Towards a Portable occam

submitted in partial fulfilment of the requirements for the degree

MASTER OF SCIENCE
of Rhodes University

by

David Timothy Hill

January 1988

Abstract

Occam is designed for concurrent programming on a network of transputers. Allocation and partitioning of the program is specified within the source code, binding the program to a specific network. An alternative approach is proposed which completely separates the source code from hardware considerations. Static allocation is performed as a separate phase and should, ideally, be automatic but at present is manual. Complete hardware abstraction requires that non-local, shared communication be provided for, introducing an efficiency overhead which can be minimised by the allocation. The proposal was implemented on a network of IBM PCs, modelled on a transputer network, and implementation issues are discussed.

Acknowledgements

I am indebted to Peter Clayton, my supervisor, for his guidance, input and encouragement. My thanks to the Rhodes University Department of Computer Science in general, for providing a pleasant, cheerful working environment.

I am grateful to the CSIR, Sasol and Rhodes University for their financial support in this project.

Finally, to Raewyn for her support, financial and otherwise, and for her tolerance.

CONTENTS

1 Introduction

Concurrent programming has become an area of prolific research over the last twenty years. New programming languages have been developed for the expression of concurrent algorithms, of which Modula-2 [WIR82], Ada [ADA84] and occam [MAY83] are currently prevalent. Concurrent programming is no longer solely the domain of the implementors of operating systems but has widespread use in both real-time and non real-time applications, including process control, digital image processing, scientific simulations/ calculations and database management systems. The reason for this has been the availability of cheap processors, making multiprocessor systems feasible, and the realization that concurrent algorithms often provide a more naturally expressed solution to many problems.

The memory of earlier multiprocessor systems was shared by the processors in the system, and languages such as Concurrent Pascal [BRI75] and Modula [WIR77] were designed with this type of target hardware in mind. The bottleneck, created by sharing either memory or a communications bus between processors, places an upper limit on the number of processors such a system can contain. Thus current multiprocessor systems have tended towards *distributed* processors, having local memory only and communicating with one another via I/O channels. CSP [HOA78] and DP [BRI78] reflect this type of multiprocessor architecture as do languages influenced by them, such as occam and *MOD [COO80]. Ada, also influenced by CSP and DP, does not completely reflect a distributed architecture and contains certain features which rely on a single shared memory. Stammers [STA85] identifies these features and suggests ways in which they could be modified so as to support implementation on distributed hardware.

Although concurrent languages should ideally be implemented on a multiprocessor system, most current implementations of these languages are on a single processor where the concurrent tasks execute pseudo-concurrently by sharing the processor via some time-slicing mechanism. Single processor implementations do not address the important issue of *allocation* (ie. which concurrent tasks execute on which processors). In *non real-time* applications, the selection of a particular allocation affects only the efficiency of the program execution, not its correct execution, since allocation does not affect the logical behaviour of the program. In *real-time* applications, however, timing constraints might not be met by a certain allocation, and hence the program will not execute correctly for that allocation.

Designers of concurrent languages can either provide facilities within the language which require the programmer to specify allocation in the source code, as in *MOD and occam, or

else can leave allocation to the implementation, as is the case with Ada and Modula-2. Leaving allocation to the implementation, or to some standard run-time support environment, allows for program portability but significantly complicates implementation. In Ada, implementation on distributed processors is further complicated by weaknesses in the language design, already referred to. As a result, no implementation yet exists which automatically distributes an Ada program amoung processors with local memory only. Bishop *et al* [BIS87] suggest an interim approach to distributing an Ada program which involves the user supplying process to processor mappings. In adapting and implementing Modula-2 for distributed systems, Mellor *et al* [MEL86] use both automatic and interactive (user supplied) techniques to allocate Modula-2 modules to processors.

By placing allocation completely in the hands of the programmer, the designers of occam allow the language to be easily implemented on a distributed system, but also create some problems. Specifying allocation in the source code results in a loss of portability since the source must be modified if it is to be executed on a transputer network of differing topology. The programmer is not abstracted from hardware considerations and is restricted in the use of the language. These restrictions are described in section 2 of this report. Specifying allocation in the source code becomes prohibitively tedious for programming a system of many (100) processors. Occam has been criticized for this and Crookes *et al* [CRO87] have developed the language *Latin* in an attempt to overcome these difficulties, encountered when using a transputer network as an array processor programmed in occam.

The ability of the programmer to specify allocation is often advantageous for real-time programming, especially where *hard* real-time constraints must be met:

- Communication times can be calculated exactly and guaranteed at the design stage since the communication paths are known.

- Response times can be guaranteed since the number of tasks sharing a particular processor is determined by the programmer and no run-time allocation overheads need be considered.

Since hard real-time programs are, by their very nature, machine specific, the loss of program portability is not a serious consideration for these applications.

For programmers not concerned with stringent timing details, but simply with the efficient execution of their programs, the disadvantages of specifying allocation in the source code,

especially via the cumbersome PLACED PAR construct of occam, become serious. This is especially true since many applications of transputers do not require that the transputer network become a special-purpose machine for solving a particular problem as determined by the occam program, but rather that it be a general purpose machine. Examples of these general purpose, low cost, high performance machines are the Meiko Computing Surface [BOT86] and the work of the ESPRIT transputer array project [BIS87]. The developers of the Meiko Computing Surface are reported [POU86,POU87] to be experimenting with a *dynamic* allocation mechanism where each transputer executes a small occam process which either accepts data sent to it for processing, or passes it on to the next transputer.

There is clearly a need to allow the mapping of occam processes to transputers to be done independently of the occam source code. This report proposes facilities to enable this and presents a partial implementation. In brief, the idea is to strive towards *portable* occam by separating the source code from all hardware-specific aspects. The allocation of occam processes to transputers is done as a separate phase which should, ideally, be automatic. In order to avoid the unnecessary recompilation of the source code each time the occam program is to be reallocated, it was decided to separate the compilation and allocation into different stages. Mellor *et al* adopt a similar approach [MEL86]. The allocator decides the allocation according to a map of the transputer network, supplied to it by the user, and run-time statistics of the behaviour of the program gathered from previous executions, as depicted in Figure 1. The proposal was partially implemented on a network of IBM PCs, each containing a four-port serial card and thus being functionally equivalent to a transputer [HIL86].

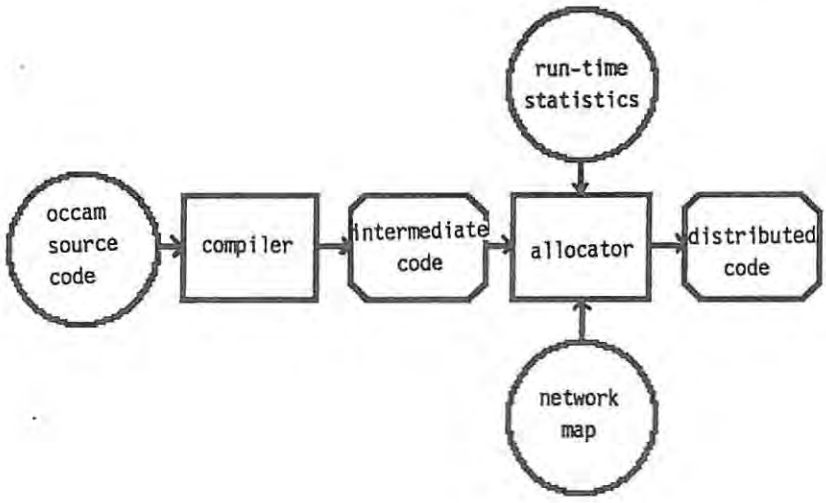


Figure 1 The Proposal

It is assumed that the reader is familiar with both occam and the transputer, and the unfamiliar reader is referred to the ample literature covering these topics [JON85, MAY84, MAY85, NEW86, TRA85].

2 Problems with the Placed Par Construct

This section criticizes the current allocation scheme of occam, provided by the PLACED PAR construct. The affect of this construct on program design and allocation is discussed and the implications of the resultant loss of portability are considered.

The configuration (allocation) construct of occam is always introduced to prospective occam programmers as an afterthought, tucked away in the last pages of the manual or textbook. Indeed, it is suggested [MAT87] that the construct be used as an afterthought, and that the program be developed without regard for the target hardware until the final configuration stage. Unfortunately the configuration construct imposes some far reaching constraints on program design, and cannot be ignored until the end.

The most far reaching of these constraints is that the programmer cannot place *any* concurrent occam process on *any* arbitrary transputer. The novice occam programmer might easily gain the impression that any component process of a PAR construct can, given a suitable transputer network, execute on its own dedicated transputer (ie. not pseudo-concurrently). Unfortunately this is not the case since not all PAR constructs can be replaced with a PLACED PAR construct. The PLACED PAR construct must be the first (outermost) construct in the occam program and its component processes are separately compiled processes which may have only channels and constants declared global to them. The PAR construct, on the other hand, may be used wherever a process is allowed and there are no restrictions on the type of declarations which may be global to a PAR construct. The only restrictions are the so called non-interference conditions, made explicit in occam2 [MAY87a], which prohibit more than one concurrent process from modifying a global variable. Thus a PAR construct may be replaced by a PLACED PAR construct only if the PAR construct is the outermost construct and its component processes (hence referred to as daughters) do not have variables or timers declared global to them.

To illustrate this point, consider the following occam program:

(i)
```
VAR offset:
SEQ
  offset := 1
  VAR x, y:
    PAR
      x := offset + 3
      y := offset + 4
```

The above program can only be configured to run on a single transputer, since the PAR

construct cannot be replaced by a PLACED PAR construct. However, if the program is rewritten as

(ii)
```
CHAN start:
PAR
  VAR offset, x:
  SEQ
    offset := 1
    START ! offset
    x := offset + 3
  VAR offset, y:
  SEQ
    START ? offset
    y := offset + 4
```

then the PAR construct can be replaced by a PLACED PAR construct and the two daughters can be made into separately compiled processes. Hence the program has the potential to be configured for a transputer network of two transputers. If an occam programmer proceeds with program development without keeping in mind the restrictions of the PLACED PAR construct, the final program might not be configurable or at least be seriously restricted. The programmer may be forced to write code in the form of program (ii) above, even though program (i) is a far more lucid expression of the algorithm.

It is perhaps possible to perform automatic source translation of occam programs, so that all daughter processes in the program are made *potential* separately compiled processes, thus allowing for the maximum degree of truly concurrent execution of the program. The large number of channels required to synchronize the execution of the daughters will, however, present problems due to the limited number of links available per transputer. This is another aspect of the PLACED PAR construct which limits the design of occam programs.

The mapping of occam channels to transputer links is done via the *port allocation* clause of the PLACED PAR construct. Only two occam channels, providing they communicate in opposite directions, may be mapped to a single transputer link. Thus two separately compiled processes, which are mapped to separate transputers, may only share as many channels as can be mapped to the links connecting the two transputers directly. The mapping remains static throughout program execution and has been identified as a problem [FAY87]. Fay and Das [FAY87] suggest either dynamic reconfiguration of transputer link hardware or else a network layer, implemented on the transputers so as to allow for differing logical and physical connections.

At present, however, the designer of an occam program must minimize the number of

channels between potential separately compiled processes so as not to fall foul of the limited number of hardware links available, even though the algorithm may be more clearly expressed by using more channels. A highly unsatisfactory alternative is to create an occam process on each transputer which multiplexes many channels onto a single channel. The low level support for synchronization provided by the transputer will, however, be lost and must be replaced by synchronizing messages generated by the occam processes doing the multiplexing.

It can be seen, then, that configuration of occam processes requires that the programmer create what are essentially separate occam programs, each executing on their own transputer and communicating with each other using a limited number of channels. Decisions on how best to divide the occam program into sub-programs and allocate them to transputers must take into account firstly the connectivity constraints of the target network, secondly the loading of each transputer and finally the logical grouping of processes. The logical grouping of processes, being the least critical aspect, would normally be poorly attended to, and an insight into the structure of a distributed occam program would, unfortunately, be rarely gained by examining its modular decomposition into separately compiled processes.

The allocation decisions, outlined above, require a fairly detailed knowledge of the run-time behaviour of the program, and the task of allocation cannot be easily done by someone other than the programmer. Should a particular transputer network be upgraded by the addition of extra transputers, the maintenance required to reconfigure the existing occam programs for the upgraded network is formidable and would probably not be done optimally. In addition, vendors of occam software are faced with a problem since the source code is required for allocation or reallocation. Thus there is a clear need for the ability to automatically configure any occam program for any transputer network in an optimal fashion.

3 A Step Towards Automatic Configuration

In this section, a system is proposed which facilitates automatic configuration by not specifying allocation within the source code, but rather leaving this to the implementation. The programmer is then free to concentrate on the *logical* structure of the program, declare as many processes and channels as might be required, use constructs which reflect the algorithm most clearly, and yet be safe in the knowledge that the program will execute on any given transputer network without modification of the source code. This scheme will be referred to as *portable occam* in order to clearly distinguish it from the existing occam language.

Unlike the current allocation scheme of occam, it is proposed that *every* daughter process be given the potential to execute on *any* transputer in the target network, as decided by the allocation algorithm employed. This is illustrated by the example program in Figure 2, where the blocked code may execute on any transputer, which is not necessarily the transputer executing the code contained in an outer block.

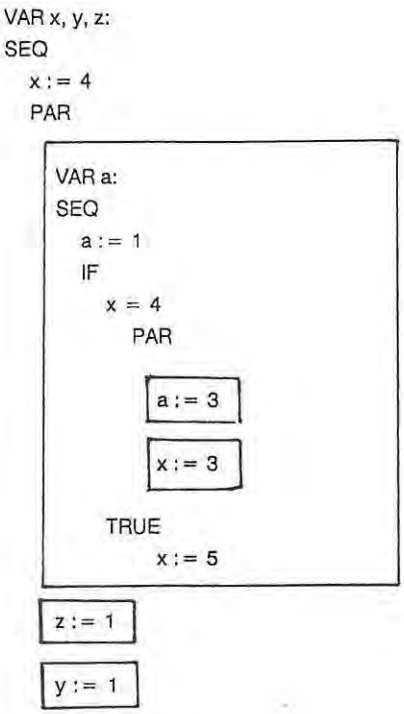


Figure 2 The Partitioning Proposed

It is necessary for a communications layer to be implemented on each transputer so as to abstract the logical connections between processes from the physical connections between transputers. Fisher [FIS86] has implemented occam on a multiprocessor token ring network

which allows for such abstraction between logical and physical connections by multiplexing occam channels onto the ring. Unfortunately the performance of the communications ring is degraded as the number of processors is increased.

Models and algorithms for optimal and suboptimal allocation are areas of research in themselves and are not investigated here, neither is a particular allocation algorithm proposed. Instead, it is assumed that if a system can be provided whereby any daughter process can be arbitrarily mapped to any transputer on any given network, then that mapping can be optimized through the use of some appropriate algorithm. Shatz *et al* [SHA87] and Chu *et al* [CHU80] survey the research into such algorithms.

3.1 Communication

The current communication philosophy of occam is that communication may only occur between directly connected transputers and that channels may not compete for communication resources. This has the advantage of removing communication bandwidth constraints on the network size, but severely restricts the set of possible allocations of a program on a given network. This restricted set might easily exclude the most efficient allocation. The ability to provide global communication is thus essential if the programmer is to be abstracted from hardware considerations and yet be given the ability to execute the program making optimal use of the available hardware.

For *portable occam*, a system is envisaged which will support the complete set of possible allocations. Hence global communication is allowed and channels may compete for communication resources.

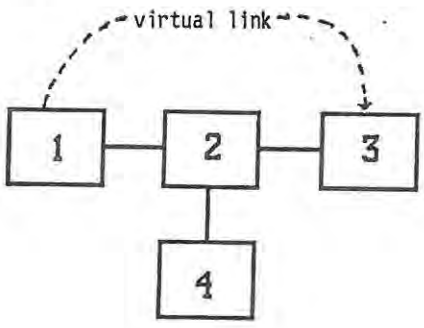


Figure 3 Message Forwarding

Communication facilities are suggested, for each transputer, to provide for a *virtual link* between every pair of transputers in the network. Those pairs which are not directly connected by a single physical link, communicate via other transputers which pass the message on. Thus a virtual link between transputer 1 and transputer 3, in Figure 3, would be created by two physical links with transputer 2 forwarding the messages.

The communication facilities must also allow any number of occam channels to be mapped to a virtual link. This sharing of links by channels can be done either on a first-come-first-serve basis or else according to priority. A priority mechanism is discussed in section 3.3.

By allowing communication between any two transputers in the network, the potential communication overhead in forwarding messages increases exponentially for each transputer as the network size increases. It is therefore imperative that the allocation algorithm adopted should attempt to minimize this overhead and the allocation should reflect a clustering, on the transputer network, of processes which communicate with each other.

3.2 Allocation

Allocation can be done either *statically* or *dynamically*, both methods having their advantages and disadvantages, and Kruskal and Weiss [KRU84] have compared the performance of these two techniques. Static allocation involves mapping processes to processors before the execution of the program. This mapping can therefore be done by the programmer, as in occam, or by the implementation before execution, which is the approach adopted by both Bishop *et al* [BIS87] and Mellor *et al* [MEL86]. The advantage of this approach is that there is no run-time scheduling overhead while the disadvantage is that, at compile-time, it is impossible to predict the run-time behaviour of the program. Hence the allocation algorithm must use predictions of such behaviour in order to determine allocation, which can lead to less efficient allocations being adopted.

In dynamic allocation, the allocation occurs at run-time with the advantage of better processor utilization since the allocation is in response to the run-time state of the processors. However, allocation decisions taken at run-time incur a certain run-time usage of both processor and communication resources. Allocation costs are thus paid each time the program is executed while costs for static allocation are paid only once for each allocation made. As a result, computationally intensive allocation algorithms are not appropriate for dynamic allocation. The dynamic allocation algorithm can either be centralized or distributed, and Gaj

et al [GAJ85] survey various implementations which use these two approaches.

Static allocation is proposed for *portable occam*, since it is more suited to transputer networks: dynamic allocation requires global communication which is expensive in a transputer network. As mentioned in section 1, the allocation stage is viewed as being separate from compilation. The allocator uses the output of the compiler together with a description of the target network, provided by the user, and produces a code packet for each transputer to execute. A disadvantage of static allocation, already mentioned, is that it is impossible, at compile-time, to predict how often a channel will be used or how much processor time a process will require. This information is obviously important to any allocation algorithm since the algorithm must minimize communication overheads while maintaining an even load balance amoung the transputers. In order to address this disadvantage, it is proposed that the run-time system, for *portable occam*, provide facilities for recording *run-time statistics* of processor and channel usage by each daughter process. Allocation is then viewed as an iterative process, with the program being configured, executed, reconfigured and re-executed until the user is satisfied with the allocation. An iterative approach to allocation is also taken by Mellor *et al* [MEL86].

3.2.1 The use of Statistics in Allocation

The idea of using statistics of the run-time behaviour of a program in order to aid the allocation process has been used by Fisher [FIS81] to aid allocation by his Bulldog compiler and Mellor *et al* [MEL86] propose this as an extension to their multiprocessor implementation of Modula-2.

Such a statistics-gathering facility obviously creates run-time overheads, and the user should be given the choice of invoking it or not. The run-time behaviour (execution pattern) of a program usually depends on the input data it is given, and thus the statistics which are calculated would differ depending on the input data used. Hence, the user has a degree of influence over the allocation: by using cumulative statistics averaged over a wide range of input data sets, the resultant configuration should be optimized for a generalized program execution pattern; by using statistics gathered using a specific set of input data, the configuration can be fine-tuned for optimal execution of a certain program behaviour.

The choice of exactly what run-time statistics should be calculated is dependent on the requirements of the allocation algorithm in use.

3.2.2 The Network Map

The allocator needs a description of the target transputer network in order to map daughters to transputers. The advent of the C004 electronic switching components, for configuring the transputer interconnections under software control, gives the allocator the freedom to configure the network topology so as to best suite the virtual interconnections required. If the links can be connected only manually, the topology must be regarded by the allocator as being fixed and must be supplied in the network map.

Other information required is the number of transputers in the network and their type. Because of the existence of special purpose transputers, such as graphic and disk controllers, the allocator needs to be aware of such transputers so as to appropriately place processes which make use of these special facilities.

3.3 Priority

The PRI PAR construct of occam assigns priority to daughter processes. The order of priority is implied by the textual order of the component processes in the source code, with the first process having the highest priority. In occam, these processes must execute pseudo-concurrently and the processor always executes the process with the highest priority which has not terminated or been blocked while waiting for I/O.

In *portable occam*, daughter processes do not necessarily execute on the same processor and a change in the semantics of the PRI PAR construct is required. The priority ordering is used to resolve any contention that might exist for the same processor or communication hardware by always giving preference to the competing process with the highest priority. This change in semantics does not guarantee that, when a process with a certain priority is executing, all processes with a lower priority are unable to execute.

3.4 Summary and Criticisms

The proposal for *portable occam* provides a more natural partitioning of an occam program into allocatable units, than does the PLACED PAR construct. The partitioning of *portable occam* gives every concurrent process the potential to execute on a separate processor and does not place restrictions on the programmer's use of the language, as does the PLACED PAR

construct. The programmer is free to specify these concurrent processes and to declare as many channels as might be required between them, without regard for a particular target network. By abstracting the use of the language from allocation issues, the proposal not only allows more freedom in the use of the language but also results in portable source code being produced. This abstraction bears the cost of making multiprocessor implementation more difficult and less efficient.

The efficiency cost is incurred by the necessity to relax the dedicated nearest-neighbour-only communication found in occam. Instead, *portable occam* allows channels to share the same communication path and to communicate with indirectly connected processors via the forwarding of messages. The message forwarding overhead depends on the allocation and is preferable to a communications bus structure since it does not necessarily place an upper bound on the number of possible processors. By clustering heavily communicating processes onto the same or nearby processor(s), the message forwarding overhead can be reduced.

The essential implementation difficulty of *portable occam* is the assumption that the implementation will provide the allocation automatically, without any user interaction. Although a limited study period did not allow for an investigation into allocation algorithms, it is perhaps worthwhile identifying some requirements of such an algorithm.

* The most important criterion is that the resultant allocation should be efficient enough to outweigh the effort of hand allocation by the user. As pointed out in section 1, efficient allocation is of particular importance to real-time programmers, since an inefficient allocation can result in timing constraints not being met and hence the incorrect execution of the program. For real-time programs, the allocation algorithm would need to be supplied with these timing constraints in order to ensure that the allocation does not cause incorrect program execution. Since the allocation algorithm can only make use of estimated information regarding run-time behaviour, it is doubtful whether satisfying such timing constraints can be guaranteed (assuming a solution exists).

* The algorithm must be able to take into account a non-homogeneous multiprocessor system, where various processors have differing capabilities, memory sizes and communication links.

* Electronically reconfigurable interprocessor connections allow the allocation algorithm to determine the network configuration. Although this should result in

more efficient allocations being found, it adds yet another dimension to the allocation algorithm. Simply finding an optimal allocation is not sufficient; an optimal combination of allocation and network configuration must be found.

* The allocation, despite all the complexities, must be done within a reasonable period.

These allocation algorithm considerations, coupled with the inefficiency imposed by global communication, support the approach of static, rather than dynamic, allocation adopted in *portable occam*.

Automatic allocation and allocation specified in the source code represent opposite ends of the allocation spectrum. The goal of automatic allocation is an ideal one and requires a great deal of further research, yet it must be achieved if concurrent software is to be truly portable. True portability implies that the user has the ability to retarget the software via an entirely mechanical process. The user should not be expected to understand the structure of the software nor to make any direct or indirect changes to the source code. In the middle of the allocation spectrum lie those approaches, such as adopted by Bishop *et al* [BIS87], Mellor *et al* [MEL86] and the implementation presented in section 4, where allocation is separated from the source code but must still be supplied by the user. A degree of portability is achieved in that the source code need not be directly modified for reallocation. However, user-supplied allocation essentially amounts to indirect modification of the source code with the user still requiring an understanding of the software in order to perform an intelligent allocation. Mellor *et al* [MEL86] propose to go one step further by allowing a combination of automatic and user-supplied allocation.

At this stage, the middle-spectrum approach is the most practical one for short-term solutions. For real-time applications, automatic allocation does not appear to be considered as a viable approach by leaders in the field [GUT86], and Kirrmann *et al* [KIR86] support the middle-spectrum approach.

4 Implementation

Portable occam has been partially implemented[1] in order to discover any implementation difficulties or inefficiencies inherent in the proposal and to provide a system on which allocation issues can be demonstrated and experimented with. It is a partial implementation of the proposal in that priorities, run-time statistics and an allocation algorithm have not been implemented. In addition, only those constructs most fundamental to the occam language are implemented and are based on the definition of occam given in the occam programming manual [OCC83]. The PRI ALT, PRI PAR and WAIT constructs were not implemented, nor were slices. Vector channels were not completely implemented and obviously the PLACED PAR construct is made redundant by automatic allocation. A restriction was placed on the use of global variables by daughter processes and is discussed in section 4.3.2. The motivation for this restriction is solely to simplify the compiler and would not be required by a more sophisticated implementation.

An important aspect of the implementation is that the network of IBM PCs used, closely reflects a transputer network (with which occam is synonymous). The relevance of Fisher's multiprocessor implementation of occam [FIS86] is restricted in that the processors are interconnected by a token ring. The choice of IBM PCs over transputers was simply for economic and practical reasons: IBM PCs are easily accessible at Rhodes University and provide well-documented hardware on which a run-time support system can be implemented; transputers are not so easily available and Inmos has been criticized [MAY87b] for the poor machine-level documentation made available.

HUL [CLA86] is a concurrent language, developed at Rhodes University, based on occam and is designed to exploit interrupt-generating boolean variables while using control structures based on human-like commands. An existing single-processor implementation of HUL [SAN85] was used as a springboard for the implementation. The HUL implementation consisted of a compiler/interpreter system written in UCSD-Pascal executing on a Sage II computer. Adapting this implementation to a multiprocessor implementation of occam involved extensive modification of both the compiler and interpreter. It was decided to use Turbo-Pascal instead of USCD-Pascal because of the facilities offered by Turbo-Pascal for access to I/O ports and operating system function calls. These facilities were required in order to implement interprocessor communication, and the HUL implementation was therefore translated into Turbo-Pascal executing on an IBM PC under MS-DOS. This implementation was then

[1] program listings available under a separate cover

extensively modified to compile, allocate, load and execute an occam program on an IBM PC network.

4.1 Implementation Outline

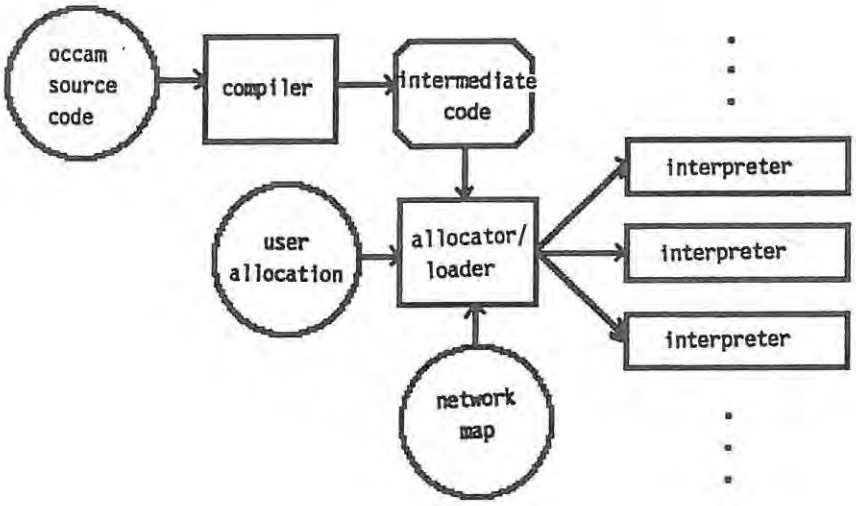


Figure 4 Implementation Components

The basic components of the implementation are illustrated in Figure 4. The compiler produces intermediate language codes (ILCs) which can be executed by a stack-based interpreter; identical copies of the interpreter are executed by each processor in the network. The allocator uses a user-supplied allocation and network map to create an allocation table and network map for each processor. The loader then sends to each interpreter the complete set of ILCs, generated by the compiler, together with an allocation table and network map. At this stage, each interpreter has the ability to execute any daughter process if instructed to do so. It is not necessary that the loader supply each processor with the code for the entire program, but this was done because the ILCs produced by the compiler are not relocatable. Fisher [FIS86], in his multiprocessor implementation of occam, uses the alternative strategy of supplying each processor only with the code it requires.

In the current implementation of *portable occam*, the allocation is performed each time the program is executed, and thus the allocation and loading stages are not separated. It is a trivial extension to separate these two phases so that allocation need not be done for every execution. The allocation/loading phase is done by a processor of the user's choice. Once all the processors in the network have been loaded, the processor which performed the loading automatically begins execution of the outermost occam process. This would be a restriction if

the allocation was done via an algorithm, rather than the user, and could easily be overcome by the loading phase instructing a processor, determined by the algorithm, to execute the outermost process.

On reaching that point in the code where a daughter process is invoked, the processor consults the configuration table to determine which processor is to execute that daughter, and instructs the relevant processor (which could be itself) to proceed. Each processor maintains a list of daughter processes it has been instructed to execute, and executes them on a time-slice basis. A daughter process may, in turn, invoke its own daughter processes (granddaughters) in the manner described above.

The implementation is a simple one and has much scope for improvement. The following sections describe the most interesting implementation issues, outline the solutions adopted, criticize these and suggest possible alternatives.

4.2 Communication Issues

4.2.1 Packet Communication

In order to facilitate the multiplexing of many occam channels onto a single virtual link, all messages are sent in the form of packets. Only complete packets may be multiplexed, not elements within a packet. The structure of a message packet is shown in Figure 5.

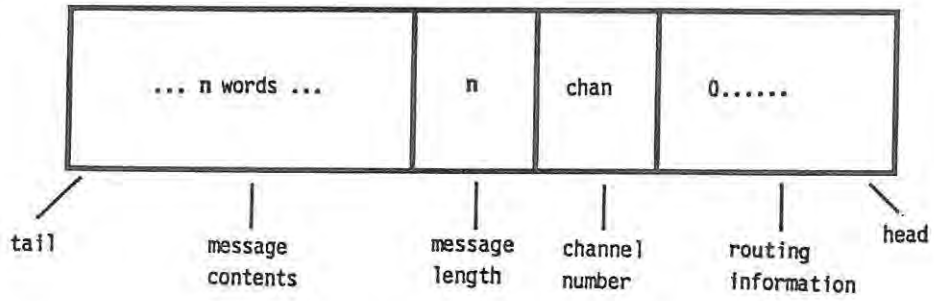


Figure 5 Structure of Message Packet

Each element of the packet is a 16-bit word. At the head of the message packet is a set of port numbers, which must be in the range 1..4. This sequence of port numbers is terminated by a zero. Each port number refers to one of the serial ports of the four-port serial card contained in each IBM PC of the network and is used to indicate the virtual link on which

the packet is being sent. On receiving a message packet, the communications routine of the processor examines the first number of the packet. A zero indicates that the message is destined for that processor, and the communications routine decodes the message packet and acts accordingly. If the first number of the message packet is non-zero, then it must be in the range 1..4 and indicates on which port the communications routine is required to forward the message. The routine does not send that first number, but forwards the remaining elements of the packet. Thus the message is routed through the network in accordance with this sequence of port numbers until the zero appears at the head of the message, indicating that it has arrived at its destination. Figure 6 shows the sequence of port numbers needed in order to create the virtual link between processor 1 and 5 in the network shown.

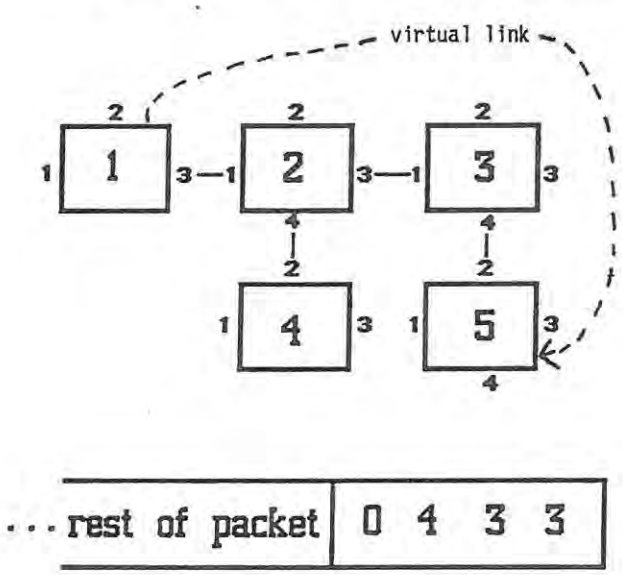


Figure 6 Message Packet Routing Information

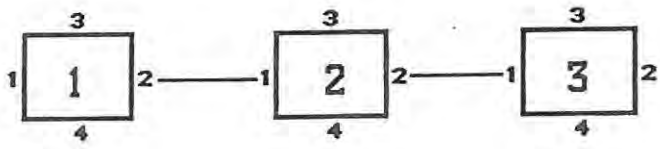
On receiving a message packet destined for itself, the communications routine examines the channel number. A channel number of zero indicates that the message is a system message, and all other numbers indicate the unique identification number assigned to each run-time instance of an occam channel. The method used to generate these numbers is described in section 4.3.4.2. These identification numbers allow many channels to be multiplexed to a single, virtual interprocessor connection. System messages are passed between the interpreters in the network and are used to communicate loading information, instructions to initiate the execution of a process and to halt all process execution.

An interrupt routine performs low-level I/O on the four-port serial card. Each port is given an input and output buffer. The interrupt routine simply sends all bytes in the output buffer

to the port and places, in the input buffer, all bytes received on that port. The interpreter polls these buffers after the execution of each ILC, and on the detection of a complete message packet, performs the necessary operation on it. An additional, virtual port (assigned the port number zero) is defined for compatibility reasons, and is used by the interpreter for sending messages to itself.

4.2.2 The Formation of the Network Map

The network map provides the mapping of virtual links to physical links and, in this implementation, must be provided in its fullest form by the user. In order to avoid calculation of the shortest paths between processors, the user is required to provide, for each ordered pair of processors, the set of port numbers which the first processor must place at the head of a message packet destined for the second processor. Figure 7 shows a network and the network map which must be supplied. This information is entered, using a text editor, into a file and read by the allocator. The text enclosed in curly brackets is for explanation purposes only and does not form part of the network map.




```
3          {number of processors in network}
0          {from processor 1 to processor 1}
3 0        {1 to 2}
3 3 0      {1 to 3}
1 0        {2 to 1}
0          {2 to 2}
30         {2 to 3}
1 1 0      {3 to 1}
1 0        {3 to 2}
0          {3 to 3}
```

Figure 7 Example Network Map

4.2.3 Message Communication and Synchronization

The implementation of synchronized communication on a multiprocessor system requires that, for a simple message exchange on the program level, a number of asynchronous protocols must be exchanged at run-time in order to ensure synchronization. Occam allows only input guards

in a guarded command which significantly simplifies these protocols, but makes the expression of some algorithms far more cumbersome [BOR86]. Bornat [BOR86] has proposed a protocol for a generalized version of occam allowing both input and output guards in a guarded command, and the significant increase in protocol complexity introduced by a bi-directional guarded command is clearly apparent. In this implementation, only the input guard to a guarded command is allowed but the protocol mechanism used allows for an extension of the implementation to include bi-directional guarded commands.

Allowing any daughter process to execute on any processor introduces a major difficulty in implementing channels. A daughter process may execute on a different processor to her parent and yet both may make use of channels declared global to them. As a result, a channel does not necessarily remain mapped to the same virtual link throughout program execution. To illustrate this, consider the following example:

Example 1

```
1   CHAN aChannel:
2   VAR x,y,z:
3   PAR
4     SEQ
5       aChannel ? x
6       PAR                          -- (i)
7         aChannel ? y               -- (iii)
8         z := 1                     -- (iv)
9     SEQ
10      aChannel ! 1                 -- (ii)
11      aChannel ! 2
```

The same convention is adopted as for Figure 2 where the blocked code indicates the daughter processes, each of which could execute on a different processor. In example 1, process (iii) sends on aChannel which is also used by her parent process (i). Assume that processes (i), (ii) and (iii) are mapped to the separate processors (1), (2) and (3) respectively. The first communication occurs when process (i) executes the ? primitive in line 5 and process (ii) executes the ! primitive in line 10. For this communication, aChannel is mapped to the virtual link between processor (1) and processor (2). However, when process (iii) executes its ? primitive (line 3) and process (ii) executes the ! primitive of line 11, aChannel is mapped to the virtual link between processor (2) and processor (3). This change in channel mapping occurs at some stage during the execution of process (i) and cannot be predicted by processor (2). Hence the channel mapping for a particular communication can only be determined when the synchronization between the two communicating processes has been established.

This implementation problem does not arise in multiprocessor implementations of standard occam because of the limited allocation allowed by the PLACED PAR construct. The allocation forces all processes which may send on a particular channel to execute on the same processor. Similarly, all processes which may receive on a particular channel must execute on one processor and thus the mapping of a channel to a physical link remains static throughout program execution.

4.2.3.1 A Hierarchial Re-routing Approach

This section presents and criticizes and initial solution devised in an attempt to solve the problem of dynamically changing channel to virtual link mappings. This approach was not implemented due to its run-time inefficiency.

In this solution, two protocol messages are used :

RTS - Request To Send. Sent by the sending process to the receiving process when it is able to send a message on a channel.

PTS - Permission To Send. Sent by the receiving process to the sending process when an RTS has been received *and* the receiving process is ready to accept the message.

The sending process sends an RTS to the receiving process, and passively waits until it receives a PTS. It then immediately sends the message and continues with execution. The receiving process is given the responsibility of ensuring synchronization by only sending a PTS when it is ready to receive and the sender has indicated its willingness to send. The asymmetry of this protocol allows for an easy implementation of the unidirectional guarded command provided by the ALT construct of occam. The process executing the ALT simply waits until synchronization is possible with one or more of the input guards, as indicated by any RTS message received. The first guarded process which is detected as being ready, is executed.

This solution relies on the sending and receiving processes being able to direct the messages to each other. As explained in section 4.2.3, the identities of the processors on which the partners in the communication are executing can only be determined after synchronization. It

was decided that the sender and receiver should always assume that their outermost partner process was still active, and direct all their message packets to the processor on which their outermost partner process executes. The identity of this processor can be determined at load time, after allocation. If the outermost process has already spawned daughter processes, and is thus temporarily inactive, the processor must re-route the message packets to the processor executing the daughter who has inherited use of that channel. Thus the message packets are routed from parent to daughter to granddaughter etc. until it reaches the currently active child process. This re-routing is illustrated in Figure 8. On terminating, a daughter must pass back to her parent any message packets which she did not satisfy. Thus message packets are treated in a fashion analogous to reference parameters.

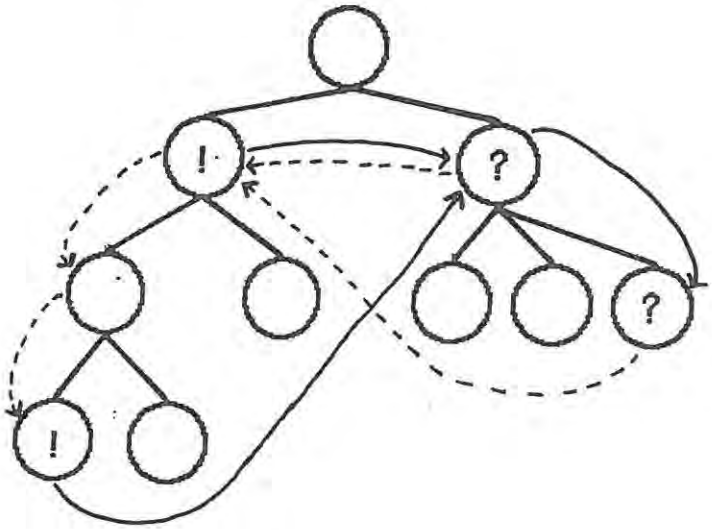


Figure 8 Illustration of Message Re-routing

This message re-routing leads to great run-time overheads in communications, which increase as the hierarchy of processes inheriting the use of a channel grows. Furthermore, the calculation of communication times is greatly complicated.

4.2.3.2 The Approach of a Third-Party Channel Supervisor

This section presents a more efficient method than that of section 4.2.3.1, and is the solution adopted in this implementation. In this approach, three components can be identified and are illustrated in Figure 9. In addition to the sender and receiver, a third-party, the matcher, is defined and may exist on a processor which does not execute either the sender or the receiver. A separate matcher exists for each instance of a channel and is a run-time facility

provided by the processor. The task of the matcher is to monitor the status of the channel and co-ordinate synchronization. The processor which was executing the process in which one or more channels are declared is assigned the task of matcher for those channels. The sender and receiver processes use the matcher to negotiate a synchronous message exchange between them, by sending all their protocol messages to the matcher. Since the matcher remains on the same processor throughout the existence of that channel, the mapping of these protocol messages to a virtual link is static and can be determined at the loading stage. When synchronization has occurred, the matcher can inform the sender of the receiver's location and the sender can then send the message directly to the receiver's processor. Bornat [BOR86] also makes use of a matcher in his protocol for bi-directional guarded commands.

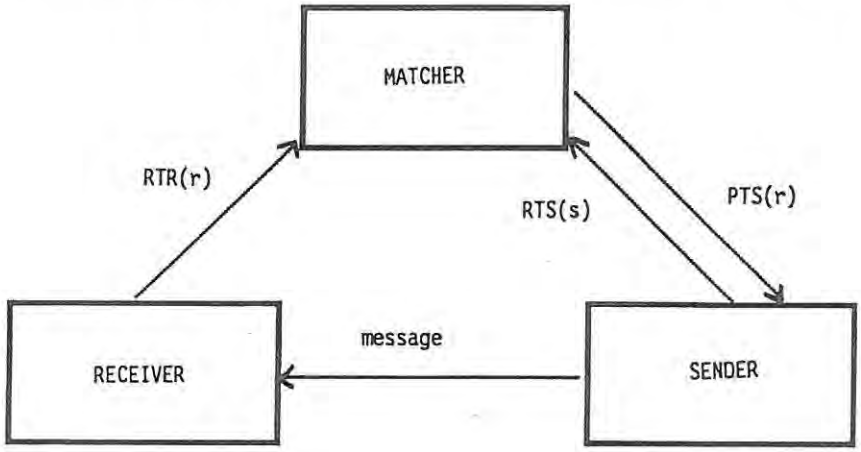


Figure 9 Illustration of Protocol

The following protocol messages are defined:

RTS(s) - Request To Send: sent by sender to matcher where s is the identification number of the sender's processor.

RTR(r) - Request To Receive: sent by the receiver to the matcher where r is the identification number of the receiver's processor.

PTS(r) - Permission To Send: sent by the matcher to the sender where r is the processor identification number, as supplied by the receiver, to which the sender must direct the actual message.

RFS(r) - Request For Status: sent to the matcher by a process executing an ALT construct. The matcher replies with an ATS if it has already received an

RTS. The reply is sent to the processor indicated by r.

ATS - Able To Send: sent by the matcher in reply to a RFS(r) query if the matcher has already received an RTS(s) message. The reply is sent to the processor indicated by r.

UTS - Unable To Send: sent by the matcher in reply to a RFS(r) query if the matcher has not yet received an RTS(s) message. The reply is sent to the processor indicated by r.

The execution of an occam ! primitive simply involves an RTS(s) being sent to the matcher, and the process is then placed on a send-queue and sleeps. When a PTS(r) message is received the process is woken, sends its message to the processor indicated by r and continues execution.

An occam ? primitive is equally simple. The process sends an RTR(r) message to the matcher, is then placed on a receive-queue and sleeps. When the actual message is received from the sender, the message is stored at the appropriate address and the process is woken.

The ALT construct is slightly more complicated. For each guarded process, the conditions of the guard are evaluated. An input guard is treated, at this stage, as a condition. It is evaluated by sending an RFS(r) to the channel matcher and waiting for a reply. An ATS reply causes the input guard to be evaluated as true and a UTS reply results in a false evaluation. The first guarded process whose conditions all evaluate to true is executed and the input guard is now treated as a ? primitive and executed.

Each channel is stored as two words. One word is the unique run-time identification number for the channel and the other word is the identification number of the processor providing the matching function.

The solution discussed here is far more efficient than that of section 4.2.3.1 since no message re-routing is involved. By creating an independent matcher, not only is the re-routing of messages overcome but the relationship between sender and receiver is also symmetric since they both depend on the matcher for synchronization. Because this structure is similar to that used by Bornat [BOR86], his protocol for bi-directional guarded commands could be implemented as an extension.

The implementation of the ALT construct could be made more efficient by using an *idle-wait* protocol rather than the existing *busy-wait* protocol. A busy-wait protocol is one in which a process must periodically poll a variable in order to establish whether another process is willing to communicate. An idle-wait protocol is one where a process simply indicates its willingness to communicate and waits for a willing partner to reply [BOR86]. An idle-wait protocol is obviously more efficient since polling uses processing and communication resources unnecessarily.

4.3 Issues Relating to Daughter Processes

4.3.1 Language Restrictions Introduced to Daughter Invocation

Daughter processes are the components into which, according to *portable occam*, the program is partitioned for allocation purposes. This partitioning has a hierarchial structure since a daughter process has the ability to spawn its own daughters. Daughter processes are invoked using the PAR construct of occam:

```
PAR
  D1
  D2
  D3
  .
  .
  .
```

where D1, D2, D3 ... indicate invocations of daughter processes. Occam allows these daughter processes to be either named processes, declared using a PROC declaration, or unnamed processes, where the definition of the process is placed at its point of invocation. The syntax of occam governing the use of global variables by daughter processes does not allow for the

easy partitioning of daughters by the compiler. Consider the following example:

Example 2

```
VAR x, y:
SEQ
  x := 1
  y := 2
  PAR
    x := x + 1          -- (i)
    VAR z:
    SEQ                 -- (ii)
      z := y
      z := z + 1
```

Here, daughter process (i) makes use of the global variable x, modifying its value and thus using it as a named process would use a reference parameter. Note that the non-interference rules of occam disallow process (ii) from either reading or modifying the contents of x.

Process (ii) makes use of the global variable y, but only reads the value of y and does not modify it. The non-interference rules of occam allow process (i) to also read the contents of y but neither process may change the value of y. Process (ii) is using y in the manner of a value parameter.

In standard occam, process (i) and process (ii) must execute on the same processor as their parent process, and the global variables can be easily accessed via common memory. However, since this implementation allows both process (i) and process (ii) to execute on different processors, the current values of all global variables used by them must be sent to their processors when they are invoked. If a global variable is modified by a daughter, its value must be returned to the parent processor when the daughter terminates. Global variables must therefore be treated as implied parameters to the daughter process.

In order to keep the compiler simple, it was decided to disallow the use of unnamed processes as daughter processes. Furthermore, named processes invoked as daughters may only reference local variables. This forces the programmer to import, as actual parameters to a daughter process, all global variables used by that daughter, including her children, and to explicitly state whether they will be used as reference or value parameters. This greatly simplifies the compiler, since only those variables used as actual parameters need be passed to the processor executing the daughter.

Two daughter processes can be prevented from both attempting to modify a global variable by introducing the following rule: a variable may be passed as a reference parameter to only one daughter process. This allows more freedom in the use of global variables than is allowed by the non-interference rules of occam. The following examples illustrate this:

Example 3

```
VAR x:
SEQ
  x := 1
  PAR
    x := x + 1
    VAR z:
    SEQ
      z := x
      z := z + 1
```

Example 4

```
PROC D1 (VAR a) =
  a := a + 1:

PROC D2 (VAL a) =
  a := a + 1:

SEQ
  x := 1
  PAR
    D1(x)
    D2(x)
```

Example 3 is illegal due to the non-interference rules of occam, since the value of x which is assigned to z is not defined, and could either be 1 or 2. Example 4, which obeys the restrictions introduced in this section, allows x to be used by both daughters and the use of parameters clearly defines the value of x, when used by process D2, as being 1.

The syntax restrictions outlined in this section result in more verbose programs but, in addition to simplifying the compiler, makes the use of global variables by concurrent processes more distinct.

4.3.2 Daughter Invocation

The compiler checks the use of actual parameters by daughter processes according to the restrictions defined in section 4.3.1. Checks are also made on the use of channel parameters by daughter processes to ensure, as far as possible, the legal use of channels and to detect some obvious cases of deadlock.

At run-time, the allocation table is used to determine the processor to which a particular invocation of a daughter process has been allocated. Each invocation of a daughter is assigned a unique number, calculated at run-time. The method used to determine these numbers is explained in section 4.3.4.3. Each number indicates an offset into the allocation

table where the identification number of the processor allocated to that daughter can be found.

The parent process, having determined where a daughter is to execute, sends that processor a system message instructing it to initiate execution of the daughter. The start address of the code for the daughter is supplied together with additional information required by the daughter to calculate unique numbers for its channels and daughters. A new, separate stack is allocated to each daughter and is simply assigned a default size, regardless of the of the daughter's space requirements for locally declared variables and channels. This simplistic approach makes inefficient use of stack memory and a better approach would be to take into account varying workspace requirements as is done by Fisher [FIS86]. It is difficult to calculate exact space requirements for parameters, as is discussed in the next section, and for arithmetic calculations.

4.3.2.1 The Parameter Passing Mechanism used

Occam allows vectors (1-dimensional arrays) of unspecified length to be used as formal parameters to named process declarations. Since the compiler cannot calculate the amount of space which must be reserved for the actual parameter passed, indirection is used, as illustrated in Figure 10.

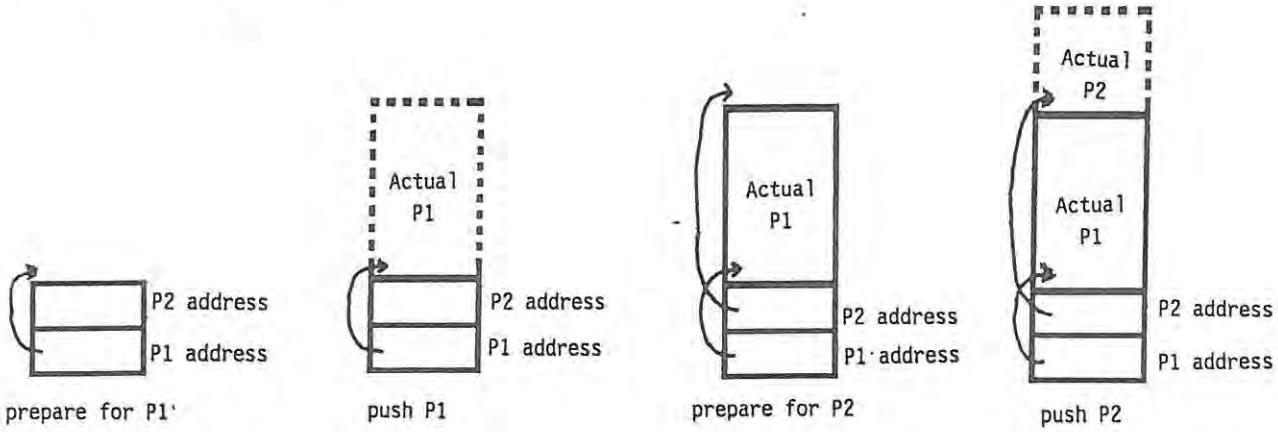


Figure 10 Parameter Passing

A parameter is passed by pointing the formal parameter location to the current top of the stack. The actual parameter is then pushed onto the stack, the next formal parameter is pointed to the top of the stack and next actual parameter is then pushed on the stack etc.

This mechanism is used for both sequential and daughter processes.

4.3.2.2 Parent-Daughter Parameter Passing

The parameter passing, described in section 4.3.2.1, relies on a strict ordering of events. The parent process cannot simply send all the parameters as a single unit, because the size of each parameter is not known by the daughter. Instead, the parameters are each sent separately. The daughter process first points the formal parameter to the top of the stack, then receives the actual parameter from the parent and stacks it, then points the next formal parameter to the top of the stack etc. The synchronous message passing of occam provides a natural mechanism to enforce this synchronization between parent and daughter in the manner described by Figure 11.

Occam Program

```
PROC D1 (VAL a VAR b) =
  .
  .
  .

PROC D2(VAL c) =
  .
  .
  .

VAR x, y, z:
SEQ
  x := 1
  y := 2
  z := 3
  PAR
    D1(x, y)
    D2(z)
```

Implementation

```
CHAN D1in, D1out, D2in:
PROC D1 (VAL a VAR b) =
  SEQ
    D1in ? a
    D1in ? b
    .
    .
    .
    D1out ! b
PROC D2 (VAL c) =
  SEQ
    D2in ? c
      .
      .
      .
    VAR x, y, z:
    SEQ
      x := 1
      y := 2
      z := 3
{ initiate execution of D1 on some processor}
      D1in ! x
      D1out ! y
{ initiate execution of D2 on some processor}
      D2in ! z
      D1out ? y
```

Figure 11 An illustration of the use of channels to pass parameters to daughters

The compiler automatically declares a channel between a parent and each of its daughters. This channel is used as a bi-directional channel to send parameters to a daughter, after invoking it, and to receive the new values of any reference parameters before the daughter terminates. The channel is used as a bi-directional channel so as to make more efficient use of the space of unique numbers used to identify channels. This is possible since only the system makes use of the channel in a defined manner.

As can be seen from Figure 11, the daughters are invoked in the sequence of their textual ordering in the PAR construct. The order of this invocation does not affect the logical behaviour of the program but might affect the efficiency of its execution. Assume process D2, in Figure 11, took very much longer to execute than process D1 and is thus the critical process in determining the completion time for the entire PAR construct. Because D1 is instantiated before D2, this completion time is unnecessarily increased and could be shortened if the order was reversed. The ideal order of daughter instantiation is thus in the order of their expected execution times, with the process having the longest execution time being instantiated first. This is obviously difficult to implement, but a better implementation strategy to that of Figure 11 might be to use an ALT construct for sending parameters to all daughters.

Alternately, the parameter passing need not be synchronized at all and could be passed to each process in one packet as an asynchronous system message. The sizes of the parameters would need to be specified in the system message. This is probably the best solution since the daughters would be instantiated virtually simultaneously, delayed only by communication delays.

4.3.2.3 Parent-Daughter Synchronization

The parent process, having spawned its daughters, must delay any execution of its subsequent processes until all daughters have terminated. Again, the synchronous communication of occam is used to implement this. Each daughter, having sent back all reference parameters to its parent, sends a synchronizing message (! ANY) to the parent process using the channel reserved for parameters. This indicates the completion of the daughter process and the parent simply receives a synchronizing message from each daughter in turn. The order in which this is done does not affect efficiency. Figure 12 shows this technique.

Occam Program	Implementation
	CHAN D1out, D2out:
PROC D1 =	PROC D1 =
.	.
.	.
.	.
	D1out ! ANY:
PROC D2 =	PROC D2 =
.	.
.	.
.	.
	D2out ! ANY:
SEQ	SEQ
PAR	{ initiate execution of D1 }
D1	{ initiate execution of D2 }
D2	D1out ? ANY
.	D2out ? ANY
.	.
. {continue further execution }	.
	. { continue further execution }

Figure 12 The use of Channels to Synchronize Parent with Daughters

4.3.3 Code Generation for Named Processes

A named process may be called as a sequential and/or daughter process. This requires that code be generated to cater for both types of invocation. The compiler, when parsing a named process, determines whether the process makes reference to any non-local variables. Should this occur, that named process is disqualified, in terms of the restrictions defined in section 4.3.1, from being called as a daughter process. In this case, code is generated which is suitable only for sequential invocation, and the compiler ensures that only such invocations occur. If, however, the named process references local variables only, it is possible for it to be invoked as a sequential or a daughter process, and code suitable for both cases must be generated.

The difference between the code for a sequential and for a concurrent process is that the concurrent process must include
code for parameter passing and synchronization via channels, as described in sections 4.3.2.2 and 4.3.2.3.

4.3.4 Name allocation

This section deals with the allocation of unique identification numbers (names) to daughter processes, channels and processors so that they may each be identified by the system. The efficient use of this name space is a problem for concurrent systems which allow recursion, including dynamic dataflow systems [VEE86]. Recursion prevents the compiler from being able to determine the number of software entities in existence at some particular stage and thus prevents efficient utilization of the name space. Occam does not allow recursion, either direct or indirect, and this greatly simplifies name allocation.

4.3.4.1 Processor Name Allocation

The allocation of names to processors does not present a problem since the number of processors in the system is specified by the user in the network map. The identification number of each processor is implied in the ordering of the information in the network map, and the loader supplies each processor with a unique identification number.

4.3.4.2 Channel Name Allocation

The channel name allocation scheme described here relies on the absence of recursion in occam. This makes it possible for the compiler to determine, for each process call, the maximum amount of channel name space required by that process for naming local channels. This name space includes the requirements of all processes called directly or indirectly by that process.

A calling process is thus able to supply each called process with an adequately sized subset of its own channel name space. This subspace is used by the called process for generating unique names for its own local channels, and for subdividing into further subspaces for processes which it calls.

Sequential processes may use overlapping channel name spaces since they do not exist concurrently. Daughter processes must be given disjoint spaces.

When parsing the body of a process, the compiler assigns a consecutive number to each local channel declared, starting from 1. This number represents an offset into the subspace

assigned to each invocation of the process.

On generating code to call a process, the compiler assigns an offset number to that invocation. This number represents an offset into the subspace of the calling process where the base of the subspace of the called process is to be found. The called process generates unique channel identification numbers by adding the offset of a channel to the base of the subspace assigned to it.

4.3.4.3 Daughter Name Allocation

The allocation table is a 1-dimensional array where each array index represents a particular invocation of a daughter process and the corresponding element contains the name of the processor to which that daughter invocation has been assigned. Daughter name allocation refers to the mapping of daughter invocations to allocation table indexes.

The method used here is similar to that used to allocate channel names, in that each daughter process invoked is supplied with a base index. The code generated for a daughter process uses offsets from this base to calculate absolute indexes into the allocation table. This allows different invocations of the same daughter process to execute the same code but, by using different bases, different sections of the allocation table are referenced. Thus children processes of these various invocations can be independently allocated.

The iterative invocation of daughters by means of a WHILE construct causes the same locations in the allocation table to be referenced and hence the allocation of daughters remains the same for each iteration.

5 Summary and Conclusions

The proposal and implementation presented here represent a small, but significant, step towards portable occam. At present, allocation must be specified from within an occam program, binding the program to a specific transputer network. This facilitates the simple, efficient multiprocessor implementation of occam but requires that program design and expression be compromised according to the target hardware. While this might be beneficial to some real-time programmers it is a weakness for non real-time applications where the benefits gained from hardware abstraction are worth the price of less efficient execution.

Complete abstraction from hardware implies that a programmer has the ability to construct and execute a program without any knowledge of the target hardware. The occam implementation is thus required to perform allocation without assistance from the user. The proposal suggests changes to occam, needed to abstract the source code from a specific transputer network, and some run-time facilities needed to support automatic configuration. The only change the proposal makes to the syntax of the language is in eliminating the PLACED PAR construct. The software partitioning and the communication restrictions, embodied by this construct, seriously limit the set of possible allocations of an occam program. Instead, it is proposed that every component process of a PAR construct be an allocatable unit, as this is the finest degree of concurrency explicit in the language. Communication is relaxed to allow any logical network of channels to be mapped to any physical communications topology. This generalized communication must be slower in terms of communication times but need not necessarily use CPU resources. It is plausible that the link adapter of the transputer could be designed to handle message forwarding and channel multiplexing autonomously.

The implementation brought to light an overhead not immediately obvious from the proposal. The finer partitioning of an occam program results in the communications path for a particular channel changing during execution. The more sophisticated protocol required to deal with this, results in more protocol messages for a single language-level communication. Furthermore, the fact that two communicating processes reside on the same processor does not imply that their communication places no burden on the communications hardware. The matcher function might be provided by a separate processor and protocol messages may, therefore, be external communications; only the actual message transfer is guaranteed to be simply a memory transfer.

The proposal suggests the use of run-time statistics in order to support a static allocation

algorithm in its decision-making, however, the implementation did not cover this aspect. It would be an interesting extension to implement some allocation algorithm and explore the affect of using run-time statistics. The success of automatic configuration ultimately rests on the proficiency of the algorithm and its ability to adapt to user requirements.

The current implementation, despite not providing complete hardware abstraction, does allow an occam program to be written independently of any hardware considerations. The program can then be separately allocated without recompiling the source code, and executed on a multiprocessor system. This provides a useful tool for demonstrating various allocations and examining program behaviour in a truly concurrent environment. It also provides a stepping-stone towards the automatic configuration of occam programs.

Bibliography

[ADA84] *Ada Programming Language Reference Manual* (1984) United States Department of Defence.

[BIS87] Bishop, J.M., Adams, S.R., Prichard, D.J. (1987) Distributing Concurrent Ada Programs by Source translation, *Software - Practice and Experience*, 17(12), 859-884.

[BOR86] Bornat,R. (1986) A Protocol for Generalized occam, *Software - Practice and Experience*, 16(9), 783-799.

[BOT86] Bottomley, R. (1986) The Meiko Computing Surface : a reconfigurable supercomputer, IEE Colloquium : The Transputer : Applications and case studies, *IEE Digest*, 91.

[BRI75] Brinch Hansen, P. (1975) The programming language Concurrent Pascal, *IEEE Trans. Software Eng.*, 1(2), 199-206.

[BRI78] Brinch Hansen, P. (1978) Distributed Processes : a concurrent programming concept, *Comm. ACM*, 21(11), 934.

[CHU80] Chu, W.W., Holloway, L.J., Lan, M., Efe, K. (1980) Task Allocation in Distributed Data Processing, *Computer*, 13(11), 57-69.

[CLA86] Clayton, P.G. (1986) *A Notation for Programming with the use of Human-like Control Structures*, Tech. Doc. 86/1, Department of Computer Science, Rhodes University, Grahamstown.

[COO80] Cook, R.P. (1980) *MOD - a language for distributed programming, *IEEE Trans. Software Eng.*, 6(6), 563.

[CRO87] Crookes, D., Morrow, P.J., Milligan, P., Scott, N.S., Kilpatrick, P.L. (1987) Notes on implementing a language for Transputer Networks, *Microprocessing and Microprogramming*, 21, 559-566.

[FAY87] Fay, D.Q.M. and Das, P.K. (1987) Hardware reconfiguration of Transputer networks for distributed object-orientated programming, *Microprocessing and Microprogramming*, 21, 623-628.

[FIS81] Fisher, J.A. (1981) Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. Computers*, 30(7), 478-490.

[FIS86] Fisher, A.J. (1986) A Multi-processor Implementation of occam, *Software - Practice and Experience*, 16(10), 875-892.

[GAJ85] Gajski, D.D., Peir, J. (1985) Essential Issues in Multiprocessor Systems, *Computer*, 18(6), 9-27.

[GUT86] Guth, R. (1986) (Ed.) *Computer Systems for Process Control*, Proc. Brown Boveri Symposium on Computer Systems for Process Control (1985: Brown Boveri Research Centre), Plenum Press, New York.

[HIL86] Hill, D.T (1986) *Simulating a Transputer*, Honours Project, Dept. Computer Science, Rhodes University, Grahamstown.

[HOA78] Hoare, C.A.R (1978) Communicating Sequential Processes, *Comm. ACM*, 21(8), 666-677.

[JON85] Jones, G. (1985) *Programming in 'occam'*, PRG-43, Oxford University Computing Laboratory, Oxford, U.K.

[OCC83] *Occam Programming Manual* (1983) Inmos Ltd., Bristol, U.K.

[KIR86] Kirrmann, H., Lalive D'Epinay, T., Stoeckler, H.P. (1986) Architectures for Process Control, in *Computer Systems for Process Control*, Plenum Press, New York.

[KRU84] Kruskal, C.P., Wiess, A. (1984) Allocating Independent Subtasks on Parallel Processors, *Proc. Int'l Conf. Parallel Processing*, 236-240.

[MAT87] Mattos, P. (1987) *Program Design for Concurrent Systems*, Tech. Note 5, Inmos Ltd., Bristol, U.K.

[MAY83] May, D. (1983) Occam, *Sigplan Notices*, 18(4), 69-79.

[MAY84] May, D. and Taylor, R. (1984) Occam, *Microprocessors and Microsystems*, 8(2), 73-79.

[MAY85] May, D. and Shepherd, R. (1985) Occam and the Transputer, in *Concurrent Languages in Distributed Systems*, eds. G.L. Reijns and E.L.Dagless, Elsevier Science Publishers, North Holland.

[MAY87a] May, D. (1987) *Occam2 Language Definition*, Inmos Ltd., Bristol, U.K.

[MAY87b] Mayer-Lindenburg, F. (1987) FIFTH on the Transputer, *Microprocessing and Microprogramming*, 19(5), 367-373.

[MEL86] Mellor, P.V., Dubery, J.M., Whitehead, D.G. (1986) Adapting Modula-2 for distributed systems, *Software Eng. Journal*, September, 184-189.

[NEW86] Newport, J.R. (1986) An introduction to Occam and the development of parallel software, *Software Eng. Journal*, July, 165-169.

[POU86] Pountain, D. (1986) Personal Supercomputers, *Byte*, 11(7), 363-368.

[POU87] Pountain, D. (1987) Power to the Transputer, *P C World*, 5(8), 124-127.

[SAN85] Sanger, W.L. (1985) *The HUL Parser*, Honours Project, Department of Computer Science, Rhodes University, Grahamstown.

[SHA87] Shatz, S.M., Wang, J. (1987) Introduction to Distributed-Software Engineering, *Computer*, 20(10), 23-31.

[STA85] Stammers, R.A. (1985) Ada on Distributed Hardware, in *Concurrent Languages in Distributed Systems*, eds. G.L. Reijns and E.L.Dagless, Elsevier Science Publishers, North Holland.

[TRA85] *Transputer Reference Manual* (1985) Inmos Ltd., Bristol, U.K.

[VEE86] Veen, A.H (1986) Dataflow Machine Architecture, *ACM Comput. Surveys*, 18(4), 365-396.

[WIR77] Wirth, N. (1977) Modula: A language for modular multiprogramming, *Software-Practice and Experience*, 7(1), 3-35.

[WIR82] Wirth, N. (1982) *Programming in Modula-2*, Springer-Verlag, New York.