High-Level Paradigms for Deadlock-Free High-Performance Systems

Peter Welch, George Justo and Colin Willcock
Computing Laboratory, The University, Canterbury, Kent, CT2 7NF

Abstract. This paper reviews the general notion of deadlock (and livelock) in parallel systems based upon synchronised message passing and relates them to the much worse problem of undetected data-loss in asynchronous or shared-memory communications. Two design paradigms (I/O-PAR and Client-Server) are presented that guarantee freedom from deadlock for synchronised communication regimes (both continuous and irregular data-flow). The paradigms are based upon the notion of ‘synchronisation classes’ for processes that are closed under certain forms of parallel composition. Checking for deadlock-freeness devolves to checking that the base processes belong to the correct classes and that the composition rules are observed. The complexity of this checking is at worst $O(n^2)$, where $n$ is the number of processes in the system, as opposed to $O(sn)$, where $s$ is the (average) number of states in each process. The latter would be required for an arbitrary parallel design. The automated checking of these design rules is therefore highly practical.

High-performance applications (e.g. physical system modelling, embedded real-time systems, …) generally consist of two components: the computationally intensive part (which is usually logically simple and can exploit the I/O-PAR paradigm) and its controlling apparatus (which can be highly complex but can exploit the Client-Server paradigm). This paper reports on the design rules for hybrid combinations of the two paradigms that preserve their deadlock-free properties. Examples will be presented. The classic ‘Dining Philosopher’ system is shown to illustrate an I/O-PAR / Client-Server hybrid that breaks these rules.

Finally, the Client-Server paradigm will be looked at from the point of view of the special language support provided by occam3.

1 Introduction

The real-world is a demanding and complex environment in which to embed computer systems. If the latter are to be of any use to the former, we must find ways of designing them that can safely manage both the performance and complexity issues. In general, parallel methods lead to implementations that are simpler than their serial equivalents, since they build up their complex behaviours in the same way as the real-world (i.e. through the simple interaction of independent and simple entities). They also lend themselves to high-performance execution since their parallel components offer many possibilities for parallel execution on multi-processors.

However, understanding and controlling the global properties of networks of ‘simple interactions’ between processes requires an additional set of skills from the parallel system engineer. The most subtle of these skills lies in guarding against accidental data-loss during process interaction and partial (or global) deadlock/livelock. For an arbitrary

---

1 occam is a registered trademark of INMOS Limited.
parallel design, there is no mechanical way to induce, from a complete knowledge of the local process interactions, the overall behaviour of the network. Many designs give rise to these problems in ways that are non-deterministic – i.e. dependent on the scheduling order of processes or the relative speed of processors. When this happens, the benefits claimed earlier for the relative simplicity of parallel design will have been lost and parallelism will appear to be a very difficult and dangerous tool to master.

Fortunately, there are disciplines of parallel design that result in systems whose deadlock/data-loss properties can be analysed, although not necessarily in a completely systematic manner (i.e. human assistance may still be needed). Foremost of these are the occam/CSP rules: process interaction only via synchronised message passing and no shared variables! These eliminate accidental data-loss because data is only transferred when space has been allocated to receive it and data cannot get zapped behind your back. Asynchronous communication requires either infinite buffering or a high-level error-recovery protocol to guarantee its security, both of which imply serious run-time penalties. A shared ‘variable’ does not have the simple engineering semantics of a variable (e.g. consecutive readings of its value ought to produce the same result) and requires very careful management. This management (e.g. semaphores) operates at too low a level and badly compromises the overall simplicity and, therefore, security of the parallel approach.

Synchronised message-passing leaves us only with deadlock/livelock and some special efficiency issues to resolve before the distributed functionality of a simple parallel design can be accepted. The efficiency problems relate to processes being unable to continue with useful work whilst awaiting synchronisation and the actual message-transfer costs. The first is avoided by having sufficient ‘Parallel Slackness’ [1] in the design (i.e. always having other processes waiting to run whenever a currently executing process becomes blocked) combined with a very low context-switch overhead (less than a micro-second on a transputer). The second can be avoided by not having too much ‘Parallel Slackness’ in the design (e.g. by using systematic ways of reducing parallelism [2]) and hardware support for simultaneous data-transfer and computation (e.g. as provided by a transputer network[3]).

This paper concentrates on paradigms for parallel design that yield systems whose operations are automatically free from deadlock.

2 Verifying the Absence of Deadlock

The choices made in the design of the occam multi-processing language mean that avoidance of deadlock (and livelock) is the responsibility of the occam engineer. In safety-critical applications, this responsibility must not be evaded since the consequences of a broken system are intolerable. occam does not allow the run-time detection of (and, hence, recovery from) arbitrary deadlock/livelock conditions, since the necessary run-time checks would require global information on the state of the network and would, therefore, be inefficient and unscalable. Instead, we must verify as formally as possible

\footnote{In general, we should err on the side of too much parallelism in our design - removing parallelism is much easier than adding it!}

\footnote{The T9000/C104 architecture provides a dramatic increase in this support compared with the earlier T2/T4/T8 series. The proposed Chamaelion design from INMOS takes this considerably further.}
that our system is free from such errors as a normal part of the design process – an ‘off-line’ activity with no run-time cost!

To ensure deadlock-freedom, we must show that there is no state into which the system can get from which no further action (i.e. communication or, perhaps, termination) is possible. Livelock-freedom is a little stronger: there must be no state from which all \textit{external} communications may be refused, even though internal activity may continue indefinitely.

The number of states in a multi-process network is bounded by the product of the numbers of the states in each component process. Even for quite modest systems (e.g. a 10 process network, where each process has 10 states), we can have the order of a billion states to check out! For almost all practical systems, exhaustive testing by generating each possible state is impossible.

Instead, our verification must consist of an exhaustive reasoning through all possible ‘categories’ of state, where the number of different categories is small. The choice of categories will depend upon the nature of the system being analysed and, in general, will require intelligent insight into its behaviour (a classical example of this being the proof of deadlock-freedom in the ‘Dining Philosophers’ problem \cite{3}). This type of verification cannot be automated!

This paper reviews some restricted sets of rules for parallel design that have the happy property of guaranteeing deadlock-freedom (because general theorems about their properties can be proved). In this case, verifying the absence of deadlock devolves to checking that the design rules have been followed – a mechanical process that can be automated with low computational cost. We claim that these design rules are suitable for most high-performance high-complexity applications.

3 Know Your Enemy

Whilst \textit{occam} cannot prevent its systems being mis-programmed and reaching a deadlocked state, it does make us well aware of the danger. It even goes so far as to provide a language primitive that explicitly generates it – \texttt{STOP}!

![Figure 1: A Deadlocked Process](image)

Consider the process in Figure 1, where the \texttt{compute} fold contains no communication. It performs no useful function, since there is no way for its environment to obtain the results of any of its internal computations. Nothing it was programmed to do in its \texttt{anything} fold, which may have included external communication, will ever take place. Worse than this, it is a danger to its environment since any committed attempt to communicate with it will deadlock the process that makes the attempt!

The process in Figure 2 cannot be distinguished by its environment from the process of Figure 1, even if the internal computations are completely different. The Figure 1
and Figure 2 processes are semantically identical – both represent deadlock and are useless and dangerous to their environments.

The processes in Figure 3 and Figure 4 represent livelock. To their environments, they behave in just the same way as the earlier deadlocked ones: they refuse all communications and compute nothing that can be accessed! From the point of view of processor resource, they are slightly worse. At least, the deadlocking processes eventually stop executing and just sit upon some memory resource – the livelocking processes continue to burn up computation resource as well!

All these processes offer the same semantic threat to their environments. They are entities that will not respond to any external enquiry and will deadlock any external agent that carelessly makes one – spreading the area of contamination. High-performance applications cannot tolerate such dangers. At best, time on a very expensive machine will have been wasted. At worst, people will be killed.

This may seem like a good reason to drop parallel algorithms altogether – or, at least, the occam model of it. Unfortunately, the alternatives are so much worse! Our ability to manage serial algorithms does not scale with their complexity. Mis-programmed parallel systems based upon asynchronous communication primitives or shared variables simply
become corrupt, but appear to carry on ‘working’! It is much better for erroneous systems or sub-systems to jam or chatter away harmlessly to themselves – at least we know that those parts of our application with which we are in touch are inviolate.

For safety-critical applications, we can build logical ‘fire-walls’ between physically separate sub-systems (that can independently fail for hardware or software reasons), so that failure in one will not bring down its neighbours [4][5]. Such fire-walls will contain an explicitly programmed asynchronous communication (constructed from synchronised ones), but for which the necessary data-loss is deliberate and controlled.

**occam** gives us excellent visibility of its communications and guarantees that each of them individually is secure. That makes a pretty good place from which to start looking for general results about deadlock/livelock.

## 4 Client-Server Networks

The problem with the deadlocked process in Figure 2 is that each of its sub-processes committed themselves to different communications with each other. The opposed directions of these communications is irrelevant – deadlock would still result even if both internal channels flowed the same way (one process trying to speak down one channel and the other listening on the wrong one). If we are going to allow multiple channels between processes (and, in general, we must), we need to impose some discipline on the way they are used. Two such regimes are the client-server principle (this section) and I/O-PAR (next section).

![Figure 5: Client-Server Communications](image)

The client-server principle relates to the pattern of communications across a collection of channels joining two processes. It is an attribute of this channel bundle, rather than the processes, and defines an ordering on the connection that is independent of the direction of the data-flows within it. One end is labelled the client and the other is called the server. The processes attached to these ends much conform to certain behaviour patterns when using the connection. Figure 5 shows a simple example.

### 4.1 Client Behaviour

A client-server transaction is always initiated by the process at the client end (process A in Figure 5). This will be a communication\(^4\) (that may be just a signal carrying no data) down a distinguished channel in the connection, called the claim-channel. (In Figure 5, request is the claim-channel and may well carry some data.)

We assume that this claim will always be accepted within a finite time (see 4.2 below). If we are in a time-critical sub-system of a real-time application, we will need to be told an upper bound to this acceptance time.

\(^4\) Usually, but not necessarily, this will be an output.
At this point, the processes at each end have synchronised at the start of their respective \textit{client-server} transaction routines. They complete the transaction with a finite sequence of communications using any of the channels in the connection set – data may be transferred in either direction. The precise order of these communications may be pre-defined or data-dependent. During this transaction sequence, the \textit{client} process may perform further relevant computation (provided this always terminates), but is not allowed to attempt communication outside the \textit{client-server} connection set. The \textit{client} may assume that all transaction communications (within the set) will complete within a bounded time.

In practice, the rich and varied message structures afforded by \textit{occam PROTOCOLs} mean that only one channel in each direction is ever needed to support such transactions (e.g. the \textit{request} and \textit{answer} channels in Figure 5). It is advisable to keep the transaction sequence as simple as possible to avoid mis-programming. A common form is just a single reply to the opening signal – for example, the transaction routine in process A may be simply:

\begin{verbatim}
SEQ
  request ! request.parameters
  answer ? results
\end{verbatim}

Note that some \textit{client-server} connections may only need to transfer data in one direction. If this is from the \textit{client} to the \textit{server}, only one channel will usually be needed. If it is from the \textit{server} to the \textit{client}, a \textit{claim}-channel carrying a data-less signal from the \textit{client} to the \textit{server} will also be needed (to open the transaction).

\subsection*{4.2 Server Behaviour}

The process at the \textit{server} end of a \textit{client-server} connection must always accept a \textit{claim} from that connection within a finite time. The simplest way to control this is to implement the \textit{server} as an ‘interrupt-handler’ for that \textit{claim} – i.e. something that remains dormant (apart from some internal housekeeping) outside each transaction. For example:

\begin{verbatim}
SEQ
  ... initialise internal state
  WHILE servicing
    SEQ
      request ? parameters -- the claim
      ... compute required information
      answer ! information -- end of transaction
      ... update internal state
\end{verbatim}

\textit{Server} processes are allowed to service more than just one \textit{client}\textsuperscript{5}. For example:

\begin{verbatim}
SEQ
  ... initialise internal state
\end{verbatim}

\textsuperscript{5}In which case, the \textit{claim} channels all have to be inputs to the server.
WHILE servicing

ALT i = 0 FOR SIZE request
    request[i] ? parameters -- claim
SEQ
    ... compute information
    answer[i] ! information -- end of transaction
    ... update internal state

This code refers to the server process in Figure 6. Each transaction is 'atomic' in the sense that it cannot be interrupted with other service transactions — a client has exclusive access to the server whilst it is being served. [Note that this code is not quite sufficient, since there is no guarantee that a particular client may not be starved by the demands of an especially active colleague. However, this is easily dealt with by standard techniques for a 'fair' ALT[6] and is not discussed here.]

The compute and update folds in the above server codes must, of course, be guaranteed to terminate. Unlike client transaction routines, we do allow servers to communicate during a transaction — they may need to obtain information not held locally to satisfy the client request. Communication may therefore take place within the compute fold, but only as the client-end of a client-server connection — see Figure 7.

For example:

{{{ compute information
IF
    ... we can handle request locally
    ... compute information here
    ... we need outside help
SEQ
    -- client transaction

Figure 6: Server for an Array of Clients

Figure 7: A Server also Acting as a Client
Thus, on some of its client-server connections, this process acts as a server and on others as a client. Client transactions may be embedded within a server transaction (but may also be done elsewhere). Since we assume that client transactions always terminate within a bounded time, we can maintain our guarantee of bounded response times to our own clients.

4.3 Client-Server Deadlock/Livelock

Deadlock/livelock analysis of a network of processes communicating solely through the client-server paradigm is particularly easy — see [7] for full details. Each client-server connection defines an ordering between two processes.

Client-Server Theorem

Any network of client-server connections that is acyclic with respect to the client-server ordering is deadlock/livelock-free.

Proof

By induction over any topological sorting (client-server ordering) of the processes. The hypothesis is that each process will accept any client signal within a finite time.

The bottom process in the topological sort makes no demands (as a client) to any other process in the network. Each service request to this process is handled either locally or by a client demand outside the network. We assume that external demands are serviced properly, since we are only concerned about deadlock arising within the network. Hence, we deduce that each service requested is completed in a (computable) finite time and the process will loop around ready to accept further claims. This is the base case of the induction argument.

The induction step has to establish the hypothesis on any other process, assuming it for all processes below it in the client-server ordering. In this case, each service request is handled either locally or by a client demand to a process lower in the ordering or outside the network. By induction and the same reasoning as for the base case, all these handling methods terminate successfully and, again, we deduce that this process will get ready — within a finite time — to accept further claims.

This completes the induction and we have the hypothesis for all processes in the network. Hence, external clients will always be serviced. External servers only attempt to communicate with processes in the network during transactions initiated by those processes and which those processes locally guarantee to complete. Thus, no external communications are ever refused by the network and it is deadlock/livelock-free.

Q.E.D.

4.4 Cross-Mounted Servers

Figure 8 shows a network that breaks the rules for the client-server theorem. For example, server.A and server.B may be separate file-servers, each with their individual
sets of *clients*. Sometimes, the files required by a *client* attached to *server.A* are only available on *server.B*. In this case, *server.A* has to become a client of *server.B* to obtain the necessary information. Similarly, *server.B* may need to obtain the services of *server.A* on behalf of its *clients*.

![Diagram](image)

**Figure 8: Cross-Mounted Servers**

So long as these cross-service demands do not occur too heavily, this system may pass its validation trials and start being used. Trouble will eventually arise when *server.A* and *server.B* both become committed to making a *claim* to each other’s service, as a result of two unfortunately timed local requests. At this point, neither *claim* ever gets accepted and the file-server network deadlocks! [Note: this is a true story!!]

![Diagram](image)

**Figure 9: Deadlock-Free Multiple Servers**

A deadlock-free design is given in Figure 9, where there is now no cycle in the *client-server* ordering. External *client* processes are no longer specifically attached to either *server.A* or *server.B*. Instead they connect through *multiplex* processes that
route transactions to the correct file-server, possibly by first consulting an independent name.server to decide which one to use.

4.5 The Clock Problem

Figure 10 shows a clock process that signals on its tick channel at a regular time interval. This time-interval is initialised by sending a value down its reset channel and can be changed by sending further values down the reset:

![Clock Diagram](image)

**Figure 10: A Clock with Speed Control**

```
PROC clock (CHAN OF BOOL tick, CHAN OF INT reset)
  TIMER tim:
  INT t, gap:
  SEQ
    reset ? gap
    tim ? t      -- only read the absolute time once
    t := t PLUS gap
  WHILE TRUE
    PRI ALT
      reset ? gap
      SKIP
      tim ? AFTER t
      SEQ
        tick ! TRUE
        t := t PLUS gap
    :
```

This process acts as a server for its reset channel (and for its internal TIMER) and as a client on its tick-channel (which it assumes will be taken before its next tick is due).

Suppose we want to use clock in an application where the process being stimulated by the ticks needs to control that tick rate – see Figure 11.

![Clock Diagram](image)

**Figure 11: A Clock Controlled by its User**
Suppose this user is simple enough to be implemented as a serial process. We have a cycle in the client/server ordering and, therefore, no guarantee against deadlock! In fact, the potential for deadlock is very real.

Server processes are allowed to make client calls for its own reasons or on behalf of its own clients. Under the conditions of the theorem, this does not matter since we know that those calls will always be answered. The 'down-time' of a server is the period between its decision to make such a call and actually making it. During this down-time, it is dependent upon other servers to enable it to complete its client transaction and resume its own role as a server.

For the clock process, this down-time runs from accepting its time-out guard and attempting the tick – say about 3 micro-seconds on a transputer. The user process can keep its down-time similarly small, since it can always check its tick channel just before attempting a reset. If these two down-times ever overlap, there will be deadlock⁶.

If the average clock rate set by the user were (say) one tick every 12 milli-seconds, the probability of an individual reset causing deadlock is about \((3 + 3)/12000\), which is 0.05%. If the user adjusted the clock rate on average once every 10 seconds, after nearly 3 hours continuous operation (i.e. 1024 resets) our chance of not being deadlocked reduces to 60%. After 24 hours, we have a less than 1% chance of still being alive.

The above rate of deadlock ought to show up under reasonably persistent system testing. However, suppose the parameters of this system were somewhat different. Suppose that the reset were generated by human (e.g. pilot) intervention with the user process and that this happened very rarely – say once every 24 hours flying-time. In this case, we would have to wait 2 flying-years before our deadlock chances reached 60%. This deadlock would easily be missed in testing and only show up several years into the actual service of the plane!! Such a rare deadlock is truly deadly. The way not to 'cure' this problem is to add extra buffering in the feedback loop – see Figure 12.

![Figure 12: Buffered Reset Connection](image)

The id process cycles through waiting for a user.reset and passing it on to the clock. It acts as a server to the user.reset channel and a client to the reset. We still have a cycle in the client/server relationships and no guarantee against deadlock.

⁶Note: we are reasoning here about occam processes and can make no assumptions about the mechanisms of any multi-processing scheduler (should those processes be allocated to the same processor). Relying on such properties to verify the absence of deadlock is implementation dependent and, therefore, unsafe.
However, the chances of deadlock are greatly reduced. The `user` process must issue a `user.reset`, change its mind fairly quickly and issue another one. The first `reset`, propagated through `id`, must arrive at the `clock` during its `down-time` (probability 0.025%). Also, the `tick` generated at the end of that period must reach the `user` during the `down-time` for its second `user.reset` (probability 0.025%). Thus, the probability of deadlock arising from a randomly generated `reset` is about 0.000006%, which still implies a 50% chance of failure after about six months flying-time (assuming `reset`s averaging once every ten seconds).

If we programmed the `user` to promise some minimum `tick`-service time between any consecutive (`user.reset`-generating) `down-times`, then deadlock could be avoided. This minimum time must cover the propagation delay in routing the `reset` through `id` plus the `down-time` of the `clock` plus the service time for the `reset` by the `clock`. But depending upon such real-time analysis for verifying this fundamental property is complex and hard to maintain. Any change to the speed or number of processors in the system, the scheduling algorithm or number of processes, the implementation of `clock` or `id` would require re-calculation of the timing constraints to be used within `user`. Administration of such side-effects<sup>7</sup> is not simple engineering!

The correct way to solve this problem is to use a design that meets the conditions of the `client-server` theorem. The system will then be free from deadlock regardless of the details listed at the end of the previous paragraph. Figure 13 applies two standard `occam` idioms: an auto-prompter (`prompt`) and an overwriting buffer (OWB), here with capacity for just one item.

![Diagram](image)

**Figure 13: Secure Reset Connection**

The `prompt` cycles through the sequence `<req!, ans?, reset!>` and acts only as a `client` on its two connections. The OWB is a pure `server`, accepting `user.reset`s at any time (which may overwrite previously sent values) and requests (provided it is not empty).

Although there are two obvious cycles of data-flow in the system, there is no cycle in the `client-server` ordering and, therefore, no deadlock. The OWB/prompt sub-system provides an `asynchronous` connection for propagating the `reset` signal. The data-loss inevitable from such a link is explicitly managed within OWB — i.e. it is under control. This data-loss (of an earlier `reset` value) is no problem in this application since it is only caused by the arrival of a change-of-mind message. The only process that may be blocked indefinitely is `prompt`, as it attempts to request a reset signal that may never

---

<sup>7</sup>Note that adding buffering is similarly no solution to the cross-mounted server deadlock of 4.3
be sent. But prompt does not have to promise service to anyone and so the design is immune to this sacrifice. This system will fly forever!

4.6 The Farm Worker

Client-Server Closure

Any collection of processes that communicate only using the client-server paradigm and has an acyclic topology (with respect to client-server relationships) itself communicates with its environment by the client-server paradigm.

Proof

We just have to show that service is guaranteed to its external clients. This is an immediate corollary of the client-server theorem, which promises that the collection is deadlock/livelock-free.

Q.E.D.

![Diagram](image)

Figure 14: Farm Worker Harness

Figure 14 shows the implementation of a worker in a simple processing ‘farm’. It consists of four standard ‘harness’ processes (buffer, prompter, catch and mux) and a work process that is special to the problem being farmed. The buffer is a pure server, servicing its external link (from which new work-packets arrive) and request-connections from work and prompter.

The prompter, as before, is a pure client, making demands on the buffer and forwarding anything it gets to the next worker in the farm (see Figure 15). The central work process has the same communication behaviour as the prompter – it is a pure
client, obtaining work-packets from buffer, processing them and outputting result-packets to mux.

The mux process services result-packets from its two input connections and multiplexes them (as a client) on to its external output link.

The catch process is a one-place buffer (like id), forwarding result-packets from the previous worker to mux. Its purpose is to service the external link in result so that it may operate in parallel with out.result. In fact, all four external links of worker may operate simultaneously (and their respective handlers operate at higher priority than work to let this happen whenever possible).

![Diagram of a farm](image)

Figure 15: A Farm

This design has no client-server cycle and worker is, therefore, deadlock/livelock-free and may be treated as a client-server component. A complete farm is shown in Figure 15, where the first worker omits the catch process and the last one omits the prompter. The worker-pipeline is a server to the farmer (who does not know which worker will service which packet) and a client to the harvester (who does not know which worker produced which result). We have a simple pipeline of parallel client-server connections and are, therefore, deadlock-free.

5 I/O-PAR and I/O-SEQ Networks

Consider a physical system whose components behave in a way determined only by their own state and those of their immediate neighbours (e.g. fluid flow, logic circuits, road-traffic, heart muscles, real-time control laws, bungee jumping, ...). Such a system can be precisely emulated by a network of communicating processes, each of which models one component, and whose topology exactly reflects that of the real system.

An I/O-PAR normal form is a process that has a cyclic serial implementation, except for its communications which always operate in parallel:

```
WHILE running
SEQ
    ... parallel i/o (once on all channels)
    ... compute
```

An I/O-SEQ normal form is similar, except that the parallel inputs are done in sequence with the parallel outputs:

```
WHILE running
SEQ
    ... parallel inputs (all input channels)
    ... compute
```
Components of the physical system described above may always be implemented in one of these normal forms. *I/O-SEQ* is usually applied in systems where the component interaction is not symmetric (e.g. logic circuits and control laws). In practice, the system domain is divided up ‘geometrically’ into regions containing similar numbers of component – the number of regions equalling the number of processors at our disposal. The components in each region are modelled together as a single normal form, with each one being placed on its own processor. A simple transformation (using the laws of *occam [10]*) is usually performed to overlap input/output with all possible computation (so we don’t have to pay the transfer fees):

```
WHILE running
  SEQ
  PRI PAR
  ... exchange boundary data (parallel i/o)
  ... compute on middle of region
  ... compute on boundaries
```

Note that the above is semantically equivalent to the pure *I/O-PAR* normal form - i.e. indistinguishable by any environment.

### I/O-PAR Theorem

Any network of *I/O-PAR normal form* processes is deadlock-free.

**Proof**

*Lemma:* any *occam* program with no ALTs is *deterministic* (in the sense of *CSP* [3]). [By definition, the parallel operator, ||, of *CSP* is deterministic. However, the hiding operator, \, may introduce non-determinism and the *occam PAR* operator automatically introduces hiding (for all internal communications). We leave it to the reader to verify that hiding all internal events between *CSP* parallel processes still leaves determinism intact.]

*Lemma:* if one infinite trace exists for a deterministic process, *all* traces can be extended – i.e. it is deadlock-free. [Again, this is left as an exercise.]

Clearly, any network of *I/O-PAR* processes contains no ALTs – so it is deterministic. If the network is scheduled as if there were a global ‘barrier synchronisation’ at the end of each normal form cycle, it is again clear that it will run forever – i.e. this gives an infinite trace. Hence, any trace resulting from any scheduling pattern can be continued.

Q.E.D.

A full proof may be found in [9], which also introduces the following notions.

**Definitions**

An ‘*I/O-R net*’ is any connected network of *I/O-PAR* and *I/O-SEQ normal forms*. 

An ‘I/O-SPANet’ is an I/O-Rnet such that there is no closed loop consisting only of I/O-SEQ normal forms and no path from external input to external output consisting only of I/O-SEQ normal forms.

Two processes are ‘p-equivalent’ if they cannot be distinguished by any I/O-Rnet environment.

**I/O-PAR/SEQ Closure**

Any I/O-SPANet is p-equivalent to an I/O-PAR normal form.

**Proof**

See [9].

Q.E.D.

Note that p-equivalence is weaker than trace-equivalence. A non-trivial I/O-SPANet cannot be serialised into an I/O-PAR normal form using transformations that obey the laws of occam. For instance, an I/O-SPANet with several I/O-PAR normal forms will allow traces in which some external communications have taken place many more times than others. A single I/O-PAR normal form will only allow traces in which the numbers of occurrences of each channel differs by at most one. Thus, we could create an environment in which the normal form deadlocks but the network does not. However, for all I/O-Rnet environments (which are the only environments for which they are designed) either they both deadlock (although not necessarily after the same trace) or they both do not deadlock.

**I/O-PAR/SEQ Theorem**

Any I/O-SPANet is deadlock/livelock-free.

**Proof**

Suppose it is not! Then there is a trace (of external communications) after which all further external communications may be refused.

Construct an I/O-Rnet environment to support this trace as follows. To each external channel of the I/O-SPANet, attach a pipe-line (‘spoke’) of I/O-PAR normal forms of length equal to the number of occurrences of the channel in the deadlocking/livelocking trace. Connect the end processes of each adjacent ‘spoke’ together to form a connected I/O-Rnet. This environment will accept the trace (with each normal form cycling by one more than its distance from the ‘rim’).

Apply this I/O-Rnet to its p-equivalent I/O-PAR normal form. Since the original network deadlocked/livelocked in this environment, so – at some stage – should the normal form. Since the normal form only has external channels and cannot cycle without using them, livelock is impossible – i.e. it must deadlock with the environment. But the environment consists solely of I/O-PAR normal forms – i.e. the combined system cannot deadlock (by the I/O-PAR theorem). Contradiction!

Therefore, the trace from which the I/O-Rnet was constructed cannot exist and the I/O-SPANet must be deadlock/livelock-free.

Q.E.D.
This closure property gives us design rules for constructing parallel emulators for a wide range of physical systems as a hierarchy of $I/O$-SPnets that are guaranteed deadlock and livelock free. Each net models the behaviour of a real system component. At the lowest level are simple normal forms emulating the simplest components of the system.

[9] gives (and implements) a constructive serialisation from an arbitrary $I/O$-SPnet to its $p$-equivalent normal form. This has practical significance since it allows us to automate the production of optimised parallel code for any chosen ‘domain decomposition’ of the system. Thus, distributing the original finely-grained massively parallel design on to any (smaller) number of parallel processors can be mechanised.

6 Hybrid Networks

The client-server paradigm supports regular designs (such as farming) but is especially useful for the safe construction of irregular networks that model complex system interfaces. The $I/O$-PAR and $I/O$-SEQ paradigm only supports the modelling of regular systems for which domain decomposition is the appropriate strategy. However, if we want the latter to be more than just an off-line activity (e.g. we want to have visual feedback on the emulation and/or interactive control over its progress and system parameters), we will need to build a rich interface. The right tool for this is client-server, which means that we must be able to design safe client-server and $I/O$-PAR hybrids.

Hybrid nets are safe provided we can maintain a separate view of the different types of network as intact sub-systems, even though the same processes may be part of differently typed sub-systems.

![Figure 16: A Hybrid Component](image)

For example, Figure 16 shows a worker process, rope, for modelling a section of an elastic rope in the simulation of a ‘bungee jump’[11]. It is $I/O$-PAR on its horizontal links, but acts as a server on its control line and client on its dump channel:

```plaintext
WHILE running
  SEQ
  PRI PAR
  PAR
    ... i/o-parallel horizontally
    ... poll control for new parameters
    ... compute on middle
    ... compute on boundaries
    ... dump current state (occasionally)
    ... update current parameters (if necessary)
```
If we ignore the control and dump channels, rope is an I/O-PAR normal form and may be replicated (horizontally) to form a deadlock-free model of the complete rope. We may now assume that the horizontal communications always succeed.

Each rope process is now connected through its dump channel to a client-server network that multiplexes the state information to a graphics device. Each rope process acts as a pure client to this sub-network and graphics is a pure server. So long as this sub-network is built according to the rules of the client-server theorem, we know that it will never refuse the dump request from rope.

[Note: rope need not make a dump on every cycle, since that may take too much time and block progress on the computation. Part of the graphics delivery network may reside on the same processor as rope, so that the physical output of state information from the processor can be executed in parallel over several cycles of rope. Only a one-place buffer (e.g. id) is needed for this.]

As far as the control line is concerned, the rope process now looks like a 'busy-waiting' server. Whenever a control message arrives (carrying new parameter information for the simulation – e.g. time-step, rope elasticity, gravity, viscosity or even new position and velocity data for some or all of the rope particles), the rope process guarantees to complete its current cycle and accept the signal.

We may therefore place above each rope process a client-server network that delivers the control information. The rope process now appears to be a pure server to this sub-network and will not deadlock it.

Figure 17: A Hybrid Network

Figure 17 shows the complete system. It is a hybrid of two separate client-server networks joined along an I/O-PAR seam. The I/O-PAR pipeline handles the main computational functions and the client-server nets deliver interactive control and graphics.
visualisation. The user interface process will have a rich internal client-server structure to manage simply its diverse responsibilities. Two of its internal processes will also have I/O-PAR normal forms with respect to their connections to the ends of the I/O-PAR pipeline.

In general, it is quite safe for a process to have client connections, server connections and I/O-PAR (or I/O-SEQ) connections — but they must, of course, all be different. So long as the individual sub-networks constructed from these hybrids satisfy either the rules for the client-server theorem or are I/O-SPnets, the overall system will remain free from deadlock and livelock.

[Note: the system in Figure 17 may be directly configured on to existing T2/T4/T8 networks using the automatic ‘virtual channel routing’ option provided by the latest occam Toolset from INMOS. Alternatively, a special-purpose routing network (constructed to a deadlock-free client-server design) can be added. For T9000/C104 networks, all the virtual channels and routers needed for direct execution of the system are provided by the hardware.]

An example of a hybrid network that does not follow these rules is the model of collegiate life described in the ‘Dining Philosophers’ story. Each philosopher is an I/O-PAR normal form who cycles through ‘thinking’ (computation), ‘grabbing the forks’ (I/O-PAR signals), ‘eating’ (computation) and ‘putting down the forks’ (I/O-PAR signals). [Note: some versions of this tale have the philosophers picking up (and putting down) their forks in a particular order — in which case, I/O-PAR is replaced by I/O-SEQ.]

On the other hand, each fork is a pure server handling signals from the philosophers on either side. A transaction covers the period from being ‘picked up’ by one of the philosophers (this is the claim signal) to being ‘put down’.

The college consists of a ring of alternating philosophers and forks. For each connection, one side thinks it is part of an I/O-PAR communication and the other thinks it is part of a client-server one — confusion reigns! None of the conditions required for the theorems that guarantee deadlock freedom apply. The college deadlocks.

7 occam3 Support for Client-Server Networks

occam3 [12] provides language support for client-server transactions. This enables the compiler to ensure that individual clients and servers conform to the behaviour patterns specified for them in sections 4.1 and 4.2, as well as to generate much faster codes for their implementation. We have both added security and added performance.

First of all, occam3 introduces channel-types, which allow us to group together as a single unit the various channels (carrying different protocols and directions of data-flow) that make up a single client-server connection.

Secondly, it allows a particular instance of such a connection to be shared between a single server process and any number of clients — see Figure 18.

Sharing automatically introduces a claim-channel into the connection (or something equivalent that performs this function). A client initiates a transaction by making a claim on the shared connection. When this claim is granted (by the server), the client has exclusive use of the channel resources provided by the connection to interact with the server. For the shared connection called X.bus in Figure 18, a client transaction looks like:
CLAIM \texttt{X.bus}

... use \texttt{X.bus} to interact with the server

The language will not allow \textit{clients} to use \texttt{X.bus} channels outside such a CLAIM. Inside the body of the CLAIM, only \texttt{X.bus} channels may be used — no other synchronising events are permitted. When the body of the CLAIM terminates, the transaction is complete and the client yields control of the connection. This sequence of claiming, using and releasing the shared medium cannot be violated.

The \textit{server} process manages in-coming \textit{claims} on a FIFO-queue. If the queue is non-empty, it must grant access on the \texttt{X.bus} to the process at the head of the queue within a bounded time. A \textit{server} transaction looks like:

GRANT \texttt{X.bus}

... use \texttt{X.bus} to interact with a client

As before, the \textit{server} may only use \texttt{X.bus} channels within such a GRANT. Unlike \textit{clients} though, a \textit{server} may indulge in other synchronisations within the body of the GRANT (like making a CLAIM to another \textit{server} on behalf of its current \textit{client}). When the body of the GRANT terminates, the transaction is complete and the \textit{server} regains its authority on \texttt{X.bus} (and should check its queue again). This sequence of granting, using and regaining control of the shared medium cannot be violated.

As a result, \textit{clients} have mutually exclusive access to the \textit{server}. The \textit{server} ‘fairly’ grants audiences to its \textit{clients} on a first-come-first-served basis. Each GRANT is managed with a constant overhead — we no longer have the ALT-penalty, whose overhead grows linearly with the number of \textit{clients}. [However, ALT is retained in \texttt{occam3}, since a FIFO management of competing events is not always what we want. Also, GRANTs may be used as guards in ALTs so that a \textit{server} may service several sets of \textit{clients} on separate shared connections — see Figure 20.]

However, \texttt{occam3} does not do everything for us! We still have to perform the higher-level checks of ensuring there are no cycles in the \textit{client-server} ordering. We also must
check the low-level algorithm in each server to guarantee that it waits on its service lines in each cycle. [Although, for simple loops, occam3 provides a SERVER process declaration that supplies this guarantee – but we shall not examine this here.]

7.1 A Simple occam3 Farm

The abstraction of a shared connection is not restricted by physical configuration. The clients and their server may be distributed over any number of processors. T9000/C104 configurations will support a distributed shared connection in hardware.

Figure 19: An occam3 Farm

Figure 19 shows an occam3 design for a farm of worker processes. It is extremely simple (as it should be) and quite takes away the fun we used to have implementing such things. Each work process, W, is a pure client (as it was in Figure 14), but this time it makes claims directly on the (remote) farmer and harvester, who are now both servers (compare with Figure 15).

The farmer and harvester simply wait on their respective service lines. The farmer responds to a claim by outputting another work-packet. The harvester responds to a claim by inputting a result-packet. As with the earlier farm, neither needs to know (and does not care) with which worker it is dealing.

The worker, W, simply cycles through claiming the farmer, processing the work-packet and claiming the harvester. To save being blocked whilst claiming, it might buffer one work and one result packet internally (either by including two one-place buffer processes or turning itself into an I/O-PAR – or CLAIM-PAR – normal form).

Clearly, the farm has no client-server cycle and is therefore deadlock/livelock-free.

7.2 An occam3 Farm of Servers

A more sophisticated farm is shown in Figure 20. This time the workers are themselves servers, S, offering their services to the world-at-large. The farmer is now a manager, M, controlling the allocation of farm services to clients, C. Clients want to be given the first available server and do not mind which one they get.

The manager, M, services two shared connections: a public one for use by clients requesting service and a private one known only to the servers. When a server is free, it reports to the manager by making a CLAIM on its private line. The manager lets these queue up until a client makes a CLAIM on the public line. To service the client, it waits for a server CLAIM (possibly with a time-out – it’s up to the client). The server tells the manager its name and issues a password (end of server transaction) and these
Figure 20: A Farm of Servers

details are forwarded to the client (end of client transaction). The client then calls on
the named server, uses the password and gets the desired service.

If there is a recession (not enough clients), the servers will end up queueing for
work. If there is a boom (too many clients), the clients will have to queue for service.
The queues are automatically managed by the mechanics of the shared channels – no
programming is needed!

This server farm can be made highly secure against client misuse, especially if occam3
CALL channels can be used for the public server connections. CALL channels should
be used for client-server connections when we do not need computational support or
independent decision making from the client during an individual transaction. The
client process is suspended during a CALL, whilst the server process runs the transaction
and has been given certain access rights to certain client resources (e.g., data-structures).
Because only the server process is active, the transaction cannot deadlock because of
mis-programming between the client and server.

The benefit now is that the server, S, can immediately terminate a client CALL if an
incorrect password is supplied (or at any time during the transaction if it so chooses).
The client, no matter how it has been programmed, cannot resist this! Since each server
generates a new password each time it reports to the manager, a client cannot by-pass
the manager and try to pick up a server directly. Even when it has been allocated a
server, it cannot hang on to it indefinitely!

The server may also set an internal time-out on the validity of its password, starting
from the end of its transaction with the manager. If the client does not CALL in time,
the server goes back to the manager. In this way, a client cannot acquire a server in
advance of its actual need and sit on it – an anti-social pattern of behaviour!!

This multi-client/multi-server system contains no client-server cycles and is safe.
Two paradigms have been presented for parallel system design using synchronised message-passing that guarantee freedom from deadlock and livelock.

I/O-PAR designs cover the computationally intensive core of super-computing or high-performance embedded applications that model physical phenomena through ‘domain decomposition’. They may be combined with client-server networks to provide interactive control and visualisation. They may be tuned automatically to produce optimised codes for different numbers of processor.

Client-server principles cover the design of processor ‘farms’ for high-performance applications whose parallel decomposition is logically regular, but requires dynamic load-balancing. They are also suitable for the safe and maintainable construction of networks with irregular topology. These greatly simplify the implementation of ever-increasing levels of sophistication in real-world interfaces (such as an X-window server [7][13]), where the specifications demand similarly irregular functionality.

The analysis of client-server principles in this paper is not complete. In particular, its closure property is too strong. For example, if we combined the prompt and OWB processes of Figure 13, we would get a component with the same client-server connections as the id process of Figure 12 — and an apparent cycle in the client-server ordering!

We need to introduce the notion of dependency between server and client connections to the same process. For a serial process, the server connections are always dependent upon the bounded acceptance of all client transactions that are attempted — any failure will break the promise it must maintain to its own clients. However, the user.reset server connection on a combined prompt/OWB process is not dependent on the reset client connection being accepted.

Server connections can also become dependent upon other server connections! For example, when the buffer process in the worker harness (Figure 14) is full, the buffer refuses service on its in.work connection. This connection then becomes dependent on its sibling connections (both server ones) being called in order for its own service to be resumed. Tracing this through, we see that the in.work server is dependent upon the out.work and out.result clients. The in.result server is dependent only upon the out.result client.

These dependencies should be part of the specification of a client-server component and the rules need to be worked out for how these dependencies are inherited by ‘properly’ composed client-server networks. The ‘proper’ composition rules must be relaxed so that they only forbid cycles of client-server connections that are all pair-wise dependent.

However, the current rules — although too restrictive in the case of client-servers — are flexible enough to allow the design of a very wide range of high-performance and safety-critical applications.

Parallelism gives us simplicity and physical concurrency. Synchronised message-passing and no shared variables give us secure communications (i.e. no unexpected data-loss). Parallel slackness, auto-serialisation, microsecond context-switches and hardware support for concurrent computation-with-communication give us efficiency. The client-server and I/O-PAR paradigms, with their respective closure rules, give us freedom from deadlock and livelock (without the need for state analysis of the system design).
occam and the transputer have always supported all these principles. occam3 and the new T9000 family support them to a considerably greater depth. The need for occam3 to be implemented in full is crucial for the widespread development of these ideas and the safe exploitation of High-Performance Computing in general.

9 Acknowledgements

We are very grateful to all our colleagues who have talked through (and used!) these ideas. Special thanks are owed to Andy Bakkers, David Beckett, Vedat Demiralp, Jon Kerridge, Roger Peel, Dyke Stiles, Shane Sturrock and Steven Turner.

References


