12

Monitoring communication

Sorting is a candidate problem for parallel solution because many algorithms have an element of divide-and-conquer. That means the task is carried out by dividing it into some number of smaller, simpler tasks each of which is repeatedly divided until only trivial tasks remain. Such a strategy rapidly discovers independent parts of the original problem which can be tackled concurrently.

This chapter describes a tree-shaped parallel sorting program, and a way of observing it whilst it is running. No claims are made for the sorting algorithm used, beyond its simplicity, because the real subject is how to observe a parallel program in operation. With very small changes to the sorting program itself, its activity can be displayed on a VDU screen, turning the program into a simulator of its own behaviour.

The example: a sorter

The program developed here will work equally well *mutatis mutandis* for sorting a collection of any type of object which has a less-than relation computable on it. In occam 2 any particular coding of the program can operate on only one data type; for definiteness and simplicity the program in this chapter deals with `INT` data with the ordinary comparison operations. The program consists of a number of simple processes linked together in a tree-shaped structure. As in the case of the matrix multiplier, no process need ever know where it is in the tree. There will be only two types of process: leaves, and internal nodes. Each process is also independent of the size of the problem, and need never store more than two values and some flags, no matter how many values are being sorted. Any particular tree will sort collections which are no larger than some fixed size; a bigger problem demands a bigger tree, but the components are unchanged.

The strategy is to distribute the numbers upwards from the root of the tree, until they are spread out, at most one to each leaf. Each process is then responsible for sending back to its parent the sequence of numbers which it has received, but sorted into ascending order. For a leaf, the task is simple, since its one number, if it has one, already constitutes a sequence in ascending order. Each internal node, relying on the fact that the subsequences which it will receive from its children will be sorted, merges two ascending sequences to generate its output sequence.

The number of data items that must be accepted by a node depends on the position of the node in the tree, and as this is not known to the process an indication of the end of the data stream must be communicated to it. A protocol for just such an indefinitely long sequence of integers was suggested in Chapter 8:

```
PROTOCOL INT.STREAM
  CASE
    another.int; INT
    no.more.ints
:
```

Each leaf process needs two-way communication with its parent,

```
PROC leaf(CHAN OF INT.STREAM from.parent, to.parent)
  ...
:
```

and each internal node needs six channels, two to provide two-way communication with its parent, and two each to and from each of its children:

```
PROC fork(CHAN OF INT.STREAM from.parent, to.parent,
                             from.left.child,
                             to.left.child,
                             from.right.child,
                             to.right.child         )
  ...
:
```

The root of the tree has both to manage its children and to interact with the outside world; we may separate these concerns by supplying one process which behaves as an internal node, and a virtual root process which takes the place of the parent of the root

```
PROC driver(CHAN OF INT.STREAM up.to.tree, down.from.tree)
  ...
:
```

and acts as a driver to control the activity of the tree.

To connect these processes, they have to be indexed, so as to correspond to arrays of channels. For simplicity, let us assume we want to be able to sort collections of size up to some power of two, and so program a complete balanced tree of depth *depth.of.tree*. Although any tree can be drawn in two dimensions, this does not map comfortably onto a two-dimensional array; there are exponentially, not quadratically, many nodes compared with the depth of the tree. The desired structure can be built within linear arrays, effectively using array indices in the place of the pointers of a language like PASCAL – a method easily extended to model any tree. First define

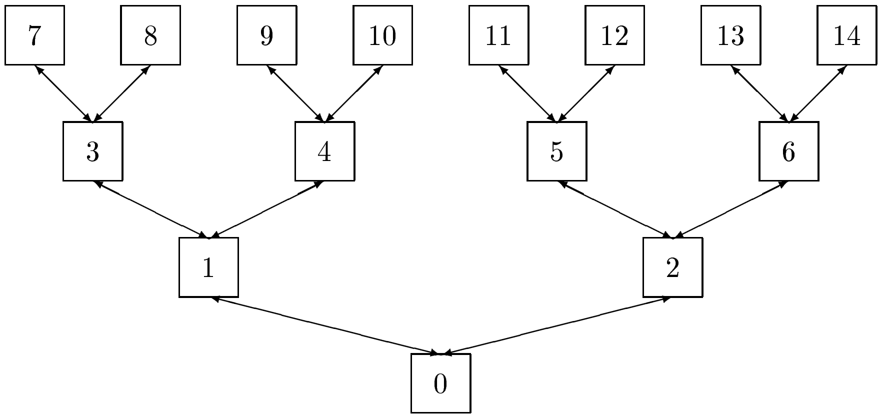
```
VAL INT number.of.leaves    IS 1 << depth.of.tree :
VAL INT number.of.forks     IS number.of.leaves - 1 :
VAL INT number.of.processes IS
                  number.of.forks + number.of.leaves :
VAL INT number.of.channels  IS number.of.processes :
```

then, numbering the processes breadth-first, upwards from the root

```
VAL INT root       IS 0 :
VAL INT first.fork IS root :
VAL INT first.leaf IS first.fork + number.of.forks :
```

the children of the internal node process i are indexed $2i + 1$ and $2i + 2$; Figure 12.1 shows the enumeration of a particular size of tree.

Figure 12.1 Enumeration of the process tree, *depth.of.tree* $= 3$



If channels indexed i are used to connect process i to its parent, these same formulae will give the indices of the channels to and from the children of internal node process i, so the program fits together as Process 12.1. Notice that the channels passed to a *fork* as *to.left.child* and *to.right.child* are consecutive elements of the *up* array, and similarly the *from.—.child* are consecutive in *down*. Since the *fork* process will deal with its children quite

Process 12.1

```
[number.of.channels]CHAN OF INT.STREAM up, down :
PAR
  driver(up[root], down[root])
  PAR i = first.fork FOR number.of.forks
    VAL INT left  IS (2 * i) + 1 :
    VAL INT right IS (2 * i) + 2 :
    fork(up[i], down[i],
         down[left], up[left],
         down[right], up[right] )
  PAR i = first.leaf FOR number.of.leaves
    leaf(up[i], down[i])
```

symmetrically, it will pay to exploit this fact by passing those two-element segments sliced from the whole, rather than the individual channels, for communication with the children; for this will allow `FOR` loops which range over the children.

This decision means that the interface to the *fork* process must be changed, and its arguments must now be specified as

```
PROC fork(CHAN OF INT.STREAM from.parent, to.parent,
          [2]CHAN OF INT.STREAM from.children, to.children )
  ...
:
```

and with the corresponding call the program becomes Process 12.2.

Process 12.2

```
[number.of.channels]CHAN OF INT.STREAM up, down :
PAR
  driver(up[root], down[root])
  PAR i = first.fork FOR number.of.forks
    VAL INT first.born IS (2 * i) + 1 :
    fork(up[i], down[i],
         [down FROM first.born FOR 2],
         [up   FROM first.born FOR 2] )
  PAR i = first.leaf FOR number.of.leaves
    leaf(up[i], down[i])
```

Components of the sorter

There are two phases of activity in the tree: first the sequence of numbers is distributed; then sorted sequences are gathered and merged. Each component process passes through the same two phases, so the *fork* procedure can be decomposed into a sequence of these two phases, as in Process 12.3.

Process 12.3

```
PROC fork(CHAN OF INT.STREAM from.parent, to.parent,
          [2]CHAN OF INT.STREAM from.children, to.children )
  SEQ
    fork.distribute(from.parent, to.children)
    fork.gather(to.parent, from.children)
:
```

During the distribution phase, each internal node receives a sequence of numbers from its parent. The simplest way of distributing the sequence amongst the children, without foreknowledge of its length, is to send one-for-left, one-for-right, alternately (Process 12.4). Notice that the termination signal is eventually broadcast to the children, whether the distributing process has zero, one or more values to share out.

Process 12.4

```
PROC fork.distribute(CHAN OF INT.STREAM from.parent,
                     [2]CHAN OF INT.STREAM to.children )
  VAL INT left IS 0 :
  VAL INT right IS 1 :
  INT child :
  BOOL more :
  SEQ
    child, more := left, TRUE
    WHILE more
      from.parent ? CASE
        INT number :
        another.int; number
          SEQ
            to.children[child] ! another.int; number
            child := child >< (left >< right)
        no.more.ints
          more := FALSE
    PAR child = left FOR 2
      to.children[child] ! no.more.ints
:
```

Since the component processes are to serve in an arbitrarily large tree, they should not count the numbers as they pass upwards. This means that during the merging phase there are again sequences of unknown length to be read, and it is correct to use the same protocol.

To preserve the ordering during the merge, numbers must be compared, and this requires that each process have at least two registers holding numbers. Since each child sends its sequence in ascending order, the head of each sequence is the minimum of those to come. The merging

Process 12.5

```
PROC fork.gather(CHAN OF INT.STREAM to.parent,
                 [2]CHAN OF INT.STREAM from.children )
  VAL INT left IS 0 :
  VAL INT right IS 1 :
  [2]BOOL more :
  [2]INT minimum :
  SEQ

    PAR child = left FOR 2
      from.children[child] ? CASE
        another.int; minimum[child]
          more[child] := TRUE
        no.more.ints
          more[child] := FALSE

    WHILE more[left] OR more[right]
      IF child = left FOR 2
        VAL INT other IS child >< (left >< right) :
        more[child] AND
                ((NOT more[other]) OR
                 (minimum[child] <= minimum[other]))
          SEQ
            to.parent ! another.int; minimum[child]
            from.children[child] ? CASE
              another.int; minimum[child]
                SKIP
              no.more.ints
                more[child] := FALSE

    to.parent ! no.more.ints
:
```

process compares the heads, passes on the smaller, and draws one more value from the selected sequence, continuing until the sequences are exhausted (Process 12.5). If there are less than *number.of.leaves* values to sort then one or more children of some *fork* will be sorting an empty sequence, so both cases of the initial discriminated input may arise.

The leaf process (Process 12.6) must be designed to simulate the behaviour of an internal node that handles a sequence of at most one number.

Finally, the driver process must generate and absorb sequences of numbers, stuffing and stripping the protocol. The sections of code missing from Process 12.7 control the behaviour of the whole program. They might, for example, read numbers from the terminal keyboard, and write them back, in ascending order, to the terminal screen.

That completes the sorting program which, whilst it may look over-complex for a single processor implementation, would look better on an array

Process 12.6

```
PROC leaf(CHAN OF INT.STREAM from.parent, to.parent)
  SEQ
    from.parent ? CASE
      INT number :
      another.int; number
        SEQ
          from.parent ? CASE no.more.ints
          to.parent ! another.int; number
      no.more.ints
        SKIP
    to.parent ! no.more.ints
:
```

Process 12.7

```
PROC driver(CHAN OF INT.STREAM up.to.tree, down.from.tree)
  SEQ
    BOOL more.output :
    INT i :
    SEQ
      i := 0
      ... initialize more.output
      WHILE (i < number.of.leaves) AND more.output
        INT number :
        SEQ
          ... think of a number, and whether any more
          up.to.tree ! another.int; number
          i := i + 1
      up.to.tree ! no.more.ints
    BOOL more.input :
    SEQ
      more.input := TRUE
      WHILE more.input
        down.from.tree ? CASE
          INT number :
          another.int; number
            ... do something with the number
          no.more.ints
            more.input := FALSE
:
```

of *number.of.processes* simple processors. Notice, particularly, that once the numbers have started to emerge from the tree in ascending order, each is available only one comparison time after its predecessor. This advantage would be more obvious were the sorter managing more complex data, where the comparison time might be very large.

Monitoring strategy

The program as it stands can be run on a single processor to simulate the activity of the ideal multi-processor implementation. By writing the missing code in the driver, you could observe numbers going into and coming out of the tree, checking that the program sorts particular sequences of numbers. That tells you nothing about what goes on inside the tree. To get a better idea of *how* the program sorts sequences you might also want to be able to watch the activity in the branches of the tree.

By analogy with the testing of electronic circuits, the idea is to probe the components of the circuit, rather than just watching the signals that pass into and out of the terminals. There are two techniques: breaking connections to measure the current flowing through them corresponds to tapping the channels to watch the traffic; attaching probes to measure the potential at various points in a circuit corresponds to noting the state in each process.

In order to observe the traffic on a channel, a process must be added which duplicates the traffic along a monitoring channel. Suppose for instance that the data stream passing along a channel between a *producer* and a *consumer* process

```
CHAN OF INT channel :
PAR
  producer(channel)
  consumer(channel)
```

is to be sampled. Then coding a *duplicate* process after the fashion of Process 12.8 allows a copy of the communications to be passed on to a third party, a process *monitor*, say

```
VAL INT collect IS 0 :
VAL INT deliver IS 1 :
[2]CHAN OF INT channel :
CHAN OF INT test.data :
PAR
  producer(channel[collect])
  duplicate(channel[collect], channel[deliver], test.data)
  consumer(channel[deliver])
  monitor(test.data)
```

Of course, the observation is not perfect: it may affect the behaviour of the program. First of all, the *duplicate* process acts as an additional buffer in the data stream. In this example it cannot matter, but were there some other communication, possibly mediated by another process, between the *producer* and *consumer*, it might matter that the output from the *producer* could

Process 12.8

```
PROC duplicate(CHAN OF INT source, sink, copy)
  WHILE TRUE
    INT datum :
    SEQ
      source ? datum
      PAR
        copy ! datum
        sink ! datum
:
```

proceed, despite the corresponding demand not being made in the *consumer*. Secondly, the *duplicate* process, as written, does not terminate, so unless it is used to observe an infinite data stream, the program will eventually become deadlocked, even had it previously terminated correctly.

In both of these ways you must be careful to design monitoring code that does not interfere excessively with the activity being observed. In general, it is necessary for the behaviour of the monitoring processes to depend on the data passing through them, and this in-stream technique should be avoided if there are many parallel data paths between pairs of processes in the program being observed. Also, whatever care is taken, the performance of the program is almost certain to change; so one cannot hope to successfully monitor time-dependent behaviours.

The other form of observation, making internal state visible, requires additional code inserted into the processes being observed. Just as observing traffic involves adding new output processes in parallel with the observed program, so observing state requires that new output processes be set in sequence with the code being observed.

```
PROC p(...)                PROC p(..., CHAN OF INT test.data)
  INT x :                    INT x :
  SEQ                        SEQ
    ...                        ...
                               SEQ
    x := e                       x := e
             becomes             test.data ! x
    ...                        ...
                               SEQ
    c ? x                        c ? x
                                 test.data ! x
    ...                        ...
:                          :
```

In order to observe the changing value of a variable each assignment to that variable should be followed by an output process signalling the change on a channel which passes out to the monitoring code. Again, the observation is invasive: you must be aware that the observed process may be delayed by executing the new output processes.

In the example of the parallel sorter, we will use both types of monitoring: the explanation of the behaviour of the merging is in terms of the sequences of values passing along channels, so the traffic along the channels will be watched; the leaves are used as storage locations, so it is appropriate to observe their state. The result of adding this monitoring code is a number of channels emerging from the tree. Messages from each will be treated similarly: to write a number to, or to remove it from, a position on the screen which will represent the place in the program which is being watched. Since changes to the screen must be made in sequence, it is appropriate to multiplex the test data from the tree, and process each new test signal in sequence.

In order to get around the problem of termination, and to show the delivery as well as the collection of values sent along tapped channels, we will adopt a slightly more complex scheme. There are three types of message to be sent along the monitoring channels: messages indicating the arrival of a number, messages indicating the departure of a number, and a final termination message. These will conform to the protocol

```
PROTOCOL DISPLAY.STREAM
  CASE
    display.number; INT
    display.empty
    display.stop
:
```

This protocol will serve equally for the intrusive probes in the leaves. The output of the multiplexer will also carry the same kinds of message, but the data must be accompanied by a note of which probe it comes from:

```
PROTOCOL MULTIPLEX.STREAM
  CASE
    multiplex.number; INT; INT  -- probe; data
    multiplex.empty; INT        -- probe
    multiplex.stop              -- all probes stopped
:
```

Putting this all together leads to the changed program structure of Process 12.9. Each *monitor* process copies data from its *collect* channels to its *deliver* channels, duplicating the activity along the corresponding *probe*. Every leaf is modified to indicate its state with similar messages. All of these messages are multiplexed onto a single channel, and then translated into sequences of instructions to display the changing state of the program on the terminal screen.

Process 12.9

```
VAL INT collect IS 0 :
VAL INT deliver IS 1 :
VAL INT number.of.probes IS number.of.channels+number.of.leaves :
VAL INT first.channel.probe IS 0 :
VAL INT first.leaf.probe    IS number.of.channels :

[2][number.of.channels]CHAN OF INT.STREAM up, down :
[number.of.probes]CHAN OF DISPLAY.STREAM probe :
CHAN OF MULTIPLEX.STREAM all.probes :
PAR
  driver(up[collect][root], down[deliver][root])
  PAR
    PAR i = first.fork FOR number.of.forks
      VAL INT first.born IS (2 * i) + 1 :
      fork(up[deliver][i], down[collect][i],
           [down[deliver] FROM first.born FOR 2],
           [up[collect] FROM first.born FOR 2]   )

    []CHAN OF DISPLAY.STREAM leaf.probe IS
         [probe FROM first.leaf.probe FOR number.of.leaves] :
    PAR i = first.leaf FOR number.of.leaves
      leaf(up[deliver][i], down[collect][i],
                                  leaf.probe[i - first.leaf] )

    []CHAN OF DISPLAY.STREAM channel.probe IS
         [probe FROM first.channel.probe FOR number.of.channels] :
    PAR i = root FOR number.of.channels
      monitor(up[collect][i], down[collect][i],
              up[deliver][i], down[deliver][i], channel.probe[i] )

  multiplex(probe, all.probes)
  display(all.probes, terminal.screen)
```

Component processes

The monitoring processes will guarantee that the communications on a *probe* will be a sequence in which *display.number* and an accompanying number alternates with *display.empty*, the whole sequence ending in a single *display.stop*. Sending an explicit termination signal insulates the monitoring code from changes to the protocol operated in the sorter.

To start with the *leaf* process, all that is needed is to indicate the arrival and departure of the stored number. There are three things which the monitoring code must do: display a number arriving while the tree is filled; clear the display when that number is unloaded; and indicate the end of the leaf's activity. The modified *leaf* is implemented by Process 12.10. Notice that if a *leaf* has no number to sort, because the input stream is smaller than

Process 12.10

```
PROC leaf(CHAN OF INT.STREAM from.parent, to.parent,
          CHAN OF DISPLAY.STREAM probe              )
  SEQ
    from.parent ? CASE
      INT number :
      another.int; number
        SEQ
          probe ! display.number; number
          from.parent ? CASE no.more.ints
          to.parent ! another.int; number
          probe ! display.empty
      no.more.ints
        SKIP
    to.parent ! no.more.ints
    probe ! display.stop
:
```

Process 12.11

```
PROC monitor(CHAN OF INT.STREAM up.collect, down.collect,
                                up.deliver, down.deliver,
             CHAN OF DISPLAY.STREAM probe                )
  PROC copy.monitoring(CHAN OF INT.STREAM collector, deliverer,
                       CHAN OF DISPLAY.STREAM probe           )
    BOOL more :
    SEQ
      more := TRUE
      WHILE more
        collector ? CASE
          INT number :
          another.int; number
            SEQ
              probe ! display.number; number
              deliverer ! another.int; number
              probe ! display.empty
          no.more.ints
            more := FALSE
      deliverer ! no.more.ints
  :
  SEQ
    copy.monitoring(up.collect, up.deliver, probe)
    copy.monitoring(down.collect, down.deliver, probe)
    probe ! display.stop
:
```

the size of the tree, it nonetheless carries on the same conversation with its parent as do all children. The monitoring code abstracts away from this, and displays no transactions for such a process.

The *monitor* process must copy the sequences of numbers passing first up and then down the tree. The necessary monitoring code is just what you would need to record changes of state in this buffering process. The activity of the *monitor*, Process 12.11, falls into three stages: being a monitored buffer for a sequence of upward communications; being a monitored buffer for a sequence of communications returning downwards; and finally marking the end of the monitoring signals on its *probe*.

Each of the probes coming out of the sorter carries a sequence of display instructions, the last of which is a *display.stop*, so they can all be treated uniformly from now on. The *multiplex* process, Process 12.12, simply gathers together all of the probe signals, tagging them with the corresponding index number for later identification. Once a *display.stop* is received from a particular probe, no more signals are read from it, and the multiplexer terminates when all probes have been shut off.

Process 12.12

```
PROC multiplex([]CHAN OF DISPLAY.STREAM probe,
               CHAN OF MULTIPLEX.STREAM all.probes )
  INT active.probes :
  [number.of.probes]BOOL more.from :
  SEQ
    active.probes := number.of.probes
    SEQ i = 0 FOR number.of.probes
      more.from[i] := TRUE
    WHILE active.probes > 0
      ALT i = 0 FOR number.of.probes
        more.from[i] & probe[i] ? CASE
          INT number :
          display.number; number
            all.probes ! multiplex.number; i; number
          display.empty
            all.probes ! multiplex.empty; i
          display.stop
            active.probes, more.from[i] :=
                                 active.probes - 1, FALSE
    all.probes ! multiplex.stop
:
```

Managing the output from the simulation

It remains only to translate the stream of probe messages into a stream of terminal-screen control messages. The first thing to do is to translate the probe numbers into positions on the screen. This happens in two stages: first the numbers are translated into positions in a terminal-independent space; those positions and the data to display are then passed to a process which maps that space onto a particular type of terminal screen. So the *display* process decomposes as Process 12.13.

Process 12.13

```
PROC display(CHAN OF MULTIPLEX.STREAM source,
             CHAN OF BYTE sink               )
  CHAN OF TERMINAL.STREAM internal :
  PAR
    independent(source, internal)
    dependent(internal, sink)
:
```

The terminal-independent space has right-handed co-ordinates, with the leaves evenly spread across the top, and the root at the middle of the bottom line, as in Figure 12.2.

Figure 12.2 The abstract screen layout

```
depth.of.tree + 2 |X  X  X  X          ...         X  X  X  X
depth.of.tree + 1 |X  X  X  X          ...         X  X  X  X
  depth.of.tree   | X     X            ...          X     X
                  :
        2         |          X         ...     X
        1         |                    X
                  ------------------------------------------
                   1 2 3 4 5 6 7  ...  number.of.leaves  ...
```

Messages from probes with indices less than *number.of.channels* are from probes within the tree, and those with higher indices are from the leaves. The top line, representing the states of the leaves, is clearly not a part of the pattern in the rest of the tree, so is dealt with differently.

For the internal probes, the simplest solution is to count up from the root. There are $2^{line} - 1$ probes represented on the bottom *line* lines of the display, so the line for a particular probe is the first for which its index lies below this number. The column is calculated by discounting the $2^{line-1} - 1$ probes displayed lower down and multiplying by a factor to account for the exponentially greater separation of nodes at greater depths. The *make.cartesian* function, Process 12.14, translates a channel probe index into an $\langle x, y \rangle$ pair satisfying

$$1 \leq x \leq ((2 \times number.of.channels) - 1)$$
$$1 \leq y \leq depth.of.tree + 1$$

and places leaf probes on the line above the leaves

$$1 \leq x \leq ((2 \times number.of.channels) - 1)$$
$$y = depth.of.tree + 2$$

each one immediately above the corresponding channel.

Process 12.14

```
INT, INT FUNCTION make.cartesian(VAL INT index)
  INT x, y :
  VALOF
    IF
      IF line = 1 FOR depth.of.tree + 1
        index < ((1 << line) - 1)
          VAL INT column IS index - ((1 << (line - 1)) - 1) :
          VAL INT spread IS number.of.leaves >> (line - 1) :
          x, y := ((2 * column) + 1) * spread, line
      index >= number.of.channels
        VAL INT column IS index - number.of.channels :
        x, y := (2 * column) + 1, depth.of.tree + 2
    RESULT x, y
:
```

Process 12.15

```
PROC independent(CHAN OF MULTIPLEX.STREAM source,
                 CHAN OF TERMINAL.STREAM sink    )
  [field.width]BYTE blanks :
  BOOL running :
  SEQ
    SEQ i = 0 FOR field.width
      blanks[i] := '*s'
    running := TRUE
    WHILE running
      source ? CASE
        INT index, number :
        multiplex.number; index; number
          INT x, y :
          [field.width]BYTE buffer :
          SEQ
            x, y := make.cartesian(index)
            make.decimal.string.int(buffer, number)
            sink ! terminal.item; x; y; buffer
        INT index :
        multiplex.empty; index
          INT x, y :
          SEQ
            x, y := make.cartesian(index)
            sink ! terminal.item; x; y; blanks
        multiplex.stop
          running := FALSE
    sink ! terminal.stop
:
```

The `TERMINAL.STREAM` protocol could be very similar to the others, indicating the presence (and value) or absence of a number; but in order to keep the terminal-dependent part of the translation as simple as possible the actual characters to display are generated here. If a number is present it must be turned into digits, otherwise blanks must be sent. All the numbers are to be written in a fixed-width field, and so the output from *independent* consists of a sequence of packets, each beginning with an $\langle x, y\rangle$ pair, followed by the *field.width* characters to be displayed there. Each packet is preceded by a *terminal.item* tag, and the whole sequence is terminated with a *terminal.stop*. So the protocol `TERMINAL.STREAM` is, in fact,

```
PROTOCOL TERMINAL.STREAM
  CASE
    terminal.item; INT; INT; [field.width]BYTE
    terminal.stop
:
```

and the terminal-independent process will then be Process 12.15.

Displaying the output on a screen

If the terminal has general cursor addressing, then the task is almost complete. For example, we will assume an ANSI standard terminal which can clear the screen, Process 12.16, and move the cursor around the screen, Process 12.17.

Process 12.16

```
PROC clear.screen(CHAN OF BYTE terminal)
  -- clear screen sequence for an ANSI terminal
  write.string(terminal, "*#1B[2J")
:
```

Process 12.17

```
PROC move.cursor(CHAN OF BYTE terminal, VAL INT x, y)
  -- left-handed co-ordinates, origin 0,0 at top left
  CHAN OF DATA.ITEM c :
  PAR
    write.formatted(terminal, "*#1B[%d;%dH", c)
    SEQ
      c ! data.int; y + 1
      c ! data.int; x + 1
:
```

The terminal-independent co-ordinate space has a size which depends only on the depth of the tree

```
VAL INT virtual.height IS depth.of.tree + 1 :
VAL INT virtual.width  IS (2 * number.of.leaves) - 1 :
```

but the terminal screen has a fixed size

```
VAL INT screen.height IS 24 :
VAL INT screen.width  IS 80 :
```

If the picture of the tree is to be fitted onto the screen, it must be scaled

```
VAL INT height.scale  IS
               (screen.height - 1) / virtual.height :
VAL INT width.scale   IS
               (screen.width - field.width) / virtual.width :
```

and translated into the left-handed co-ordinate system of the terminal. The necessary terminal-dependent part of the display process is Process 12.18.

Process 12.18

```
PROC dependent(CHAN OF TERMINAL.STREAM source,
               CHAN OF BYTE terminal          )
  -- terminal-dependent code for driving ANSI terminal
  BOOL more :
  SEQ
    clear.screen(terminal)
    more := TRUE
    WHILE more
      source ? CASE
        INT x, y :
        [field.width]BYTE buffer :
        terminal.item; x; y; buffer
          SEQ
            move.cursor(terminal, (x - 1)  * width.scale,
                (virtual.height - (y - 1)) * height.scale )
            write.string(terminal, buffer)
        terminal.stop
          more := FALSE
    move.cursor(terminal, 0, screen.height - 1)
:
```

The division of work between *independent* and *dependent* is such that, if it is at all reasonable to draw such pictures on a particular terminal, the program can be modified to do so simply by writing the appropriate *dependent* process. Even should the terminal not have full cursor control, but only the ability to move the cursor in small steps, *dependent* can be made to keep track of the position of the cursor.

Driving the simulation

For the purpose of the simulator, a simple coding of the driver process invents a random sequence of numbers for input to the tree, and discards the sorted sequence which is returned by the tree.

A common way of generating an unpredictable sequence of numbers is to use a linear feedback shift register, Process 12.19, with an uncontrolled initial state. An arbitrary initial state can be obtained by reading the real-time clock. Since the state of the shift register can never change if it is zero, the time has its bottom bit set to guarantee a non-zero initial state.

Process 12.19

```
VAL INT mask IS BITNOT ((BITNOT 0) << 9) :

PROC shift(INT state)
  SEQ i = 1 FOR 9
    state := ((state << 1) /\ mask) \/
             (((state >> 4) >< (state >> 8)) /\ 1)
:
```

Process 12.20

```
PROC driver(CHAN OF INT.STREAM up.to.tree, down.from.tree)
  TIMER clock :
  SEQ
    INT event, number :
    SEQ
      clock ? event
      number :=  (event /\ mask) \/ 1
      SEQ i = 0 FOR number.of.leaves
        SEQ
          event := event PLUS second
          shift(number)
          up.to.tree ! another.int; number
          clock ? AFTER event
      up.to.tree ! no.more.ints
    INT event :
    SEQ
      clock ? event
      SEQ i = 0 FOR number.of.leaves
        SEQ
          event := event PLUS second
          INT discard :
          down.from.tree ? CASE another.int; discard
          clock ? AFTER event
      down.from.tree ? CASE no.more.ints
:
```

The *driver*, Process 12.20, can discard the result of the sort, because all the information has already been displayed as it passes out of the root process. This coding of the driver completely fills the tree with a known number of values, and removes the same number, so the inputs on *down.from.tree* are guaranteed to have the expected tag.

The driver pauses after injecting each number into the tree and after removing each number from the tree, so as to allow time to see what is happening. There is nothing scientific about the choice of a one second delay; it was adjusted to give an acceptable display on a particular implementation.