11

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 vector of Cartesian co-ordinates, encoded as floating-point numbers, then these rotations and displacements can be achieved by matrix multiplication and vector addition. For each point $\mathbf{u}$, with co-ordinates $\langle u_0, u_1, u_2 \rangle$, it is necessary to calculate the corresponding vector $\mathbf{v}$ of transformed co-ordinates $\langle v_0, v_1, v_2 \rangle$ given by

$$v_i = \sum_{j=0}^{2} a_{ij} u_j + k_i$$

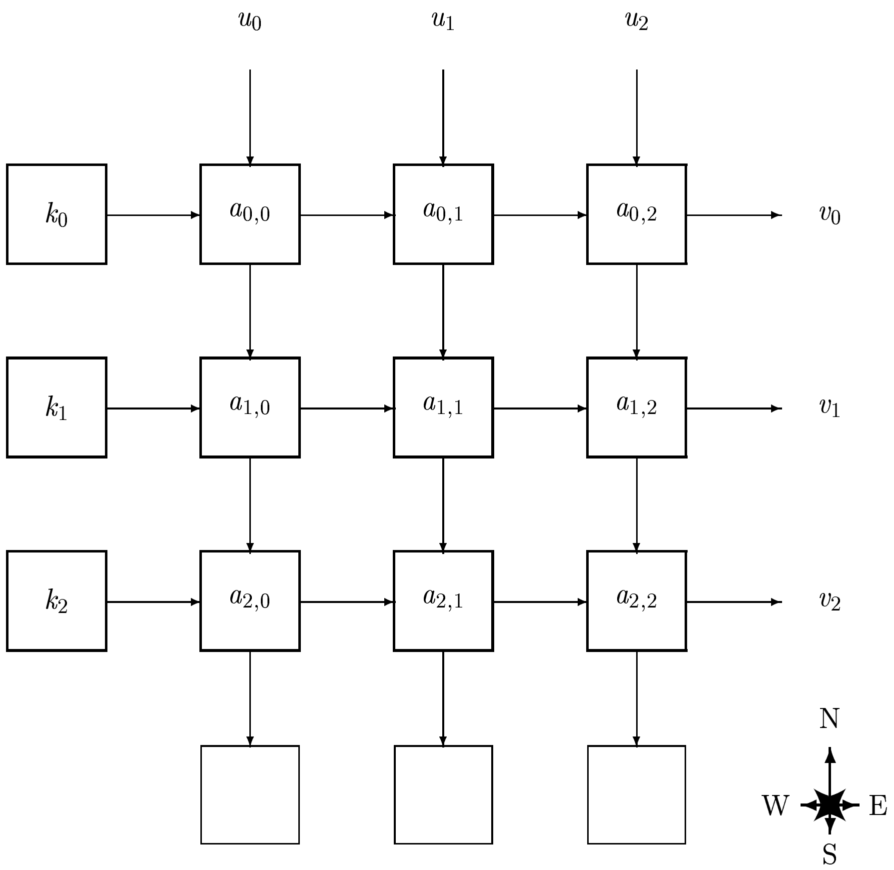
This requires nine multiplications and nine additions for each point in the representation of the object.

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 graphics screen 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.

Figure 11.1 shows a natural configuration of processors to perform this task: 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 co-ordinate of the vector $\mathbf{u}$ are poured

Figure 11.1 Design layout for the matrix multiplier



into the array from the top, passing down along the north-to-south channels, and successive values of the transformed co-ordinates of $\mathbf{v}$ emerge from the west-to-east channels at the right 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 u_j from its northern neighbour and passing it on to its southern neighbour; performing its own multiplication; getting a partial sum from its western neighbour, adding its own contribution and passing the sum on to the east. These tasks might be performed sequentially:

```
REAL32 uj,  aij.times.uj,  vi :
WHILE TRUE
  SEQ
    SEQ
      north ? uj
      south ! uj
    aij.times.uj := aij * uj
    SEQ
      west ? vi
      east ! vi + aij.times.uj
```

Because the condition on the loop is a constant `TRUE`, this process never terminates; it repeatedly performs the three tasks in strict sequence. This is to be expected, if we regard the process as a design for hardware; a multiplier chip should no more cease operation autonomously than, say, an and-gate.

Since the hardware we are designing is to be highly parallel for speed, it should be worth extracting a little more parallelism within the components. The input from *north* has to happen before each of the actions that use the value of *uj* that is received, and the input from *west* has to precede the use of *vi*. There is no reason, however, to wait for the value of *vi* 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 *uj* 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 (Process 11.1). Since different sub-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 western border (Process 11.2) – again, for simplicity, we take these to be constant – and a sink must be provided at the southern end of each column of multipliers to receive the redundant u_j from

Process 11.1

```
PROC multiplier(VAL REAL32 aij,
                CHAN OF REAL32 north, south, west, east )
  REAL32 uj,  aij.times.uj,  vi :
  SEQ
    north ? uj
    WHILE TRUE
      SEQ
        PAR
          south ! uj
          aij.times.uj := aij * uj
          west ? vi
        PAR
          east ! vi + aij.times.uj
          north ? uj
:
```

Process 11.2

```
PROC offset(VAL REAL32 ki, CHAN  OF REAL32 east)
  WHILE TRUE
    east ! ki
:
```

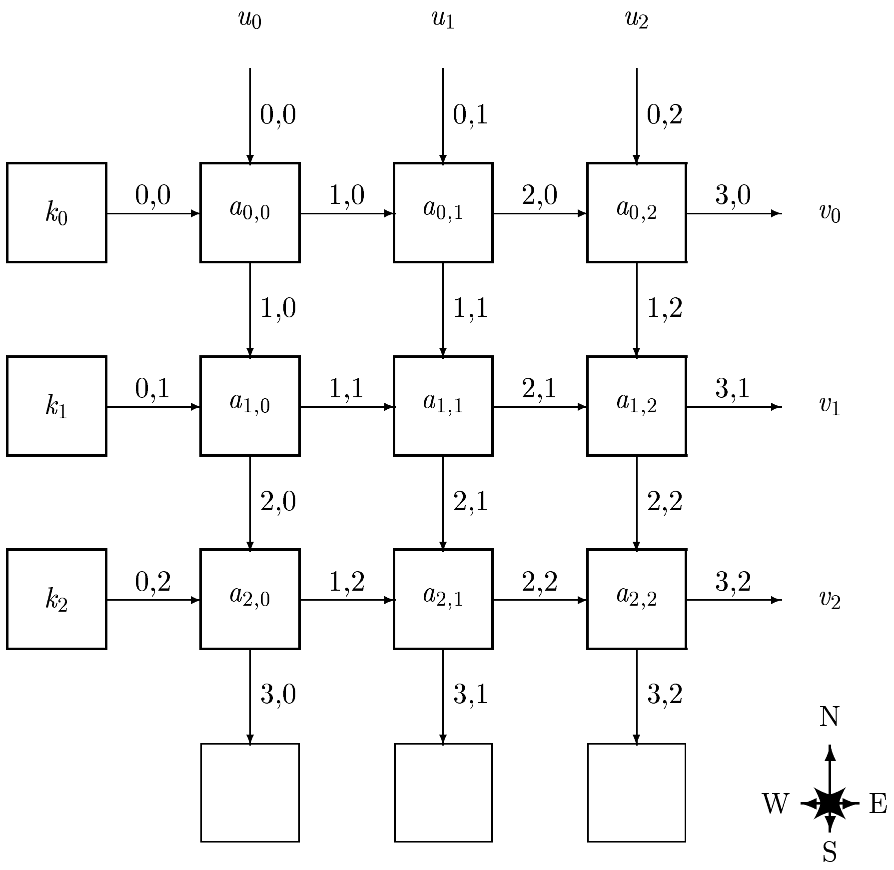
Process 11.3

```
PROC sink(CHAN OF REAL32 north)
  WHILE TRUE
    REAL32 discard :
    north ? discard
:
```

the southernmost multiplier processes (Process 11.3). 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 using `CHAN` arrays, suitably indexed. Two arrays are used, one for north-to-south traffic and one for west-to-east. As channel arrays can be two-dimensional, one obvious solution to the indexing would be to have the process handling a_{ij} served by input channels subscripted first by i and then by j, and output channels with the appropriate index incremented by one. This has the disadvantage that each one-dimensional sub-array of the west-to-east channel array comprises the four channels that carry some particular k_i through, accumulating terms, until it emerges as v_i; thus it is a collection of channels which are used sequentially, as far as any particular calculation is concerned, and it is not useful to be able to refer to it as a whole. If, however, the order of the subscripts to this array is reversed, each one-dimensional sub-array carries all the results at some point in the calculation, and in particular the easternmost carries the final answers. The preferred solution is therefore that of Figure 11.2, and so the channels connected to a typical multiplier process are as shown in Figure 11.3, while the matrix a naturally corresponds to the elements of an occam 2 two-dimensional array.

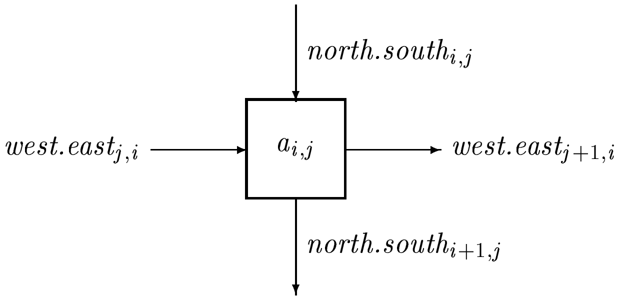
Figure 11.2 Channel indices for the matrix multiplier



The whole multiplier is thus described by Process 11.4. It is the task of the *produce.u* process to output the three co-ordinates of successive values of $\mathbf{u}$ in parallel, and that of the *consume.v* process to input the three transformed co-ordinates of successive values of $\mathbf{v}$.

Having supplied suitable definitions of the two processes *produce.u* and *consume.v*, this program can be run with any occam 2 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

Figure 11.3 A typical multiplier cell



Process 11.4

```
VAL INT n IS 3 :
VAL [n][n]REAL32 a IS
          ... values of the matrix :
VAL [n]REAL32 k IS
          ... values of the offsets :

[n+1][n]CHAN OF REAL32 north.south, west.east :
PAR
  produce.u(north.south[0]) -- producer of co-ordinates u[j]

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

  consume.v(west.east[n])   -- consumer of co-ordinates v[i]
```

in communicating and scheduling. On the special hardware for which it is designed, however, it will be very much faster. The longest data path from input to output is that traversed by u_0 on the way to contributing to v_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 [7].