7 Parallel sorting

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 identifies 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. I make no claims 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.

Sorting strategy

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. Again, each process is 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. 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, 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 already constitutes a sequence in ascending order. Each internal node, relying on the sorted subsequences that it will receive from its children, merges two ascending sequences to generate its output sequence.

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

```
PROC leaf(CHAN up, down) =
  ...
```

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 up, down, left.down,  left.up,
                         right.down, right.up) =
  ...
```

For simplicity, the root process is treated as an internal node, with a virtual root process acting as the parent of the root

```
PROC driver(CHAN up.to.tree, down.from.tree) =
  ...
```

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

To connect these processes, they have to be indexed, so as to correspond to linear arrays of channels. For simplicity, I have made the program a complete balanced tree

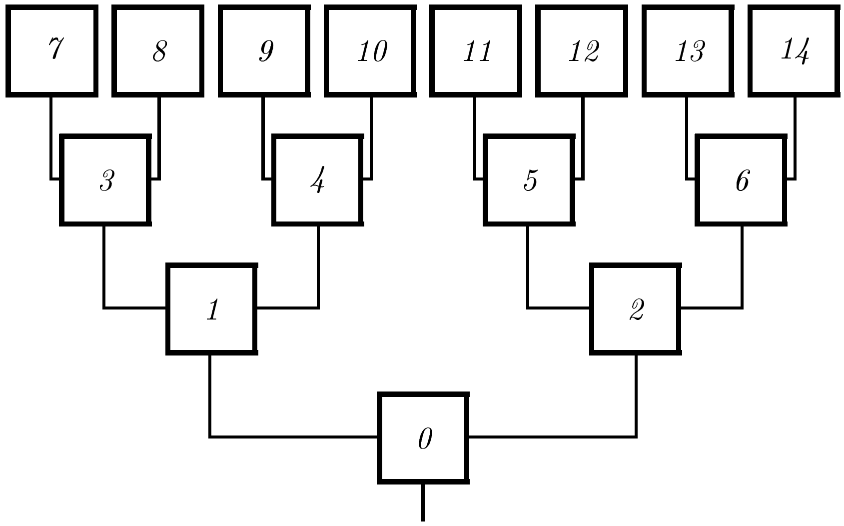
```
DEF depth.of.tree = 3 :

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 :
```

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

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

the children of the internal node process i are indexed $2i + 1$ and $2i + 2$. If channels indexed i are used to connect process i to its parent, these same formulae will give the indexes of the channels to and from the children of internal node process i.

```
CHAN up[number.of.channels], down[number.of.channels] :
PAR
  driver(up[root], down[root])
  PAR i = [first.fork FOR number.of.forks]
    fork(up[i], down[i], down[(2*i)+1], up[(2*i)+1],
                         down[(2*i)+2], up[(2*i)+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.

```
PROC fork(CHAN up, down, left.down,  left.up,
                         right.down, right.up) =
  SEQ
    fork.distribute(up, left.up, right.up)
    fork.gather(down, left.down, right.down)  :
```

During the distribution phase, each internal node receives a sequence of numbers from its parent. Notice that since a fork process knows neither where it is in the tree, nor how big the tree is, it cannot know how many numbers to expect. Accordingly, the sequence is passed around with each number preceded by a `TRUE` value, and the last followed by a `FALSE` value. Such a sequence can be read by

```
VAR more :
SEQ
  up ? more
  WHILE more
    VAR next :
    up ? next; more
```

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.

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

Notice that this process passes the guarantee of correctly interpolated `TRUE` and `FALSE` values on to its children. Induction from this observation demonstrates that if the root process operates the protocol correctly, so will each of its descendants.

Since I assumed that the component processes would 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 I will use a similar protocol. The returning sequences of numbers will be coded by preceding each number by a `TRUE` value, and adding a pair consisting of a `FALSE` value and a subsequent dummy at the end of the sequence.

In order to do the merging, preserving ordering, numbers must be compared, and this requires that each merging 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, so the merging process compares the heads, passes on the smaller, and draws one more value from the selected sequence, continuing until the sequences are exhausted.

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

Notice a final `ANY` sent downwards at the end of the sequence, which accounts for the parent process being able to receive a pair of values, ... *more*, ... *minimum*, even when the first is `FALSE`. This trick is simpler than making the parent's behaviour conditional on the first value, and there is no great cost since after sending down its `FALSE` value there is nothing left for the child process to do.

The leaf process must be designed to simulate the behaviour of an internal node that only handles a sequence of one number

```
PROC leaf(CHAN up, down) =
  VAR number :
  SEQ
    up ? ANY; number; ANY
    down ! TRUE; number; FALSE; ANY :
```

Finally, the driver process must generate and absorb sequences of numbers, stuffing and stripping the protocol.

```
PROC driver(CHAN up, down) =
  SEQ
    SEQ i = [0 FOR number.of.leaves]
      VAR number :
      SEQ
        ... think of a number
        up ! TRUE; number
    up ! FALSE
    SEQ i = [0 FOR number.of.leaves]
      VAR number :
      SEQ
        down ? ANY; number
        ... do something with the number
    down ? ANY; ANY                        :
```

The missing code controls the behaviour of the whole program. It 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 overcomplex for a single processor implementation, would look better on an array 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. The 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, something like

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

This process can be inserted into a data stream passing along a channel

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

allowing the data to be read by another process

```
CHAN channel.a, channel.b, test.data :
PAR
  producer(channel.a)
  duplicate(channel.a, channel.b, test.data)
  consumer(channel.b)
  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 through a third party, between the *producer* and *consumer*, it might matter that the output from the *producer* could 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.

In order to make internal state visible, it is necessary to add code to 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(...) =
  VAR x :
  SEQ
    ⋮

    x := e

    ⋮

    c ? x

    ⋮
```

```
PROC p(..., CHAN test.data) =
  VAR x :
  SEQ
    ⋮
    SEQ
      x := e
      test.data ! x
    ⋮
    SEQ
      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, I 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, each carrying signals indicating the presence or absence of a number. 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.

These decisions lead to the following, changed, program structure

```
DEF number.of.probes = number.of.channels +
                       number.of.leaves :

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)
```

Each *monitor* process copies data from its ...*a* channels to its ...*b* 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.

Component processes

There are three types of message to be sent along the *probe* channels: messages indicating the presence of a number, messages indicating the absence of a number, and a final termination message. Each of these will be indicated by starting it with one of three values

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

and the monitoring processes will guarantee that the communications on a *probe* will be a sequence of triples, *display.number* followed by a number followed by a *display.empty*, the whole sequence being followed by 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 the number arriving while the tree is filled; clear the display when the number is unloaded; and indicate the end of the leaf's activity.

```
PROC leaf(CHAN up, down, probe) =
  VAR number :
  SEQ
    up ? ANY; number   -- this ANY is TRUE
    probe ! display.number; number
    up ? ANY           -- this one is FALSE

    down ! TRUE; number
    probe ! display.empty
    down ! FALSE; ANY

    probe ! display.stop          :
```

The *monitor* process must copy the sequences of numbers passing first up and then down the tree using the correct protocol for each case. The necessary monitoring code is just what you would need to record changes of state in this buffering processes, so again divides into the same three stages

```
PROC monitor(CHAN up.a, down.a, up.b, down.b, probe) =
  SEQ
    VAR more :
    SEQ
      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 ! FALSE

    VAR more, number :
    SEQ
      down.a ? more; number
      WHILE more
        SEQ
          probe ! display.number; number
          down.b ! more; number
          probe ! display.empty
          down.a ? more; number
      down.b ! FALSE; ANY

    probe ! display.stop          :
```

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 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.

```
PROC multiplex(CHAN probe[], all.probes) =
  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
      VAR instruction :
      ALT i = [0 FOR number.of.probes]
        more.from[i] & probe[i] ? instruction
          IF
            instruction = display.number
              VAR number :
              SEQ
                probe[i] ? number
                all.probes ! instruction; i; number

            instruction = display.empty
              all.probes ! instruction; i

            instruction = display.stop
              SEQ
                more.from[i] := FALSE
                more := more - 1

    all.probes ! display.stop           :
```

Display management

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; then that space is mapped onto the terminal screen.

```
PROC display(CHAN source, sink) =
  CHAN 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.

```
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 index 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 right line for a particular probe is the first for which its index lies below this number. The right column is calculated by discounting the $2^{line-1} - 1$ probes displayed on the lower lines and multiplying by a factor which accounts for the exponentially differing separation of nodes at different depths

```
PROC make.cartesian(VALUE index, VAR x, y) =
  IF
    IF line = [1 FOR depth.of.tree + 1]
      index < ((1 << line) - 1)
        VAR c :
        SEQ
          c  := index - ((1 << (line - 1)) - 1)
          x := ((2 * c) + 1) *
                         (number.of.leaves >> (line - 1))
          y := line
    index >= number.of.channels
      SEQ
        x := (2 * (index - number.of.channels)) + 1
        y := depth.of.tree + 2        :
```

The *make.cartesian* process translates a probe index into an x, y pair

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

The other terminal independent part of the translation is to turn into digits the numbers that are to be displayed. All the numbers are written in a fixed width field

```
PROC independent(CHAN source, sink) =
  VAR instruction :
  SEQ
    source ? instruction
    WHILE instruction <> display.stop
      SEQ
        sink ! TRUE
        VAR index, x, y :
        SEQ
          source ? index
          make.cartesian(index, x, y)
          sink ! x; y
        IF
          instruction = display.number
            VAR number :
            SEQ
              source ? number
              write.signed(sink, number, field.width)
          instruction = display.empty
            SEQ i = [0 FOR field.width]
              sink ! ‘*s’
        source ? instruction
    sink ! FALSE                             :
```

The output from this process consists of a sequence of packets, each beginning with an x, y pair, followed by *field.width* number of characters to be displayed there. Each packet is preceded by a `TRUE` value, and the whole sequence is terminated with a `FALSE`.

If the terminal has cursor addressing, then the task is almost complete. Here, for example, is the necessary terminal dependent part of the display process for a *digital VT52* terminal

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

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

  DEF screen.height = 24, screen.width = 80 :
  DEF height.scale = screen.height / virtual.height,
      width.scale  = screen.width / virtual.width :
  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) * width.scale,
                   (virtual.height - y) * height.scale)
        SEQ i = [1 FOR field.width]
          VAR ch :
          SEQ
            source ? ch
            terminal ! ch
        source ? more
    goto.xy(terminal, 0, screen.height - 1)       :
```

The divison of work 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.

For the purpose of the simulator, the simplest coding of the driver process invents a random sequence of numbers for input to the tree. A common way of generating an unpredictable sequence of numbers is to use a linear feedback shift register

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

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

with an uncontrolled initial state. An arbitrary initial state can be obtained by reading the real-time clock. Since the shift-register will not change state if the initial state is all zeroes, the time is $\vee$-ed with a one to guarantee a non-zero initial state.

```
PROC driver(CHAN up, down) =
  SEQ
    VAR event, number :
    SEQ
      TIME ? event
      number :=  (event /\ mask) \/ 1
      SEQ i = [0 FOR number.of.leaves]
        SEQ
          event := event + second
          shift(number)
          up ! TRUE; number
          TIME ? AFTER event
      up ! FALSE
    VAR event :
    SEQ
      TIME ? event
      SEQ i = [0 FOR number.of.leaves]
        SEQ
          event := event + second
          down ? ANY; ANY
          TIME ? AFTER event
          down ? ANY; ANY                          :
```

The driver discards 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 pauses after injecting each number into the tree, and after removing each number from the tree, so as to give you time to see what is happening. There is nothing scientific about the choice of a one second pause: I adjusted it to get a satisfactory display from the particular implementation that I was using.