6 Parallel matrix multiplication

In systems which manipulate and display geometrical data, one of the common routine tasks is the application of linear transformations to the data. A system containing a representation of a three-dimensional object may need to rotate or displace that representation so as to select a point of view from which to project a two-dimensional picture of the object onto the plane of a terminal screen, or a plotter. If the positions of the parts of the object are represented by a sequence of Cartesian co-ordinates, then these rotations and displacements can be achieved by matrix multiplication. For each point, with co-ordinates $\langle x_0, x_1, x_2 \rangle$, it is necessary to calculate the corresponding transformed co-ordinates $\langle y_0, y_1, y_2 \rangle$ given by

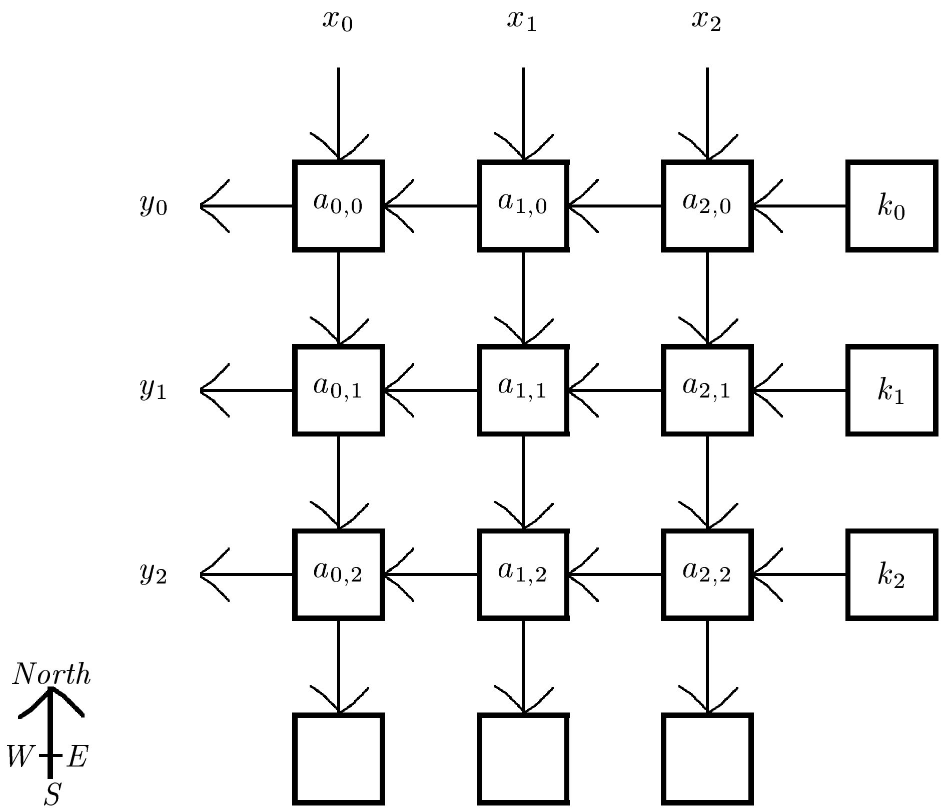
$$y_i = \sum_{j=0}^{2} a_{ij} x_j + k_i$$

This requires nine multiplications and nine additions for each point in the representation of the object.

If the transformation is being applied once to an object with a view to printing an image on a slow, hard copy device such as a pen plotter, then the time taken to do the transformation is probably not important, and it does not matter much how the matrix multiplication is organized. On the other hand, if the image is being displayed on a cathode ray tube, and the observer is allowed to change his point of view from the console, then speed is important. Ideally, the transformation should be applied to every relevant point of the object as the position of that point is required for refreshing the display, so that the observer sees the effect of a change in the transformation as soon as possible.

If there are of the order of a thousand points in the representation of the image, then this means something of the order of a hundred thousand matrix multiplications in a second. For practical purposes, this requires that special hardware be dedicated to performing the matrix multiplications on a stream of co-ordinates on its way to the display. In such an arrangement, the time taken to perform the nine individual multiplications will dominate the time taken by all of the communications and additions involved. There is therefore an advantage in arranging that as many as possible of the multiplications can happen at once.

A natural configuration of processors to perform this task is a square array, mimicking the matrix a, one processor being responsible for each element of the matrix, and performing the multiplication by that element.



Successive values of each x co-ordinate are poured into the array from the top, passing down along the north to south channels, and successive values of the transformed y co-ordinates emerge from the east to west channels at the left of the array. In this diagram, each processor is labelled with the parameter for which it takes responsibility. For simplicity, the transformation is assumed to be constant: a mechanism for changing the parameter values might involve a further array of channels at right angles to the plane of the array of processes. These would connect each relevant processor to a controlling process.

Each multiplier cell has three tasks to perform during each complete matrix multiplication: getting the next co-ordinate x_j from its northern neighbour and passing it on to its southern neighbour; performing its own multiplication; getting a partial sum from its eastern neighbour, adding its own contribution and passing the sum on to the west. These tasks might be performed sequentially

```
VAR xj,  aij.times.xj,  yi :
WHILE TRUE
  SEQ
    SEQ
      north ? xj
      south ! xj

    aij.times.xj := aij * xj

    SEQ
      east ? yi
      west ! yi + aij.times.xj
```

Because the condition on the loop is a constant `TRUE`, this process never terminates; it repeatedly performs the three tasks in strict sequence.

Since this is a design for highly parallel hardware, it should be worth extracting a little more parallelism. The input from *north* has to happen before each of the actions that use the value of *xj* that is received, and the input from *east* has to precede the use of *yi*. There is no reason, however, to wait for the value of *yi* before doing the multiplication (which was assumed to be the most expensive part of the task). Similarly, the multiplication must be completed before the variable *xj* is reused for the next input from *north*, and of course the multiplication must be complete before its result is used in the addition. The process can be rearranged to 'overlap' the multiplication with activity of which it is independent, provided only that these constraints are respected.

```
PROC multiplier(VALUE aij,
                CHAN north, south, west, east) =
  VAR xj,  aij.times.xj,  yi :
  SEQ
    north ? xj
    WHILE TRUE
      SEQ
        PAR
          south ! xj
          aij.times.xj := aij * xj
          east ? yi
        PAR
          west ! yi + aij.times.xj
          north ? xj                           :
```

Since different components of a *multiplier* implemented in hardware would be used by the arithmetic and each of the communications, the branches of the `PAR` constructs naturally execute simultaneously.

Notice that the multiplier process does not need to know where it is in the array—it is independent of i and of j. This means that an implementation in hardware could use nine identical circuits.

In order to complete the matrix multiplier, a source of the k_i offset values is needed along the eastern border

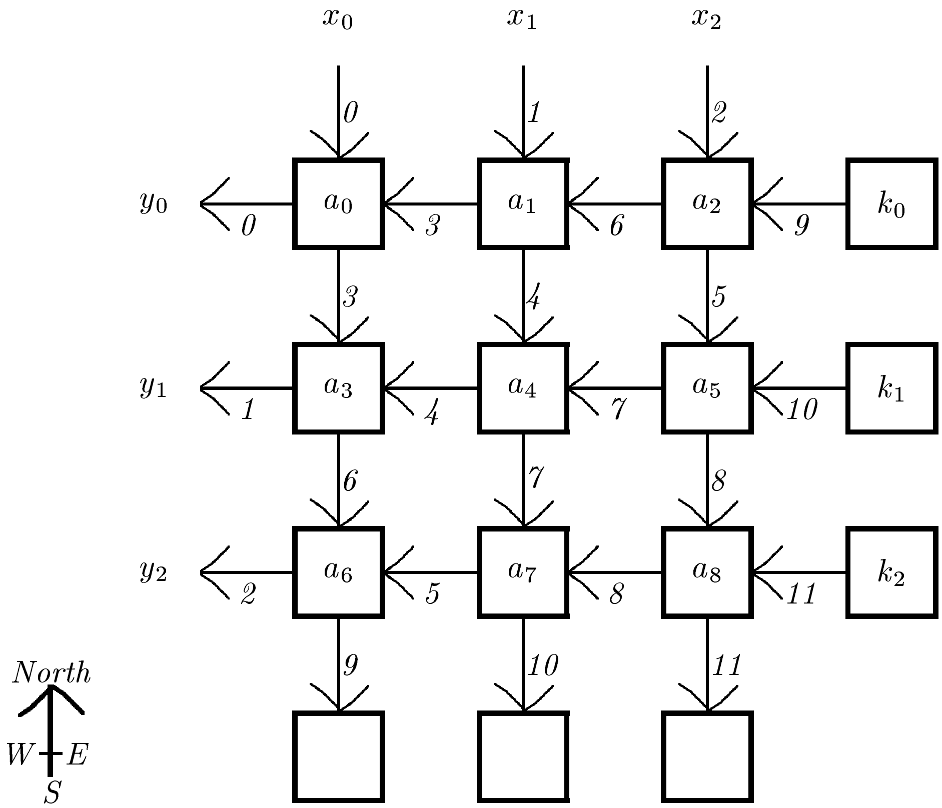
```
PROC offset(VALUE ki, CHAN west) =
  WHILE TRUE
    west ! ki                      :
```

and a sink must be provided at the southern end of each column of multipliers to receive the redundant x_j from the southernmost multiplier processes

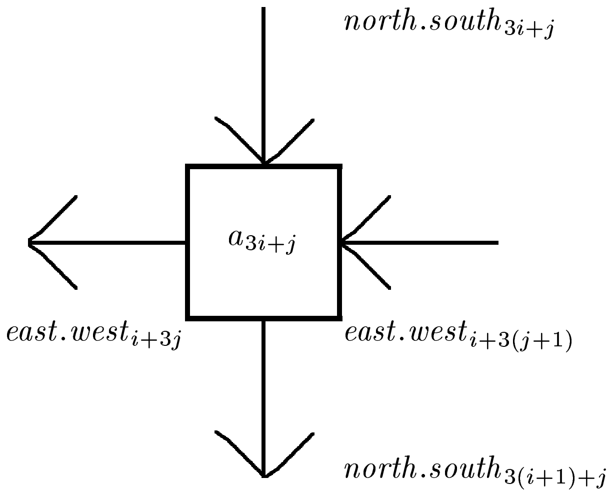
```
PROC sink(CHAN north) =
  WHILE TRUE
    north ? ANY        :
```

Although the *sink* does nothing with the values received, its input actions are necessary so that the corresponding output can happen in its neighbouring *multiplier*. A row of *sink* processes yields a simpler solution than one which involves two kinds of multiplier process, one for the north of the array, and another for the southernmost row.

Connecting these components to form the matrix multiplier is a matter of choosing an enumeration for the channels, and using `CHAN` arrays, suitably indexed. One solution is



The channels connected to a typical multiplier process are



Elements of the matrix a have also been enumerated, so as to correspond to the elements of an *occam* one-dimensional array. The whole multiplier is described by the program

```
DEF n = 3 :
VAR a[n * n],  k[n] :
SEQ
  -- initialize a and k

  CHAN north.south[(n+1)*n], east.west[n*(n+1)] :
  PAR
    PAR j = [0 FOR n]  -- producer of co-ordinates x[j]
      produce.xj(j, north.south[j])

    PAR                -- the matrix multiplier
      PAR i = [0 FOR n]
        offset(k[i], east.west[(n*n)+i])
      PAR i = [0 FOR n]
        PAR j = [0 FOR n]
          multiplier( a[(n*i)+j],
                      north.south[(n * i) + j],
                      north.south[(n * (i + 1)) + j],
                      east.west  [i + (n * j)],
                      east.west  [i + (n * (j + 1))])
      PAR j = [0 FOR n]
        sink(north.south[(n*n)+j])

    PAR i = [0 FOR n]  -- consumer of co-ordinates y[i]
      consume.yi(i, east.west[i])
```

It is the task of each *produce.xj* process to output successive values of the co-ordinate corresponding to its first parameter, and that of each *consume.yi* process to input successive values of the transformed co-ordinate.

By devising suitable definitions for the processes *produce.xj* and *consume.yi*, this program can be used on any *occam* implementation as a simulation of the parallel matrix multiplier hardware. Of course, if it is executed on a single processor computer, then it will be very much slower than a simpler sequential program, because of the additional work in communicating and scheduling. On the special hardware for which it is designed, however, it would be very much faster. The longest data path from input to output is that traversed by x_2 on the way to contributing to y_2. This path involves six communications, three additions, and a single multiplication, all of which must happen in sequence. The program is designed on the assumption that the time taken for the multiplication would dominate all others, under which assumption it would be almost nine times faster than a sequential implementation.

The matrix multiplier example appears in essentially this form as an example in C.A.R. Hoare, Communicating sequential processes, *Communications of the ACM*, **21** (8) August 1978, pp. 666–677.