EFFICIENT MULTIPLE BUFFERING IN OCCAM

Geraint Jones, Programming Research Group, Oxford

One of the virtues, it is said, of the transputer is that it can do all that communication *at the same time* as the all that arithmetic – or whatever it is a processor does. The great virtue, it is also said, of occam is that it is ideal for programming such concurrent activity. This is an exploration of these assertions, prompted by a fragment of code in a paper[1] presented at the tenth occam user group meeting.

Keeping things going

Suppose T is the type of some fixed size buffer – it will probably be `[N]BYTE` for some N or other – and that *in* and *out* are channels with that protocol, one placed at an input link and one at an output link of a single transputer. A program like

```
WHILE TRUE
  T x :
  SEQ
    in ? x
    p(x)
    out ! x
```

reads an infinite sequence of buffer values from the input link, applies p to each of them, and then copies them out of the other link.

The input link is working while the input from *in* is executing, the processor is working while the call of p is executing, and the output link is working while the output to *out* is executing. However, since only one of these three processes is executing at a time, only one of the three *engines* – the link drivers or the processor – is in use at a time. (I am assuming that N is so large that the time spent by the processor on loading the link registers is insignificant compared with the time it takes to ship the data along a serial link.) If executing p takes a time about equal to the time it takes to ship a value of type T over a link, and nothing else is happening in the transputer, you are getting only a third of the potential performance of your microprocessor.

The answer, since each engine 'uses up' a process to keep it busy, is to write a program with three parallel processes in it. The program in figure 1 also reads an infinite sequence of buffer values from the input link, applies p to each of them, and copies them out of the other link. However it can be doing these three things at once. In order to do that it is capable of reading up to three more items than it has output, unlike the earlier sequential process which could only read one item ahead.

That, it would seem, is that: all three engines can now be running at once, so this has clearly tripled the performance of the previous worst case, has it not? Well, no, regrettably not quite: although it is possible for all three engines to run at once, it is not possible for all three engines to be running at once all the time.

```
CHAN OF T c.xy, c.yz :
PAR
  WHILE TRUE
    T x :
    SEQ
      in ? x
      c.xy ! x
  WHILE TRUE
    T y :
    SEQ
      c.xy ? y
      p(y)
      c.yz ! y
  WHILE TRUE
    T z :
    SEQ
      c.yz ? z
      out ! z
```

Figure 1: a three stage pipeline

```
VAL INT n IS 3 :
[n+1]CHAN OF INT c :
PAR
  PAR i = 0 FOR n
    SEQ
      c[i] ! i
      WHILE TRUE
        INT ix.wi :
        SEQ
          c[i+1] ? ix.wi
          c[i] ! ix.wi

  CHAN OF INT new IS c[0] :
  CHAN OF INT c.xy, c.yz :
  CHAN OF INT used IS c[n] :
  [n]T b :  -- * shared * --
  PAR
    WHILE TRUE
      INT ix.x :
      SEQ
        new ? ix.x
        in ? b[ix.x]
        c.xy ! ix.x
    WHILE TRUE
      INT ix.y :
      SEQ
        c.xy ? ix.y
        p(b[ix.y])
        c.yz ! ix.y
    WHILE TRUE
      INT ix.z :
      SEQ
        c.yz ? ix.z
        out ! b[ix.z]
        used ! ix.z
```

Figure 2: a pointer-passing pipeline

The problem is that between each of what you might call the useful activities, the three processes have to spend some time in a communication on $c.xy$ or $c.yz$. Even if the processes are keeping up with each other, so that no time is spent waiting for the other partner in the synchronization, some time has to be spent on the transfer of data: over $c.xy$ from x to y, and over $c.yz$ from y to z. Although these are channels internal to the transputer – so the transfers are block moves and are presumably much faster than the link transfers – each link engine is guaranteed to be idle for one block move time per link communication, and the processor is guaranteed to be 'otherwise occupied' for two block move times per call of p.

Keeping things going at all costs

The solution that a number of people have adopted is, in effect, to replace the communication of T values by communication of pointers to variables containing those values. The pointer can be quite small in comparison to the value, so the time overhead is greatly reduced.

Of course, there are no *pointers* as such in occam, so you have to use indices into an array instead of pointers, and must explicitly manage the pool of variables to which they point. For the three place pipeline, three buffer variables are needed at a time so an array $[3]T$ is declared to be the pool of shared variables. Each T variable in the program is replaced by an `INT` variable which will serve as a pointer into that pool, and most accesses to the T variable are replaced by accesses to the pool indexed by the `INT` variable. The exception is that instead of communicating a T value over an internal channel, it is necessary only to pass the value of its index in the pool.

A mechanism has to be provided to allocate the buffers on first use, and to collect them after last use. In our case, the program ends up looking something like figure 2, where the replicated `PAR` arranges that the sequence of `INT`s available at *new* is $0, 1, 2$ followed by the sequence returned at *used*. (Exercise for the reader: show that only one of $ix.x$, $ix.y$, $ix.z$ has any particular value at any one time when the corresponding process indexes the array b.) Roger Peel's program[1] was actually, apart from an allocation mechanism different in detail, the same as this program except that he had declared n to be two rather than three. This has the effect of throttling the pipeline so that at most two of the three activities can happen at once. (Exercise: what does a T-passing program equivalent to Roger's pointer-passing program with $n = 2$ look like?)

The striking similarity between the programs in figures 1 and 2 is meant to inspire your confidence that they have a similar meaning. I hope that you can see that this pattern might be followed with any program where a number of values of the same type were being passed between a number of processes, provided all the processes were on a single transputer so that the pool of buffers was in a shared store. In general the pool handler process has to hand out indices over a number of different *new* channels, to a number of different processes: wherever the pooled values are created or copied a new buffer must be allocated. Similarly, wherever a value is destroyed a buffer must be released so the pool handler has also to collect indices from a number of *used* channels.

On the assumption that the time taken to do something with an `INT` is dominated by the time taken doing something to a T, the overheads are now negligible, and we get the more-or-less tripled performance we expected.

The cost of keeping things going

The problem with a program that passes array indices and shares an array is precisely the problem with any program using pointers: since the identity of a particular variable can be determined by data, it is not always possible to check statically that anti-aliasing and disjointness conditions hold. In this particular case, for example, the only way to find out which part of b is accessed by

```
INT ix.y :
SEQ
  c.xy ? ix.y
  p(b[ix.y])
  c.yz ! ix.y
```

is to run the program and see what turns up on $c.xy$.

This means that there is no way that a compiler can check that the parallel processes in figure 2 make disjoint use of the locations of b. I should emphasize that it *is* possible for the programmer to check this disjointness: that was the essence of an exercise in the last section. So if you are prepared to take the burden of responsibility for proving that your program obeys the rules, albeit in an uncheckable way, you can turn off the usage checks of the compiler and plough on. You can spend the time that the compiler takes to generate code, laboriously proving by hand that it is indeed a safe program.

In this respect, it is a pity that the INMOS compiler does not (as far as I know) let you suppress the checking of just one variable – in this case b – without suppressing the checks on all variables in a compilation. The price for using the pointer trick would seem to be that the programmer is obliged to do all the checks the compiler would otherwise have done. (But, of course, a disciplined programmer is doing that anyway while writing the program!)

Keeping things going with a clean conscience

The strategy for getting around this cost is to eliminate the transfer of data by 'taking the process to the data' rather than the other way about.

Recall that each of the pairs of communications in an occam program serves two purposes: as well as the data transfer, there is also a synchronization. This is encapsulated in the equivalence of the program in figure 3 to that in figure 4. Now, since x is not free in the second `PAR`, the assignment can be discarded by replacing the uses of y by uses of x, so this program is in turn equivalent to that in figure 5. If Q and S have the same form, you can proceed to eliminate the communications on c, one by one.

Suppose you have a program consisting of a `PAR` of two `WHILE` loops, the bodies of which communicate with each other exactly once, like that in figure 6. You can unwind each of the loops once to get a program with a distinguished first communication on the channel, which you can then eliminate. That makes a program with one communication fewer, and a component which is again a `PAR` of `WHILE` loops,

```
CHAN OF T c :
PAR
  T x :
  SEQ
    P(x)
    c ! x
    Q(c)
  T y :
  SEQ
    R()
    c ? y
    S(c, y)
```

Figure 3

```
T x, y :
SEQ
  PAR
    P(x)
    R()
  y := x
  CHAN OF T c :
  PAR
    Q(c)
    S(c, y)
```

Figure 4

```
T x, y :
SEQ
  PAR
    P(x)
    R()
  CHAN OF T c :
  PAR
    Q(c)
    S(c, x)
```

Figure 5

```
CHAN OF T c :
PAR
  WHILE TRUE
    T x :
    SEQ
      in ? x
      c ! x
  WHILE TRUE
    T y :
    SEQ
      c ? y
      out ! y
```

Figure 6

```
T x, y :
SEQ
  in ? x
  WHILE TRUE
    SEQ
      PAR
        in ? y
        out ! x
      PAR
        in ? x
        out ! y
```

Figure 7

in which you can unwind the loops again. Eventually you can get the program into a form where you can see how to wind up the loops again, and show the `PAR`-of-`WHILE`s program equivalent to the `WHILE`-of-`PAR`s program in figure 7. (Exercise: do so; you will need to unwind the loops about three times.)

We have seen that a two-stage pipeline (figure 6) can be implemented by a sequence of parallel processes (figure 7), and since this has the form of figure 8 it is tempting to try replacing the three-place pipeline in figure 1 by figure 9. Indeed this is the closest approximation to the original of which I know, which uses no internal copying of data but can be checked by a compiler. (It is to be found in, for example, reference [3].)

The reason this program is different from the original pipeline is that it is not 'a buffer' in the sense of reference [2]. Just like the pipeline, this one can input up to three more items than it has output; however once it has taken any input, it can never have input *fewer* than one more item than it has output.

So long as the additional latency does not matter, this is a statically checkable program which can keep two links and their processor busy at the same time without any additional copying of data. If a call of p takes the same length of time as

```
VAL INT n IS 2 :
[n]T b :
SEQ
  in ? b[0]
  WHILE TRUE
    SEQ i = 0 FOR n
      PAR
        in ? b[(i+1)\n]
        out ! b[i]
```

Figure 8

```
VAL INT n IS 3 :
[n]T b :
SEQ
  in ? b[0]
  PAR
    in ? b[1]
    p(b[0])
  WHILE TRUE
    SEQ i = 0 FOR n
      PAR
        in ? b[(i+2)\n]
        p(b[(i+1)\n])
        out ! b[i]
```

Figure 9

the transfer of a value of type T over a link, this program has the same tripled performance as figure 2; in fact it should be marginally better.

Why a clean conscience cannot be cheap

The following is an informal argument that there is no reasonable statically checkable program with the same behaviour as the pipeline in figure 1 and the same efficiency as the pointer-passing program in figure 2. By *behaviour* I mean the possible orders in which observable communications – those on *in* and *out* – can happen; and by *efficiency* I mean the throughput and latency, as measured by the passage of real time, and the amount of storage consumed.

A reasonable program can use no more than three T variables, and must use at least three variables to have enough internal concurrency. There are therefore three variables – call them x, y and z – and so nine distinct types of activity – for each variable an input from *in*, a processing activity and an output to *out* – which are repeated over the passage of time. I will refer to these activities as, for example ix_n for the nth input into x, px_n for the corresponding processing activity, and so on.

The inputs have to happen in a sequence, and there is no point doing anything but using the variables in rotation, which implies an ordering: ix_n before iy_n before iz_n before ix_{n+1} and so on (up to a relabelling of the variables). The same holds for outputs, which clearly have to happen in the same order. There is also a sequence implied by the history of each variable: for example ix_n before px_n before ox_n before ix_{n+1} and so on.

These orderings are shown in figure 10. Each solid vertical arrow indicates an order enforced by sharing of a variable; each solid horizontal arrow is an order enforced by the sharing of a channel. For disjointness, orderings imposed by sharing a variable or channel must be implemented by executing the corresponding processes in sequence, either in a `SEQ` or something equivalent like a `WHILE`-loop. The broken arrows are orderings which might be implemented by communication; these are not important for the argument, and indeed it would not help even if we were prepared to relax these constraints. Notice that the same activity – ix_{n+1} – appears twice in

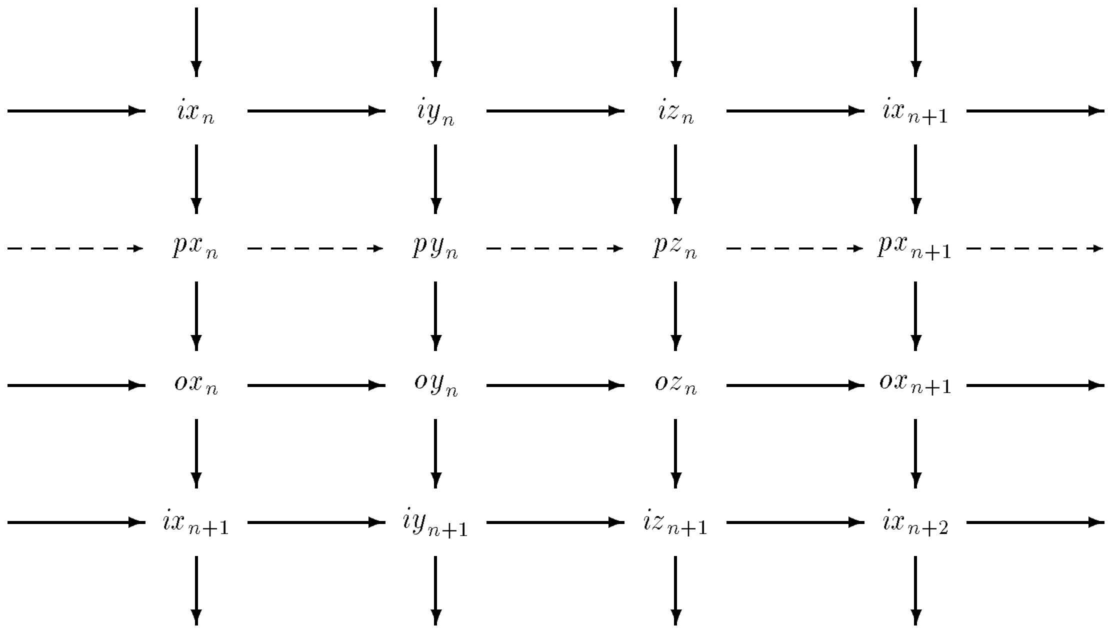

Figure 10: the sequencing of activities in figures 1 and 2

the figure to make the picture clearer: the whole picture should be imagined as being wrapped around a cylinder.

Now, consider how you might implement these constraints. The effect of the `SEQ` in figure 5 is to constrain Q to happen when and only when both P and R have terminated, and similarly that S happen only when both P and R have terminated. In a picture there would be arrows from each of P and R to Q, and in general a process pointed at by some arrows must appear at the beginning of a `PAR` which follows immediately in a `SEQ` after a `PAR` which ends with the processes from which the arrows start.

Consider iy_{n+1} in the figure: it must come just after the end of a two branch `PAR` in which the branches finish with ix_{n+1} and oy_n. Similarly, oz_n must come just after the end of a two branch `PAR` in which the branches finish with oy_n and pz_n. But this means that oy_n must be the last process in two different `PAR` constructs in the program, which is not possible.

If there were a checkably disjoint program with the behaviour described by the graph sketched in figure 10, it would have to be possible to implement by `SEQ` ordering the constraints shown as solid lines. Assuming that such a program exists has led to a contradiction, so we conclude that there is no such program. (Exercise: show that the equivalence of the programs in figures 6 and 7 does not contradict this argument because the sequencing graph corresponding to figure 6 is a special case.)

A question of philosophy

There seems to be a growing body of opinion that the usage checker is an optional extra to occam and that whenever the programmer convinces himself that the restrictions are too confining but that his dubious program will work he should turn it off and still claim his program to be occam. This is, in some people's opinion, a similar level of 'language abuse' to that of the Fortran programmer who discovers that he can pass a routine as an argument to itself, and so introduce recursion. It can be argued that the occam definition is too restrictive and should not have specified, as it does[5, p. 75]:

> An array may be used in more than one component of a parallel, if and only if the subscripts used to select components of the array can be determined at compile time. Otherwise the array may only be used in one component of the parallel.

but until a revised constraint allowing justifiable exceptions to the rule is promulgated, anything which does not satisfy the Fourth Law of Parallel Usage is presumably not occam.

References

[1] Roger Peel, *Issues Raised while Implementing a Layered Protocol Using occam and the Transputer*, in ed. André Bakkers, OUG-10 *Applying Transputer Based Parallel Machines*, IOS, 1989.

[2] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, *A theory of communicating sequential processes*, Journal of the ACM, **31**(3), July 1984, pp. 560–599.

[3] Phil Atkin, *Performance maximisation*, INMOS Technical Note 17, INMOS document 72-TCH-017-00; also the last chapter of [4].

[4] INMOS Limited, *Transputer Technical Notes*, Prentice Hall, 1989.

[5] INMOS Limited, *occam 2 Reference Manual*, Prentice Hall, 1988.

Acknowledgement

This note started life in a conversation in the back of the lecture theatre at the Enschede meeting, and contains material contributed to a subsequent electronic mail conversation. There have been contributions from – amongst others – Roger Peel, Michael Goldsmith, Roger Shepherd, John Wexler, Phil Winder, Philip Ross, Tony Lomax, Mark Edmonds, Adrian Lawrence and Steven Beard; but since I have not given them an opportunity to correct me, I lay a personal claim to my mistakes.

Geraint Jones Tel: +44 865 273851
Programming Research Group geraint@uk.ac.oxford.prg
11 Keble Road
Oxford OX1 3QD
United Kingdom