8

Programming structures

The simplest thing that you can usefully want a process to be doing, at the same time as another process is doing something else, is to copy data from one channel to another. This is just a matter of repeatedly taking input from one channel, storing it in a local variable, and then sending the value of the variable along another channel. Think of

```
INT local :
SEQ
  source ? local
  sink ! local
```

Why you might possibly want this done should be apparent: the local variable acts as a buffer in the data stream passing along the two channels. This copying process can be packaged as a named process that can be used to buffer an infinite stream of integers passing between two processes:

```
PROC buffer(CHAN OF INT source, sink)
  WHILE TRUE
    INT local :
    SEQ
      source ? local
      sink ! local
:
```

Now, whereas the producer and the consumer process are tightly synchronized in a program like

```
CHAN OF INT data.stream :
PAR
  producer(data.stream) -- sends output to data.stream
  consumer(data.stream) -- takes input from data.stream
```

with neither able to get ahead of the other, by adding a buffer

```
CHAN OF INT data.from.producer, data.to.consumer :
PAR
  producer(data.from.producer)
  buffer(data.from.producer, data.to.consumer)
  consumer(data.to.consumer)
```

the two are slightly decoupled. The producer is now able to run up to one item of data ahead of the consumer. (It remains difficult to let the consumer get ahead of the producer!)

More buffering is easily provided by inserting more buffers in the data path. This is like a 'fall-through' first-in–first-out store, where each item of data is passed along a sequence of registers until it reaches the last unoccupied location. Several items can be in independent passage at once – several of the component processes of the buffer can be active – if the buffer is fairly empty.

```
CHAN OF INT data.stream[number.of.buffers + 1] :
PAR
  producer(data.stream[0])
  PAR index = 0 FOR number.of.buffers
    buffer(data.stream[index], data.stream[index + 1])
  consumer(data.stream[number.of.buffers])
```

There is an apparent paradox here: although channels implement only unbuffered communication, it is not possible to write a process that joins two channels together without introducing some buffering. The best that can be done is something like the *buffer* procedure, which ensures that the same data are communicated on two channels, without being able to make the communications happen at the same time. Specifically, a process is a *buffer* if it

- consumes data from one channel and performs outputs, on the other channel, of the same data in the same sequence
- is prepared to perform an output whenever it has consumed more than it has subsequently output
- is prepared to consume more input whenever it has disposed of everything that it has input

So the procedure *buffer* is a buffer for infinitely long streams of INTs, and

```
CHAN OF INT data.stream[number.of.buffers + 1] :
PAR index = 0 FOR number.of.buffers
  buffer(data.stream[index], data.stream[index + 1])
```

is a buffer between *data.stream*[0] and *data.stream*[*number.of.buffers*] for infinitely long streams of `INT`s.

If there is a bound on how much more data a buffer can consume than reproduce before it is *full* – no longer ready for more – this is called the *length* of the buffer. The length of *buffer* is one, the length of an array of *number.of.buffers* of them is *number.of.buffers*.

Synchronization without data

Even when there is no transfer of data, a communication between an output process and an input process synchronizes the two processes concerned. The protocol

```
PROTOCOL SIGNAL
  CASE
    signal
:
```

allows of only one message, so a channel declared with this protocol

```
CHAN OF SIGNAL handshake :
```

can be used in only one sort of output process

```
handshake ! signal
```

and only one sort of input process

```
handshake ? CASE signal
```

Inputs on channels with this or an equivalent protocol are usually used as guards in alternatives. Since there is no transfer of information between an input and output on a signal channel, the processes are interchangeable except of course that only the input can appear as a guard.

In some notations for parallel programming, the fundamental synchronizing primitive is the *semaphore*, a device for ensuring that only one of a collection of parallel processes has access to a resource at any one time. Process 8.1 is initially prepared to accept a signal on any of the *handshake* channels. After that, the next communication has to be on the same channel. That means that the semaphore can only ever have performed an odd number of communications on at most one of its channels. If each channel is used for output by a 'user' of the semaphore, then each user knows that if it has completed an odd number of outputs, no other user has done so.

In a program like Process 8.2 at most one of the instances of *critical*(i) can be executing at any time. This sort of guarantee is just what you need to ensure that only one process was using a single shared physical resource,

Process 8.1

```
PROC binary.semaphore([]CHAN OF SIGNAL handshake)
  WHILE TRUE
    ALT i = 0 FOR SIZE handshake
      handshake[i] ? CASE signal
        handshake[i] ? CASE signal
:
```

Process 8.2

```
[number.of.users]CHAN OF SIGNAL handshake :
PAR
  PAR i = 0 FOR number.of.users
    CHAN OF SIGNAL resource IS handshake[i] :
    WHILE TRUE
      SEQ
        unimportant(i)
        resource ! signal
        critical(i)
        resource ! signal
  binary.semaphore(handshake)
```

Process 8.3

```
PROC receive(CHAN OF INT another.int, CHAN OF SIGNAL no.more.ints)
  BOOL more :
  SEQ
    more := TRUE
    WHILE more
      ALT
        INT x :
        another.int ? x
          ...
        no.more.ints ? CASE signal
          more := FALSE
:
```

Process 8.4

```
PROC send(CHAN OF INT another.int, CHAN OF SIGNAL no.more.ints)
  SEQ
    WHILE ...
      SEQ
        ...
        another.int ! i
    no.more.ints ! signal
:
```

like a terminal screen: if each process were to guarantee to write characters on the screen only when it was in its critical routine, then no process would be allowed to interrupt a message from another.

A particularly common use of interrupting signals is to mark the end of a sequence of communications. Process 8.3 consumes a sequence of integers from *another.int*, until it receives a signal from *no.more.ints*. This sort of process would probably be used in a program like

```
CHAN OF INT another.int :
CHAN OF SIGNAL no.more.ints :
PAR
  send(another.int, no.more.ints)
  receive(another.int, no.more.ints)
```

to consume the output of a process, Process 8.4, which executes some indefinite number of outputs to *another.int*, and then having sent them all, sends a signal on *no.more.ints*.

In fact, in this case, there is a neater way of coding this example. Because both of the channels pass between the same two processes, and are used in a strict sequence, they can be replaced by a single channel with a discriminated protocol

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

An output on one channel is replaced by an output of one variant, on the other channel by a signal of the other variant (Process 8.5); and the alternative of

Process 8.5

```
PROC send(CHAN OF INT.STREAM c)
  SEQ
    WHILE ...
      SEQ
        ...
        c ! another.int; i
    c ! no.more.ints
:
```

the two inputs becomes an input discriminating between the two variants of the discriminated protocol (Process 8.6). This code is neater, if a trifle less modular, and an implementation would probably execute it more efficiently than the earlier coding.

Process 8.6

```
PROC receive(CHAN OF INT.STREAM c)
  BOOL more :
  SEQ
    more := TRUE
    WHILE more
      c ? CASE
        INT x :
        another.int; x
          ...
        no.more.ints
          more := FALSE
:
```

Synchronization by access to shared data

The semaphore is an awkward primitive with which to work; the generality of an occam alternative process can usually express rather better what is intended by keeping the code which handles the shared resource in the same process as the resource. Process 8.7 implements an averaging register; it accepts a sequence of integers from *another.int*. At any time this sequence can be interrupted by a signal on *request.mean* in response to which the averaged sum is sent by an output on *mean.of.stream*. By sequencing all accesses to the variables *sum* and *count*, the alternative process guarantees that an average is not calculated with inconsistent values of the variables. At the same time it encapsulates all accesses to those variables, so that a reader of the program can immediately see that the variables always contain values which guarantee that each output to *mean.of.stream* is the mean of the integers read from *another.int*.

Process 8.7

```
PROC average(CHAN OF INT another.int,
             CHAN OF SIGNAL request.mean,
             CHAN OF INT mean.of.stream  )
  INT sum, count :
  SEQ
    sum, count := 0, 0
    WHILE TRUE
      ALT
        INT x :
        another.int ? x
          sum, count := sum + x, count + 1
        count > 0 & request.mean ? CASE signal
          mean.of.stream ! sum / count
:
```

A perfect example of this sort of arbitration with encapsulation is a buffer process, which effectively shares a local variable between a writer and a reader; the single variable in the earlier example of a buffer of length one required little management, but a longer buffer would be more complex. The usual algorithm for implementing a buffer of length *size* characters uses an array

```
[size]BYTE store :
```

indexed by variables

```
INT reader, writer :
```

which range over zero to $size - 1$ so that the oldest value, about to leave the buffer, will be found at $store[reader]$, and the next to enter the buffer will be written to $store[writer]$. It will be convenient to keep track of the number of unoccupied locations in the buffer by a further variable

```
INT count :
```

whose value ranges from zero, for a full buffer, to *size* for an empty one.

There are two activities in which the buffer must be able to participate: provided that it is not full, that is that $count > 0$, it must be possible to add another value to the buffer

```
SEQ
  source ? store[writer]
  count, writer := count - 1, (writer + 1) \ size
```

and provided that $count < size$ it must be possible for the oldest value to be read from the buffer

```
SEQ
  sink ! store[reader]
  count, reader := count + 1, (reader + 1) \ size
```

The buffer must allow the producing and consuming processes to control its activity, selecting between writing and reading, provided only that there is room to write, or something to read, respectively.

Arbitration like this requires an `ALT` process, with the reading and writing actions being guarded by the conditions under which they may happen – $count > 0$ and $count < size$ – and the readiness of the consumer and the producer. Since the consumer is ready when the buffer can output to *sink* and the producer when the buffer can input from *source* it is tempting to write

```
ALT
  count > 0   &   source ? store[writer]
    count, writer := count - 1, (writer + 1) \ size
  count < size   &   sink ! store[reader]
    count, reader := count + 1, (reader + 1) \ size
```

but output processes cannot be used to guard alternatives. The solution is to have a control signal from the consuming process indicating that it is ready to accept an input from *sink*.

```
ALT
  count > 0   &   source ? store[writer]
    count, writer := count - 1, (writer + 1) \ size
  count < size   &   request ? CASE signal
    SEQ
      sink ! store[reader]
      count, reader := count + 1, (reader + 1) \ size
```

There is no need for the corresponding request before a write to the buffer, because the input along *source* serves perfectly well in place of a control signal. It is the responsibility of the consumer, whenever it reads from the circular buffer, Process 8.8, to precede the input from *source* by an output on *request*, performing two communications in sequence

```
SEQ
  request ! signal
  source ? ...
```

The ultimate consumer can be relieved of this burden at the expense of an extra process like *request.buffer*, Process 8.9. This process executing concurrently with the circular buffer has a behaviour, Process 8.10, which is indistinguishable – apart, perhaps from performance – from that of a chain of $size + 1$ single-item *buffer* processes acting in parallel.

Using parallelism for program modularity

Imagine that for some obscure reason you wanted a procedure

```
PROC write.even.parity.string(CHAN OF BYTE c, []BYTE s)
```

that, like *write.string*, sent the characters from its string argument to its output channel, but which only sent the even parity ones and discarded the odd parity ones. (The *parity* of a character is odd or even according as the number of one bits in the representation of the character is odd or even.)

Process 8.8

```
PROC circular.buffer(CHAN OF BYTE source,
                     CHAN OF SIGNAL request,
                     CHAN OF BYTE sink      )
  -- Copy from source to sink buffering up to size items. A signal
  -- is needed on request before each item is read from sink.
  INT reader, writer, count :
  [size]BYTE store :
  SEQ
    count, reader, writer := size, 0, 0
    WHILE TRUE
      ALT
        count > 0   &   source ? store[writer]
          count, writer := count - 1, (writer + 1) \ size
        count < size   &   request ? CASE signal
          SEQ
            sink ! store[reader]
            count, reader := count + 1, (reader + 1) \ size
:
```

Process 8.9

```
PROC request.buffer(CHAN OF SIGNAL request,
                    CHAN OF BYTE source, sink )
  WHILE TRUE
    BYTE datum :
    SEQ
      request ! signal
      source ? datum
      sink ! datum
:
```

Process 8.10

```
PROC multiple.buffer(CHAN OF BYTE source, sink)
  -- Copy from source to sink, buffering up to size+1 items.
  CHAN OF SIGNAL request :
  CHAN OF BYTE data :
  PAR
    circular.buffer(source, request, data)
    request.buffer(request, data, sink)
:
```

Any pair of different values will do for representing *even* and *odd*,

```
VAL even IS 0 :
VAL odd  IS 1 >< even :
```

Each bit in the representation of the character has to be considered, so the simplest thing to do is to consider them one at a time. (An expert bit-twiddler may care to code an algorithm logarithmic in the number of bits in the character.) The expression `(pattern >> i) /\ 1` is zero or one according as the i^{th} bit of *pattern* is zero or one, so Process 8.11 returns the parity of its argument.

Process 8.11

```
INT FUNCTION parity(VAL BYTE ch)
  INT p :
  VALOF
    SEQ
      p := even
      VAL INT bits.in.byte IS 8 :
      VAL INT pattern IS INT ch :
      SEQ i = 0 FOR bits.in.byte
        p := p >< ((pattern >> i) /\ 1)
    RESULT p
:
```

You could certainly code *write.even.parity.string* directly, as shown in Process 8.12, but this is rather a specialized process. It can only be used for selecting even parity characters from arrays of characters, because the code for selecting characters according to their parity is mixed in with the code for turning a string into a sequence of outputs.

A more modular program might re-use *write.string* to convert the string into a sequence of outputs, and a separate process, Process 8.13, to filter the characters on their parities. These components can be plugged together to make a process that almost implements *write.even.parity.string*

```
PAR
  SEQ
    write.string(both, s)
    end.of.both ! signal
  divide.on.parity(both, end.of.both, c, rubbish)
```

One process converts the string s into a sequence of communications on the channel *both*; another process divides these characters so that only the even parity characters are communicated on c.

The problem is that the odd parity characters are communicated on *rubbish*. Since no process is taking input from *rubbish* the *divide.on.parity* process will be held up indefinitely by the first odd parity character which

Process 8.12

```
PROC write.even.parity.string(CHAN OF BYTE c, []BYTE s)
  SEQ i = 0 FOR SIZE s
    VAL BYTE ch IS s[i] :
    CASE parity(ch)
      even
        c ! ch
      odd
        SKIP
:
```

Process 8.13

```
PROC divide.on.parity(CHAN OF BYTE source,
                      CHAN OF SIGNAL end.of.source,
                      CHAN OF BYTE even.sink, odd.sink )
  -- Copy the even parity chars from source to even.sink,
  -- the odd parity ones to odd.sink, until end.of.source
  BOOL more.expected :
  SEQ
    more.expected := TRUE
    WHILE more.expected
      ALT
        BYTE ch :
        source ? ch
          CASE parity(ch)
            even
              even.sink ! ch
            odd
              odd.sink ! ch
        end.of.source ? CASE signal
          more.expected := FALSE
:
```

Process 8.14

```
PROC discard(CHAN OF BYTE source, CHAN OF SIGNAL end.of.source)
  BOOL more.expected :
  SEQ
    more.expected := TRUE
    WHILE more.expected
      ALT
        BYTE ch :
        source ? ch
          SKIP
        end.of.source ? CASE signal
          more.expected := FALSE
:
```

it encounters. Parallel programs abound with instances where a discarding process, such as Process 8.14, are useful. There is no real equivalent in sequential programming; this process discards all of its input, never sends any output, never changes the values of any global variables. Its sole function is to achieve the necessary synchronizations.

Process 8.15

```
PROC write.even.parity.string(CHAN OF BYTE c, []BYTE s)
  CHAN OF BYTE both, odd.ones :
  CHAN OF SIGNAL end.of.both, end.of.odd :
  PAR
    SEQ
      write.string(both, s)
      end.of.both ! signal
    SEQ
      divide.on.parity(both, end.of.both, c, odd.ones)
      end.of.odd ! signal
   discard(odd.ones, end.of.odd)
:
```

The procedure *write.even.parity.string*, Process 8.15, can be put together from these components so that the effect of

```
write.string(output, "Bootifrolo")
```

for example, can be achieved by

```
write.even.parity.string(output, "Booting from Floppy")
```

In this particular case, the gain in modularity may not seem adequate to justify the expense, both in programming effort and execution time. The advantage is clearer in cases where the program must perform a number of tasks each of which divides its input data into chunks, and where the boundaries of these components do not coincide.

Using parallelism to resolve structure clash

A structure clash happens whenever a program must perform operations on data that must be divided into mutually overlapping components. In a text processing program, for example, it may prove necessary to do something to every line of a document, and something else to every sentence.

The natural way to code each of these tasks, individually, is to write programs whose structure reflects the structure of the document. To perform an action on every line:

```
WHILE ... there are more lines
  SEQ
    ... read a line
    ... process the line
```

and to perform an action on every sentence:

```
WHILE ... there are more sentences
  SEQ
    ... read a sentence
    ... process the sentence
```

Since sentences do not need to contain only complete lines, nor lines complete sentences, it is difficult to combine these two programs in a single sequential program. The somewhat unsatisfactory best that can be done in a sequential program is to treat the document as a sequence of words, these being the largest common sub-components of both lines and sentences.

```
WHILE ... there are more words
  SEQ
    ... read a word
    ... if it completes a line process the line
    ... if it completes a sentence process the sentence
```

In a parallel program, the structure of both component processes can be retained by performing both of the divisions of the document concurrently.

```
CHAN OF LINES lines :
CHAN OF SENTENCES sentences :

PAR
  ... copy the document to lines and sentences

  WHILE ... there are more lines
    SEQ
      ... read a line from lines
      ... process the line

  WHILE ... there are more sentences
    SEQ
      ... read a sentence from sentences
      ... process the sentence
```

The simplest case of a structure clash arises from attempting to pack data into fixed-sized blocks that will not accommodate an exact whole number of items. It might be necessary, for example, to pack a stream of characters into half-kilobyte blocks for transmission or storage on a medium which accepts only such blocks. Consider first a case in which there is no structure clash: the medium is represented as a channel that accepts only outputs of arrays of half a kilobyte, and characters are represented by codes in the range from 0 to 255, so that a whole number of characters exactly fill a block.

The way to perform actions sequentially on the components of an array of bytes declared by

```
VAL INT bytes.in.a.block IS 512 :
[bytes.in.a.block]BYTE buffer :
```

is to use a sequential 'array' of processes created by the constructor

```
SEQ byte.number = 0 FOR bytes.in.a.block
```

so this packing might be done by a process like Process 8.16.

Process 8.16

```
PROTOCOL BLOCK IS [bytes.in.a.block]BYTE :

PROC pack.bytes.into.blocks(CHAN OF BYTE byte.source,
                            CHAN OF SIGNAL end.of.source,
                            CHAN OF BLOCK block.sink     )
  BOOL more.bytes.expected :
  SEQ
    more.bytes.expected := TRUE
    WHILE more.bytes.expected
      [bytes.in.a.block]BYTE buffer :
      SEQ
        SEQ byte.number = 0 FOR bytes.in.a.block
          ALT
            BYTE byte IS buffer[byte.number] :
            more.bytes.expected & byte.source ? byte
              SKIP
            more.bytes.expected & end.of.source ? CASE signal
              more.bytes.expected := FALSE
            NOT more.bytes.expected & SKIP
              SKIP
        block.sink ! buffer
:
```

The branch of the alternative that does all the work is the first, that guarded by an input from *byte.source* which inputs the next byte into the particular component of the buffer which is being considered. Since the guard does all the work, there is nothing left to be done in the guarded process, so this is `SKIP`. The condition before the `SKIP` guard ensures that it is ready when and only when there are no more bytes to be packed into the last block.

That process always sends a partly or completely empty block as its last output. The sending of a completely empty block could be prevented by

Process 8.17

```
VAL INT bytes.in.a.block IS 512 :

PROC pack.bytes.into.blocks(CHAN OF BYTE byte.source,
                            CHAN OF SIGNAL end.of.source,
                            CHAN OF BLOCK block.sink     )
  BOOL more.bytes.expected :
  SEQ
    more.bytes.expected := TRUE
    WHILE more.bytes.expected
      [bytes.in.a.block]BYTE buffer :
      ALT
        byte.source ? buffer[0]
          SEQ
            SEQ byte.number = 1 FOR bytes.in.a.block - 1
              ALT
                BYTE byte IS buffer[byte.number] :
                more.bytes.expected & byte.source ? byte
                  SKIP
                more.bytes.expected & end.of.source ? CASE signal
                  more.bytes.expected := FALSE
                NOT more.bytes.expected & SKIP
                  SKIP
            block.sink ! buffer
        end.of.source ? CASE signal
          more.bytes.expected := FALSE
:
```

looking ahead for the next byte (Process 8.17). Even so, in case the entire message does not exactly fill a whole number of blocks, it has to be possible for a process that unpacks the characters from the blocks to deduce from those characters that it has reached the actual end of the character stream before the end of the last block.

Full blocks can be unpacked by Process 8.18 so a stream of a number bytes that happened to fit exactly into a number of blocks could be passed by

```
CHAN OF BLOCKS medium :
CHAN OF SIGNAL end.medium :
PAR
  SEQ
    pack.bytes.into.blocks(bytes.in, end.of.in, medium)
    end.medium ! signal
  unpack.bytes.from.blocks(medium, end.medium, bytes.out)
```

This is almost a buffer, but not quite, because it might not be prepared to output its first byte until it has received a full block's worth of bytes of input.

Process 8.18

```
PROC unpack.bytes.from.blocks(CHAN OF BLOCK block.source,
                              CHAN OF SIGNAL end.of.source,
                              CHAN OF BYTE byte.sink       )
  BOOL more.blocks.expected :
  SEQ
    more.blocks.expected := TRUE
    WHILE more.blocks.expected
      ALT
        [bytes.in.a.block]BYTE buffer :
        block.source ? buffer
          SEQ byte.number = 0 FOR bytes.in.a.block
            byte.sink ! buffer[byte.number]
        end.of.source ? CASE signal
          more.blocks.expected := FALSE
:
```

Process 8.19

```
PROC set.bit(BYTE byte, VAL INT bit.number, VAL BOOL bit)
  IF
    NOT bit
      byte := BYTE ((INT byte) /\ (~(1 << bit.number)))
    bit
      byte := BYTE ((INT byte) \/ (1 << bit.number))
:
```

Now consider the problem of trying to achieve a higher packing density, given that it happens that only character codes less than 128 are going to be sent, so that seven bits will suffice rather than eight. Seven bit values will not fit neatly into bytes, nor indeed into half-kilobyte blocks. The problem can, however, be decomposed into two simpler separate problems in which there is no structure clash: turning seven bit character values into a sequence of bits, and packing a sequence of bits into blocks.

The packing of bits into blocks can be done in almost exactly the same way as that suggested for packing bytes into blocks. A byte can be considered to be an array of bits, indexed by using the bit-pattern manipulating operations. The assignment

```
byte := BYTE ((INT byte) /\ (~(1 << bit.number)))
```

sets the *bit.number*$^{\text{th}}$ bit of *byte* to zero, whilst

```
byte := BYTE ((INT byte) \/ (1 << bit.number))
```

sets that same bit to one, so the conditional in Process 8.19 stores one or zero in the *bit.number*$^{\text{th}}$ bit of the *byte* according as the given *bit* is true or not. The buffer could be treated as a two-dimensional array, so it would be possible to write a two-dimensional SEQ–FOR array of processes, Process 8.20, over the bit-number and byte-number to pack the buffer.

Process 8.20

```
VAL INT bits.in.a.byte IS 8 :
VAL INT bytes.in.a.block IS  512 :
PROTOCOL BIT IS BOOL :

PROC pack.bits.into.blocks(CHAN OF BIT bit.source,
                           CHAN OF SIGNAL end.of.source,
                           CHAN OF BLOCK block.sink     )
  -- Copy data from bit.source in complete blocks onto
  -- block.sink, until a signal arrives on end.of.source
  BOOL next.bit :
  ALT
    bit.source ? next.bit   -- Read ahead the first bit
      BOOL more.bits.to.pack :
      SEQ
        more.bits.to.pack := TRUE
        WHILE more.bits.to.pack
          VAL INT block.size IS bytes.in.a.block :
          [block.size]BYTE buffer :
          SEQ
            SEQ byte.number = 0 FOR bytes.in.a.block
              IF
                more.bits.to.pack
                  SEQ bit.number = 0 FOR bits.in.a.byte
                    IF
                      more.bits.to.pack
                        SEQ
                          set.bit(buffer[byte.number],
                                           bit.number, next.bit )
                          ALT
                            bit.source ? next.bit
                              SKIP
                            end.of.source ? CASE signal
                              more.bits.to.pack := FALSE
                      NOT more.bits.to.pack
                        SKIP
                NOT more.bits.to.pack
                  SKIP
            block.sink ! buffer
    end.of.source ? CASE signal -- No bits at all
      SKIP
:
```

A more modular solution, Process 8.21, would separate the two loops into two processes: one to turn the sequence of bits into a sequence of bytes; the other to turn the sequence of bytes into a sequence of blocks.

Process 8.21

```
PROC pack.bits.into.blocks(CHAN OF BIT bit.source,
                           CHAN OF SIGNAL end.of.source,
                           CHAN OF BLOCK block.sink     )

  PROC pack.bits.into.bytes(CHAN OF BIT bit.source,
                            CHAN OF SIGNAL end.of.source,
                            CHAN OF BYTE byte.sink       )
    BOOL more.bits.expected :
    SEQ
      more.bits.expected := TRUE
      WHILE more.bits.expected
        BYTE byte :
        ALT
          BOOL bit :
          bit.source ? bit
            SEQ
              set.bit(byte, 0, bit)
              SEQ bit.number = 1 FOR bits.in.a.byte - 1
                ALT
                  more.bits.expected & bit.source ? bit
                    set.bit(byte, bit.number, bit)
                  more.bits.expected & end.of.source ? CASE signal
                    more.bits.expected := FALSE
                  NOT more.bits.expected & SKIP
                    SKIP
              byte.sink ! byte
          end.of.source ? CASE signal
            more.bits.expected := FALSE
  :

  CHAN OF BYTE bytes :
  CHAN OF SIGNAL end.of.bytes :
  PAR
    SEQ
      pack.bits.into.bytes(bit.source, end.of.source, bytes)
      end.of.bytes ! signal
    pack.bytes.into.blocks(bytes, end.of.bytes, block.sink)
:
```

Process 8.22

```
BOOL FUNCTION get.bit(VAL BYTE byte, VAL INT bit.number) IS
                          (((INT byte) >> bit.number) /\ 1) <> 0 :
```

Turning seven-bit characters into a sequence of bits is also a simple task, since there is again no structure clash. The result of a call of *get.bit*, Process 8.22, is true or false according as the *bit.number*$^{\text{th}}$ bit of the rep-

Process 8.23

```
INT VAL bits.in.a.character IS 7 :

PROC unpack.bits.from.chars(CHAN OF BYTE char.source,
                            CHAN OF SIGNAL end.of.source,
                            CHAN OF BIT bit.sink         )
  BOOL more.chars.expected :
  SEQ
    more.chars.expected := TRUE
    WHILE more.chars.expected
      ALT
        BYTE character :
        char.source ? character
          SEQ bit.number = 0 FOR bits.in.a.character
            bit.sink ! get.bit(character, bit.number)
        end.of.source ? CASE signal
          more.chars.expected := FALSE
:
```

Process 8.24

```
PROC pack.chars.into.blocks(CHAN OF BYTE char.source,
                            CHAN OF SIGNAL end.of.source,
                            CHAN OF BLOCK block.sink     )
  CHAN OF BIT bits :
  CHAN OF SIGNAL end.of.bits :
  PAR
    SEQ
      unpack.bits.from.chars(char.source, end.of.source, bits)
      end.of.bits ! signal
    pack.bits.into.blocks(bits, end.of.bits, block.sink)
:
```

resentation of *byte* is one or zero, so the character code can be treated as though it were an array of seven bits:

```
VAL INT bits.in.a.character IS 7 :
SEQ bit.number = 0 FOR bits.in.a.character
  bit.sink ! get.bit(byte, bit.number)
```

Process 8.23, which turns a stream of characters into a stream of the bits which make up their codes, least significant bit of the character first, has exactly the same structure as the earlier unpacking process.

Packing seven-bit characters into half-kilobyte blocks is simply a matter of unpacking the characters into a stream of bits, and assembling the blocks from the stream of bits, the one activity in parallel with the other (Process 8.24). Substantially the same program structure can clearly be used to turn the stream of blocks back into a stream of seven bit character codes, since that is just another, similar packing problem.

The solution to each packing problem is of one of the three forms that we have shown here: grouping small objects to make larger ones; dividing large objects to make small ones; or a problem in which a structure clash requires that both the input data and the output data be divided into common sub-components. The parallel solution to the third kind of problem makes it no more complicated than the others.

Distributed implementation of buffers

It sometimes happens that it is necessary to communicate between processes where it is impracticable for them to share a channel. As mentioned earlier, the best that can be done is to implement a buffer between the processes. Consider, for example, two groups of processes executed by two distinct transputers. Each process in one group needs to communicate with the corresponding process in the other group, but the single serial link connecting the two transputers will only support a single channel in that direction. (At its simplest, a serial link between transputers does support exactly one channel in each direction.) The strategy in this case is to multiplex the use of the channel. Each output-producing process sends over the same channel, but identifies its message with its own name.

To be concrete, let each of the concurrent producers be identified by an `INT` index, and let the messages be counted arrays of characters

```
PROTOCOL MESSAGE IS INT::[]BYTE :

PROC producer(CHAN OF MESSAGE output)
  ... write some messages to output
:

PROC consumer(CHAN OF MESSAGE input)
  ... read some messages from input
:
```

The aim is to produce a program which behaves as nearly as possible like

```
PAR i = 0 FOR width
  CHAN OF MESSAGE left, right :
  PAR
    producer(left)
    buffer(left, right)
    consumer(right)
```

where *buffer* is a buffer. For simplicity, assume that none of the producers or consumers ever terminates. The program must have the form

```
CHAN OF ... c :
PAR
  PAR
    PAR i = 0 FOR width
      producer(...)
    ...
  PAR
    ...
    PAR i = 0 FOR width
      consumer(...)
```

with only one channel global both to the array of producers and to the array of consumers.

Since a number of producers cannot use the same channel directly – only one process may write to a channel at a time – there must be a mediating process, and this process can add the tags identifying the source and consumer. Similarly there must be a mediating process directing the tagged messages to their ultimate destinations. Process 8.25 shows how these would be used.

Process 8.25

```
PROTOCOL TAGGED.MESSAGE IS INT; INT::[]BYTE :

PROC multiplex([]CHAN OF MESSAGE local,
               CHAN OF TAGGED.MESSAGE link )
  ...
:
PROC demultiplex(CHAN OF TAGGED.MESSAGE link,
                 []CHAN OF MESSAGE local     )
  ...
:
CHAN OF TAGGED.MESSAGE link :
PAR
  [width]CHAN OF MESSAGE local :
  PAR
    PAR i = 0 FOR width
      producer(local[i])
    multiplex(local, link)
  [width]CHAN OF MESSAGE local :
  PAR
    demultiplex(link, local)
    PAR i = 0 FOR width
      consumer(local[i])
```

The multiplexing process must continually arbitrate between the producers to decide which message is sent over the shared link, so should clearly be a repeated alternative process, Process 8.26, and the demultiplexing process must dole out the received messages according to their tags, Process 8.27.

Process 8.26

```
PROC multiplex([]CHAN OF MESSAGE local,
               CHAN OF TAGGED.MESSAGE link )
  WHILE TRUE
    ALT source = 0 FOR SIZE local
      INT count :
      [longest.message]BYTE string :
      local[source] ? count::string
        link ! source; count::string
:
```

Process 8.27

```
PROC demultiplex(CHAN OF TAGGED.MESSAGE link,
                 []CHAN OF MESSAGE local     )
  WHILE TRUE
    INT destination, count :
    [longest.message]BYTE string :
    SEQ
      link ? destination; count::string
      local[destination] ! count::string
:
```

At a first glance it might appear that this is all that is needed. It is relatively easy to show that any message which passes into the *multiplexer* along an input channel and which passes out of the *demultiplexer* arrives on the right channel. Similarly, the sequence of messages along corresponding pairs of channels is the same, because no message can ever overtake another with the same tag. However, this is not quite enough to be able to show that the multiplexer and demultiplexer together implement an array of buffers.

There is a problem still, and one which could not have arisen in a sequential program. Consider what might happen if one of the consumers, say that indexed by one, were never to accept any input. If the multiplexer were ever to accept input from the corresponding producer, that message would never be delivered, so the demultiplexer would wait indefinitely to perform an output on *local*[1]. This means that no further messages could ever be delivered to any other consumer, for example that indexed by zero. For that reason, the multiplexer and demultiplexer would not implement a buffer between the processes with index zero, nor by symmetry between any other pair.

The system with a multiplexer and demultiplexer has more opportunities to become deadlocked than the system with buffers, but since these are all caused by a *consumer* refusing a message this is not a problem in a system where each *consumer* is guaranteed eventually to consume each message sent to it, independently of the others. This would be the case if each consumer were always ready to receive any number of messages, or if there were some higher-level protocol that ensured that no producer ever sent any message that was not guaranteed to be accepted. In fact, it can be shown that this is the best that you can achieve with only one channel between the producers and consumers.

Whilst it is possible to avoid opportunities to become deadlocked by interposing another process which relieves the consumer of its responsibility to accept messages, to do so requires a second channel between the two parts of the program. A safe demultiplexer could provide a buffer between the link and the ultimate consumer; a safe multiplexer would accept and deliver over the link only those messages for which there was room in the demultiplexer's buffer. To code these requires a channel from the demultiplexer to the multiplexer indicating the state of the buffers.

An adequate buffer need contain room for only one message. An instance of Process 8.28 is always prepared to accept a message from *source* if it has sent more signals – one more signal, in fact – on *request* than it has received messages from *source*.

Process 8.28

```
PROC request.buffer(CHAN OF SIGNAL request,
                    CHAN OF MESSAGE source, sink )
  WHILE TRUE
    INT count :
    [longest.message]BYTE string :
    SEQ
      request ! signal
      source ? count::string
      sink ! count::string
:
```

A possible coding of the demultiplexing process contains an array of these buffers driven by a naïve demultiplexing process, and an additional process to multiplex signals from the buffers onto the *back* channel, Process 8.29. On the assumption that the signals sent on the *back* channel are always accepted, it is easy to see that this process must, without blocking, accept from the *link* channel any message with an index that it has sent over the *back* channel. Moreover, it only sends a given index on the *back* channel once more than it has received messages with that index.

A similarly coded multiplexing process, Process 8.30, uses an array of buffers to throttle the input to a naïve multiplexing process. A message is only sent over the *link* channel if a corresponding index has been received over the *back* channel. Moreover, if messages are never blocked on the *link* channel this process is always prepared to receive indices from the *back* channel, so

Process 8.29

```
PROC safe.demultiplex(CHAN OF TAGGED.MESSAGE link,
                      CHAN OF INT back,
                      []CHAN OF MESSAGE local     )
  [width]CHAN OF SIGNAL ack :
  [width]CHAN OF MESSAGE fwd :
  PAR
    demultiplex(link, fwd)
    PAR i = 0 FOR width
      request.buffer(ack[i], fwd[i], local[i])
    WHILE TRUE
      ALT i = 0 FOR width
        ack[i] ? CASE signal
          back ! i
:
```

Process 8.30

```
PROC safe.multiplex([]CHAN OF MESSAGE local,
                    CHAN OF TAGGED.MESSAGE link,
                    CHAN OF INT back            )
  [width]CHAN OF SIGNAL ack :
  [width]CHAN OF MESSAGE fwd :
  PAR
    PAR i = 0 FOR width
      request.buffer(ack[i], local[i], fwd[i])
    multiplex(fwd, link)
    WHILE TRUE
      INT i :
      SEQ
        back ? i
        ack[i] ? CASE signal
:
```

Process 8.31

```
CHAN OF TAGGED.MESSAGE link :
CHAN OF INT back :
PAR
  [width]CHAN OF MESSAGE local :
  PAR
    PAR i = 0 FOR width
      producer(local[i])
    safe.multiplex(local, link, back)
  [width]CHAN OF MESSAGE local :
  PAR
    safe.demultiplex(link, back, local)
    PAR i = 0 FOR width
      consumer(local[i])
```

long as no index is received more than once more than a message with that index has been sent on the *link* channel.

Each half of the system has been shown to be safe, provided that it can rely on the safeness of the other half. Combining these arguments shows that a system built with 'safe' components, Process 8.31, is guaranteed not to become deadlocked.

Even this multiplexer and demultiplexer do not together implement an array of buffers. If one of the pairs of producer and consumer with a given index were to communicate an indefinitely long sequence of messages, the alternative processes might always be able to select the channels with that index. That would mean that although progress was always being made – the system did not become deadlocked – yet no messages would be transferred on any of the other virtual channels. In such a circumstance, the other processes are said to be *starved*. If a system of processes ignores its environment altogether – if it starves its environment – because it is busy with an infinite sequence of internal communications, the system is said by some to be *livelocked*.

The safe multiplexer and demultiplexer together cannot become livelocked, because they can never make an infinite sequence of communications without communication with a producer or consumer. If a producer and a consumer are added to them which communicate an infinite stream of messages, then that system is allowed to become livelocked.

In fact, this is a problem only if a group of the producers and consumers are able to keep the multiplexer and demultiplexer busy, to the exclusion of some others. Normally, it will be enough to observe that if the traffic does not exceed the bandwidth of the multiplexer and demultiplexer, starvation will not occur. If there is possibility of starvation, however, it is necessary to resort to asymmetric alternatives.

If the *multiplexer* component were *fair*, there should be no way that it could infinitely often decline to select any given channel which was ready. In fact, fairness is not particularly useful because a fair multiplexer could still take arbitrarily long to get around to noticing a particular channel. What is needed in practice is *short-term* fairness: a bound on the time for which a channel can be passed over.

The symmetric alternative constructor, `ALT`, is associative, commutative, and idempotent: all that determines the meaning of an alternative process is the *set* of guards which appear in it, not how they appear in nested alternatives, the order in which they appear, or indeed how often they appear. For that reason, we say that

```
ALT j = 0 FOR n
  c[j] ? ...
    ...j...
```

is the same process as

```
ALT
  c[i] ? ...
    ...i...
  ALT j = 0 FOR n
    c[j] ? ...
      ...j...
```

provided $0 \leq i < n$. Either would do in place of the other. An asymmetric alternative process on the other hand cannot pass over its first guard; so

```
WHILE TRUE
  SEQ i = 0 FOR n
    PRI ALT
      c[i] ? ...
        ...i...
      ALT j = 0 FOR n
        c[j] ? ...
          ...j...
```

cannot pass over any of the channels $c[i]$ for more than n consecutive iterations of the inner loop. To this extent it is a short-term fair implementation of the process

```
WHILE TRUE
  ALT j = 0 FOR n
    c[j] ? ...
      ...j...
```

If both of the alternatives in the system – one in the procedure *multiplex*, and the other in *safe.demultiplex* – are recoded in this way (see page 237) then the multiplexer and demultiplexer together implement an array of independent buffers. A system which could become neither deadlocked nor livelocked if connected by an array of buffers is also well behaved if connected by a safe, fair multiplexer and demultiplexer.

This rather lengthy discussion outlines a general strategy for demonstrating that a system is correct.

- It is important that the data values passed in messages and stored in variables cannot be incorrect.

- It must be shown that the system cannot become deadlocked: that there is always something that the system can do.

- It must be shown that the system cannot become livelocked: that by doing something that it can do, the system makes progress.

In a well structured system, it should be possible to demonstrate these three things independently, and in a modular fashion. For example, although *liveness* – a guarantee that a system cannot become deadlocked – is a global property of the whole system, in a well constructed system the liveness of the system should follow from demonstrable properties of the components.

Using parallelism to eliminate recursion

If a procedure has no free variables, and only value and variable parameters, a call to it can be thought of as a pair of communications; an output to send it the values of its parameters, and an input to receive in return the final values of its variable parameters.

To be precise,

```
PROC accumulate(INT x, VAL INT dx)
  x := x + dx
:
...
accumulate(acc, expr)
...
```

has the same effect on the value of the variable *acc* as

```
PROTOCOL int.pair IS INT; INT :
PROC server(CHAN OF INT.PAIR call, CHAN OF INT return)
  INT x, dx :
  SEQ
    call ? x; dx
    return ! x + dx
:
...
CHAN OF INT.PAIR call :
CHAN OF INT return :
PAR
  server(call, return)
  SEQ                     -- 'user' process
    call ! acc; expr
    return ? acc
...
```

A similar equivalence holds for a procedure or group of procedures which use and modify a free variable which is not used in any other way. In ALGOL, variables like this were called *own* variables, and they were difficult to use because of the problem of giving the variable an initial value. A procedure with an own variable can be implemented by communicating with a process which services a sequence of requests. Rather than write

```
INT accumulator :
PROC accumulate(VAL INT increment)
  accumulator := accumulator + increment
:
INT FUNCTION total() IS accumulator :
SEQ
  accumulator := 0
  ...
  accumulate(expr)
  ...
  use(total())
  ...
```

it would be natural in occam to write something like Process 8.32. Three points are worth making about this program. Setting of the variable to an initial value has been hidden away in the server, making it obvious that no accesses can be made to the variable until after it has been given a value. Not every call needs to be implemented as a pair of communications; since calling *accumulate* returns no result, a single output will suffice. Thirdly, termination of the server has to be caused by its user, so that their parallel composition can terminate; in this case the termination is caused by an explicit signal.

An array of mutually communicating processes can in this way be used to implement what might have been written, were recursive procedures allowed, as a recursive procedure. Consider implementing a process to maintain a set of at most n characters. There should be a way of adding a character to the set, and a way of testing whether a particular character is already in the set. To do this in the style of communication with a server process, we would need a *byte.set* with the interface described by

```
PROTOCOL CALLS
  CASE
    add.element; BYTE
    is.element; BYTE
:
PROTOCOL RETURNS
  CASE
    is.element.return; BOOL
:
PROC byte.set(CHAN OF CALLS call, CHAN OF RETURNS return)
  ...
:
```

Process 8.32

```
PROTOCOL CALLS
  CASE
    accumulate; INT
    total
    terminate
:

PROC server(CHAN OF CALLS call, CHAN OF INT return)
  BOOL more :
  INT accumulator :
  SEQ
    more, accumulator := TRUE, 0
    WHILE more
      call ? CASE
        INT dx :
        accumulate; dx                -- accumulate(dx)
          accumulator := accumulator + dx
        total                         -- return total()
          return ! accumulator
        terminate                     -- done
          more := FALSE
:

CHAN OF CALLS call :
CHAN OF INT return :
PAR
  server(call, return)
  SEQ
    ...
    call ! accumulate; expr
    ...
    INT x :
    SEQ
      call ! total
      return ? x
      use(x)
    ...
    call ! terminate
```

where sending *add.element*; x to an instance of the *byte.set* would add the character x to it, and sending *is.element*; x to it would cause the return of *is.element.return*; `TRUE` or *is.element.return*; `FALSE` from it according as x was an element of it or not.

Suppose we had a procedure *small.set* which could itself handle up to one element of a set, and which would pass on any other elements to a slave process which would handle the rest of the set. Such a procedure would look

Process 8.33

```
PROC singleton(VAL BYTE x,
               CHAN OF CALLS call, slave.call,
               CHAN OF RETURNS return, slave.return )
  -- represents the singleton set containing x
  WHILE TRUE
    call ? CASE
      BYTE y :
      add.element; y
        IF
          x = y
            -- y is already in the set
            SKIP
          x <> y
            -- add y to the slave set
            slave.call ! add.element; y
      BYTE y :
      is.element; y
        IF
          x = y
            -- y is in the set
            return ! is.element.return; TRUE
          x <> y
            -- y is in the set iff it is in the slave set
            BOOL found :
            SEQ
              slave.call ! is.element; y
              slave.return ? CASE is.element.return; found
              return ! is.element.return; found
:

PROC small.set(CHAN OF CALLS call, slave.call,
               CHAN OF RETURNS return, slave.return )
  WHILE TRUE
    call ? CASE
      BYTE x :
      add.element; x
        singleton(x, call, slave.call, return, slave.return)
      BYTE x :
      is.element; x
        return ! is.element.return; FALSE
:
```

like Process 8.33. When the *small.set* is empty, it is prepared to accept an *add.element* instruction and become a *singleton* set containing that element; at the same time, it is prepared to accept enquiries about set membership, and replies that the element is not in the set. The only element which can be added to a *singleton* is its member x; requests to add other elements to the set are passed on to the slave process which represents the rest of the set. A *singleton* process can tell that its own element is in the set; if asked about any other element it passes on the enquiry to the rest of the set, and passes the answer back.

If only we were allowed to write recursive procedures, we might implement the slave set as a *byte.set*, and implement a *byte.set* as a *small.set* in parallel with this, Process 8.34. Beware, this text would not define a recursive procedure in occam!

Process 8.34

```
PROC byte.set(CHAN OF CALLS call, CHAN OF RETURNS return)
  CHAN OF CALLS slave.call :
  CHAN OF RETURNS slave.return :
  PAR
    small.set(call, slave.call, return, slave.return)
    byte.set(slave.call, slave.return)
:
```

Now, if an instance of *byte.set* is required to store a set of up to n characters the slave set within it need store no more than $n-1$ characters. If the programmer were to substitute the body of *byte.set* in place of the 'recursive' call of it

```
PAR                        -- set of up to n elements
  small.set(...)
  PAR                      -- set of up to n-1 elements
    small.set(...)
    byte.set(...)          -- set of up to n-2 elements
```

the new instance of *byte.set* would need to be able to deal with only $n-2$ characters. The process of substitution can be repeated n times, until the problem has been reduced to one of implementing a process like *byte.set* which needs to be able to store no more than no characters at all. Since this particular instance of *byte.set* could be implemented by an instance of *small.set*, the task is done. Process 8.35 implements a set that can store up to n characters. Notice that there is only one process using each of the channels $calls[n]$ and $returns[n]$, but this does not matter because no communications are ever attempted on these channels.

In effect, this procedure uses an array of $(n+1)$ *small.set* processes to implement a set of n elements: one process looks after the first element and communicates with the user; it uses an array of one fewer *small.set* processes as its slave. This sort of technique can be used to eliminate any linear recursions for which there is a known bound on the depth of the recursion. In Chapter 12 there is a slightly more complicated example which implements two recursive calls made in parallel with each other by using a tree-shaped network of processes.

Process 8.35

```
PROC byte.set(CHAN OF CALLS call, CHAN OF RETURNS return)
  -- a set of up to n characters
  [n+1]CHAN OF CALLS calls :
  [n+1]CHAN OF RETURNS returns :
  PAR
    small.set(call, calls[0], return, returns[0])
    PAR i = 1 FOR n
      small.set(calls[i-1], calls[i], returns[i-1], returns[i])
:
```

Process farming

This chapter has been about the structures from which parallel programs are composed. You should think of things like buffers and multiplexers as being components in the parallel programmer's toolkit. They serve the same skeletal purpose in concurrent programs as are served by the familiar idioms of sequential programs, things like well-known sorting and searching algorithms. To close, here is an entirely unstructured scheme, which has turned out to be an unpredictedly simple way of using the parallel computers which have become available over the past few years.

There are some kinds of problems which are computationally expensive, but which can very easily be divided into a large number of small independent problems. You might call these 'embarrassingly parallel' problems, because almost any strategy for solving such a program on a multiprocessor machine will take some advantage of the parallelism in the machine. This is why such problems are often used in demonstrations of parallel machines.

For example a 'ray-traced' picture is constructed by performing, for each point on a screen, a lengthy calculation on a few data which describe the objects in the picture. The calculation follows the history of the ray of light which comes to the observer's eye through that point in the picture. By finding from which sources of light it has come, from which surfaces it has been reflected, and through which transparent bodies it has been transmitted, the colour and brightness of that ray can be constructed. The resulting pictures are startlingly realistic renderings of the non-existent scenes.

If the calculations at each point in the picture are treated as independent tasks, a ray tracing program can be executed more quickly by using more processors until the number of processors approaches the number of points on the screen, which is typically several millions.

A particularly simple general strategy, Process 8.36, is to run a *worker* process on each processor of the machine; each *worker*, Process 8.37, is a little sequential process accepting and processing a sequence of the small parts of the problem.

One additional process, Process 8.38, can then handle a pool of tasks

Process 8.36

```
PROTOCOL TASKS
  CASE
    new.task; INT; REAL64
    no.more.tasks
:
PROTOCOL RESULTS IS INT; REAL64 :

[n]CHAN OF TASKS to.slave :
[n]CHAN OF RESULTS from.slave :
PAR
  farmer(tasks, to.slave, from.slave, results)
  PAR j = 0 FOR n
    worker(to.slave[j], from.slave[j])
```

Process 8.37

```
PROC worker(CHAN OF TASKS from.farmer, CHAN OF RESULTS to.farmer)
  BOOL more :
  SEQ
    more := TRUE
    WHILE more
      from.farmer ? CASE
        INT i :
        REAL64 task, result :
        new.task; i; task
          SEQ
            result := f(task)
            to.farmer ! i; result
        no.more.tasks
          more := FALSE
:
```

and 'farm' them out to the workers. First a task is given to each of the n workers. Then whenever a worker finishes its present task, the result is recovered and a new task issued. This continues until all the tasks have been allocated, after which the farmer collects the remaining results and signals to the workers that they should terminate.

Provided that a calculation of f takes a long time, compared with the cost of communicating with the *farmer*, this program could be placed on n or $n+1$ processors and would execute almost n times as quickly as the equivalent sequential program

```
SEQ i = 0 FOR SIZE tasks
  results[i] := f(tasks[i])
```

even though the calls of f might take widely different amounts of time. There is scope for tuning a farm like this, for example for each *worker* to keep a

Process 8.38

```
PROC farmer(VAL []REAL64 task,
                      []CHAN OF TASKS to.slave,
                      []CHAN OF RESULTS from.slave,
                                              []REAL64 result )
  -- This process assumes n <= (SIZE task)
  SEQ
    PAR j = 0 FOR n
      to.slave[j] ! new.task; j; task[j]
    SEQ i = n FOR (SIZE task) - n
      ALT j = 0 FOR n
        INT k :
        from.slave[j] ? k; result[k]
          to.slave[j] ! new.task; i; task[i]
    SEQ i = 0 FOR n
      ALT j = 0 FOR n
        INT k :
        from.slave[j] ? k; result[k]
          to.slave[j] ! no.more.tasks
:
```

Process 8.39

```
[number.of.workers + 1]CHAN OF ADDR.TASK.STREAM outgoing :
[number.of.workers + 1]CHAN OF ADDR.RESULT.STREAM incoming :
PAR
  farmer(task, outgoing[0], incoming[0], result)
  PAR i = 0 FOR number.of.workers
    work.detail(outgoing[i], outgoing[i+1],
                incoming[i+1], incoming[i] )
  farmers.boy(outgoing[number.of.workers],
              incoming[number.of.workers] )
```

task in hand so that it never has to wait for the *farmer*.

More generally, all of the processors in a machine need not be directly connected, so it might not be possible to have a channel from the *farmer* to each *worker*. A farm can be constructed on any connected set of processors by grouping the workers into trees which are connected at their roots to the farmer. The *farmer* sees each of these trees as a worker which is capable of performing some number of tasks simultaneously. Each of the nodes of the tree must, in parallel with executing its own *worker*, farm tasks out to the sub-trees below it and recover their results.

Alternatively, the workers might be arranged in pipelines or rings, each node being additionally responsible for passing streams of tasks from the farmer to its successors, and streams of results from it predecessors to the farmer. The appendix contains a re-coding of the farm (page 241) to be executed on a row of processors each connected to its neighbours by a single pair of channels. In Process 8.39 each *work.detail* runs a single *worker*, but as well as dealing locally with a stream of tasks addressed to its worker, it also passes tasks to the sub-farm of processes with higher indices than its own. Each *work.detail* might be executed on a different transputer; each pair of channels $incoming[i]$ and $outgoing[i]$ could be implemented by a single link between an adjacent pair of processors.