Exploiting concurrency: a ray tracing example

INMOS Technical Note 7

Jamie Packer

October 1987
72-TCH-007-01

Contents

1 Introduction

The INMOS transputer [1] is a family of VLSI microcomputers with processor, memory and communication links for direct connection to other transputers on a single chip, Figure 1. Concurrent systems can be constructed from a collection of transputers which operate concurrently and communicate through links. To provide maximum speed with minimum hardware the transputer uses point to point serial communication links.

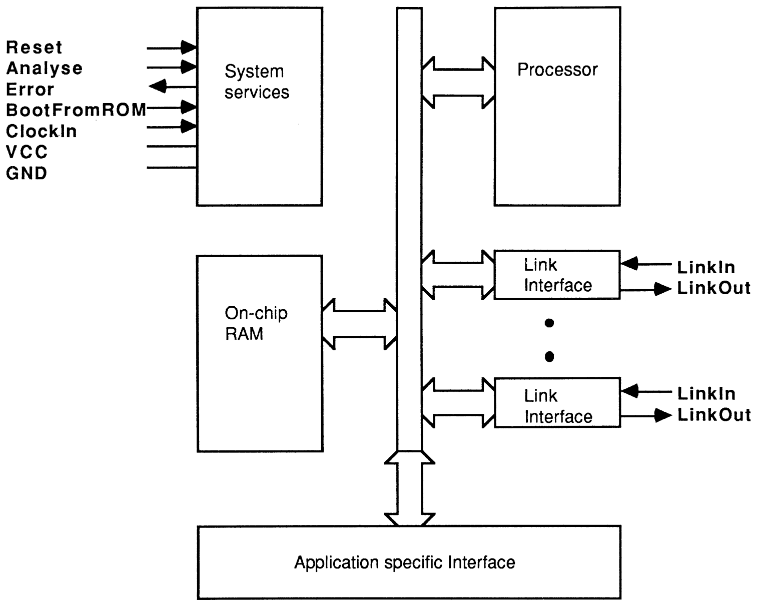

Figure 1: Transputer architecture

The first transputer available was the IMS T414, a 32 bit microprocessor with a throughput of 10 MIPs (million instructions per second). It has 2 kilobytes of fast (50ns cycle) on-chip static RAM and four INMOS serial links. The 32 bit multiplexed address/data bus allows up to 4 gigabytes of external memory to be accessed. The IMS T800 transputer is compatible with the T414 but includes floating point hardware and 4K of internal RAM.

This note describes the implementation of a computer graphics program on an array of transputers. The technique used to distribute the work among the transputers is known as a processor farm and is independent of the application. The same approach is suitable for any algorithm which can be subdivided into independent subproblems. For example, another graphics program, the Mandelbrot set, has been distributed in the same way as well as a financial forecasting program and a simulation of metal deposition. The entire program is written in occam [2], a language designed to simplify the programming of concurrent systems. Again, however, the main part of the program could have been written in any suitable language such as C or Fortran. Only those parts of the program which deal explicitly with concurrency and the distribution of work are easier to describe in occam.

The graphics program described here was written to provide a demonstration of the performance obtainable by using large numbers of transputers. We used a technique known as ray tracing which can generate very realistic images but requires massive amounts of computer power. This is an ideal application for transputers as the calculations for each pixel (picture element) on the screen are independent of one another and so can be done in parallel on separate processors. In addition, the complexity of the task means that the time spent calculating is much greater than that spent passing data between processors.

The completed program has two important properties. Firstly, processing speed is directly proportional to the number of transputers used. Virtually any desired performance can be simply obtained by the addition of more transputers. The second feature, which came about as a side effect of the program structure, is that the system is remarkably robust. Individual transputers can be removed from the system, while the program is running, and the system will continue to function although with reduced performance and possibly some loss of data.

2 Logical architecture

2.1 Ray tracing

The basic ray tracing algorithm used is that described by Turner Whitted in his classic paper [3]. A brief description of the technique is given here.

The colour and intensity of each pixel on the screen is determined by calculating the path of a ray projected from the screen through a pinhole, see Figure 2. This ray is tested for intersection with each object in the world model by solving the equation of the line and the surface of the object. This is the reason spheres are so common in ray tracing programs; they are simple to intersect. When the closest point of intersection is found the ray will be reflected and several new rays may be produced. If the object is transparent then a ray is generated which passes through the object, its path modified by the laws of refraction. The effects of shadow casting are handled by sending rays from the point of intersection towards each light source in turn. If this ray intersects an object which is nearer than the light source then this will cast a shadow on the first object. A recurring problem in computer graphics

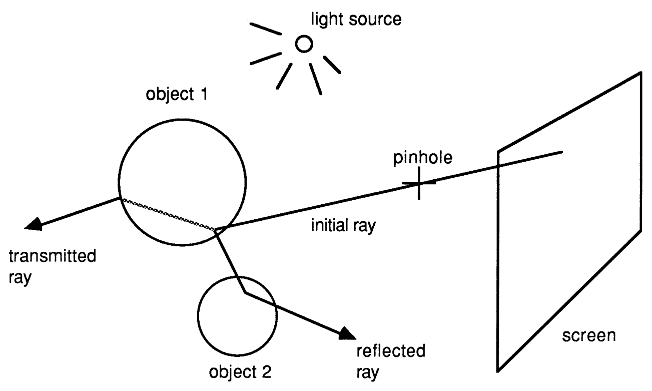

Figure 2: Ray tracing

is aliasing, which appears as coarse steps in the image. This is caused by undersampling of the image and can be reduced by increasing the sampling frequency, i.e. tracing several rays for each point on the screen. The number of extra rays traced can be reduced by only oversampling when aliasing is likely to be most objectionable, for instance where there is a sharp change in intensity at the boundary of objects.

In this way a tree of rays, and the surfaces with which they have intersected, is generated for each pixel. The final colour of the image at this point is calculated by traversing the tree and applying a shading model at each node. This model uses the intensity and positions of the various light sources and the coefficients of reflection and transmission for the objects intersected to determine the intensity of the pixel.

Our implementation of this algorithm, in its simplest form, is not particularly efficient. The time taken to render a scene increases exponentially with the number of objects and light sources as each ray has to be tested for intersection with every object and then a ray fired at every light source to test for shadowing. This shadow ray then has to be tested against every object. Also all calculations are done in floating point for simplicity which traditionally imposes a considerable performance penalty. It has been estimated that a straightforward ray tracing program like this will spend over 75% of the time performing tests for intersection, so performance is very dependent on the speed of floating point operations. The T414 has some extra instructions to provide microcode support for floating point operations and the T800's on-chip FPU enables floating point operations to be performed at about the same speed as integer operations.

There are many ways in which the basic ray tracing algorithm can be improved (e.g. by the use of space subdivision or bounding volumes around objects) so many implementations could well be faster on a single processor.

However, a more sophisticated implementation would also benefit from the use of multiple transputers.

2.2 Introducing concurrency

The calculations performed for each pixel on the screen are completely independent so they can be performed in any order and on any number of processors. One way of distributing the work to a number of processors is shown in Figure 3.

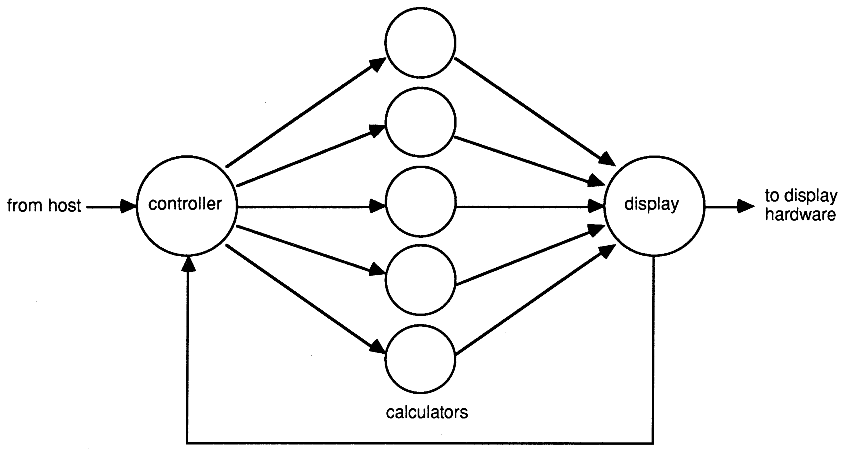

Figure 3: Logical architecture

This requires three different processes running concurrently on one, or more, processors: a controller which interfaces with the user or host computer to provide a description of the scene being viewed and allocate work to processors; an intersect and shading calculator, which can be replicated any number of times, to render the pixels; and a display process which collects the results from each rendering process and drives the graphic display. It can be seen that this structure is not related to the ray tracing algorithm and is, in fact, suitable for any problem which can be broken into independent subproblems. A system like this in which a controller farms out work to a number of application specific processes has become known as a processor farm.

Every calculating process is first given the description of the scene and then processing work can be allocated by the controller which gives each calculator pixels to evaluate. When the the calculations have been completed the results are passed out to the display process. The display process then informs the controller that there is now a free processor and another pixel is sent out for evaluation. The amount of computation required varies from pixel to pixel and this method automatically balances the load amongst the processors and ensures they are all kept busy.

An interesting idea here is that the pixels do not need to be generated in sequence and, if they are generated in some pseudo-random order, a good impression of the final picture can be obtained well before every pixel has been evaluated. This could be particularly useful in a computer aided design system where the user wishes to generate different views of an object in rapid succession.

3 Physical architecture

3.1 General description

It appears, at first sight, that the above architecture cannot be mapped directly onto a network of transputers because of the fixed number of links available. However, it is very simple to arrange for the controller to communicate with any transputer in a network by passing messages through the intervening transputers. For simplicity, the ray tracing program was mapped on to a linear array of transputers as shown in Figure 4. Each transputer link implements two occam channels, one in each direction, so this mapping only uses two of the four links available on a T414 or T800.

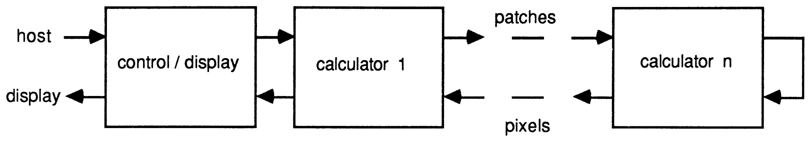

Figure 4: Physical architecture

Here the control and display processes are executed in parallel on one transputer and the rest of the transputers do the intersection and shading calculations. In fact the first transputer also does these calculations and the same, parameterised, program is loaded onto every transputer. However, it is simpler to view the system as shown above. This method of mapping processes onto transputers requires that each transputer also execute routing processes. These pass commands and data along the array from the controlling process and pass results back for display. This implies some sort of command protocol for identifying the nature and destination of data. This is simplified by using a linear connection of transputers; the routing process on each transputer only needs to decide whether a message is to be accepted locally or passed on to be dealt with elsewhere. A different array structure (e.g. a 2D array or a hypercube) could reduce the distance that messages have to pass and increase the bandwidth of data through the controller but at the cost of a slightly more complex protocol and routing process.

A few important points need to be made. Firstly, the work involved in designing and implementing this protocol is trivial compared to that required for the actual ray tracing algorithm and this will be true for any realistic program. Secondly, although two extra processes are being executed in parallel with the main ray tracing process, they actually consume very little processor time. Transputer processes are descheduled whilst waiting for communications to take place and so do not use the processor. They are automatically rescheduled, by the scheduling hardware, when the communication is complete. Also, external communication is done by the autonomous link DMA engines which can transfer data independently of, and concurrently with, the processor. This implies that the processing resource used by the communication depends more on the number of communications than the amount of data transmitted in each message.

The occam description of this transputer configuration has a constant defining the number of transputers in the network, which is all that needs to be changed if the size of the network is changed.

3.2 The control/display transputer

There are two processes executed by the control/display transputer, see Figure 5. The first of these, sendPatches, interfaces to the host computer to receive the description of the scene being modelled and other commands. It passes the world model out to all the other transputers and then sends out requests for pixels to be evaluated. Square areas of the screen, "patches", rather than individual pixels, are given to each transputer to enable "slices" or blocks of data to be transmitted. A slice communication transmits an array of data as a single operation. As there is the same processor overhead for setting up the links to transmit a single byte as for a million bytes, this make s the most efficient use of the transputer link engines. It also allows the processor to continue calculating at very nearly full speed while the communication takes place, with only occasional interruptions to manage the routing processes.

The other process, called loadBalance, coordinates the sending of data to the other transputers and the display of the generated pixels. If there are n transputers then loadBalance initially passes on 2n pixel patch requests from the process sendPatches. It then waits until a result is returned before handing out another request. So this process acts like a valve, only allowing work to be passed out when there are transputers able to accept it. Each of the n calculating transputers can accept two patch requests as described

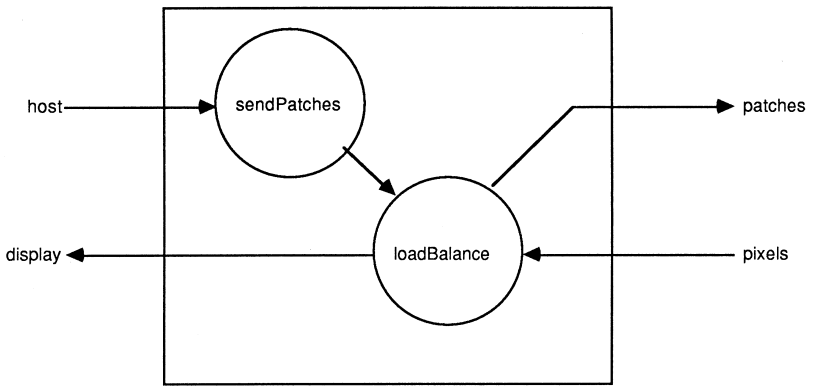

Figure 5: Processes running on control transputer

below.

3.3 The calculating transputers

The work on each of these transputers is organised as three processes shown in Figure 6. The most important of these is render which is sent patches to evaluate via the throughput process. The render process is a completely sequential piece of code and could be written in any standard programming language which supports communication over occam channels. It does all the calculations to find intersections, build the tree of rays and then traverse this tree to get the final pixel value. When all the pixels in the patch are evaluated then the pixels are passed out to the feedback process and another patch is requested from throughput. The feedback process multiplexes the local results and those received from other transputers and passes them back towards the display transputer. This process is very simple, using an occam ALT construct to wait for an input from either of the two channels.

The task of the throughput processes is to route patch requests through the pipeline to a free processor, i.e. one that is able to accept a pixel patch for evaluation. Patches can be routed to the next processor; routed to the local render process or buffered for local processing later. Initially each processor starts in the state busy = FALSE (not currently processing a patch) and buffered = FALSE. Patches are routed by throughput according to these state variables: if not busy then the patch is sent to the render process; if not busy and not buffered the patch is saved for later processing; otherwise the patch is passed on for processing elsewhere. Each processor therefore accepts two patches at startup, the first is passed immediately to render for evaluation and the second is held until needed. Any further patches received are passed on to be evaluated elsewhere until the processor becomes free

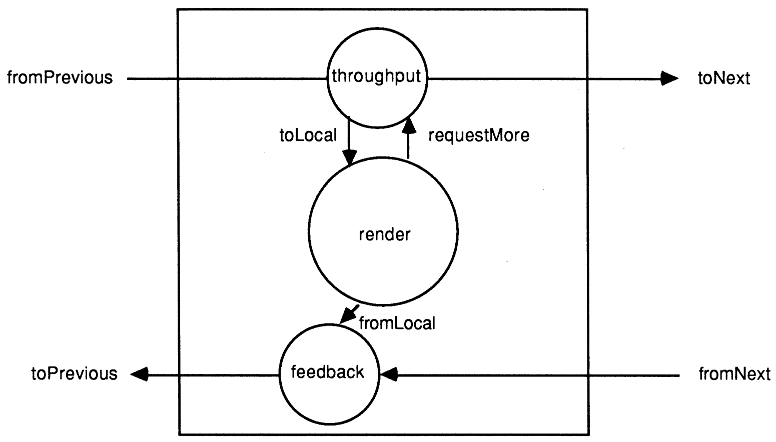

Figure 6: Processes running on the calculating transputers

again. After the first patch has been completed by the render process it sends a request to throughput for another. This is shown in the simplified piece of occam below:

```
BOOL busy, buffered, running :
BYTE byte :
[3]INT patch, buffered.patch :
SEQ
  -- initialise state variables
  busy := FALSE
  buffered := FALSE
  running := TRUE
  WHILE running
    ALT
      -- a request for another patch from render
      requestMore ? byte
        IF
          -- we have some work buffered, pass it on
          buffered
            SEQ
              toLocal ! buffered.patch
              buffered := FALSE
          -- else indicate that the renderer is free
          TRUE
            busy := FALSE
      -- a message from the pipeline
      fromPrevious ? CASE
        -- if it is a patch...
        rt.render; patch
          IF
            -- this processor not busy, pass to render
            NOT busy
              SEQ
                toLocal ! patch
                busy := TRUE
            -- if can't handle it here, pass to next transputer
            busy AND buffered
              toNext ! patch
            -- save patch for later processing
            busy AND (NOT buffered)
              SEQ
                buffered.patch := patch
                buffered := TRUE
        -- the terminate message
        rt.stop
          running := FALSE
```

Provided that the time taken to render a pair of patches is greater than the time before throughput receives a new patch, the render process is always kept busy. This provides distributed control of work allocation; each processor simply passes on any work that it cannot handle to be done elsewhere. It doesn't need to know where the work will be done or any other details of the system configuration. Because no more work requests are sent out than can be handled, the last processor in the network will never find itself with work to pass on to a nonexistent processor.

4 Maximising performance

The processing speed of the system is directly related to the number of transputers used; ten transputers perform ten times faster than one. A number of factors contribute to this aspect of the system.

The work is given to the transputers in large chunks which require only three words of data (the X and Y coordinates and size of the patch) to specify the position of all the pixels in the patch. If the work were distributed on a pixel by pixel basis then two words of data would be required for every pixel. This would mean a much larger ratio of communication to processing.

Use of slice communication for data means there is less processor overhead per byte sent and allows a greater amount of concurrency between the link engines and the processor. Allocating the work in chunks made this even more important as entire patches of pixels were returned to the control/display transputer as a single communication.

The message routing processes are run at high priority to ensure that an incoming message can be examined and forwarded immediately it is received.

The input guards of the ALT constructs in these processes are also carefully ordered in priority to ensure that patches are returned to the control processor as quickly as possible.

As well as holding an item of work in throughput, software buffers were added to any channels which communicate via a transputer link. These decouple the communication taking place via the link from the processes using the channel, thus allowing more overlap between processing and link communication. Channel buffers are frequently used, and easy to implement in occam.

These issues and others, such as efficient use of on chip RAM, are discussed in more detail in another INMOS technical note [4].

The performance of the system has been measured with up to 80 T414-15 transputers and the results are summarised below. These times were measured by the ray tracing system itself using the low priority transputer timer which has a resolution of 64 microseconds. The image generated consisted a simple scene containing 4 spheres and a single light source at a screen resolution of 256 x 256. The time taken with each number of transputers was averaged over four runs. The processing speed in the table below is the number of pixels generated per second, linearity was calculated as relative speed / transputers * 100.

transputers	**speed**	**relative speed**	**linearity %**
1	164.0	1.00	100.0
2	327.6	2.00	99.9
4	654.0	3.99	99.7
8	1296.4	7.91	98.8
16	2601.6	15.87	99.2
32	5189.5	31.65	98.9
64	10300.0	63.15	98.7
80	12500.0	76.37	95.5

The ray tracer has also been run on T800 processors showing a factor of about 6 or 7 speed improvement due to the on-chip floating point unit.

5 Fault tolerance of the system

It should be possible to exploit the number of processors in a multi-transputer system to introduce a degree of redundancy. The system described above is already remarkably robust. If a transputer fails then the system will progressively deadlock only if the neighbour, on the controller side, attempts to communicate with it. This is unlikely to occur, however, because results are passed back to the display by the shortest route, and new pixel patches are not sent out until results are returned. If a transputer is stopped while it is actually communicating, or between sending out results and being given its next patch of pixels then the system will deadlock. Otherwise, apart from the loss of of the processing power of the transputers on the far side of the fault, and the associated data, the system continues to operate.

In order to make the system more robust it must be possible to detect when a failure has occurred. This can be done by using a timeout on all communications. Secondly it must be possible to ensure that, even if a communication does fail, all the input and output processes will terminate. As this cannot be achieved directly in occam, INMOS provide a number of predefined procedures which perform the desired functions. These allow an input or output to be attempted within a time limit, and recovery from a failed communication. They are described more fully in INMOS Technical Note 1, "Extraordinary use of transputer links" [5]. The use of these procedures means that failure of a transputer can be detected by its neighbour. The controlling transputer could then be informed and so take action to recover or regenerate the lost data.

Detection of the failure of a transputer implies that facilities could be added to allow the defective transputer to be bypassed. This can be done with no extra hardware as shown in Figure 7. If a transputer decides that its neighbour has failed then it switches to the other link to communicate with the next transputer along. Alternatively, if boards with more than one transputer are used (for example the IMS B003) it may be better to arrange the link connections so that an entire board is bypassed if a failure is detected. Obviously, this will not be sufficient if two adjacent transputers or boards were to fail, but this unlikely event could be catered for with extra hardware to allow link connections to be switched externally thus allowing any number of devices to be bypassed.

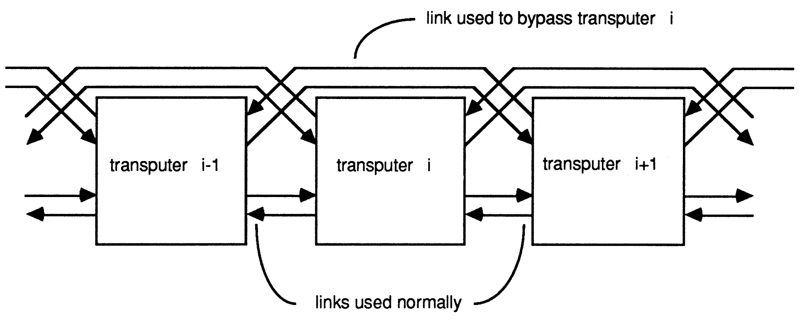

Figure 7: Bypassing a failed transputer

6 Ray tracing in occam

The occam language enables a system to be described as a collection of concurrent processes which communicate with one another, and with the outside world, via channels. Occam programs are built from three primitive processes: assignment (variable := expression); input (channel ? variable); and output (channel ! expression).

Each occam channel provides a one way communication path between two concurrent processes. Communication is synchronised and unbuffered. The primitive processes can be combined to form constructs which are themselves processes and can be used as components of another construct. Conventional sequential programs can be expressed by combining processes with the sequential constructs SEQ, IF and WHILE. Concurrent programs are expressed using channel communication, the parallel construct PAR and the alternative construct ALT. An alternative process may be ready for input from a number of channels, input is taken from the first of the channels to become ready.

This is an outline of the occam program for each transputer, and the description of the entire transputer system. The procedures all have several more parameters (such as screen size, maximum number of reflections etc.) but, for simplicity, only the essential outline is given here.

In order to pass the various types of messages (e.g. object definitions, patch requests and pixel values) around the system a variant protocol was used:

```
PROTOCOL trace.p
  CASE
    rt.stop
    rt.done
    rt.render;   [3]INT              -- rt.render; x; y; patchSize
    rt.data;     INT; INT::[]INT     -- rt.data; type; data
    rt.pixels;   INT; INT; INT::[]INT -- rt.pixels; x; y; n::data
    rt.message;  INT::[]INT          -- rt.message; n::chars
:
```

Each message then consists of: the message tag followed by the arguments. For example a 16x16 patch is sent as: `out ! c.render; [x; y; 16]`.

The code running on the control/display transputer is:

```
PROC control (CHAN OF trace.p fromHost, toDisplay,
                   toCalculators, pixelsIn)

  ...  definition of sendPatches procedure
  ...  definition of loadBalance procedure

  CHAN OF trace.p data :
  PAR
    sendPatches (fromHost, data)
    loadBalance (data, toCalculators, pixelsIn, toDisplay)
:
```

Each of the calculating transputers runs the following code:

```
PROC calculate (CHAN OF trace.p fromPrev, toNext, fromNext, toPrev)

  ...  throughput procedure
  ...  render
  ...  feedback

  CHAN OF trace.p toLocal, fromLocal, requestMore :
  PRI PAR
    -- run these at high priority for
    -- fastest response to messages
    PAR
      throughput (fromPrev, toNext, toPrev, toLocal, requestMore)
      feedback (fromLocal, fromNext, toPrev)

    -- and this one at low priority
    render (toLocal, fromLocal, requestMore)
:
```

The system description is as follows:

```
... define constants for the link addresses

VAL number.transputers IS 42 :
VAL last IS number.transputers - 1 :

CHAN OF trace.p host, display, loopback :
[number.transputers]CHAN OF trace.p forward, return :

PLACED PAR

  --  processor 0 is the control/display processor
  PROCESSOR 0 T4
    PLACE host       AT link0in  : --  data from host
    PLACE display    AT link2out : --  to display
    PLACE forward[0] AT link1out : --  patches out
    PLACE return[0]  AT link1in  : --  pixel values back
    control (host, display, forward[0], return[0])

  -- the main body of the pipeline of calculators
  PLACED PAR i = 1 FOR number.transputers - 2
    PROCESSOR i T8
      PLACE forward[i]   AT link0in  : --  patches in
      PLACE return[i]    AT link1out : --  pixels out
      PLACE forward[i+1] AT link1out : --  patches out
      PLACE return[i+1]  AT link0in  : --  pixels in

      calculate (forward[i], forward[i+1], return[i+1], return[i])

  -- the last transputer is a special case as it
  -- has no one else to talk to. The fact that the
  -- channel ‘loopback’ is not placed means that
  -- an internal ("soft") channel will be created.
  -- In fact this channel is never used but is
  -- required as a parameter.

  PROCESSOR last T8
    PLACE forward[last] AT link0in  :
    PLACE return[last]  AT link0out :

    calculate (forward[last], loopback, loopback, return[last])
```

Further information on the program is available from the Central Applications Group at INMOS Ltd in Bristol.

References

[1] Transputer reference manual
INMOS Ltd Bristol

[2] Occam reference manual
INMOS Ltd Bristol

[3] ”An Improved Illumination Model for Shaded Display” Turner Whitted.
Communications of the ACM, pp. 343-349, June 1980, 23(6)

[4] Technical note 17: ”Performance Maximisation” Phil Atkin
INMOS Ltd Bristol

[5] Technical note 1: ”Extraordinary use of transputer links” Roger Shepherd
INMOS Ltd Bristol