On the Serialisation of Parallel Programs

P.H.Welch and G.R.R.Justo
Computing Laboratory, University of Kent at Canterbury, CT2 7NF.

Abstract. This paper argues that one of the key techniques for making the most efficient use of multi-processor architectures is the *serialisation* of parallel code! Parallel algorithms are presented as having strong engineering merits that will form the natural basis for systems design in the future. *Parallelisation* of serial code is regarded as having only short-term value (for ‘‘dusty-decks’’, whose correctness cannot be verified) as well as being mathematically intractable. *Serialisation*, on the other hand, is much easier to automate and can be profitably employed today. Several serialising transforms for *occam* processes are presented and applied to various simulation and image compression tasks.

2 .nr H1 1 0. Introduction

This paper reviews and interprets some of the *practice* and *experience* of programming parallel computing systems we have obtained at the University of Kent over the past six years. We present in a semi-formal, but disciplined, manner some of the practical skills we believe should be regularly applied to the development of parallel programs. We are by no means alone in our beliefs. We are alarmed, however, that they do not seem to be recognised by the ‘‘mainstream’’ computer science community.

The chief lessons are these :–

- parallelism is a major structuring method that enables us to manage complexity (in the design, verification and maintenance of systems);
- system design, therefore, should be (highly) parallel from the start;
- in general, there should be many more logical processes than physical processors (‘‘parallel slackness’’);
- to optimise performance, parallel sub-networks running on individual processing nodes may need serialising. Tools to automate (or, at least, help in) such serialisation are badly needed.

Expressed positively like this, these do not seem to be too contentious. It is the negative conclusions we can draw from them, however, that seem to raise eyebrows :–

- design standards that exclude parallelism also exclude security for complex applications. This leads to growing losses — both financial and human life;
- efficient and robust systems cannot be built by ‘‘first getting them to work serially on one processor’’ and then ‘‘parallelising’’ them;
- existing ‘‘dusty-deck’’ codes, that represent massive financial investments that ‘‘cannot afford to be wasted’’, also represent massive serial codes that are becoming unmaintainable and are certainly unverifiable. These are technical dead-ends — as commercial pressures will gradually make clear to all those who persist with them;
- tools to assist the parallelisation of large-scale serial code are very difficult to make, will be very expensive to buy and will not be needed by the time they are half-made to work.

Regardless of the reaction of your eyebrows to the above assertions, please read on!

1. Some Merits of Parallel Design

Parallelism does not extend the range of functions that can be computed. The *parallel* operator in *CSP* [0] is completely defined in terms of its *serial* choice operator. The only motive for its introduction is that it simplifies the expression (i.e. the ''programming'') of the behaviour of most processes (above a low level of complexity) and, hence, our ability to reason about them.

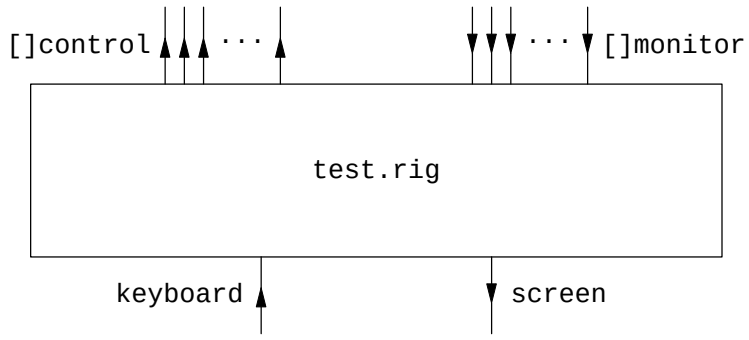
The parallel construct in *occam* [1] is directly based upon *CSP* theory and directly reflects the above properties. Parallelism (or, at least, *occam* parallelism) should be regarded as a high-level programming structure and used freely. It may be ''compiled'' down to low-level serial code (just as `WHILE` and `FOR` loops may be implemented by unstructured `GOTO`s), but that low-level code is almost always much more complicated and harder to understand. Nevertheless, this serialisation can always be done and sometimes there are good reasons for doing it — see below. The reverse operation, parallelisation, requires the ''de-compilation'' of low-level code back to high-level structures — an activity that never produces satisfactory results!

We are arguing the case for parallelism on the grounds that it simplifies and clarifies the development of complex systems — not that it makes them go faster! History supports this view. Software parallelism was first experimented with in the early 1970s in an effort to make operating systems work — or, at least, to make them work for longer periods between crashes! These systems were supporting uni-processor computers, so that the question of exploiting concurrency to improve *performance* did not arise. Indeed, a performance penalty (due to the overheads for managing the software concurrency) was cheerfully accepted if the overall *reliability* could be increased to tolerable levels.

We are very fortunate these days that parallel hardware lets us apply concurrency to increase the performance of computer systems. In our excitement over all the MIPS and MFLOPS that are now at our disposal, we must not forget the powerful benefits for clear thinking that were the original motivation for going parallel.

2. Some Designs Just Have to be Parallel

The following `test.rig` provides a user-interface for controlling and monitoring the state of a continuously running machine :-

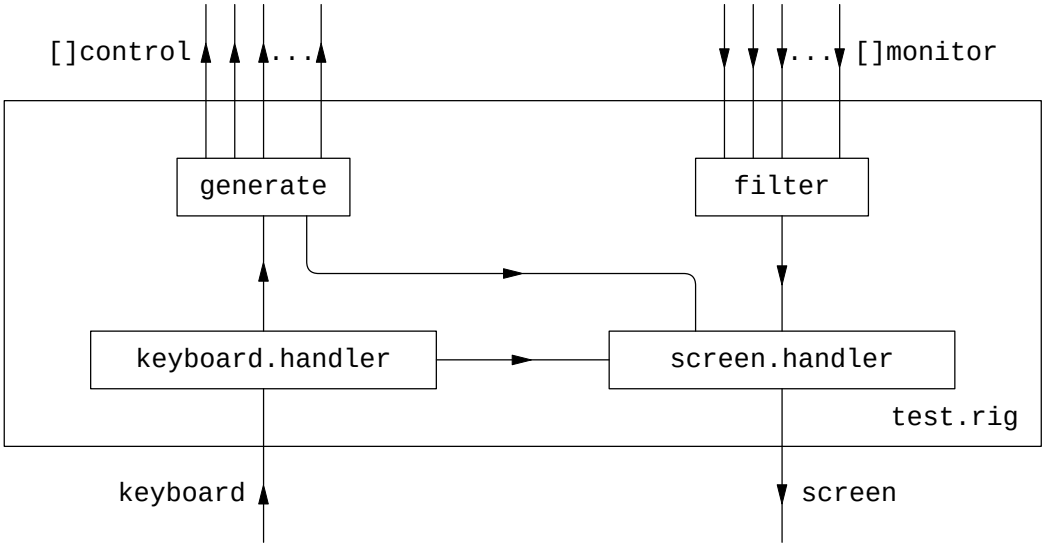


Its required behaviour is as follows :–

- the user supplies keystrokes to the `keyboard` channel and receives display information from the `screen` channel;

- responding to user keystrokes, the `test.rig` generates `control` messages to the machine under test and updates the user's display to indicate what it has done. Erroneous keystrokes ''bleep'' the user's display;
- at the same time, the `test.rig` receives continuous information from its `monitor` channels about the machine state. This information flow is too great to display in its raw form and has to be filtered and summarised before being dynamically presented to the user in some meaningful way;
- the user may freeze the display at any moment by pressing a ''pause'' key — the next keypress resumes normal operations.

The next figure describes a reusable design for the implementation of such a `test.rig`. It shows a natural parallel construction out of four processes — each one performing its own logically self-contained function.



The `keyboard.handler` :–

- validates and forwards characters from `keyboard` to `generate`;
- invalid characters (pressed by the user by mistake) are not passed on — instead an ''error'' signal is output to the `screen.handler`;
- if the ''pause'' character arrives, it outputs a ''pause'' signal to the `screen.handler`, waits for another keystroke and sends a ''resume'' signal.

The `generate` process :–

- receives validated characters from the keyboard;
- interprets these as instructions to modify an internal data-base recording the state of various control options in the machine under test;
- issues appropriate commands down the relevant `control` line;
- formats a display packet to reflect any changed control value and sends this to the `screen.handler`.

The `filter` process :–

- continuously receives data from its `monitor` channels about the state of the machine;
- filters this data by integrating it into a ''history'' database (internal to this process);

- reports meaningful summaries about changing machine state in display packets to the `screen.handler`.

The `screen.handler` :-

- multiplexes formatted display packets straight through to its `screen` channel;
- ‘‘error’’ signals from the keyboard are interpreted by ‘‘bleeping’’ the screen;
- a ‘‘pause’’ signal causes this process to lock on to its channel from the keyboard and await a ‘‘resume’’ signal — freezing further screen output;
- the `key.handler` signals take priority over display packets.

We claim that this design is much simpler than any equivalent serial one. Each process has responsibility for one distinct area of operation. Its data-structures are its own affair and its algorithm is expressed from its own (‘‘object-oriented’’) point of view — not that of an external controller. Strong engineering principles are followed in this design: processes have tightly controlled external interfaces (only channels) and high internal cohesion (with all design details private). Each process is now sufficiently simple so that a serial (*occam*) implementation is probably clearer than any natural language specification.

The same could not be said about any serial implementation for the whole `test.rig`! That would require an integration of the algorithms and data structures of the four processes into a single thread of control. Such an integration would invert their object-oriented character and gravely damage their clarity. Worse still, in order to maintain the same freedom to synchronise with its environment that the parallel implementation enjoyed, it would sometimes have to `ALT` across all its channels — both input and output! It must maintain this freedom — deadlock would threaten if it ever committed itself solely to output a control adjustment to any part of the machine that happened to be close to another part from which feedback monitoring was being obtained!! The output guards may be removed by further transformation [2], but it would now have got very obscure indeed.

It would therefore be very risky to attempt a serial implementation of the `test.rig`. The parallel implementation is the correct one — even though we never have any need or intention of distributing it over more than one processor!

3. Serialisation and General Purpose Parallel Computing

If you have one processing unit, then you have an excuse for trying to devise your algorithms with a single thread of control. If you have two processors, then two process logic would seem appropriate. If you have eight processors, you can make a case that the most effective way to exploit them to solve a particular problem is to program it up as eight parallel processes. What is not credible, however, is to replace each ‘‘eight’’ in the preceding sentence by, say, ‘‘twenty three’’ or ‘‘one hundred and eighty seven’’!

Even if we stick to one parallel computing architecture and one particular installation of that architecture, the number of working compute nodes allocated to us for any particular run will be somewhat variable. To cope with these conditions, we must design our algorithms with (apparently) excessive parallelism — at least ten times as many processes as we are ever likely to be allocated processors. Then, without re-designing the software, it becomes possible to configure it to the resources we are actually given. Ideally, this should happen automatically as the system is being loaded (when its resources become apparent). Even better, we can envisage the possibility of dynamic balancing of the software processes against the given hardware (e.g. during your run, some nodes may fail or be taken away from you by the operating system or you may even be granted extra ones!).

For the moment, with current *occam/transputer* systems, we need a few minutes notice of the resources we are going to be given in order to change some configuration constants or, in the worst case, perform some mechanical code transformations and re-compile. However, the parallel slackness means that no re-design is necessary.

There is one other compelling reason for designing with an excess of parallelism. Only parallel ''farming'' algorithms run compute nodes that operate most of their time completely independently. All other parallel paradigms require significant interaction between processors. Consider the view from a particular processing node. Because the time to acquire information held on other nodes is so great compared with the time to load information from our own node, we must find some useful work for our node to get on with whilst awaiting external information or accept a low efficiency of use from our node. With a *large* number of processes being managed by our node, it is very unlikely they will *all* become blocked awaiting external events at the same time. Hence, there is always something profitable to be doing and we obtain high efficiency. See Valiant's papers [3, 4] for a detailed analysis on the merits of this ''parallel slackness''.

We now have three grounds for designing systems with a high degree of parallelism :–

- it is good (software) engineering — i.e. it makes system design, verification and maintenance easier;
- it gives us portability across different physical configurations of a particular multi-processor architecture (ultimately, *occam* will give us portability across different architectures as well);
- it enables a high efficiency of use for each individual processor — i.e. it makes the system go faster!

Serialisation of these excessively parallel designs now becomes a viable optimisation technique. It is not always applicable *but*, with so many processes allocated to each processor, it *can* :–

- *save time:* by eliminating context switches and the copying of data packets between processes;
- *save space:* by having a common data area for shared data-structures, rather than separate buffers in each process.

With a sub-microsecond overhead in *transputers*, context switching is not really a problem. However, significant savings can sometimes be achieved on the other two items.

Beware that serialisation — whilst always possible — will not always prove to be an optimisation. The user-interface component described in the previous section is constrained to operate at the speed of a user-terminal. Serialising its processing logic will not address that bottleneck!

Beware also that serialisation can — and usually does — lead to an explosion in the length and complexity of the resulting code. This can be so excessive as to render the whole operation impractical. In the following sections, we describe some examples where the synchronisation characteristics of the processes we are combining are sufficiently well-behaved to allow the serialisations to work. Note that the resulting code should be considered as ''compiled'' code — software engineering principles are not upheld and this is not the level at which the components should be maintained.

In the above paragraphs, we have only been discussing the processes that *directly* contribute to the algorithm that solves the original problem. On any particular node, there will be

a further collection of (high-priority) processes to manage external communication and events. This is because a physical node in a credible multi-processor machine will itself be a parallel device. It may have only one compute engine, but it will certainly have multiple communications engines that can operate at the same time. Therefore, there is a minimum level of parallelism to which each node must be programmed if it is to be used to its full advantage.

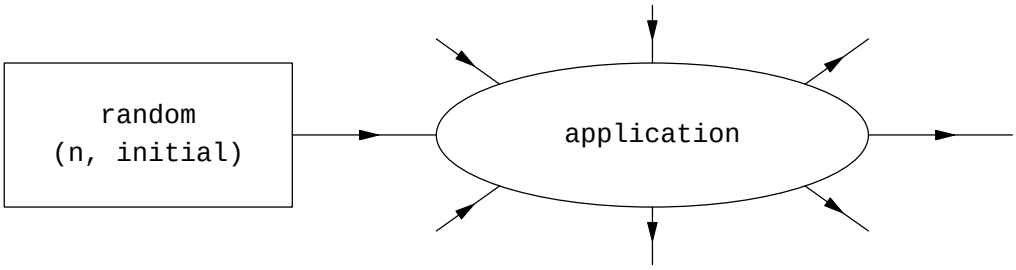
Thus, even an efficient ‘‘farm’’ worker on a *transputer* needs a harness of eight support processes (to drive all its links bi-directionally and in parallel with its main task). For *occam* and the current generation *transputer* (T2s, T4s and T8s), these high-priority buffers, auto-prompters, multiplexors and forwarders have become well known, very simple and standardised. So much so that in the new generation T9000 *transputers*, some of these processes are in hardware! No attempt, of course, should be made to serialise any (remaining) high-priority processes with the background application-specific tasks.

4. Serial “In-Lining” of a Simple Server

In simulating the growth of ‘‘diffusion limited aggregates’’ [5], the computationally intensive innermost loop consisted of executing a random walk over a regular lattice. Anything that could be done to speed up these walks had a direct and equal effect on the speed of the entire simulation.

Each step of the walk consisted of obtaining three random bits to decide the direction of the step, making the step (i.e. updating some coordinates) and checking to see if you had reached a ‘‘sticky’’ tile (which indicated the end of the walk). We used the random number algorithm from [6] that produces acceptable sequences for our application, whilst being computationally light. Despite this lightness, most of the time for each step was spent computing these numbers — there being so little else to do!

The proper way to implement the random number generator is as a *server*, continuously pushing its results towards its *client* :–



where :–

```
PROC random (VAL INT n, initial, CHAN OF INT out)
  -- outputs n random bits per communication
  INT seed:
  ...  other state declarations
  SEQ
    seed := initial
    ...  initialise rest of state
    WHILE TRUE
      INT word:
      SEQ
        ...  compute n random bits in word & update seed etc.
        out ! word
:
```

and :–

```
PROC application (CHAN OF INT from.random, ...)
  ...  local declarations
  ...  body
:
```

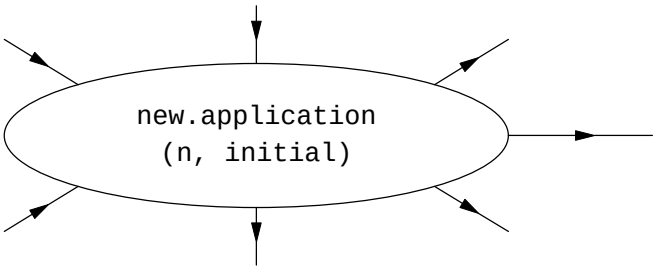
and where, deep inside the `body`, the innermost loop goes :–

```
WHILE walking
  INT next:
  SEQ
    from.random ? next
    ...  rest of step
```

This is good engineering. The application has no responsibility for maintaining the random number seed nor for the random number logic. The seed is a private data-structure, encapsulated and hidden by the random number server that alone needs to know about it.

However, we want to remove the overhead of running the server as a separate process from its client. The serialisation in this case is quite easy. We first have to decide which (if any) of two threads of control to retain as defining the structure of the unified thread of control. The logic of the client application is fairly complex outside its innermost loop and would not take kindly to the inversion of its logic if it were not chosen. On the other hand, the server control structure is rather trivial and can, therefore, take the necessary knocks.

So, the `application` stays in charge! It must inherit the parameters of its absorbed server — apart, of course, from the connecting service channel that now disappears :–



Internally, it picks up any persistent data-structures from the server (i.e. `seed` etc.) and installs any server initialisation code :–

```
PROC new.application (VAL INT n, initial, ...)
  ...  (old) local declarations
  INT seed:                             -- from random
  ...  other state declarations         -- from random
  SEQ
    seed := initial                     -- from random
    ...  initialise rest of state       -- from random
    ...  (old) body
:
```

We must ''in-line'' the server loop code wherever the `body` used to demand service :–

```
WHILE walking
  INT next:
  SEQ
    {{{  from.random ? next
    INT word:
    SEQ
      ...  compute n random bits in word & update seed etc.
      next := word
    }}}
    ...  rest of step
```

We no longer have a context switch to be performed and the server communication has been replaced by an assignment.

Finally, we observe that the transient data-structure `word` (inherited from the server) can be dispensed with, along with the data-copying assignment, and we compute the result directly where it is needed :–

```
WHILE walking
  INT next:
  SEQ
    ...  compute n random bits in next & update seed
    ...  rest of step
```

From an engineering point of view, this code is not as manageable as the original. Client and server data-structures are mixed up and so is the logic that operates on them. However, our walking speed has increased from 93,000 steps per second to 127,000!

5. Serialisation of Pipe-Lined Logic

5.0. *Basic Principles*

Some pipelines are designed specifically for the buffering characteristics they introduce and their ability to service their supplier and consumer processes in parallel. For example, this technique enables *transputers* to communicate and compute at the same time. Serialisation is probably not the right way to try to optimise these pipelines — see [7], [8] and [9] for a discussion on this.

Other pipelines are introduced to separate the phases of a particularly complex function into manageable stages. We concentrate on these and show how to serialise them so as to preserve their overall functionality, but not worry too much about the buffering services they originally provided. The environment in which such a pipeline is applied is only interested in the mathematical transformation being performed — indeed, one of the optimisations being sought through this serialisation is the elimination of superfluous data-buffers and data-copying. Formally, the semantics of the (originally pipelined) component will be preserved with respect to an environment that is always willing to accept its output.

Consider a component process with a single input and a single output channel. We call such a component a *p-q-transformer* if it synchronises with its environment by cycling through the sequence: first do *p* inputs and then do *q* outputs.

If it is implemented with code of the form :–

```
PROC transform (CHAN OF A in, CHAN OF B out)
  ...  state declarations
  SEQ
    ...  initialise state
    WHILE running
      SEQ
        ...  do p inputs
        ...  compute
        ...  do q outputs
:
```

where we also allow computation to be interleaved amongst the above `inputs` and `outputs`, we say the transformer is in *normal* form.

Serialising a pipeline of *normal* form *1-1-transformers* is fairly easy. It becomes a new *normal* form *1-1-transformer* that contains all the state variables of the original pipeline components (modulo some name changes to avoid any clashes). All initialisations on these states are first performed (in any sequence) and its main cycle then :–

- inputs (as in the first component of the pipeline);
- performs the sequences of computations made from the individual computations from each component in the pipeline. The order of this sequence is the same as the order of the components in the pipeline. The communications between pipeline components become assignments between corresponding state variables;
- outputs (as in the last component of the pipeline).

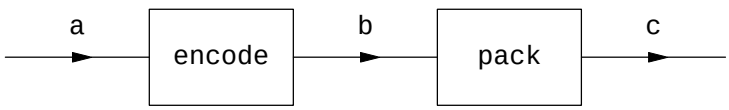
In the computation phase above, there is plenty of opportunity for state-variable and assignment elimination.

If a transformer is not in *normal form*, then part of its state is governed by where it is in its code. By introducing further state variables to represent these positions and testing these within its `compute` section, any non-*normal* form transformer can always be transformed into *normal* form.

5.1. *Structure Clash within the Pipeline*

The result of normalising and serialising a pipeline of *1-1-transformers* will be more complex than the original code. Things get really exciting, however, when we do the same for a pipeline of *p-q-transformers* with differing *p* and *q* values!

Consider part of an image compression pipeline :–



where channels `a`, `b` and `c` respectively carry the protocols :–

```
PROTOCOL PICTURE IS [height][width]BYTE:
PROTOCOL BITS IS BOOL:
PROTOCOL PACKET IS [packet.size]INT:
```

A stream of (fragments of) pictures arrive on channel a and are ''Huffman-encoded'' into a compressed bit-stream on channel b. The encoding operates on differences between neighbouring pixels — small ones are Huffman-encoded, larger differences are transmitted plain (preceded by an ''escape'' code). The bit-stream from b is packed into a decently sized packet for onward transmission down c (and out of the *transputer*).

The encode process is a *1-"many"-transformer*, where *"many"* is data-dependent. The pack process is a *packet.size-1-transformer*. There is a serious structure clash here! The parallel design protects us completely from its difficulties :–

```
PROC encode (CHANNEL OF PICTURE in, CHAN OF BITS out)
  WHILE TRUE
    [height][width]BYTE picture:
    SEQ
      in ? picture
      SEQ i = 0 FOR height
        VAL [width]BYTE line IS picture[i]:
        ...  compress line
:
```

where :–

```
{{{  compress line
INT last.pixel:
SEQ
  last.pixel := 127
  SEQ j = 0 FOR width
    VAL INT pixel IS INT line[j]:
    SEQ
      VAL INT diff IS (pixel – last.pixel) + 255:
      VAL INT n IS n.bits[diff]:
      INT code:
      SEQ
        code := h.code[diff]
        ...  emit bottom n bits of code
      last.pixel := pixel
}}}
```

and where :–

```
VAL [510]INT n.bits IS [ ... ]:
VAL [510]INT h.code IS [ ... ]:
```

are compile-time constant tables holding, respectively, the number of bits and the actual code values for each possible change in pixel intensity. Finally :–

```
{{{  emit bottom n bits of code
SEQ k = 0 FOR n
  SEQ
    out ! (code /\ 1) = 1
    code := code >> 1
}}}
```

The structure of the above code is derived naturally from the specification of encode. The same thing happens for :–

```
PROC pack (CHAN OF BITS in, CHAN OF PACKET out)
  WHILE TRUE
    [packet.size]INT packet:
    SEQ
      SEQ p = 0 FOR packet.size
```

```
          INT word IS packet[p]:
          SEQ
            word := 0
            ...  input bits into word
        out ! packet
:
```

where :–

```
{{{  input bits into word
INT bit:
SEQ
  bit := 1
  SEQ q = 0 FOR WORD.SIZE
    BOOL b:
    SEQ
      in ? b
      IF
        b
          word := word \/ bit
        TRUE
          SKIP
      bit := bit << 1
}}}
```

This completes the programming. The structure clash between the synchronisation characteristics of the two elements is absorbed by the run-time scheduler. The use of parallelism to design such a clean solution to this problem was first described (to our knowledge) in the book by Jones and Goldsmith [10].

5.2. *A Serialising Optimisation*

The problem with leaving the code like this is that the bit-stream channel (whether mapped on to memory or an external link) imposes a bottleneck on the data-flow! It must be removed — i.e. we must serialise the `encode` and `pack` processes.

We have to chose which process structure to preserve — it does not really matter which. Let us choose to preserve `encode` (since it has three nested loops in its cycle and `pack` has only two).

The state of the `pack` process is represented by its variables `packet`, `word`, `bit`, `p` and `q`. Import these variables into what used to be the structure of the `encode` process and is now the serialised :–

```
PROC encode.pack (CHAN OF PICTURE in, CHAN OF PACKET out)
  INT word, bit, p, q:
  [packet.size]INT packet:
  SEQ
    word, bit, p, q := 0, 1, 0, 0
    ...  structure of the encode process
:
```

The encode structure is unchanged except for its single output (deep inside its emit fold). This output triggered a cycle of the pack process — it is replaced by a fold that contains that logic with its housekeeping all inverted :–

```
{{{  out ! (code /\ 1) = 1
SEQ
  ...  'pack' response to the communication
  ...  'pack' housekeeping
}}}
```

where :–

```
{{{  'pack' response to the communication
SEQ
  IF
    (code /\ 1) = 1
      word := word \/ bit
    TRUE
      SKIP
  bit := bit << 1
}}}
```

and :–

```
{{{  'pack' housekeeping
SEQ
  q := q + 1
  IF
    q = WORD.SIZE
      SEQ
        packet[p], word, bit, q := word, 0, 1, 0
        p := p + 1
        IF
          p = packet.size
            SEQ
              out ! packet
              p := 0
          TRUE
            SKIP
    TRUE
      SKIP
}}}
```

That completes the transformation. Designing such complex serial code directly would not be a good idea!!

The alternative transformation — i.e. retaining the structure of pack and inverting encode into it — leads to a very different serial structure. This is left as an exercise for the reader! Note, however, that the transformations (via the original parallel code) will prove the equivalence of two very different serial versions.

5.3. *Further Optimisations Now Become Possible*

Of course, now that the code is serial and the innermost loops from the two original processes have been interleaved and can see each other's data-structures, further optimisations become possible. For instance, the resulting innermost loop (in the emit fold) transfers n bits from code over to word one bit at a time! Clearly, this loop can be removed and the transfer done in one go :–

```
{{{ emit bottom n bits of code
SEQ
  word := word \/ (code << q)
  q := q + n
  IF
    q >= WORD.SIZE
      SEQ
        q := q - WORDSIZE
        packet[p], word := word, code >> (n - q)
        p := p + 1
        IF
          p = packet.size
            SEQ
              out ! packet
              p := 0
          TRUE
            SKIP
    TRUE
      SKIP
}}}
```

Note that `code` should now be declared as a `VAL` and that the `bit` pointer and, of course, the innermost loop control variable `k` are no longer needed.

All these codes plus the necessary buffers can be fitted into the on-chip memory of a T2 *transputer*. On a 20 MHz T800 (alas, we have no T2s), the original clean parallel code took 7.3 μsecs. to produce one compressed bit of output. The first serialisation reduced this to 4.6 μsecs. The last optimisation above (that was enabled by the serialisation) reduced this further to 2.1 μsecs. We could go on, but again we leave this to the interested reader. [Of course, all run-time checks — including those for array-bound violation — were *left on* for the above timings. Switching them off is always a false economy!]

6. Arbitrary Topologies with Well-Behaved Synchronisation

6.0. *Basic Principles*

Our final example is taken from the field of continuous system simulation (e.g. distribution networks for gas or electricity, urban traffic flow, digital circuit emulation ...). The simple way to design the simulation is to create a network of software processes that directly mirrors the physical network of processes in the real system. Any topology — including those with feedback — must be allowed.

In general, attempts to optimise an arbitrary process network by serialisation will lead to an impractical explosion in the size of the resulting code. However, the synchronisation characteristics of the processes studied here are simple and regular — each process communicates continuously and in parallel with all its topological neighbours. This is generalised *"systolic"* computing — irregular networks with feedback are allowed as well as regular meshes. For such systems, serialisation does not cause a bang!

In [11], the notion of an *I/O-PAR* process was introduced. Informally, an *I/O-PAR* process is one that, whenever it communicates, communicates on all its channels in parallel. The following two processes are *I/O-PAR* and in *normal* form :–

```
PROC A ( ... )
  ...  declarations A
  SEQ
    ...  initialise A
    WHILE TRUE
      SEQ
        ...  parallel i/o A
        ...  compute A
:
```

```
PROC B ( ... )
  ...  declarations B
  SEQ
    ...  initialise B
    WHILE TRUE
      SEQ
        ...  parallel i/o B
        ...  compute B
:
```

A key property of *I/O-PAR* processes is that any parallel network of them is deadlock-free and remains *I/O-PAR* — that is why it is so easy and safe to design with them!

Clearly, a network of *I/O-PAR* processes can synchronise with its environment more freely than one in *normal* form. At any particular moment, such a network may have communicated on one of its channels several more time than it has communicated on one of its other channels (where ‘‘several’’ is bounded by the maximum ‘‘diameter’’ of the network). However, a network in this condition will always be offering its environment communications on its more backward channels (that would enable the number of times they have been used to catch up with the leader). For an *I/O-PAR* process in *normal* form, the ‘‘several’’ is limited to one.

If we place a collection of *I/O-PAR* processes in an environment that is itself *I/O-PAR* (with respect to its connections to that collection), then that collection may be serialised into an *I/O-PAR* process in *normal* form without changing the semantics of the whole system.

These results are more formally presented in [12], together with the serialising transformations and some proofs! Here we are somewhat less formal. Suppose we want to run processes `A` and `B` in parallel :–

```
PROC A.B ( ... )
  ...  'internal' channels for connecting A and B
  PAR
    A ( ... )
    B ( ... )
:
```

where the parameters for `A.B` are the union of those for `A` and those for `B`, less their interconnecting channels.

To serialise them, we extract a set of execution paths that can be expressed in *I/O-PAR normal* form. This certainly loses some of the paths that were available to the original parallel code — but since we are only going to run the derived code in an *I/O-PAR* environment that is not going to exploit those extra paths, this does not matter! The serialised code is :–

```
PROC A.B ( ... )
  ...  declarations A
  ...  declarations B
  SEQ
    ...  initialise A
    ...  initialise B
    ...  serialised A and B loop
:
```

Since they concern separate sets of state variables, the order of the `initialise` sections derived from `A` and `B` is irrelevant. Since no communications are involved (i.e. the external

environment cannot detect what is happening), it is safe to serialise them. The same is true for the respective `compute` sections inside the loop :–

```
{{{  serialised A and B loop
WHILE TRUE
  SEQ
    PAR
      ...  parallel i/o A (except 'internals')
      ...  parallel i/o B (except 'internals')
      ...  'internal' assignments
    ...  compute A
    ...  compute B
}}}
```

The position of the respective parallel `i/o` sections clearly represents a synchronisation behaviour with its environment that the original parallel code could have chosen. That is all we promised to do!

In parallel with those communications are a set of assignments between the state variables of `A` and `B`. These are derived from the original ''internal'' communications between `A` and `B`. Again, because no external communications are involved, it is safe to serialise these assignments (in any order — because the anti-alias and usage rules of *occam* ensure there can be no data-dependencies!). Also, because there are no usage conflicts with the `i/o` (currently happening in parallel), it is safe to move these assignments to the start of the `compute` region of the cycle :–

```
{{{  serialised A and B loop
WHILE TRUE
  SEQ
    PAR
      ...  parallel i/o A (except 'internals')
      ...  parallel i/o B (except 'internals')
    ...  'internal' assignments
    ...  compute A
    ...  compute B
}}}
```

This last change is, of course, undetectable by its environment and the code is now *normal* form *I/O-PAR* — as required.

Another key property of processes, discussed in [11, 12], is *I/O-SEQ*. This is similar to *I/O-PAR* except that input communications are serialised before output ones. However, input communications are still all parallel — i.e. when one input happens, all inputs must happen. The same is true for outputs. The following process is in *normal I/O-SEQ* form :–

```
PROC C ( ... )
  ...  declarations C
  SEQ
    ...  initialise C
    WHILE TRUE
      SEQ
        ...  parallel inputs C
        ...  compute C (part 0)
        ...  parallel outputs C
        ...  compute C (part 1)
:
```

The second general result is this: if we run an *I/O-SEQ* process in parallel with an *I/O-PAR* process that supplies all its input, they may be serialised into an *I/O-PAR* process in *normal* form (again modulo an environment that is itself *I/O-PAR*).

Suppose that these conditions apply to processes `A` and `C` above. A valid (sub-)set of execution paths is given by the serialisation :–

```
PROC A.C ( ... )
  ...  declarations A and C
  SEQ
    ...  initialise A and C
    WHILE TRUE
      SEQ
        PAR
          ...  parallel i/o A (except 'internals')
          SEQ
            ...  'internal' assignments (from A to C)
            ...  compute C (part 0)
            PAR
              ...  parallel outputs C (except 'internals')
              ...  'internal' assignments (from C to A)
        ...  compute A and C (part 1) - any order
:
```

Again, we may move the internal assignments and computations around a bit :–

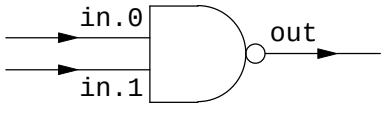
```
    WHILE TRUE
      SEQ
        ...  'internal' assignments (from A to C)
        ...  compute C (part 0)
        ...  'internal' assignments (from C to A)
        PAR
          ...  parallel i/o A (except 'internals')
          ...  parallel outputs C (except 'internals')
        ...  compute A and C (part 1) - any order
```

while the parallel usage rules ensured that there were no data-dependencies to prevent us!

6.1. *Applying the Transforms*

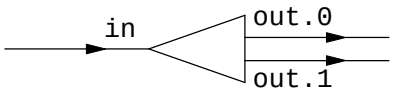
We will take a concrete example from [11]. Fundamental gates used in digital logic circuits are emulated by *I/O-PAR* processes. For instance, a two-input `nand` gate is given by :–

```
PROC nand (CHAN OF INT in.0, in.1, out)
  INT a.0, a.1, b.0, b.1:
  SEQ
    b.0, b.1 := undefined, undefined
    WHILE TRUE
      SEQ
        PAR
          in.0 ? a.0
          in.1 ? a.1
          out ! ~(b.0 /\ b.1)
        PAR
          in.0 ? b.0
          in.1 ? b.1
          out ! ~(a.0 /\ a.1)
:
```



Because *occam* channels have to be ''point-to-point'', branches in wiring have to be represented by active processes :–

```
PROC delta (CHAN OF INT in, out.0, out.1)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      PAR
        out.0 ! x
        out.1 ! x
:
```

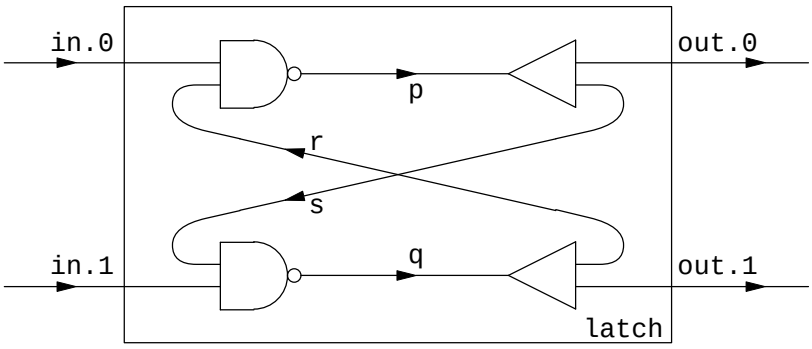


The *I/O-SEQ* nature of the above process corresponds to a digital logic component with zero propagation delay. Such components, therefore, have no impact on the timing characteristics of the circuit being emulated and may be freely used.

The previous `nand` process corresponds to a component with a propagation delay equal to one (emulated) sample interval between incoming logic values. Variable length propagation delays can be easily modelled by adding *I/O-PAR* ''delay-line'' processes, parametrised to the required value.

A four-valued logic is emulated in these processes: `TRUE` and `FALSE` (represented by `11` and `00` respectively) and two ''undefined'' levels (represented by `10` and `01`). Notice that, for a word length of 32, up to 16 independent sets of wavefront trials can be conducted simultaneously.

A simple circuit with feedback is the `latch` :–



which can, of course, be instantly implemented :–

```
PROC latch (CHAN OF INT in.0, in.1, out.0, out.1)
  CHAN OF INT p, q, r, s:
  PAR
    nand (in.0, r, p)
    nand (s, in.1, q)
    delta (p, out.0, s)
    delta (q, r, out.1)
:
```

To serialise this, let us first join the *I/O-PAR* logic gate with its adjacent *I/O-SEQ* ''fan-out'' process. This transformation is based upon the second one given in the previous subsection, extended in the obvious way to cope with the two *I/O-PAR* phases of the `nand` cycle :–

```
PROC nand.delta (CHAN OF INT in.0, in.1, out.0, out.1)
  INT a.0, a.1, b.0, b.1, a:
  SEQ
    b.0, b.1 := undefined, undefined
    WHILE TRUE
      SEQ
        a :=  ~(b.0 /\ b.1)           -- first phase
        PAR
          in.0 ? a.0
          in.1 ? a.1
          out.0 ! a
          out.1 ! a
        a :=  ~(a.0 /\ a.1)           -- second phase
        PAR
          in.0 ? b.0
          in.1 ? b.1
          out.0 ! a
          out.1 ! a
:
```

This is now in (two-phase) *I/O-PAR normal* form. Notice that the variables `a.0` and `a.1` need only have very local scope — that of the first parallel communications in the loop and its following assignment. Next, by moving the first assignment in the loop to the end of the loop (and, of course, duplicating it in the initialisation part), we observe that the same is true for the variables `b.0` and `b.1`. Localising both pairs of definitions and renaming them to `c.0` and `c.1`, we end up with a loop whose body is a sequence of two identical phases. This collapses to a simple *I/O-PAR normal* form :–

```
PROC nand.delta (CHAN OF INT in.0, in.1, out.0, out.1)
  INT a:
  SEQ
    a := ~(undefined /\ undefined)
    WHILE TRUE
      INT c.0, c.1:
      SEQ
        PAR
          in.0 ? c.0
          in.1 ? c.1
          out.0 ! a
          out.1 ! a
        a :=  ~(c.0 /\ c.1)
:
```

Now, the `latch` runs two instances of this `nand.delta` in parallel. This may now be serialised by applying the first transform from the previous section. We have to rename the internal state-variables to avoid clashes — we do this by adding the suffix `.hi` to those from the ‘‘higher’’ `nand.delta` and `.lo` to the ‘‘lower’’ one :–

```
PROC latch (CHAN OF INT in.0, in.1, out.0, out.1)
  INT a.hi, c.0.hi, c.1.hi:
  INT a.lo, c.0.lo, c.1.lo:
  SEQ
    a.hi, a.lo := other.undefined, other.undefined
    WHILE TRUE
      SEQ
        PAR
          in.0 ? c.0.hi
          out.0 ! a.hi
          in.1 ? c.1.lo
          out.1 ! a.lo
        c.1.hi, c.0.lo := a.lo, a.hi
        a.hi :=  ~(c.0.hi /\ c.1.hi)
        a.lo :=  ~(c.0.lo /\ c.1.lo)
:
```

Clearly, the variables `c.1.hi` and `c.0.lo` may be dispensed with and their assignment costs saved — the `a.lo` and `a.hi` values being used directly in the final assignments. Renaming `c.0.hi` and `c.1.lo` as `t.hi` and `t.lo` respectively and localising their declaration, we are left with :–

```
PROC latch (CHAN OF INT in.0, in.1, out.0, out.1)
  INT a.hi, a.lo:
  SEQ
    a.hi, a.lo := other.undefined, other.undefined
    WHILE TRUE
      INT t.hi, t.lo:
      SEQ
        PAR
          in.0 ? t.hi
          in.1 ? t.lo
          out.0 ! a.hi
          out.1 ! a.lo
        a.hi, a.lo :=  ~(t.hi /\ a.lo), ~(a.hi /\ t.lo)
:
```

Looking at the resulting code, it is possible that it could have been coded like that in the first place. However, the serialised code only collapsed to this simple form because of the symmetry in the original circuit. Less regular circuits would require serial code we would not like to compose directly! For example, a latch circuit whose gates imposed different propagation delays!!

The resource demands from the two versions of the above `latch` component are significantly different. The parallel version requires 308 bytes of workspace and processes incoming signal sample ‘‘wavefronts’’ at the rate of one every 36 μsecs. The final serial version only requires 84 bytes of workspace and cycles in 15 μsecs.

We would expect similar benefits to be obtained from serialising larger circuits — enabling them to be emulated in the same (real) time from the same (*transputer*) hardware resources. Without automatic tools, however, we would not like to try it!

7. Discussion

A recent article [13] on parallel computing in the popular computing magazine *BYTE* ends with the following paragraph :– "*The hardware issue has already been solved, thanks to the INMOS transputer. Software remains the final hurdle to clear if parallel processing via multicomputers is to emerge as a popular alternative to sequential processing.*"

This point-of-view is a little worrying. Whilst the article mentions *C* and *FORTRAN*, it makes no no reference to *occam*. Yet *occam* was devised specifically to address the software and hardware issues associated with parallel computing [14, 15, 16, 17] — the security weaknesses in ''standard'' programming languages disqualifying them from being robust platforms on which to build concurrent systems. *Occam* was developed simultaneously with the *transputer* and the latter would not exist (as we know it) without the former. If you only pick up half the groceries, don't complain if you get hungry!

Our experience from working with *occam* is that hardware issues and software issues are no different from one another. We adopt the same approach for each. Both are designed as parallel systems — it's just that the hardware designs tend to stay parallel, while the software elements sometimes get serialised a little bit!

The second sentence of the above quotation endorses the common belief that it is something in the *parallelism* that causes the difficulty in the software. The theme of this paper is that this is false — it is the attempt to write complex *serial* code directly that causes (and always has caused) the problems.

We have argued that parallelism is a high-level programming concept. It enables us to capture complex system behaviour much more directly, concisely and simply than any equivalent (low-level) serial code. On the practical side, to develop application software that will be portable and efficient across different architectures and configurations of multiprocessor, we need *much more* parallelism in our algorithms and data-structures than we are ever likely to be offered in hardware.

We have considered three different applications (the simulated growth of diffusion limited aggregates, image compression and digital logic emulation) and demonstrated three different parallel paradigms (client/server, pipe-lines and arbitrary feedback networks) that yield, respectively, well-engineered solutions for them. For each of these cases, we have shown how to transform (''compile'') them into equivalent serial code that is more efficient in terms of its space and time requirements, but is more complex and less well-engineered. In general, we have little confidence in our ability to produce such serial code directly and none in our ability to maintain it.

The serialising optimisations we have used are all constructively defined and can clearly be automated. We see an urgent need for tools to do these transformations for us. In the medium term, serialising will be an everyday activity for parallel programmers and programmers make too many mistakes on their own! We also need these serialising tools integrated into a *secure development environment* alongside their complementary (''folding'') editor, compiler and maintenance tools — i.e. the INMOS *TDS* [18], or something that shares its philosophy, must be re-born. Eventually, serialisation may be hidden from us by being incorporated into the compiler, loader or dynamic load balancer.

''Computer algebra'' tools (e.g. [19]) have been developed and are in significant use by mathematicians to help them manipulate their formulae. It seems extraordinary that computer scientists (who made those tools for the mathematicians) have not demanded similar help. We are far too confident in our abilities to manipulate our formulae (i.e. programs) —

evidence of our *inability* is widespread.

The real reason for the lack of program manipulation tools (in *significant* use) is that the programming languages (in *significant* use) do not allow formulae (i.e. programs) with the same simple algebraic properties as are enjoyed in mathematics. Languages such as *C* and *FORTRAN* have ill-defined and highly complex semantics that rule out the prospect for any formal analysis or manipulation.

On the other hand, transformation tools exist for some *functional* programming languages and also, of course, for *occam* [20, 21]. *Occam* is the exception to the general statement in the preceding paragraph. It is the only programming language in significant (industrial) use that *only* allows formulae (i.e. programs) with simple algebraic properties. Nevertheless, the use of the Oxford tools [21] is not very widespread — it seems not to have gone much beyond INMOS (and its sub-contractors), where it has played a crucial role in the design of major features of the T9000 *transputer* [22]. The Oxford tools do not include the serialising transforms described in this paper. We are keen to use such tools at Kent and work is in progress here to produce them.

Finally, we summarise our approach to (parallel) computing applications :–

- design a solution incorporating as much parallelism as naturally falls out from the application — this is usually massive;
- balance this across the number of processors at our disposal — this is easy so long as the parallelism in the algorithm greatly exceeds the parallelism in the hardware;
- for each individual node, serialise the worker processes so long as this yields significant optimisations — i.e. a complete serialisation is not always necessary and may be counter-productive. We need tools to assist us in this.

We have been fortunate in being able to avoid working with ‘‘dusty decks’’. Extracting parallel code from them (in order to exploit parallel hardware) is as hard as extracting ‘‘high-level’’ source code from a raw assembler listing. It’s quicker and safer and cheaper to go back to the original problems and re-write them from scratch using the higher paradigm. We must, of course, use a proper multi-processing language that allows the use of formal methods and enables us, and automated tools, to work.

8. Acknowledgements

The work of one of the authors (GRRJ) has been funded by the Brazilian Research Council (CNPq), under grant No. 205034/88-8, and we are especially grateful for their support.

We are also indebted to the community of parallel system engineers within the Computing Laboratory at the University of Kent, who have created the culture from which the particular experiences reported in this paper have been drawn. That work has been variously supported by the Computer Board Initiative on Software Environments for Parallel Computers, the SERC/DTI Transputer Initiative, the Royal Armament Research and Development Establishment and the COMETT training programme of the EEC.

9. References

[0] C.A.R.Hoare: *“Communicating Sequential Processes”*; Prentice-Hall; 1985.

[1] INMOS Ltd.: *“occam2 Reference Manual”*; Prentice-Hall; ISBN 0-13-629312-3; 1987.

[2] G.Jones: *"On Guards"*; in 'International Workshop on the Parallel Programming of Transputer-Based Machines', the Proceedings of the 7th. Occam User Group Technical Conference, Grenoble, France; edited by T.Muntean; IOS Press, Holland; September 1987.

[3] L.G.Valiant: *"Bulk-Synchronous Parallel Computing"*; in 'Parallel Processing and Artificial Intelligence'; edited by M.Reeve and S.E.Zenith; John Wiley & Sons Ltd., England; ISBN 0-471-92497-0; July 1989.

[4] L.G.Valiant: *"A Bridging Model for Parallel Computation"*; Commun. ACM 33, 8, pp. 103-111; August 1990.

[5] D.R.Morse, A.M.Welch and P.H.Welch: *"Diffusion Limited Aggregation: an Example of Real-Time Parallelisation"*; in 'Real-Time Systems with Transputers', the Proceedings of the 13th. Occam User Group Technical Conference, York, England; edited by H.Zedan; pp. 248-261; IOS Press, Holland; September 1990.

[6] S.K.Park and K.W.Miller: *"Random Number Generators: Good Ones are Hard to Find"*; Commun. ACM 31, 10, pp. 1192-1201; October 1988.

[7] R.Peel: *"Issues Raised while Implementing a Layered Protocol using Occam and the Transputer"*; in Proceedings of the 10th. Occam User Group Technical Conference, Twente, The Netherlands; edited by A.Bakkers; IOS Press, Holland; April 1989.

[8] G.Jones: *"Efficient Multiple Buffering in Occam"*; in Occam User Group Newsletter, No. 11, pp. 36-44 (July 1989).

[9] P.H.Welch: *"Securely Managed Pointers"*; in Occam User Group Newsletter, No. 15 (July 1991).

[10] G.Jones and M.H.Goldsmith: *"Programming in Occam2"*; Prentice-Hall; ISBN 0-13-730334-3; 1988.

[11] P.H.Welch: *"Emulating Digital Logic Using Transputer Networks (Very High Parallelism = Simplicity = Performance)"*; in 'Parallel Architectures and Languages Europe - Volume 1'; Lecture Notes in Computer Science, vol. 258, pp.357-373; Springer-Verlag; June 1987.

[12] G.R.R.Justo and P.H.Welch: *"Synthesis of Deadlock-Free Parallel Programs"*; in Proceedings of the 3rd. pan-Hellenic Conference on Information Technology, Athens, Greece; pp. 46-59; May 1991.

[13] R.M.Stein: *"Scaling Up: Get the Message?"*; in Byte, June 1991, pp. 231-244; McGraw-Hill; June 1991.

[14] C.A.R.Hoare et al. : *"Laws of Programming"*; Commun. ACM 30, 8, pp.672-686; August 1987.

[15] A.W.Roscoe and C.A.R.Hoare: *"Laws of Occam Programming"*; Technical Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford, OX1 3QD, ENGLAND; 1986.

[16] R.Shepherd: *"Security Aspects of Occam"*; Technical Note 28 (72-TCH-028-00); INMOS Ltd., Bristol; 1987.

[17] G.Barrett: B.Sufrin, *"Formal Support for Distributed Systems: occam and the Transputer"*. In 'TRANSPUTING '91'; IOS Press, Holland; ISBN 90-5199-056-6; April 1991.

[18] INMOS Ltd.: *"Transputer Development System"*; Prentice-Hall; ISBN 0-13-928-995-X 1988.

[19] S.Wolfram: *"Mathematica — a System for Doing Mathematics by Computer"*; Addison-Wesley; 1991

[20] M.H.Goldsmith: *"Occam Transformation at Oxford"*; in 'International Workshop on the Parallel Programming of Transputer-Based Machines', the Proceedings of the 7th. Occam User Group Technical Conference, Grenoble, France; edited by T.Muntean; IOS Press, Holland; September 1987.

[21] M.H.Goldsmith: *"The Oxford Occam Transformation System — User Documentation "*; Programming Research Group, Oxford University; January 1988.

[22] A.W.Roscoe, M.H.Goldsmith, A.D.B.Cox and J.B.Scattergood: *"Formal Methods in the Development of the H1 Transputer"*. In 'TRANSPUTING '91'; IOS Press, Holland; ISBN 90-5199-056-6; April 1991.