Parallel sorter

```
PROC fork(CHAN up,down, left.down,left.up, right.down,right.up)
  PROC fork.distribute(CHAN up, left.up, right.up)
  PROC fork.gather(CHAN down, left.down, right.down)
PROC leaf(CHAN up, down, probe)

PROC monitor(CHAN up.a, down.a, up.b, down.b, probe)
PROC multiplex(CHAN probe[], all.probes)
PROC independent(CHAN source, sink)
  PROC make.cartesian(VALUE index, VAR x, y)
PROC dependent(CHAN source, terminal)
  PROC clear.screen(CHAN terminal)
  PROC goto.xy(CHAN terminal, VALUE x, y)
PROC display(CHAN source, sink)
PROC driver(CHAN up, down)
  PROC shift(VAR state)

PAR
  driver(up.a[root], down.b[root])
  PAR i = [first.fork FOR number.of.forks]
    fork( ... )
  PAR i = [first.leaf FOR number.of.leaves]
    leaf( ... )
  PAR i = [root FOR number.of.channels]
    monitor( ... )
  multiplex(probe, all.probes)
  display(all.probes, terminal.screen)
```

```
PROC fork(CHAN up, down, left.down, left.up, right.down, right.up) =

  PROC fork.distribute(CHAN up, left.up, right.up) =
    -- share out a sequence of numbers as two sequences
    -- to the left, to the right
    DEF leftward = 0, rightward = NOT leftward :
    VAR more, inclination :
    SEQ
      inclination := leftward
      up ? more
      WHILE more
        VAR number :
        SEQ
          up ? number
          IF
            inclination = leftward
              left.up ! TRUE; number
            inclination = rightward
              right.up ! TRUE; number
          up ? more
          inclination := NOT inclination
      PAR
        left.up ! FALSE
        right.up ! FALSE                          :

  PROC fork.gather(CHAN down, left.down, right.down) =
    -- merge two ascending sequences, from left and right
    -- into one ascending sequence
    VAR left.more, left.minimum, right.more, right.minimum :
    SEQ
      PAR
        left.down ? left.more; left.minimum
        right.down ? right.more; right.minimum
      WHILE left.more OR right.more
        IF
          left.more AND
                ((NOT right.more) OR (left.minimum <= right.minimum))
            SEQ
              down ! TRUE; left.minimum
              left.down ? left.more; left.minimum
          right.more AND
                ((NOT left.more) OR (left.minimum >= right.minimum))
            SEQ
              down ! TRUE; right.minimum
              right.down ? right.more; right.minimum
      down ! FALSE; ANY                                         :

  -- PROC fork() -- actions for a medial node in the sorting tree
  SEQ
    fork.distribute(up, left.up, right.up)
    fork.gather(down, left.down, right.down)  :
```

```
DEF display.number = 1, display.empty = 2, display.stop = 3 :


PROC leaf(CHAN up, down, probe) =
  -- actions for a terminal node in the sorting tree
  VAR number :
  SEQ
    up ? ANY; number                 -- expect a sequence of one number
    probe ! display.number; number   -- pass to the monitoring code
    up ? ANY
    down ! TRUE; number              -- return it as a sequence
    probe ! display.empty            -- indicating its departure
    down ! FALSE; ANY
    probe ! display.stop          :


PROC monitor(CHAN up.a, down.a, up.b, down.b, probe) =
  -- in-channel monitoring code, in the form of a buffer
  SEQ
    VAR more :
    SEQ               -- first watch an upward-bound sequence of values
      up.a ? more
      WHILE more
        VAR number :
        SEQ
          up.a ? number
          probe ! display.number; number
          up.b ! more; number
          probe ! display.empty
          up.a ? more
      up.b ! more
    VAR more, number :
    SEQ               -- then watch a downward-bound sequence
      down.a ? more; number
      WHILE more
        SEQ
          probe ! display.number; number
          down.b ! more; number
          probe ! display.empty
          down.a ? more; number
      down.b ! more; number
      probe ! display.stop                           :
```

```
DEF depth.of.tree = 4 :

DEF number.of.leaves    = 1 << depth.of.tree ,
    number.of.forks     = number.of.leaves - 1 ,
    number.of.processes = number.of.forks + number.of.leaves ,
    number.of.channels  = number.of.processes ,
    number.of.probes    = number.of.channels + number.of.leaves :



PROC multiplex(CHAN probe[], all.probes) =
  -- gather all probe signals onto a single channel
  VAR more, more.from[number.of.probes] :
  SEQ
    more := number.of.probes
    SEQ i = [0 FOR number.of.probes]
      more.from[i] := TRUE
    WHILE more > 0              -- while not all probes are dead
      VAR instruction :
      ALT i = [0 FOR number.of.probes]
        more.from[i] & probe[i] ? instruction
                                -- take a probe instruction
          IF
            instruction = display.number    -- if this is a number
              VAR number :
              SEQ
                probe[i] ? number           -- copy the number
                all.probes ! instruction; i; number
                                -- tagging it with the probe number

            instruction = display.empty     -- if this is a blank
              all.probes ! instruction; i
                                -- tag it with the probe number

            instruction = display.stop      -- if the probe is dead
              SEQ
                more.from[i] := FALSE       -- then expect no more
                more := more - 1            -- and decrease the count
                                            --   of working ones
    all.probes ! display.stop      :
```

```
DEF field.width = 3 :

PROC independent(CHAN source, sink) =


  PROC make.cartesian(VALUE index, VAR x, y) =
    -- turn a probe index into Cartesian co-ordinates
    -- in a terminal-independent space
    IF
      IF line = [1 FOR depth.of.tree + 1]
        index < ((1 << line) - 1)         -- then probe is from a
          VAR c :                         --   channel at this depth
          SEQ
            c  := index - ((1 << (line - 1)) - 1)
            x := ((2 * c) + 1) * (number.of.leaves >> (line - 1))
            y := line
      index >= number.of.channels         -- then probe is from a leaf
        SEQ
          x := (2 * (index - number.of.channels)) + 1
          y := depth.of.tree + 2                         :


  VAR instruction :
  SEQ
    source ? instruction
    WHILE instruction <> display.stop
      SEQ                        -- turn every probe signal into ...
        sink ! TRUE              --     ... a TRUE value
        VAR index, x, y :
        SEQ
          source ? index
          make.cartesian(index, x, y)
          sink ! x; y            --     ... a co-ordinate-pair
        IF                       --     and field.width characters:
          instruction = display.number
            VAR number :
            SEQ
              source ? number
              write.signed(sink, number, field.width) -- a numeral
          instruction = display.empty
            SEQ i = [0 FOR field.width]               -- or blanks
              sink ! '*s'
        source ? instruction
    sink ! FALSE                         :
```

```
DEF virtual.height = depth.of.tree + 1,
    virtual.width  = (2 * number.of.leaves) - 1 :


PROC dependent(CHAN source, terminal) =
  -- terminal dependent code for driving a VT52

  DEF screen.height = 24, screen.width = 80 :

  DEF control = NOT ((NOT 0) << 5), escape = control /\ '[' :

  PROC clear.screen(CHAN terminal) =
    -- clear screen sequence for a VT52
    terminal ! escape ; 'H' ; escape ; 'J'           :

  PROC goto.xy(CHAN terminal, VALUE x, y) =
    -- lefthanded co-ordinates, origin 0, 0 at top left
    terminal ! escape ; 'Y' ; '*s' + y ; '*s' + x    :

  VAR more :
  SEQ
    clear.screen(terminal)
    source ? more
    WHILE more
      SEQ
        VAR x, y :
        SEQ
          source ? x; y
          goto.xy(terminal, (x - 1) * (screen.width / virtual.width),
               (virtual.height - y) * (screen.height / virtual.height))
        SEQ i = [1 FOR field.width]
          VAR ch :
          SEQ
            source ? ch
            terminal ! ch
        source ? more
    goto.xy(terminal, 0, screen.height - 1)            :


PROC display(CHAN source, sink) =
  CHAN internal :
  PAR
    independent(source, internal)
    dependent(internal, sink)     :
```

```
PROC driver(CHAN up, down) =

  DEF mask = NOT ((NOT 0) << 9) :

  PROC shift(VAR state) =
    SEQ i = [1 FOR 9]
      state := ((state << 1) /\ mask) \/
                                 (((state >> 4) >< (state >> 8)) /\ 1)  :

  SEQ
    VAR event, number :                  -- first fill the tree
    SEQ
      TIME ? event
      number :=  (event /\ mask) \/ 1  -- initialize the random number
      SEQ i = [0 FOR number.of.leaves]
        SEQ
          event := event + second
          shift(number)                  -- pick a new number
          up ! TRUE; number              -- send it into the tree
          TIME ? AFTER event             -- and wait for a second
      up ! FALSE
    VAR event :                          -- then empty the tree
    SEQ
      TIME ? event
      SEQ i = [0 FOR number.of.leaves]
        SEQ
          event := event + second
          down ? ANY; ANY                -- take a number from the tree
          TIME ? AFTER event             -- once a second
      down ? ANY; ANY              :
```

```
DEF root = 0 ,
    first.fork = root ,
    first.leaf = first.fork + number.of.forks :

CHAN  up.a[number.of.channels], down.a[number.of.channels],
      up.b[number.of.channels], down.b[number.of.channels],
      probe[number.of.probes], all.probes  :

PAR
  driver(up.a[root], down.b[root])

  PAR i = [first.fork FOR number.of.forks]
    fork(up.b[i], down.a[i],
         down.b[(2*i)+1], up.a[(2*i)+1],
         down.b[(2*i)+2], up.a[(2*i)+2])

  PAR i = [first.leaf FOR number.of.leaves]
    leaf(up.b[i], down.a[i],
         probe[number.of.channels + (i - first.leaf)])

  PAR i = [root FOR number.of.channels]
    monitor(up.a[i], down.a[i], up.b[i], down.b[i], probe[i])

  multiplex(probe, all.probes)

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