUE
   SEQ
     PAR
       in ? ch2
       out ! ch1
     PAR
       in ? ch1
       out ! ch2
```



The first consists of two concurrent versions of the previous example, joined by a channel to form a "double buffer". The second is perhaps a more conventional version. As 'black boxes', each with an input and an output channel, the behaviour of these two programs is identical; only their internals differ.

4.1 Synchronised communication

Synchronised, zero-buffered, communication greatly simplifies programming, and can be efficiently implemented. In fact, it corresponds directly to the conventions of self timed signalling [2]. Zero buffered communication eliminates the need for message buffers and queues. Synchronised communication prevents accidental loss of data arising from programming errors. In an unsynchronised scheme, failure to acknowledge data often results in a program which is sensitive to scheduling and timing effects.

Synchronised communication requires that one process must wait for the other. However, a process which requires to continue processing whilst communicating can easily be written:

```
PAR
  c ! x
  P
```

5 The Alternative Construct

In occam programs, it is sometimes necessary for a process to input from any one of several other concurrent processes. This could have been provided by a channel 'test', which is true if the channel is ready, false otherwise. However, this is unsatisfactory because it requires a process to poll its inputs "busily"; in some (but by no means all) cases this is inefficient.

Consequently, occam includes an alternative construct similar to that of CSP [3]. As in CSP, each component of the alternative starts with a guard - an input, possibly accompanied by a boolean expression. From an implementation point of view, the alternative has the advantage that it can be implemented either "busily" by a channel test or by a "non-busy" scheme. The alternative enjoys a number of useful semantic properties more fully discussed in [4][5]; in particular, the formal relationship between parallel and alternative is shown below:

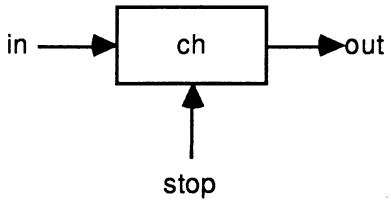
```
                          ALT
                            c ? x
                              PAR
PAR                             P
  SEQ                           SEQ
    c ? x                         d ? y
    P           =                 Q
  SEQ                         d ? y
    d ? y                       PAR
    Q                             Q
                                  SEQ
                                    c ? x
                                    P
```

This equivalence states that if two concurrent processes are both ready to input (communicate) on different channels, then either input (communication) may be performed first.

One feature of CSP omitted from occam is the automatic failure of a guard when the process connected to the other end of the channel terminates. Although this is a convenient programming feature, it complicates the channel communication protocol, introducing the need for further kinds of message. In addition, it can be argued that many programs are clearer if termination is expressed explicitly.

A simple example of the alternative is shown below; this is a 'stopable' buffer program

```
WHILE going
   ALT
    in ? ch
      out ! ch
    stop ? ANY
      going := FALSE
```



in which `stop ? ANY` inputs any value from the channel `stop`, and as a result causes the loop to terminate.

5.1 Output Guards

Output guards are a very convenient programming tool. In particular, they allow programs such as the following buffer process to be written in a natural way.

```
WHILE TRUE
  ALT
    count>0 & output ! buff [ outpointer ]
      SEQ
        outpointer := (outpointer + 1) REM max
        count := count - 1
    count<max & input ? buff [ inpointer ]
      SEQ
        inpointer := (inpointer + 1) REM max
        count := count + 1
```

It is very tempting to include output guards in a communicating process language, and attempts have been made to include output guards in occam. The major difficulty is in the distributed implementation; in a program such as

```
PAR
  ALT
    c ! x1
    d ? x2
  ALT
    c ? y1
    d ! y2
```

what is expected to happen in the event that two identical processors both enter their alternative at exactly the same time? Clearly some asymmetry

must be introduced; the easiest way to do this is to give each processor in a system a unique number. Even so, the provision of output guards greatly complicates the communications protocol. For this reason, output guards are omitted from occam, and the above buffer must be written as shown below.

```
PAR
  WHILE TRUE
    ALT
      count>0 & req ? ANY
        SEQ
          reply ! buff [ outpointer ]
          outpointer := (outpointer + 1) REM max
          count := count - 1
      count<max & input ? buff [ inpointer ]
        SEQ
          inpointer := (inpointer + 1) REM max
          count := count + 1
  WHILE TRUE
    SEQ
      req ! ANY
      reply ? ch
      output ! ch
```

On the other hand, an occam implementation with only input guards can be used to write the communications kernel for a "higher level" version of occam with output guards. An example of an algorithm to implement output guards in CSP is given in [6]; and one for occam is given in [7].

6 Channels and hierarchical decomposition

An important feature of occam is the ability to successively decompose a process into concurrent component processes. This is the main reason for the use of named communication channels in occam. Once a named channel is established between two processes, neither process need have any knowledge of the internal details of the other. Indeed, the internal structure of each process can change during execution of the program. The parallel construct, together with named channels provides for decomposition of an application into a hierarchy of communicating processes, enabling occam to be applied to large scale applications. This technique cannot be used in languages which use process (or 'entry') names, rather than channels, for communication.

In specifying the behaviour of a process, it is important that a specification of the protocol used on the channel exists, and the best way to do this varies

from program to program (or even from channel to channel!). For example, Backus-Naur Form is often suitable for describing the messages which pass between the individual processes of a linear pipeline of processes. On the other hand, for more complex interactions between processes, it is often useful to describe the interactions by an occam "program" in which all unnecessary features are omitted. This often enables the interactions between processes to be studied independently of the data values manipulated. For example:

```
SEQ
  request ?
  WHILE TRUE
    PAR
      reply !
      request ?
```

describes a process which inputs a request, and then endlessly inputs a new request and outputs a reply, in either order. Such a process would be compatible, in some sense, with any of the following processes:

```
WHILE TRUE
  SEQ
    request !
    reply ?
```

```
SEQ
  request !
  WHILE TRUE
    SEQ
      request !
      reply ?
```

```
SEQ
  request !
  WHILE TRUE
    PAR
      request !
      reply ?
```

More design aids are needed to assist in the specification and checking of channel protocols.

7 Arrays and Replicators

The representation of arrays and 'for' loops in occam is unconventional. Although this has nothing to do with the concurrency features of occam, it seems to have significant advantages over alternative schemes.

To eliminate trivial programming errors, it is desirable that there is a simple relationship between an array declaration and a loop which performs some operation for every element of an array. This might lead a language designer to a choice of

```
ARRAY a [base TO limit] ...

FOR i IN [base TO limit] ...
```

It is also useful if the number of elements in an array, or the number of iterations of a loop, is easily visible. For this reason, a better choice might be

```
ARRAY a [base FOR count] ...

FOR i IN [base FOR count] ...
```

For the loop, this gives a further advantage: the 'empty' loop corresponds to count=0 instead of limit¡base. This removes the need for the unsatisfactory 'loop':

```
FOR i IN [0 TO -1]
```

Implementation can be simplified by insisting that all arrays start from 0. Finally, in occam the `FOR` loop is generalised, and its semantics simplified. An occam 'replicator' can be used with any of `SEQ`, `PAR`, `ALT` and `IF`; its meaning is defined by:

```
X n = b FOR c     =       X
  P(n)                      P (b)
                            P (b+1)
                            ...
                            P (b+c-1)
```

where `X` is one of `SEQ`, `PAR`, `ALT` and `IF`, `n` is a name and `b`, `c` expressions. This definition implicitly defines the 'control variable' `n`, and prevents it being changed by assignments within `P`.

The introduction of arrays of variables and channels does complicate the rules governing the correct use of channels and variables. Simple compile-time checks which are not too restrictive are:

- No array changed by assignment (to one of its components) in any of the components of a parallel may be used in any other component.
- No two components of a parallel may select channels from the same array using variable subscripts.
- A component of a parallel which uses an array for both input and output may not select channels from the array using variable subscripts.

where a variable subscript is a subscript which cannot be evaluated by the compiler.

8 Time

The treatment of time in occam directly matches the behaviour of a conventional alarm clock.

Time itself is represented in occam by values which cycle through all possible integer values. Of course, it would have been possible to represent time by a value large enough (say 64 bits) to remove the cyclic behaviour, but this requires the use of multiple length arithmetic to maintain the clock and is probably not justified.

Using an alarm clock, it is possible at any time to observe the current time, or to wait until the alarm goes off. Similarly, a process must be able to read the clock at any time, or wait until a particular time. If it were possible only to read the clock, a program could only wait until a particular time "busily". Like the alternative construct, the "wait until a time" operation has the advantage that it can be implemented "busily" or "non-busily".

A timer is declared in the same way as a channel or variable. This gives rise to a relativistic concept of time, with different timers being used in different parts of a program. A localised timer is much easier to implement than a global timer.

A timer is read by a special 'input'

```
time ? v
```

which is always ready, and sets the variable v to the time. Similarly, the 'input'

```
time ? AFTER t
```

waits until time t.

The use of an absolute time in occam instead of a delay is to simplify the construction of programs such as

```
WHILE TRUE
  SEQ
    time ? AFTER t
    t := t + interval
    output ! bell
```

in which n rings of the bell will always take between (n*interval) and n* (interval+1) ticks. This would not be true of a program such as

```
WHILE TRUE
  SEQ
    DELAY interval
    output ! bell
```

because of the time taken to ring the bell.

It is not possible, in occam, for a process to implement a timer. This would require a 'timer output' such as

```
timer ! PLUS n
```

which advances the timer by n ticks. There is no obvious reason why this could not be included in occam. It would be particularly useful in constructing timers of different rates, or in writing a process to provide 'simulated time'.

9 Types and Data structures

The occam described so far makes few assumptions about data types. Any data type could be used - provided that values of that type can be assigned, input and output according to the rule

```
PAR
  c ! x          =       y := x
  c ? y
```

To preserve this rule, and keep the implementation of communication simple, it is best for assignment not to make type conversions.

The initial version of occam provides untyped variables and one dimensional arrays. No addressing operations are provided, as this would make it impossible for the compiler to check that variables are not shared between concurrent processes.

Occam has been extended to include data types. The simple variable is replaced with boolean, byte and integer types, and multi-dimensional arrays are provided. Communication and assignment operate on variables of any data type, allowing arrays to be communicated and assigned.

A detailed description can be found in [8].

10 Implementation of occam

The implementation of concurrent processes and process interaction in occam is straightforward. 'this results from the need to implement occam on the transputer using simple hardware and a small number of microcoded instructions. Conveniently, the transputer instructions used to implement occam can be used as definitions of the 'kernel primitives' in other implementations of occam. A discussion of the implementation of occam can be found in [9]. However, some measure of the efficiency of the occam primitives is provided by the performance of the Inmos transputer: about 1 microsecond/component of PAR, and 1.5 microseconds for a process communication.

Another interesting feature of occam is that the process interactions directly represent hardware mechanisms, which is one reason why occam is being used as a hardware description language.

10.1 Compile time allocation

For runtime efficiency, the advantages of allocating processors and memory at compile time are clear. To allow the compiler to allocate memory, some implementation restrictions are imposed. Firstly, the number of components of an array, and the number of concurrent processes created by a parallel replicator, must be known at compile time. Secondly, no recursive procedures are allowed. The effect of these restrictions is that the compiler can establish the amount of space needed for the execution of each component of a parallel construct, and this makes the run-time overhead of the parallel construct very small.

On the other hand, there is nothing in occam itself to prevent an implementation without these restrictions, and this would be fairly straightforward for a single computer with dynamic memory allocation.

A distributed implementation of 'recursive occam' might allow a tree of processors to be described by:

```
PROC tree (VALUE n, CHAN down, CHAN up)
  IF
    n=0
      leaf ( down, up )
    n>0
      CHAN left.down, left.up
      CHAN right.down, right.up
      PAR
        tree (n-1, left.down, left.up)
        tree (n-1, right.down, right.up)
```

```
        node ( down, up,
               left.down, left.up,
               right.down, right.up )
```

If the depth of the tree is known at compile time (as it normally would be if the program is to be executed on a fixed size processor array), the same effect can be achieved by a non-recursive program such as:

```
DEF p = TABLE [1, 2, 4, 8, 16, 32, 64, 128]:

-- depth of tree = n
CHAN down [n*(n-1)] :
CHAN up   [n*(n-1)] :

PAR
  PAR i = [0 FOR n-1]
    PAR j = [0 FOR p[i]]
      branch ( down [p[i] + j], up [p[i] + j],
               down [p[i+1]+(j*2)], up [p[i+1]+(j*2)],
               down [p[i+1]+(j*2)+1], up [p[i+1]+(j*2)+1] )
  PAR i = [0 FOR p[n]]
    leaf ( down [p[n]+i], up [p[n]+i] )
```

Obviously, a pre-processor could be used to provide a correctness preserving transformation between these two programs.

If the depth of the tree above were not known, it is not clear how such a program could be mapped onto a processor array, either explicitly by the programmer or implicitly by the implementation. Fortunately, this problem can be left for the future; many applications require only simple compile time allocation of processors and memory space.

11 Program Development

The development of programs for multiple processor systems is not trivial. One problem is that the most effective configuration is not always clear until a substantial amount of work has been done. For this reason, it is very desirable that most of the design and programming can be completed before hardware construction is started.

This problem is greatly reduced by the property of occam mentioned above: the use of the same concurrent programming techniques for both a network and a single computer. A direct consequence of this is that a program ultimately intended for a network of computers can be compiled and executed efficiently by a single computer used for program development.

Another important property of occam in this context is that occam provides a clear notion of "logical behaviour"; this relates to those aspects of a program not affected by real time effects. It is guaranteed that the logical behaviour of a program is not altered by the way in which processes are mapped onto processors, or by the speed of processing and communication.

This notion of "logical behaviour" results from the relatively abstract specification of parallel and alternative; it allows almost any scheduling system to be used to simulate concurrency. For the parallel construct, an implementation may choose the order in which the individual actions of the components are executed. If several components are ready (not waiting to communicate), the implementation may execute an arbitrary subset of them and temporarily ignore the rest. For the alternative, an implementation may select any ready component; there is no requirement to select the "earliest", or to select randomly.

11.1 Configuration

The configuration of a program to meet real time constraints is provided by annotations to the parallel and alternative constructs. For the parallel construct, the components may be placed on different processors, or may be prioritised. For the alternative construct, the components may be prioritised. A better version of the 'stoppable' buffer shown earlier would therefore be:

```
WHILE going
  PRI ALT
    stop ? ANY
      going := FALSE
    in ? ch
      out ! ch
```

A prioritised alternative can easily be used to provide either a prioritised or a 'fair' multiplexor:

```
WHILE TRUE -- prioritised
  PRI ALT i = 0 FOR 10
    in [i] ? ch
      out ! ch

WHILE TRUE -- 'fair'
  PRI ALT i = 0 FOR 10
    in [(i+last) REM 10] ? ch
      SEQ
```

```
      out ! ch
      last := (i+1) REM 10
```

In practice, only limited use is made of prioritisation. For most applications, the scheduling of concurrent processes and the method of selecting alternatives is unimportant. This is because, assuming that the system is executing one program, the processes which are consuming all of the processing resources must eventually stop, and wait for the other processes to do something. If this is not the case, the other processes are redundant, and can be removed from the program. An implementation should not, of course, allow a processor to idle if there is something for it to do. But this property is true of any programming language!

Scheduling is important where a system executes two disjoint processes, or has to meet some externally imposed constraint. Both of these occur, for example, in an operating system which deals with disjoint users, and needs to take data from a disk at an externally imposed rate.

12 Occam Programs

Despite being a fairly small language, occam supports a very wide variety of programming techniques. Most important, the programmer may choose between a concurrent algorithm or an equivalent sequential one. A final program often consists of a mixture of the two, in which the concurrent algorithm describes a network of transputers, each of which executes the sequential algorithm.

In practice, it is often best to write the concurrent algorithm first. The reason for this is that only the concurrent program provides freedom in the implementation. A pipeline of ten processes could be executed by a pipeline constructed from up to ten transputers; the number being chosen according to the performance required. It is very unlikely that a sequential program can easily be adapted to produce a concurrent program, never mind one suitable for execution by a network of transputers with no shared memory.

The following example is a concurrent searching algorithm. It uses the tree program shown earlier. The data to be searched is held in the leaf processors; the node processors are used to disperse the data to the leaves and collect the replies.

```
PROC leaf (CHAN down, up) =
  VAR data, enq:
  SEQ
    ... -- load data
```

```
    WHILE TRUE
      SEQ
        down ? enq
        up ! (enq = data)

PROC node (CHAN down, up,
           CHAN left.down, left.up,
           CHAN right.down, right.up) =
  WHILE TRUE
    VAR enq, left.found, right.found:
    SEQ
      down ? enq
      PAR
        left.down ! enq
        right.down ! enq
      PAR
        left.up ? left.found
        right.up ? right.found
      up ! left.found OR right.found
```

However, it is unlikely to be economic to store only one data item in each leaf. Although each leaf could itself execute the above algorithm using a tree of processes, this would not be very efficient. What is needed in each leaf is a conventional sequential searching algorithm operating on an array of data:

```
PROC leaf (CHAN down, up) =
  VAR enq, data [length], found:
  SEQ
    ... -- initialise data
    WHILE TRUE
      SEQ
        found := FALSE
        down ? enq
        SEQ i = [0 FOR length]
          found := (data [i] = enq ) OR found
        up ! found
```

It now remains to choose the number of items held in each leaf so that the time taken to disperse the enquiry and collect the response is small relative to the time taken for the search at each leaf. For example, if the time taken for a single communication is 5 microseconds, and the tree is of depth 7 (128 leaves) only 70 microseconds is spent on communication, about one tenth of the time taken to search 1000 items.

12.1 Example - Systolic arrays

A very large number of concurrent algorithms require only the simplest concurrency mechanisms: the parallel construct and the communication channel. These include the 'systolic array' algorithms described by Kung [10]. In fact, occam enables a systolic algorithm to be written in one of two ways, illustrated by the following two versions of a simple pipeline, each element of which performs a 'compute' step. First, the traditional version:

```
VAR master [ n ]:
VAR slave [ n ] :
WHILE TRUE
  SEQ
    PAR i = 0 FOR n
      compute ( master [ i ], slave [ i ] )
    PAR
      input ? master [ 0 ]
      PAR i = 0 FOR n-1
        master [ i + 1 ] := slave [ i ]
      output ! slave [ n ]
```

This pipeline describes a conventional synchronous array processor. The compute operations are performed in parallel, each taking data from a master register and leaving its result in a slave register. The array processor is globally synchronised; in each iteration all compute operations start and terminate together, then the data is moved along the pipeline. The initialisation of the pipeline is omitted, so the first n outputs will be rubbish.

The main problem with the above program is the use of global synchronisation, which gives rise to the same implementation difficulties as global communication; it requires that the speed of operation must be reduced as the array size increases. A more natural program in occam would be

```
CHAN c [ n + 1 ] :
PAR i = 0 FOR n
  WHILE TRUE
    VAR d:
    VAR r:
    SEQ
      c [ n ] ? d
      compute ( d, r)
      c [ n + 1 ] ! r
```

In this program, c[0] is the input channel, c[n+1] the output channel. Once again, all of the compute operations are performed together. This time there

is no need for initialisation, as no output can be produced until the first input has passed right through the pipeline. More important, the pipeline is self synchronising; adjacent elements synchronise only as needed to communicate data. It seems likely that many systolic array algorithms could usefully be re-expressed and implemented in this form.

12.2 Example - Occam compiler

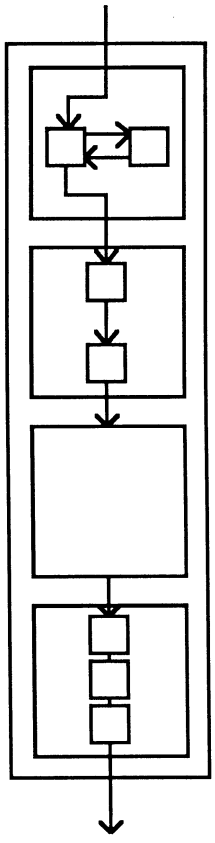
The structure of the occam compiler is shown below. It demonstrates an important feature of the occam support system; the ability to 'fold' sections of program away, leaving only a comment visible. This enables a program, or part of a program, to be viewed at the appropriate level of detail.

```
-- occam compiler
CHAN lexed.program:
CHAN parsed.program:
CHAN scoped.program:
PAR
  -- lexer
  CHAN name.text:
  CHAN name.code:
  PAR
    -- scanner
    -- nametable

  -- parser
  CHAN parsed.lines :
  PAR
    -- line parser
    -- construct parser

  -- scoper

  -- generator
  CHAN generated.constructs :
  CHAN generated.program :
  PAR
    -- construct generator
    -- line generator
    -- space allocator
```



The compiler also illustrates an important programming technique. The nametable process contains data structures which are hidden from the rest of the program. These structures are modified only as a result of messages from the lexical analyser. They are initialised prior to receipt of the first message.

```
-- nametable
SEQ
  -- initialise
  WHILE going
    -- input text of name
    -- look up name
    -- output corresponding code
  -- terminate
```

From the outside, the compiler appears to be a single pass compiler. Internally, it is more like a multiple pass compiler; each process performs a simple transformation on the data which flows through it. The effect of decomposing the compiler in this way was that each component process was relatively easy to write, specify and test; this meant that the component processes could be written concurrently!

13 Conclusions

In many application areas, concurrency can be used to provide considerable gains in performance provided that programs are structured to exploit available technology. For many -application areas (especially signal processing and scientific computation) suitable algorithms already exist, but many areas remain to be explored.

Writing programs in terms of communicating processes tends to produce programs with a large number of concurrent processes, ranging in size from 1 to 1000 lines. Consequently, it is particularly important that the concurrent processing features in the language are efficiently implementable. Occam demonstrates that this efficiency can be achieved for a widely applicable language.

In occam programs, the process/channel structure tends to be used as a major program structuring tool, procedures being used in the normal way within the larger concurrent processes. The process/channel structure seems to be effective for managing the construction of large programs, although more experience is needed in this area.

References

[1] Occam Programming Manual. Prentice-Hall International 1984.

[2] C A Mead and L A Conway: Introduction to VLSI Systems. Addison Wesley 1980 Section 5.

[3] Communicating Sequential Processes, C A R Hoare, Communications of the ACM Vol. 21, 8 (August 1978) p. 666.

[4] Denotational Semantics for Occam, A W Roscoe. Presented at NSF/SERC Seminar on Concurrency, Carnegie-Mellon University, July 1984. To be published.

[5] The Laws of Occam Programming, A W Roscoe and C A R Hoare, Programming Research Group, Oxford University, 1986.

[6] An Effective Implementation for the Generalised Input-Output Construct of CSP. G N Buckley and A Silberschatz, ACM Transactions on Programming Languages and Systems Vol. 5, 2 (April 1983) p. 224.

[7] A Protocol for Generalised Occam, R Bornat, Department of Computer Science, Queen Mary College, London 1984.

[8] Occam 2 Reference Manual, INMOS Ltd, 1987

[9] The Transputer Implementation of occam, Technical note 21. INMOS Ltd

[10] Lets Design Algorithms for VLSI Systems, H T Kung, in: C A Mead and L A Conway: Introduction to VLSI Systems. Addison Wesley 1980 Section 8.3.