PROGRAMMING LANGUAGES

OCCAM

SUBMITTED BY-

Atishaya Jain
MIT-26
Hitesh Gupta
MIT-11
Introduction to OCCAM

The multi-processing language occam provides a strong formal basis for the secure and efficient development of high-performance parallel applications. Industrial experience over the past eight years (limited to transputer-based hardware) has amply demonstrated its benefits for the engineering of scalable computing applications across a wide spectrum of problem area. Methodologies and supporting tools for specific areas (e.g. modelling, real-time control, databases, hardware design...) have been developed and remain the subject of vigorous and high-quality research (chiefly UK led).

This is a proposal to build a set of architecturally neutral tools, based on occam, that support the design, implementation, optimisation and maintenance of high-performance applications. Systems constructed with these tools will be portable across all current and currently foreseen (MIMD) parallel computing platforms, where they will operate with high levels of efficiency.

This programme is partly motivated by fears shared by all partners in the consortium - particularly the industrial partners - that the engineering and commercial benefits previously enjoyed through the use of occam on transputer-based platforms will be denied to them in the future unless the language is 'opened' to the wider parallel computing community. The positive motivation arises from occam being the only parallel-processing language in widespread industrial and academic use whose semantics is sufficiently well-defined, simple and powerful to play this unifying role.

Language shapes our thoughts - it is our fundamental tool for formulating ideas, communicating them and applying them to our advantage. Language must evolve to capture higher levels of abstraction as they are discovered and found to be useful - but language must remain as simple and precise as possible lest it contributes more to our problems then our solutions.

occam was designed in the early 1980s to meet the challenge of parallel computing. It is built upon important theoretical insights gained during the previous decade (primarily those of CSP and CCS), whose significance has been confirmed by the subsequent decade of practical application. Occam provides and enforces a rich set of abstractions for parallel processes, synchronization and communication. Its compilers incorporate sophisticated analysis tools to ensure conformity with those abstractions. These tools are only made possible through the clean semantics of occam and have no equivalents for parallel extensions to standard imperative languages (such as C and FORTRAN), even though those extensions may target similar paradigms to those of occam. The fact that occam is an astonishingly small language is a significant help in creating these tools.

A key principle is the abbreviation mechanism of occam, that enables tight control over the aliasing of names to objects. occam uses this control to eliminate the aliasing of updatable objects, thereby greatly simplifying the semantics of assignment, bringing this
into line with our engineering intuition. This yields immediate benefits for the safe and efficient implementation of parameter passing, but the major reward lies in the enabling of data and channel usage checking for parallel programs. These latter guarantees that non-determinism is not introduced simply through the use of parallelism. It results in system properties that are stable against the vagaries of the ordering in which processes are scheduled and against the number and performance characteristics of the processors on which processes happen to be allocated. When seeking tools to assist the production of applications that are portable across parallel architectures, this uniquely qualifies Occam as the (imperative) language on which they should be based.

Occam offers a range of other tools for parallel, distributed, real-time and safety-critical computing: explicit control over the level of non-determinism; dynamically sized and dynamically structured message-passing that is checked at compile-time (note: most other systems support messages only as arrays of bytes whose structure is open to mis-construction by the sender and mis-interpretation by the receiver); a high-level view of time, time-outs, interrupts and priority scheduling; really lightweight processes whose scheduling can be managed with a context-switch overhead of less than 10 micro-seconds on most modern processors (and less than 1 micro-second on a transputer); a formal denotational semantics, based on the (timed) traces-failures-divergences model of CSP.

At present, these tools are restricted for use only on transputer-based hardware. Our first group of work packages builds on preliminary efforts by the academic partners and others to open these tools for other processors.

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.

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. Occam and Transputer are registered trademarks of Inmos Limited.

**Introduction to the Programming Language Occam**

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 in 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 its 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. Process P1 can send a message by way of channel C to process P2. 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.

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. 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
6 Introduction to the Programming Language Occam
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:

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

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

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

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

```plaintext
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> :
```

Introduction to the Programming Language Occam 7

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:

```plaintext
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:

```plaintext
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,

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

8 Introduction to the Programming Language Occam 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.

```plaintext
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”.
P1 P2 comms buffer.in buffer.out
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
Introduction to the Programming Language Occam 9
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
time ? AFTER begin.time + (10 * sec)
SKIP

10 Introduction to the Programming Language Occam
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 pi 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>
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”.

```occam
#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
    Introduction to the Programming Language Occam 11
    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.

12 Introduction to the Programming Language Occam
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

The strategic importance of Occam

The parallel concepts of occam are architecturally neutral of the platforms on which its programs are executed. They can be implemented with high levels of efficiency both on distributed and (virtual) shared memory MIMD systems - or, for ultimate performance, on custom/customisable silicon [10]. The formal semantics of occam [5] offer the prospect of ‘closing the gap’ between software and hardware systems [7] and of forming the backbone of a suite of formal tools that will integrate the development and verification of systems from high-level specification down to transistors - taking in general-purpose parallel computing on the way.

occam has no rival technologies competing for this role. Its loss as a widely-used and commercially supported tool would therefore be seriously damaging to our industrial and academic futures. Unless occam is opened up and made efficient for all architectures, this loss is very likely. Our Work Packages I-IV address this ‘opening up’. They involve a substantial amount of work that can no longer be funded on the backs of other programmes. They specifically address the fundamental aims of this PSTPA programme.
Neither INMOS nor any commercial software vendor (that we have contacted) is in a position to allocate resources to such work as a purely internal project.

**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.