Introduction to the Programming Language Occam

By

Dr. Daniel C. Hyde
Department of Computer Science
Bucknell University
Lewisburg, PA 17837
hyde@bucknell.edu

Updated March 20, 1995

Table Of Contents

1 Introduction

Occam[1] is a parallel programming language developed in Great Britain. This Chapter describes the language, the circumstances surrounding its creation and its relationship to Inmos' Transputer. Occam is a good language for exploring the ideas of the message passing style of parallel programming. It has the important advantage that for a modest investment one can write truly parallel programs executing on an ensemble of Transputers.

1.1 The Inception of Occam

Occam is a parallel programming language developed by David May [May, 83] at Inmos Limited, Bristol, England. The language is one of several parallel programming languages based on Tony Hoare's CSP (Communicating Sequential Processes) [Hoare, 78]. A more careful treatment of CSP is in Hoare's book on CSP [Hoare, 85]. Using CSP as a basis, the researchers at Inmos developed an Occam concurrency model. From the Occam model, they developed the programming language Occam. The name is derived from William of Occam, a thirteenth century philosopher. Occam's Razor or the ancient philosophical principle of "keep things simple," is attributed to William. A primary goal of the Occam language is to keep the language simple, hence the name.

1.2 Occam and the Transputer

From the Occam model, Inmos developed a hardware chip to support their concurrency model. This hardware is in the form of a very large scale integration (VLSI) integrated chip (IC) called the Transputer [Walker, 85; Whitney-Strevens, 85]. The Transputer (Inmos part number T800) is a 32-bit microprocessor (20 MHz clock) that provides 10 MIPS (million instructions per second) and 2.0 MFLOPS (million floating point operations per second) processing power with 4K bytes of fast static RAM (Random Access Memory) and concurrent communication capability all on a single chip. Though Occam is a high-level language, it can be viewed as the assembly language for the Transputer. Unlike most microprocessors, e. g., the M68000, the definition of the operations of the Transputer is in terms of the Occam model and not machine language. Because the Transputer is designed to execute Occam, the compiler can generate very efficient and compact machine code. Besides being a high performance microprocessor (half the speed of a VAX 8600), the Transputer has on its chip four (4) serial bi-directional links (each 20 Megabits per second) to provide concurrent message passing to other Transputers. The "channels" in the Occam language are mapped to these hardware links which connect by way of twisted pairs of wires to other Transputers. The Transputer hardware supports concurrency by scheduling (time-slicing), in round-robin fashion, an arbitrary number of Occam concurrent processes. The language and the hardware are so designed that an Occam program consisting of a collection of concurrent processes may execute on one Transputer (via time slicing between the different concurrent processes) or be spread over many Transputers with little or no change in the Occam code. Therefore, the designer can develop his or her Occam program on one Transputer, and if higher performance is required, can spread the Occam processes over a network of interconnected Transputers.

---

[1] Occam and Transputer are registered trademarks of Inmos Limited.

The original Transputer (T414), having no floating point unit and only 2 Kbytes of RAM, became available in 1985. Two years later, the T800 Transputer was introduced and is used quite widely in many vendor products. Inmos is currently developing a new, faster version of the Transputer called the "T9000," which is scheduled to be available the middle of 1992. The T9000 will be a 150 MIPS microprocessor with a 20 MFLOPS floating point unit. The four links of the T800 will be replaced by more "virtual" links, with each link's speed at 100 Megabits per second. Memory on chip will be increased from 4 K bytes to 16 K bytes and will include memory mapping and memory protection.

1.3 Versions of Occam

The first version of Occam was distributed by Inmos to research laboratories and universities in 1983 [May, 83]. This version of Occam has become known as Occam 1 and was distributed in various forms. One form, the Portakit, was a FORTRAN source program which was ported to many machines but was very slow and contained errors in the Occam model. Another common Occam 1 version was the VAX VMS version, which Inmos distributed to universities for a modest cost of $100. This version corrected most of the errors of the Portakit.

From the experience gained in the three intervening years (1983-1986), David May and his group at Inmos have developed an enhanced version of Occam called Occam 2 [May, 86]. In Occam 2, the Occam model of concurrency has not changed; however, Inmos has added many features which we have come to expect in a modern high-level programming language, notably types and type checking (strong type checking as in Pascal). For the numerical programmer, important new features added were floating point arithmetic and multi-dimensional arrays. In 1988, David May added protocols on the Occam channels, user defined functions and "include" files to Occam 2 [Inmos, 88]. David May is currently working on Occam 3.

Inmos currently supports Occam on a variety of platforms including VAX VMS, IBM PC compatibles and SUN workstations. These all require a board (containing one or more Transputers) to be installed in a slot in the platform. Two development systems are available. The older one is called TDS (Transputer Development System), which has it own folding editor, compiler and linker all integrated into one bundle. The newer system is called the Inmos Toolset, which is an unbundled set of tools with the advantage that users are allowed to use tools under their familiar development environment, e. g., "make" in Unix.

This book will only discuss the latest version of Occam 2. Also, it uses the Inmos Toolset version of Occam in all of its examples.

2 The Occam Concurrency Model

The Occam model supports concurrency, i. e., true parallelism on several processors or simulated parallelism on one processor by way of time-slicing. The model is based on concurrent processes. For an excellent introduction into concurrent programming, see Ben-Ari's book entitled *Principles of Concurrent and Distributed Programming* [Ben-Ari, 1990].

In Occam, communication between concurrent processes is achieved by passing messages along point to point channels. Point to point means that the channel's source and destination must be at one point or reside in one concurrent process. Below, process P1 can send a message by way of channel C to process P2.

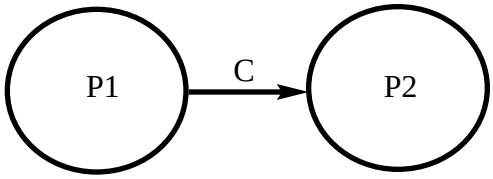

Fig. 1 Process P1 Sends a Message Along Channel C to Process P2

To alleviate many problems caused by interference when sharing variables between concurrent processes, all communication between concurrent processes in Occam must be by way of channels. Hence, there are no shared variables in Occam. (Actually, two concurrent processes can both read a variable, but one cannot read while the other writes into a shared variable.) Therefore, Occam reflects the message-passing model of parallel computation that is supported by the Transputer hardware with its local memory, i. e., no global memory, and communication links.

The communication on an Occam channel is synchronous. When either the sender or the receiver arrives at the proper place in the code, the first to arrive waits for the other. Once synchronized, the message is transferred between the two, then they continue executing. This action is similar to an Ada rendezvous. A consequence of this style of communication is that it provides no automatic buffering. If buffering is desired, an intermediate process may be inserted between the two processes. One advantage of this scheme is that one language feature performs both synchronization and message passing.

3 The Occam Language

This section is an overview of the Occam 2 language. For a more definitive statement of the language, see the Occam 2 Reference Manual [Inmos, 88].

3.1 Cosmetic Features

All reserved words must be in capital letters. Spaces are delimiters. Each construct must be indented two spaces to show structure. This indenting has the potential to cause grief for programmers, but any problems are alleviated by Inmos' good folding editor. Occam is line oriented, which means each statement starts on a new line, possibly indented. Continuation to the next line is possible by breaking an expression at an operator, semicolon or comma. Comments are designated by -- to the end of the line.

3.2 Program Structure

The structure of a program is a process with declarations preceding it.

```
<declares>
<process>
```

An example:

```
INT j:
SEQ
  j := 1
  j := j + 1
```

3.3 Occam Process

The Occam "process" can be considered a generalization of "statement" in other languages, e. g., Pascal. However, the Occam process may not fit your intuitive idea of process, e. g., an Occam process is different from a process as used in most operating systems texts. For example, two Occam processes need not be "concurrent processes." If the Occam concept of process confuses you at first, think of a process as an "action," i. e., something that is done.

3.4 Primitive Actions

In Occam, there are five primitive actions (called primitive processes in Occam after Hoare's CSP): assignment, receive, send, SKIP and STOP.

PRIMITIVE	SYNTAX	EXAMPLE
assignment	<variable> := <expression>	x := y + 1
receive	<channel> ? <variable>	Ch ? x
send	<channel> ! <expression>	Ch ! y + 1
SKIP	SKIP	SKIP
STOP	STOP	STOP

a). assignment - assigns a variable the value of an expression.

b). receive - receives value from a channel. Uses "?" to signify a query.

c). send - sends expression value on a channel. Uses "!" to signify an exclamation.

d). SKIP - do nothing and terminate the process, i. e., no operation.

e). STOP - do nothing and never terminate the process, i. e., never get to the next process.

You may wonder why "!" and "?" rather than, for example, SEND and RECEIVE. The notation is straight from Hoare's CSP.

In Occam expressions, there is *no* operator precedence! Therefore, you *must* use parentheses to specify the order of operation. For example:

```
x := 2 * y + 1
```

is illegal. You must use parentheses as in the following:

```
x := (2 * y) + 1
```

In sends and receives, one may use ";" to separate expressions or variables, as in :

```
ch ! x; y; x + y
```

3.5 Data Structures

Occam is strongly typed like Pascal and you must declare every variable. Declares are of the form:

<type> <one or more identifiers separated by commas> :

Types available include INT for integer, BOOL for Boolean, BYTE for character, REAL32 for 32-bit reals, REAL64 for 64-bit reals and CHAN for channels. For example, to declare the variables “x” and “y” to be integer type and “q” a channel with a message protocol of a single integer, we use the following:

```
INT x, y:
CHAN OF INT q:
```

The only data structures available are arrays. This shortcoming of Occam is rumored to be alleviated in the forthcoming Occam 3. Array bounds always start at zero as in the language C.
Multi-dimensioned arrays are available as in the following:

```
VAL n IS 100:
INT i, j:
[n][n] REAL32 a:
[n+1] CHAN OF ANY links:

SEQ
  SEQ i = 0 FOR n
    SEQ j = 0 FOR n
      a[i][j] := 0.0 (REAL32)
 -- other code
```

In the above code segment, we defined a constant n to be 100 and declared a 100 by 100 matrix “a” and a vector called “links” of 101 channels which allow ANY type of value to be passed along them. The last three lines initialize the whole array “a” to zero using nested replicated SEQs (See section 3.14 on replicators). Also, since the language has different types of real numbers, we had to explicitly state that 0.0 was a REAL32.

Declarations of identifiers can be done anywhere in the code, i. e., they do not have to be all at the beginning of a procedure as in Pascal. The scope of an identifier is from where it is declared to and including the body of the process where it was declared unless redeclared inside. In the above, the scope of “n” includes the declares after and the body of the outer SEQ.

Since a major design goal of Occam was secure concurrent programs, the language does not allow pointers.

3.6 Constructors

Constructors group processes into a process, i. e., they allow a hierarchical nesting of processes like the BEGIN-END nesting of statements in Pascal. All constructors may group N processes (except WHILE).

3.7 SEQ

If several processes are to be executed in sequential order, the SEQ constructor is used. For example,

```
SEQ
  a := 3
  b := a + 5
  c := a - 5
```

performs three primitive processes in sequential order. To show the nested structure of the constructs, the code is indented two spaces.

```
SEQ      -- Processes p1 and p2 are done sequentially.
  p1
  p2
```

The two dashes together indicate the start of a comment on the end of a line.

3.8 PAR

If several processes are to be performed concurrently (true parallelism or simulated parallel by time-slicing), the PAR construct is used. Below are two primitive processes in parallel.

```
PAR
  INT x:
  ch1 ? x       -- receive from channel ch1
  INT y:
  ch2 ? y       -- receive from channel ch2
```

In general, any number of processes - primitive or constructors - can be executed in parallel. The whole construct terminates when all of the PAR's components terminate.

```
PAR    -- p1, p2 and p3 are conceptually performed in parallel.
  p1   -- The whole construct terminates when all
  p2   -- three p1, p2 and p3, terminate.
  p3
```

Below is an example using a parallel constructor and channel communication. The two WHILE loops are performed in parallel with the top process communicating with the bottom process through the channel "comms".

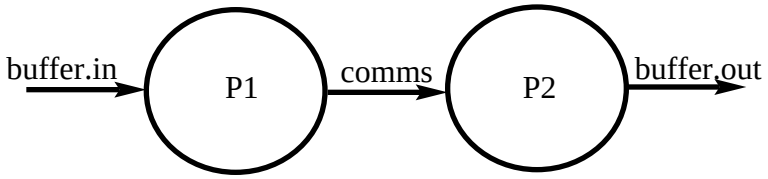

Fig. 2 Two buffer Processes

```
CHAN OF BYTE comms, buffer.in, buffer.out:
PAR
  WHILE TRUE
    BYTE x:
    SEQ
      buffer.in ? x
      comms ! x
    WHILE TRUE
      BYTE y:
      SEQ
        comms ? y
        buffer.out ! y
```

3.9 IF

To select a process depending on a Boolean expression, Occam has an IF constructor. Where Pascal has a binary choice, the Occam IF is a multi-way selector.

```
IF                -- standard IF with N way choice.  The pi is selected
  <Boolean exp1>  -- for the first Boolean expression true.
    p1
  <Boolean exp2>
    p2
  <Boolean exp3>
    p3
```

The process is selected in which the first Boolean expression is evaluated true. If no Boolean expression is true, then the IF construct acts like a STOP (an abort).

```
IF
  a > b
    c := 3
  a < b
    c := 4
  TRUE
    SKIP
```

Here, the Boolean expression TRUE will catch the case where “a” equals “b” and acts like an “otherwise” in other languages. Programmers should always include a TRUE as the last case.

3.10 WHILE

To repeat a process, Occam has the WHILE construct.

```
WHILE <Boolean Exp> -- Standard WHILE loop.
      p
```

For example:

```
WHILE i < 10
  i := i + 1
```

3.11 ALT

Many times one needs to select a process (an ALTernative) depending on a condition which is more than a Boolean expression (as in an IF). These conditions are called guards (from Dijkstra’s guards). Guards in Occam can depend on receive channels being ready, timers, or combinations of both with Boolean expressions.

Starting at the top guard and evaluating each guard, the ALT selects one and only one process, depending on the first guard to be satisfied. If no guards are satisfied, then the ALT waits for a guard to be satisfied.

```
ALT
  ch1 ? x
    A[1] := x
  ch2 ? x
    A[2] := x
  time ? AFTER begin.time + (10 * sec)
    SKIP
```

In the above ALT, if the send for ch1, or ch2 is ready, the receive guard will be performed and the assignment statement after it. If no send is ready after 10 seconds, the timer guard becomes satisfied and the SKIP is selected.

```
ALT
  Flag1 & ch1 ? x
    a := 3
  (Flag2 OR Flag3) & Ch2 ? y
    a := 4
  SKIP
    a := -1
```

In this ALT example, Boolean flags are used to control the receives. If the first two guards are not satisfied, the SKIP guard immediately is satisfied and “a” is assigned -1.

In general, exactly one p_i is selected to be performed, depending on the guards. If more than one guard is true, the Occam manual says one is selected, i. e., non-determinism is introduced.

```
ALT         -- Exactly one pi is selected for the satisfied
  <guard1>  -- guard. If more than one guard is satisfied,
    p1      -- then one is selected arbitrarily.  If no
  <guard2>  -- guard is satisfied, then ALT waits until a
    p2      -- guard is satisfied.
  <guard3>
    p3
      . . .
  <guardn>
    pn
```

The forms of guards are:

```
<receive>
<Boolean exp> & <receive>
SKIP
<Boolean exp> & SKIP
```

where a receive may be a timer.

A variation of the ALT is the PRI ALT which gives priority to the first guards. In all current implementations of Occam there is no difference between the ALT and PRI ALT: this means that the alternatives are not treated fairly, for the top guards are checked first. This can lead to starvation of the guards near the bottom if the top guards are very greedy and occur often.

3.12 Simple Occam Program

A simple program to compute the square root of the numbers from 1 to 10 is shown. Note that the indentation shows the structure and that all keywords must be capitalized. Also, note that the library routines (start with “so.”) use the “SP” protocol on the channels “keyboard” and “screen”.

```
#INCLUDE "hostio.inc" -- contains SP protocol
PROC Sqrt.program(CHAN OF SP keyboard, screenn)
  -- Occam 2 program to compute square roots of numbers 1 thru 10
  -- Dan Hyde, Jan 5, 1991

  #USE "hostio.lib"  -- I/O library - always include
```

```
  BYTE key, result:
  REAL32 A:
  SEQ
    so.write.string.nl(keyboard, screen, "Value   Square Root")

    SEQ i = 1 FOR 10
      SEQ
        so.write.string(keyboard, screen, "i = ")
        so.write.int(keyboard, screen, i, 2)
        A := REAL32 ROUND i          -- type conversion from
                                     -- integer to real
        so.write.real32(keyboard, screen, SQRT(A), 4, 6)
        so.write.nl(keyboard, screen)

    so.exit(keyboard, screen, sps.success)
:
```

3.13 PROCs

A PROC names one process and allows parameters to be passed by-value or by-reference. It is similar to a SUBROUTINE in FORTRAN, i. e., only static allocation of variables and no recursive activations allowed.

Using PROCs, the buffer example of section 3.8 is redone. Note the passing of channels in this example.

```
-- other parts to program
PROC buff(CHAN OF BYTE in, out)
  WHILE TRUE
    BYTE x:
    SEQ
      in ? x
      out ! x
:   -- end of buff
CHAN OF BYTE comms, buffer.in, buffer.out:
PAR
  buff(buffer.in, comms)
  buff(comms, buffer.out)
```

A by-value parameter uses the keyword VAL in the PROC heading.

3.14 Replicators

All the constructors, except WHILE, can replicate the construct to combine nearly identical ones. The replicator variable (“i” in the example below) may be substituted in the process.

For example, n parallel processes use a vector of n channels to send the elements of vector A conceptually in parallel. In this case, n must be a constant.

```
VAL n IS 10:  -- number of processes
[n] CHAN OF REAL32 Ch:
PAR i = 0 FOR n
  Ch[i] ! A[i]
```

Below is an example with a replicated IF. Here, we are writing our own input-an-integer routine instead of using the library routine so.read.echo.int.

```
PROC Input.integer(INT Out)
  VAL Cr IS '*c':
  VAL [10] BYTE Digits IS ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:
  INT Sum, Sign:
  BYTE Ch:
  BYTE result:
  SEQ
    Sum := 0
    Sign := 1
    so.write.char(keyboard, screen, '?')    -- print a prompt
    so.getkey(keyboard, screen, Ch, result)
    so.write.char(keyboard, screen, Ch)     -- echo character
    IF
      Ch = '-'
        SEQ
          Sign := -1
          so.getkey(keyboard, screen, Ch, result)
          so.write.char(keyboard, screen, Ch)  -- echo character
      TRUE
        SKIP
    WHILE Ch <> Cr         -- read digits until carriage return
      SEQ
        IF
          IF i = 0 FOR 10  -- replicated IF; acts like 10 IF statements
            Ch = Digits[i] -- compare to each digit
              Sum := (10 * Sum) + i
          TRUE           -- must be an error
            so.write.string(keyboard, screen, "Bad integer")
        so.getkey(keyboard, screen, Ch, result)
        so.write.char(keyboard, screen, Ch)  -- echo character
    Out := Sign * Sum
:  -- end of Input.integer
```

3.15 Time

A programmer can declare TIMER variables to utilize a real time clock. This special variable can be used in ALT guards as in the following:

```
  VAL one.sec IS 15625: -- number of ticks in second on Transputer
  INT begin.time:
  TIMER time:
  SEQ
    -- some code
    time ? begin.time
    PRI ALT -- wait for a second for send on channel Ch then go on
      Ch ? y
        p1          -- some process
      time ? AFTER begin.time + one.sec
        SKIP
```

4 A Methodology for Developing Occam Programs

We present a common approach to developing Occam programs. The three steps are the following:

Step 1: *Draw boxes of concurrent processes and lines for channels of communication. Design the general overall approach.*

Step 2: *Write a PROC for each box. Design boxes that will be replicated and be independent of position in the diagram.*

Step 3: *Devise a "harness" of PARs, replicated PARs, and vector of channels for the parallelism and lines of communication.*

5 Extended Example Demonstrating the Methodology

To demonstrate the methodology, we present an example of a four stage pipeline to find square roots using a Newton Raphson approximation. For the square root of *x*, we will use the following:

$$estimate_{i+1} = estimate_{i} + \frac{x}{2 \cdot estimate_{i}}$$

The overall approach is to pass *x* and the first approximation *x*/2 into an iteration box which passes *x* and its *estimate* on to another iteration box as shown.

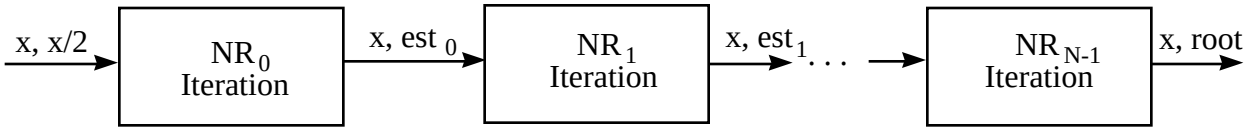

Fig. 3 Pipeline for Newton Raphson Approximation of Square Root

After *N* iterations, we receive the answer in *root*.

We will now develop the Occam program using the three steps stated above.

Step 1: Draw boxes of concurrent processes and lines for channels of communication. Design the general overall approach.

We designate an Occam concurrent process for each iteration box. We will need a concurrent process at each end of the pipeline -- one to read the keyboard and another to write to the screen. A vector of channels called "Links" will connect the pipeline together. For this example, *N*, the number of iterations, is 4 or four stages in the pipeline.

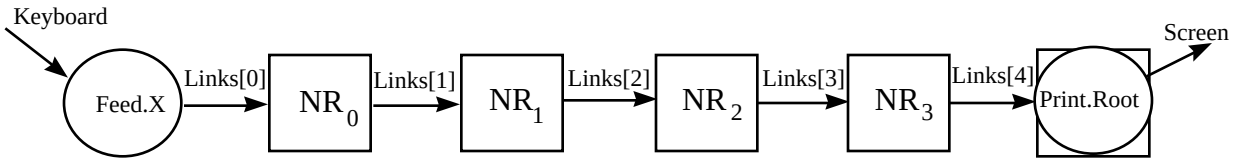

Fig. 4 Process Diagram for Pipeline Program

Step 2: *Write a PROC for each box. Design boxes that will be replicated and be independent of position in the diagram.*

In this case, we need to write three PROCs - one for each shape of box.

```
PROC Feed.X(CHAN OF ANY Out)
  -- read in real numbers and feed X and X/2 to
```

```
  -- input of pipeline
  REAL32 X:
  BOOL error:
  WHILE TRUE
    SEQ
      so.write.string.nl(keyboard, screen, "Type in a value for X")
      so.read.echo.real32(keyboard, screen, X, error)
      so.write.nl(keyboard, screen)
      Out ! X
      Out ! X / 2.0 (REAL32)
:  -- end of Feed.X
```

```
  PROC Print.Root(CHAN OF ANY In)
    -- receive X and Root and print them
    REAL32 X, Root:
    WHILE TRUE
      SEQ
        In ? X
        In ? Root
        so.write.string(keyboard, screen, "Square root of ")
        so.write.real32(keyboard, screen, X, 4, 6)
        so.write.string(keyboard, screen, "is ")
        so.write.real32(keyboard, screen, Root, 4, 6)
        so.write.nl(keyboard, screen)
  :  -- end of Print.Root

  PROC NR(CHAN OF ANY In, Out)
    -- read in X and current Estimate, compute next iteration and ship out
    REAL32 X, Estimate:
    WHILE TRUE
      SEQ
        In ? X
        In ? Estimate
        Out ! X
        Out ! Estimate + (X / (2.0 (REAL32) * Estimate))
  : -- end of NR
```

Step 3: *Devise a "harness" of PARs, replicated PARs, and vector of channels for the parallelism and lines of communication.*

Below is the whole Occam2 program. Each fold contains the Occam2 code for that PROC. The harness consists of three processes in a PAR where one is a replicated PAR. The PROC NR is replicated N times with the appropriate indices of the vector of channels.

```
#INCLUDE "hostio.inc" -- contains SP protocol
PROC Newton2(CHAN OF SP keyboard, screen)
  -- Pipeline example based on "Occam Programming Manual", Inmos,
  --                            Prentice Hall, 1984, section 2.8.
  -- Computes square root of number by pipeline
  -- of Newton Raphson iterations
  -- Programmed by Dan Hyde, Bucknell University, June, 1987
  -- updated Jan 5, 1991

  #USE "hostio.lib"
  -- insert code for PROC Feed.X
  -- insert code for PROC Print.Root
  -- insert code for PROC NR
  VAL N IS 4:             -- number of iterations in pipeline
  [N + 1] CHAN OF ANY Links:
  PAR                     -- Harness
    Feed.X(Links[0])
    PAR i = 0 FOR N
      NR(Links[i], Links[i + 1])
    Print.Root(Links[N])
: -- end of Main
```

Even though the above program looks fine, it does not work! Why?

The two concurrent processes Feed.X and Print.Root violate channel usage rules in Occam. Both try to output to the screen (as well as input from the keyboard). Therefore, the channel "screen" is not point to point (see section 2). When we try to compile this,

the compiler detects the problem and writes the error message “Usage error, parallel inputs on ‘keyboard.’” To correct this, the two processes Feed.X and Print.Root were merged into one called the “Controller” as shown below.

```
PROC Controller(CHAN OF ANY Out, In)
  -- read in real numbers and feed X and X/2 to input of pipeline
  -- receive X and current estimate and print them
  REAL32 X, Root:
  BOOL error:
  WHILE TRUE
    SEQ
      so.write.string.nl(keyboard, screen,"Type in a value for X")
      so.read.echo.real32(keyboard, screen, X, error)
      so.write.nl(keyboard, screen)
      Out ! X
      Out ! X / 2.0 (REAL32)
                          -- wait to compute the square root
      In ? X
      In ? Root
      so.write.string(keyboard, screen, "Square root of ")
      so.write.real32(keyboard, screen, X, 4, 6)
      so.write.string(keyboard, screen, "is ")
      so.write.real32(keyboard, screen, Root, 4, 6)
      so.write.nl(keyboard, screen)
:  -- end of Controller
```

Here is the complete corrected program.

```
#INCLUDE "hostio.inc" -- contains SP protocol
PROC Newton2(CHAN OF SP keyboard, screen)
  -- Pipeline example based on "Occam Programming Manual", Inmos,
  --                           Prentice Hall, 1984, section 2.8.
  -- Computes square root of number by pipeline
  -- of Newton Raphson iterations
  -- Programmed by Dan Hyde, Bucknell University, June, 1987
  -- updated Jan 5, 1991

  #USE "hostio.lib"

  PROC Controller(CHAN OF ANY Out, In)
    -- read in real numbers and feed X and X/2 to input of pipeline
    -- receive X and current estimate and print them
    REAL32 X, Root:
    BOOL error:
    WHILE TRUE
      SEQ
        so.write.string.nl(keyboard, screen, "Type in a value for X")
        so.read.echo.real32(keyboard, screen, X, error)
        so.write.nl(keyboard, screen)
        Out ! X
        Out ! X / 2.0 (REAL32)
                              -- wait to compute the square root
        In ? X
        In ? Root
        so.write.string(keyboard, screen, "Square root of ")
        so.write.real32(keyboard, screen, X, 4, 6)
        so.write.string(keyboard, screen, "is ")
        so.write.real32(keyboard, screen, Root, 4, 6)
        so.write.nl(keyboard, screen)
  :  -- end of Controller

  PROC NR(CHAN OF ANY In, Out)
    -- read in X and current Estimate, compute next iteration and ship out
    REAL32 X, Estimate:
    WHILE TRUE
      SEQ
        In ? X
        In ? Estimate
        Out ! X
        Out ! Estimate + (X / (2.0 (REAL32) * Estimate))
  : -- end of NR

  VAL N IS 4:             -- number of iterations in pipeline
  [N + 1] CHAN OF ANY Links:

  PAR                     -- Harness
    Controller(Links[0], Links[N])
    PAR i = 0 FOR N
      NR(Links[i], Links[i + 1])
: -- end of Main
```

A second possible way to correct the program would be to design a new concurrent process which acted as a screen manager, i. e., a de-multiplexer of several new channels from Feed.X and Print.Root. This solution is not shown here.
Below is a run with $N = 4$, i. e., with four NR processes in the pipeline.

```
Type in a value for X? 4.0
Square root of 4.0 is 2.0
Type in a value for X? 10.0
```

```
Square root of 10.0 is 3.162278
Type in a value for X? 100.0
Square root of 100.0 is 10.030495
Type in a value for X? 1000.0
Square root of 1000.0 is 41.22295
Type in a value for X? 10000.0
Square root of 1.0 E+04 is 323.05426
Type in a value for X? 100000.0
Square root of 1.0 E+05 is 3135.61792
Type in a value for X? 1000000.0
Square root of 1.0 E+06 is 3.12606E+04
```

Notice as *N* becomes large, the four stages (iterations) do not give a very good estimate of the square root. For example, the square root of 10000 should be 100 not 323.05426.

A second run with ten NR processes or with $N = 10$.
(Only the line "VAL N IS 4:" was changed.)

```
Type in a value for X? 4.0
Square root of 4.0 is 2.0
Type in a value for X? 10.0
Square root of 10.0 is 3.162278
Type in a value for X? 100.0
Square root of 100.0 is 10.0
Type in a value for X? 1000.0
Square root of 1000.0 is 31.622776
Type in a value for X? 10000.0
Square root of 1.0 E+04 is 100.0
Type in a value for X? 100000.0
Square root of 1.0 E+05 is 316.229248
Type in a value for X? 1000000.0
Square root of 1.0 E+06 is 1033.84106
```

The above pipe example is actually a silly way to compute the square root of a number. On one processor with no parallelism, the program is slow compared to the sequential algorithm because of time spent in process scheduling and in communication. A speaker at an Occam conference presented the following timing analysis of the pipeline example in the Occam Programming Manual [Inmos, 1988]:

Time for pipe Newton-Raphson on one Transputer was 170 microseconds.

Time for sequential Newton-Raphson on one Transputer was 60 microseconds.

Time for pipe Newton-Raphson on 12 Transputers was 30 microseconds.

He observed, "We need to think about our designs carefully."

6 Conclusion

The Occam programming language is a viable language to express concurrency. It is especially valuable for teaching the concepts of parallel algorithms in the message passing paradigm.

One of Occam's redeeming aspects is its simplicity. One only needs to compare it to other programming languages that support concurrency, e. g., Ada, to be convinced of Occam's simplicity. The Occam concurrency model makes it easy to think and reason about an Occam program as a network of concurrent processes which only passes messages, i. e., the programmer does not have to deal with shared variables and their associated problems.

In Occam, one expresses concurrency explicitly at the statement level, i. e., the PAR construct, while in other languages, e. g., Ada, concurrency can only be expressed implicitly at the procedural level. In a language like Ada, the "heavy" machinery to support concurrency tends to discourage the programmer from using concurrency. With Occam, the PAR is a natural and easy construct to use.

Another important goal of Occam is security. Security is not easy to achieve in a concurrent system, however, Occam makes it a lot easier than most concurrent languages. In order to support this security, favorite language features, e. g., pointers, dynamic memory allocation, dynamic process allocation and recursive functions, are missing from Occam. While a concurrent language like Logical Systems C may be more flexible and support all of these wonderful features, concurrent C programs are very hard to debug. Where sequential C gives the programmer the power and flexibility to "shoot oneself in the foot," concurrent C compounds the problems many fold. The security that Occam brings to programming concurrent program is especially important in a student environment where they are learning the concepts of concurrency while mastering the language.

References

Dijkstra, E. W. , "Guarded Commands, Nondeterminacy and Formal Derivations of Programs," Communications of the ACM, Vol. 18, 1975, pp. 453-7.

Hoare, C. A. R., "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, 1978, pp. 666-677.

Hoare, C. A. R., *Communicating Sequential Processes*, Prentice Hall, 1985.

Inmos Limited, *Occam 2 Reference Manual*, Prentice Hall, 1988.

Jones, Geraint and Michael Goldsmith, *Programming in Occam 2*, Prentice Hall, 1988.

May, David, "Occam," *SIGPLAN Notices*, Vol. 18, No. 4, April, 1983, pp. 69-79.

May, David, *Occam2 Product Definition Manual*, Inmos, June, 1986.

May, David and Richard Taylor, "Occam: An Overview," *Microprocessors and Microsystems*, Vol. 8, No. 2, March, 1984, pp. 73-79.

Walker, Paul, "The Transputer," *BYTE*, May, 1985.

Whitney-Strevens, Colin, "The Transputer," *SIGARCH Newsletter*, Vol. 13, No. 3, June, 1985, pp. 292-300.