A Systolic Array Solution for the Assignment Problem

G. M. MEGSON[1] AND D. J. EVANS[2]*

[1] *Computing Laboratory, University of Newcastle upon Tyne*

[2] *Parallel Algorithms Research Centre, Loughborough University of Technology, Loughborough, Leicestershire*

A systolic array for the solution of the assignment problem is presented. The algorithm requires $O(n^2)$ time and an orthogonally connected array of $(n+2) * (n+2)$ cells consisting of simple adders and control logic. The design is area-efficient and incorporates the new concept of a Systolic Control Ring (SCR) to generate the necessary systolic wavefronts in any orientation within the design, while special cells are positioned only on the periphery of the design.

The resulting Assignment Problem Iteration (API) is suitable for Wafer Scale integration using only single input and output for data permitting low bandwidth.

Received April 1987, revised July 1990

1. INTRODUCTION

In this report we present a systolic design to implement the assignment problem. This problem exhibits a number of features that indicate possible systolic solutions. For instance, it involves only add and subtract operations and, as we will see, some comparisons implying an area-efficient set of basic cells in the design. The assignment problem also requires a square table representation rather than rectangular in the case of the standard and revised forms of the systolic simplex algorithm investigated by the authors,[1,2] resulting in simpler computational wavefront synchronisation, hence a tight, simple control.

The assignment problem solution technique used for the array construction is the Hungarian algorithm, which proves to be a simple and effective computational procedure, again a favourable quality for a systolic design. In contrast, we note that the assignment can be converted into a transportation problem with n supplies and n demands all equal to 1 by essentially replacing the integer programming problem by a linear program. At this time the systolic implementation of the transportation algorithm has proved at best complex. We also note that under certain circumstances the corresponding transportation problem can become highly degenerate.

2. THE ASSIGNMENT PROBLEM

For completeness we state the details of the assignment problem as follows: let there be n tasks which must be performed by n individuals. The cost of individual i performing task j is denoted by c_{ij}. The problem is to assign people to the tasks in a way that minimises the cost of completing the tasks.

Let

$$x_{ij} = \begin{cases} 1 & \text{if person } i \text{ does task } j \\ 0 & \text{otherwise} \end{cases} \qquad i = 1(1)n, \quad j = 1(1)n.$$

We minimise the total cost, according to the constraints that one person is assigned to one task, and each task is assigned to one person.

* To whom correspondence should be addressed.

Then minimise

$$f = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i = 1(1)n$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad j = 1(1)n$$

$$x_{ij} = 0 \text{ or } 1 \quad i = 1(1)n, \quad j = 1(1)n$$

Remark. The matrix $c = c(ij)$ is called the cost matrix.

The Hungarian algorithm is an efficient method for solving this problem, and can be simply stated as follows.

Step 1

(*a*) For each row in the cost matrix c_{ij} locate the smallest number in the row, and subtract it from each number in the row.

(*b*) For each column in the new matrix locate the smallest element in the column and subtract it from each element in the column. The resulting matrix is called the reduced-cost matrix.

Step 2

Find the minimum number of lines through rows and columns of the reduced-cost matrix, such that all the zeros have a line through them. If the number of lines is n then stop (optimal solution found); else proceed to Step 3.

Step 3

A new reduced-cost matrix is now constructed. Locate the smallest number in the matrix without a line through it. Subtract the number from all uncovered numbers, and add to it numbers at the intersection of two lines (twice covered).

Go to Step 2. [N.B. A covered number is one with a line through it.]

The final solution is then constructed by assigning a worker to a job so that the reduced cost is zero. This is done by checking first the rows and then the columns, looking for rows or columns with only a single zero in them; the assignment is then the (i,j) ordered pair corresponding to the zero element.

3. THE SYSTOLIC ARCHITECTURE

The systolic design is split into two systolic arrays. The first array is a linearly connected array of n cells which computes the reduced-cost matrix in step 1 of the algorithm, essentially performing a pre-processing function. The second array is an $(n+2)*(n+2)$ orthogonally connected mesh of cells performing steps 2 and 3 and is the core of the algorithm; we refer to this array as the Assignment Problem Iteration (API) array. We could also consider a third array to perform post-processing to recover the final assignments, but this is a relatively straightforward task and is not pursued here.

3.1 Pre-processing array

Here we use a linearly connected systolic array of n cells to transform the starting-cost matrix to the reduced-cost matrix. The construction requires four passes through the array. *Pass* 1. Find the minimum element of each row, storing it in the cell. *Pass* 2. Subtract the stored value from all elements in the row. *Pass* 3. Find the minimum in each column, and store it in the cell. *Pass* 4. Subtract the stored element from all the elements in the column.

It appears from the array in Fig. 1 that the basic cell requires a subtractor and a comparator when choosing the minimum. However, if we include with the subtractor a status bit to register when a negative value is produced, we can detect a 'less than' condition without an additional comparator. Thus, each basic cell in the preprocessor is a simple subtractor with registers.

The change from pass 2 to pass 3 is crucial, as we turn the matrix input from row ordering to column ordering, which allows the same array to be used for the row and column passes. As the data is a square matrix or table the last column element leaves the array in a row pass, as the first element of the column is required to enter the array. It follows that we have sufficient time to reorganise the row form in pass 2 to the column form in pass 3 'on the fly' by the Host machine or a buffer. The data output by the pre-processing array at each pass is looped back into the array input; on the last pass the loop can be switched out to provide a suitable interface for loading the API array via its host interface.

The total time for all the passes is $T = 5n$ (a cycle is the cost of an add or subtract). If the last pass is used for loading the API array, the last n cycles can be overlapped with the start of steps 2 and 3, the API section of the algorithm.

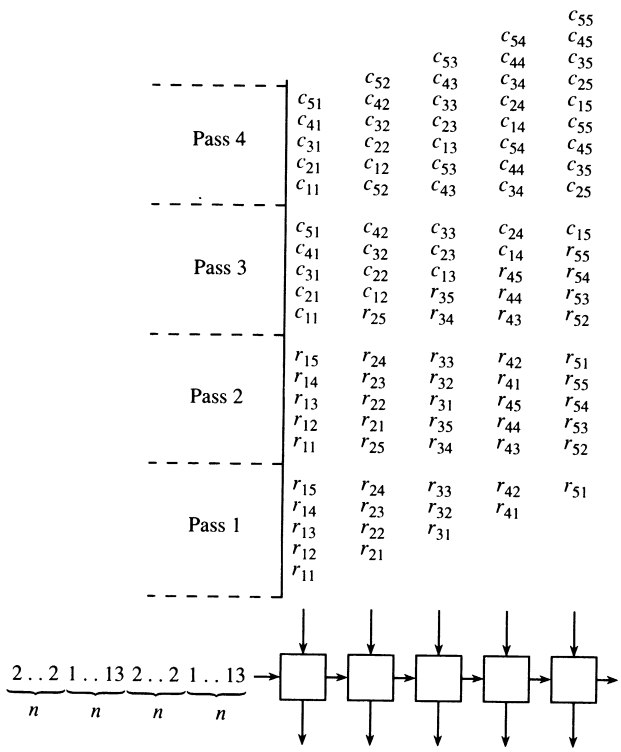

Figure 1. Pre-processing array ($n = 5$).

3.2 Assignment Problem Iteration (API) array

The API computes the optimal solution to the assignment problem given the reduced-cost matrix generated by the pre-processing array. A global view of the API array is shown in Fig. 2, where the major sub-arrays of the design are defined and will be discussed below. The basic structure is an orthogonally connected mesh of n by n cells placed inside a systolic control ring, in an area-efficient manner. The systolic control ring (SCR) as its name suggests controls the computation, its special structure allowing it to generate control wavefronts in any orientation travelling across the tableau representing the reduced-cost matrix. The SCR can therefore easily move control around the mesh by a series of point-to-point connections.

The remaining two steps of the algorithm forming the iteration are split into a number of distinct computational phases related to the state of the SCR during the algorithm iteration. The movement of control around the ring allows the wavefronts corresponding to different phases to be overlapped on the tableau, but in general interleaving and interference from distinct wavefronts is avoided, except where conflicts can be easily resolved with simple control sequences.

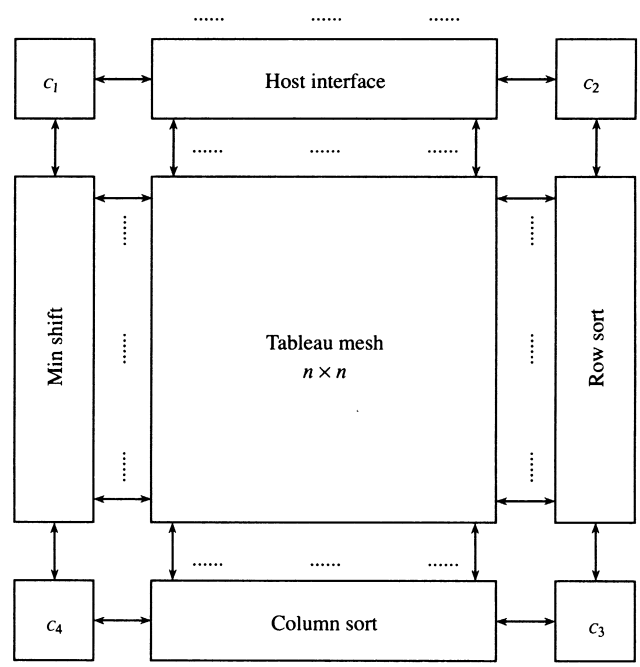

Figure 2. Assignment Problem Iteration (API) array.

Finally, the C_i, $i = 1(1)4$ are simple controllers performing combinational logic functions on control data streams only. The row, column sorters and Min-shift sub-arrays are linear systolic arrays of n cells. The API iteration begins with control signals from C_1, which is suitably close to the Host interface in case we want to reset the array from the Host, e.g. for loading data. The partitioning of the iteration algorithm into phases is as follows.

Phase O. Loading of the reduced-cost matrix. A number of options are available for this phase – the SCR is not required, and techniques for loading square meshes of processors with data are well understood.

Phase I. (Start of the iteration algorithm). In this phase two wavefronts are generated, moving from C_1 towards C_2 across the tableau.

(*a*) *Wavefront 1* (W_1). C_1 generates a control value moving horizontally to C_2, and vertically to C_4, through the Host interface and Min-shift arrays. These travelling signals generate the wavefront W_1 moving through the tableau. At the start, the min-shift cell immediately below C_1 contains the smallest uncovered element in the reduced-cost matrix. As W_1 moves across the tableau it performs the reduced-cost matrix modification, according to covered line positions. On the first cycle of the API algorithm Min-shift will contain only zeros and the first modification has no effect.

(*b*) *Wavefront 2* (W_2). On the cycle, after C_1 has generated controls for W_1, it generates a different control value also travelling on the SCR to C_2 and C_4. W_1 represents the end of step 3 of the algorithm, and W_2 the second wavefront generated in a similar manner is the start of step 2. W_2 uses a signal moving through the host interface to C_2, which is also sent vertically through the tableau to count the number of zeros in each column.

Phase II. This begins when the signal generated in phase I for W_2 reaches C_2. C_2 relays the signal to the row-sort array, which propagates the signal from C_2 to C_3. While the signal travels down to C_3, W_2 and W_1 continue moving in the tableau. At the start of Phase II the first column has completed its count of zeros in the column; we call this value the Column Zero Weight. Thus, as the control signal moves on the C_2–C_3 portion of the SCR the column sorter array is loaded with column zero weights. As the weights are loaded the sorter cell checks if the weight is zero and if on a previous iteration the column was covered by a line; if this is the case a signal is propagated back into the tableau to erase the line. The sort cell then waits a cycle for the adjacent cell to be loaded, and checks for line removal, then the cell starts sorting. As sorting takes place, the maximum column weight is bubbled right, and the minimum column weight bubbled left. As the weights are swapped, signals are output to the tableau to swap the corresponding column elements. When the signal from C_2 reaches C_3 a wavefront W_3 of swap controls is halfway across the tableau, and the last column weight is in the rightmost cell of the column sorter.

Phase III. The control signal continues on the SCR from C_3 to C_4 through the column sorter which continues to sort. The sorter being a linear array is used to perform an ODD–EVEN transposition sort or parallel bubble-sort, which requires n steps for n numbers when all the cells are working in parallel. It follows from the start-up of the sorter in Phase II that we must have moved the maximum zero right to the rightmost cell by the start of Phase III. Hence, as the control moves down the C_3–C_4 portion of the SCR we can close down the column-sorting cells while completing the sort. Thus, when the signal reaches C_4 the column weights are fully sorted, the sorting cells are off and the last possible column swap has entered the tableau.

Phase IV. The control signal now travels the section C_4–C_1 of the SCR, during which it passes through Min-shift causing a wavefront W_5 to be propagated right. W_5 performs the same task as W_2, collecting the row zero weights by counting zeros in each row. When the control reaches C_1, completing the first cycle of the SCR, the bottom row of tableau cells have completed the zero count.

Phase V. This phase is analogous to Phase II. C_1 generates a new control signal travelling C_1–C_2 on the next control cycle. Thus W_5 continues its journey right, and the row sorter is loaded with the row zero weights from bottom to top. Each weight is checked to see if it is zero, and as the row was previously covered a row line erase can be sent left to remove it. When the new signal reaches C_2 as for the column sorter the row sorter has loaded all the row weights, and all the cells are computing in parallel. A side effect of the sorting is that the minimum row weight has reached the sorting cell immediately below C_2. The wavefront W_6 is propagated left to perform relevant row swaps. At the end of this phase, W_6 is halfway across the tableau.

Phase VI. The control signal is now to travel C_2–C_3 and can be used to close down the row sorters, while also completing the swap wavefront moving left. At the end of the phase, the control value is in C_3, and the last row swap control has entered the tableau. All the row sorting cells are off, and the list of row weights is fully sorted, with the maximum row weight in the cell immediately above C_3.

Phase VII. We are now in a position to complete Step 2 of the algorithm by drawing the minimum number of lines. Notice that the heads of the sorted row and column zero weights are locally placed for C_3 to take control of the algorithm, which it now does. The basic technique for drawing the minimum number of lines will be discussed in detail later; we sketch the method here (see Fig. 3).

(*a*) C_3 collects the maximum row and column weights from the adjacent cells, and finds the maximum.

(*b*) If both values are zero then there are no zeros without lines covering them, go to Phase VIII. If n lines have been drawn, go to stop (optimal solution) otherwise draw a line.

(*c*) A line is drawn by zeroing the row or column weight which was a maximum, and sending a control value into the tableau from the sorter element cleared. For a column sorter cell the signal propagates up the column towards the host interface, and for a row cell towards the Min-shift array along the row. As the control moves it marks the cell element with a line state, and if the cell element is zero we send a control to the adjacent sorter cell and modify the column or row weight to remove the zero.

(*d*) At the same time as the tableau is being marked with an implicit line, the sorter cells are switched on by controls from C_3 along the C_3–C_4 and C_3–C_2 portions of

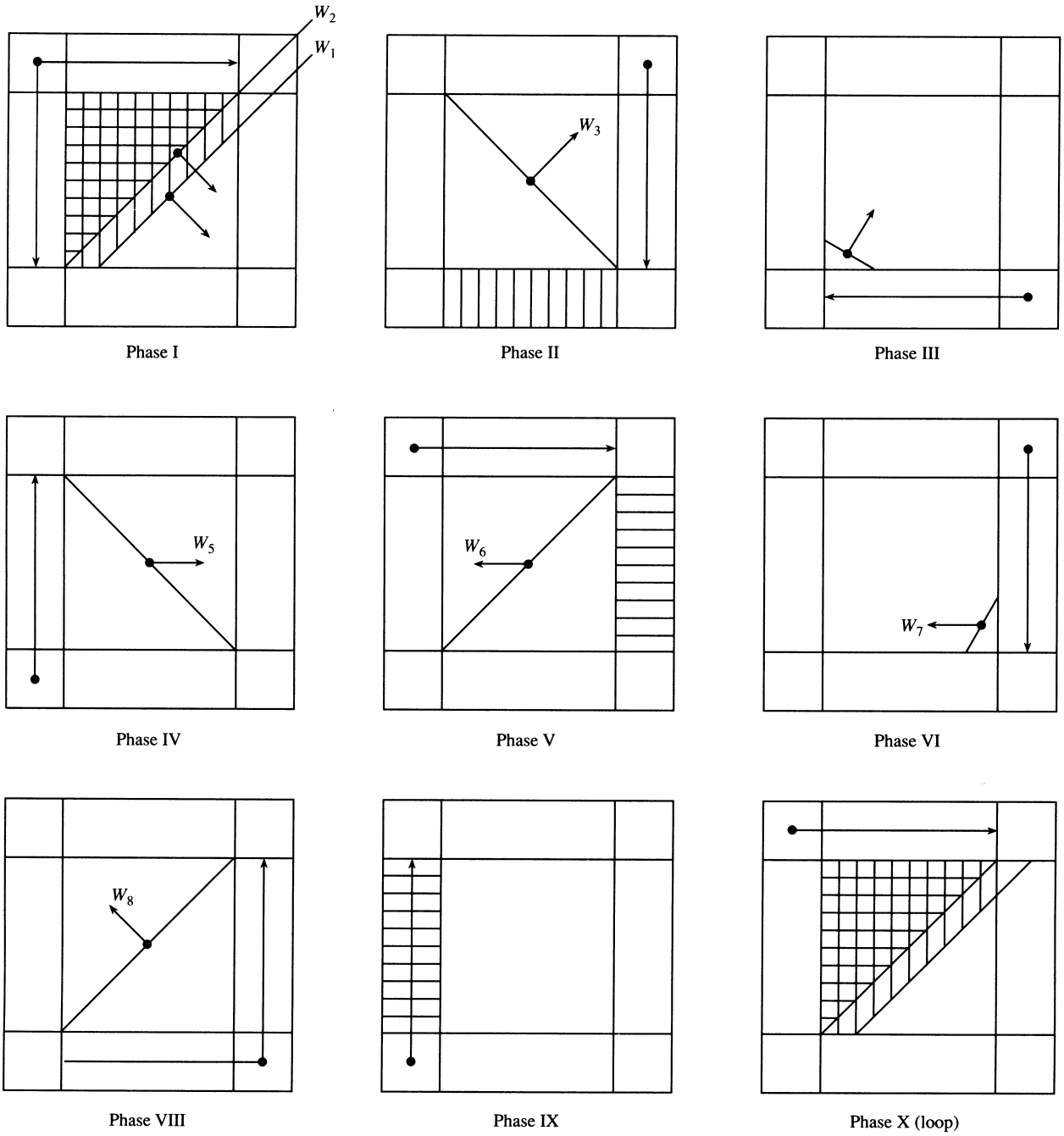

Figure 3. The computational phases for API. (Phase VII not shown is the line-drawing section.)

the ring and weight modify commands, and the lists are updated.

(*e*) The sorters are closed down by the returning signal from C_4 and C_2; as the round trip for the signal is $2n$ cycles of the basic cell the new list must be sorted. The previous maximum was set to zero and so will have bubbled down the list, hence we can loop back to (*a*).

Phase VIII. If we reach this phase we have reduced both the column and row lists to zero by covering all the zeros in less than n lines. Hence a modification of the cost matrix is required. C_3 now releases control and sends the control signal from Phase VI along C_3–C_4, producing a wavefront W_8, which moves the minimum element left. When the signal reaches C_4 the minimum of the last tableau row is available.

Phase IX. The control value now completes its second circuit of the SCR by moving along C_4–C_1; as it moves it loads the minimum elements of each row into Min-shift and also moves the minimum of the row minima up towards C_1. When the control completes its second cycle the minimum element of the tableau taken over uncovered values is in the Min-shift cells, and particularly the one adjacent to C_1. We now go to Phase I.

Stop. If stop is reached in Phase VII we close down the API array and get ready to output the final tableau.

This completes the phases of the algorithm; some remarks at this point are pertinent.

(1) The column and row sorting cells also contain an index for the row and column, so that the row and column swaps can be recorded and the final result recovered.

(2) In Phase II and Phase V, the column and row

sorters check the zero weights to see if they are zero and that the row or column had a line through it on the previous iteration. This implies that: (*a*) each sorter cell is marked when a line is drawn from it; (*b*) a row or column can only have a zero weight and be covered by a line if all its zeros were on intersections with other lines orthogonal to the line from the sorter cell: hence the line must be removed.

(3) In order to remove lines we have to compute totals or weights for rows and columns with lines already through them; these weight totals must be zeroed when a line is removed.

The points above seem to be making the array cell structures more complicated. In fact it can be shown that we will never have to remove a line, and hence line-drawing marks in the sorters can be removed, and we can compute zero weights only from uncovered rows and columns. Hence the checking in Phase II and Phase V can be removed, making the sorter start up a simple load procedure. Likewise the line removal wavefronts never occur in the dataflow.

Remark. This is an important point to realise so that cells are not overly complex.

3.3 Systolic line drawing

The method used in Phase VII to draw lines is to place a line through as many zeros as possible each time a line is drawn. We do this by maintaining a list of row and column weights for the number of zeros in each unmarked row and column, with a line being placed in the row or column with the maximum number of zeros over all the weights. These weights are then adjusted by removing the covered zeros, and the lists resorted. However can we be sure that we always draw the minimum number of lines?

Theorem 1

The systolic API array always constructs the minimum number of lines to cover all the zeros.

Proof

Case (*a*). Suppose we do not cover all the zeros, then on evaluating the new row and column weights and sorting, some row or column would be non-zero and we would be forced to draw another line. Only when all the row and column weights are zero are all the zeros covered.

Case (*b*). Suppose we drew more than the minimum number of lines. Let the minimum number of lines be $K_{\min}$ and suppose we drew K lines $K > K_{\min}$. Suppose for simplicity and w.l.o.g. that $K_{\min} + 1 = K$. Then a line was drawn that was redundant; in order to do this all the zeros had to be covered at the time of drawing, hence all the row and column weights were zero and we could not have drawn the line. This is a contradiction, hence $K = K_{\min}$ follows, as we always put a line through the maximum available zeros at the time.

After defining the algorithm we stated that once a line is drawn it is never erased. Why is this so? To answer this we prove the following theorem.

Theorem 2

Once a line is drawn on the tableau it will never be removed.

Proof

To update the cost matrix we perform one of the following computations.

(*a*) Add the minimum uncovered element to an element on an intersection of two lines.

(*b*) Subtract the minimum uncovered element from another uncovered element.

It follows that once a zero has had a single line drawn through it, it will never be made non-zero. While a zero on the intersection of two lines can be made non-zero, if the only zeros in a row or column are on intersections, a modification of the reduced-cost tableau could create a line on a row or column which contained no zeros and a line would have to be removed. However, a line with zeros only at intersections with other lines is redundant and would not be drawn by Theorem 1. It follows that once a line is drawn it is not removed.

Remark. Theorem 2 allows the timing of the systolic array to be calculated, and allows the suggested simplifications to the systolic array to be made.

3.3.1 'On-the-fly' construction of lines

The lines drawn on the reduced-cost matrix are represented by the line state of the tableau cells. The line state is used to identify covered elements of the cost matrix during the modification (Phase I; W_1) of the cost matrix. Essentially there are only three states for a tableau element: uncovered, no line; covered, with a single line; covered, by two lines, which is an intersection. The line state of each tableau element and hence processor can thus be represented by a line-state variable consisting of only two bits with the above states 00, 01 and 10 respectively. Line drawing then becomes trivial as the line state can be set by simply shifting the line-state variable left, and initially introducing a 1 for the first drawing. Thus a three-bit circulating shift register in the tableau cell is used to represent lines, according to the following states: 100, no line; 001, single line; 010, intersection. (Notice that if we had to remove a line we would need 'shift right' signals; here only a single shift direction suffices.) For purposes of swapping rows or columns during the sorting of row and column weights, we consider these three bits to be tagged to the actual tableau element. The lines are simply constructed by tying the control inputs to the cell to the shift register, so that they shift when the line-drawing control from the row or column sorter cell adjacent to controller C_3 is received. The tableau cells in the row and column adjacent to the row and column sorter cells, by the mechanics of the array are the only ones to receive the line-drawing commands, but as the lines are bubbled away from boundaries all the cells in the tableau must contain a line-state register to record lines for modification. As lines are marked on the tableau cells, cells which contain the zeros on the line must generate signals to modify the row/column weight in the adjacent cell. The tableau cell must therefore be able to detect a zero element, which is simply done by taking the NAND of the bits in the tableau element. It follows that the line drawing can be completed with only a 3-bit circulating shift register and a NAND gate, in each cell.

As the row or column is bubbled away from the row sort/column sort boundary, taking the now zero but

previously maximum weight with it, the list which is having its weights modified may need to be fully re-sorted. The round trip for the control signals generated from controller 3, to reach controllers C_4 and C_2 and back, is $2n$ which gives plenty of time for the lists to be re-sorted, and the line-drawing signals to move out of the immediate vicinity of the C_3 controller. It follows that we can draw lines at the rate of 1 every $2n+k$ cycles, where k is a small constant for synchronisation of the row and column sorters and the C_3 cell to decide on the next line position.

3.4 Complexity of the tableau element

We have already seen in the section on drawing lines that the tableau requires a three bit circulating shift register and a NAND gate to detect lines and generate signals to modify the weights when a zero is covered. The tableau cell also requires some additional hardware to perform the modification of the cost matrix, compute row and column weights in early phases and locate the minimum uncovered element used in the modification.

(i) *Modification of the cost matrix.* This is easily performed. We already have the line status in three bits, using two of these to decide if an element is covered by one or two lines, or not at all. We also have a signal which will indicate if the element is zero. Thus we can use these combined signals to indicate addition, subtraction, or no operation, as follows.

Line state	Zero	Action
00	X	Subtract minimum element from cell element
01	X	Add zero to tableau element
10	X	Add minimum element to cell element

X = don't care.

(ii) Row and column weights are also simply computed using the same signals.

Line state	Zero	Action
00	1	Add 1 to row or column index
01	X	Null (add zero)
10	X	Null (add zero)

Note that we can use line state and zero to generate the 1 to be added to total, in the cell.

(iii) The modification of the matrix requires the location of the minimum element; this in turn means that the tableau cell must shift the minimum element left. To find the minimum we compare the value coming from the right with the cell element and moving the minimum left. This is done by simply subtracting the cell element from the incoming element; if the result is negative the incoming value is less than the cell element, otherwise the cell element is the minimum and we send the correct value left. All this can be detected by noting that all the cost matrix variables are positive, so a negative result is unique in determining the minimum. Finally, the most significant bit generated by the adder/subtractor will be set high for a negative answer, hence a single control bit is sufficient for detecting the minimum.

It follows that all the operations required for the tableau element can be constructed using an adder/subtractor, together with some switching for inputs and the line state, zero and $\pm$ status bit. Thus the cycle time of the API array is the time for a single addition or subtraction and the necessary time for setting controls. The tableau cell is thus very simple in structure. By similar arguments we can also conclude that the row sort, column sort and Min-shift arrays can also be represented by an adder/subtractor arrangement with the same cycle time. The fact that all the basic cells are adder/subtractors with associated control logic is significant, because the algorithm appears to imply a basic cell with at least a comparator and adder/subtractor arrangement, so the above arrangement can save about half the area. We assume a comparator is approximately the same area as an adder.

Remark. We stated that each tableau cell reads a value from its right and passes the minimum of the cell element and the incoming value to the left. What happens on the right boundary, where tableau cells have row-sort cells to the right? We may think that a row weight may propagate through the tableau to become the erroneous minimum. In fact this is not possible, as we find the minimum only after drawing lines, hence if we seek the minimum element of the tableau all the row weights are zero. Also, the minimum uncovered element must be non-zero (line-drawing algorithm), hence incorrect results can easily be avoided by rejecting zero minima (see Fig. 4).

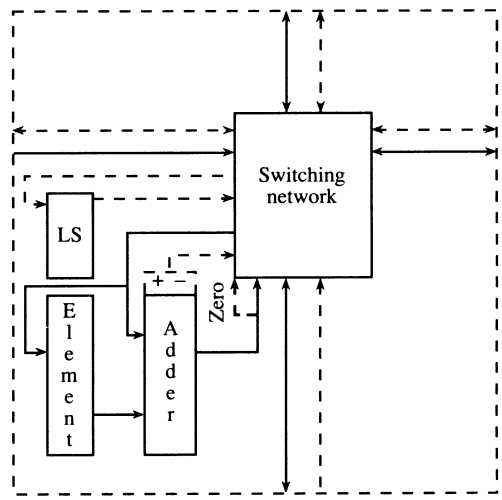

Figure 4. Tableau cell arrangement. LS, line state; ---, control; ——, data.

4. COMPUTATION TIME AND AREA CONSIDERATIONS

The use of the SCR in the API array means that the timing of the array is easy to compute. The control signal travels around the SCR exactly twice to complete the steps 2 and 3 of the algorithm; on the second cycle we perform line drawing, which is a variable time on each iteration of the matrix. If we call the total time spent in the line drawing phase T_{ld}, the time for a single API iteration is

$$T_i = 8n + \theta_i T_{ld} + 8, \quad 0 < \theta_i < 1,$$

as the time to traverse a quarter (or side) of the SCR is $n+1$ basic cell cycles and a single cycle around the SCR is $4(n+1)$.

Now if we perform z iterations the total time is

$$\sum_{i=1}^{z} T_i = \sum_{i=1}^{z} 8n + \sum_{i=1}^{z} \theta_i T_{ld} + 8z$$

$$= 8nz + \sum_{i=1}^{z} \theta_i T_{ld} + 8z.$$

The time taken to draw a single line systolically is also easily computed. It is the time taken to select the next row and column and then traverse a side of the API twice, i.e.

$T_l = 2(n+1)+k,$

where k = constant (non-zero, positive).

But if we stop when we have drawn n lines, and if, once a line is drawn, it is never removed, then

$$\sum_{i=1}^{z} \theta_i T_{ld} = nT_l = 2n^2+n+kn.$$

Thus, for z iterations, we have

$$T = 8nz+8z+2n^2+n+kn$$
$$= 2n^2+n(k+1+8z)+8z.$$

Hence if we perform the following iterations:

$$(z=1) \quad T_{ld} = 2n^2+(k+9)n+8$$

$$(z=n) \quad T_{up} = 10n^2+(k+9)n.$$

Hence the API computation time is bounded as given by the expression

$$2n^2+(9+k)n+8 \leqslant T \leqslant 10n^2+(9+k)n.$$

The time taken to pre-compute the initial reduced-cost matrix is $5n$ and, overlapping the last n with the loading of the API, we only require an extra $2n$ to complete the loading and performing unloading to give the total $7n$. Thus a bound for the complete algorithm is

$$2n^2+(16+k)n+8 \leqslant T \leqslant 10n^2+(16+k)n.$$

Notice that this timing is in terms of cycles which are as long as is required to perform an addition and some trivial control switching, so the systolic design should be fast.

In terms of area we already have n adder-type cells to produce the reduced-cost matrix in the pre-processing stage, while the API requires approximately $[(n+2)+(n+2)]$ added cells (including switching, registers, etc.), which is useful in terms of area efficiency. We also assume that the controllers are sufficiently simple to be bounded by the area of four adder arrangements, and it is trivial to notice that C_2 and C_4 satisfy sub-adder area, while C_1 and C_3 – the major controllers – are more complex. In any event, the controller area is not significantly more than the area for the adder-type cells, so our area estimation is essentially correct.

4.1 Wafer Scale Integration (WSI)

It is acknowledged that the assignment problem will often be used on large problem sizes (n), and thus the number of basic elements in the API is large and the resulting area for the problem may be large. In earlier sections it was noted that the pre-processing array could be linked directly to the Host interface section of the API array for each loading of the reduced cost matrix. This implies a high bandwidth for large n and a large number of pins for a chip-based implementation of the API array. However, we assumed that loading would be done in a row-by-row order in $O(n)$ time. The API algorithm is itself $O(n^2)$, so a more expensive loading scheme with smaller bandwidth can be justified, in particular we consider a method known as 'snake-loading' requiring $O(n^2)$ time which requires only a single input–output line, as is shown in Fig. 5.

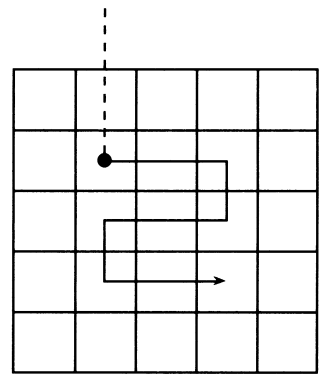

Figure 5. Snake loading using one I/O connection ($n = 3$).

Since we now only require a single I/O connection which is easily within pin restrictions, and the interface between API and Host is still local to C_1 the start controller, the possibility of producing a Wafer API (WAPI) Systolic Array which could have a very large number of cells permitting large assignment problems to be solved is feasible using this principle. Notice that the API is well suited to VLSI implementation due to the arrangement of different cell types, with a core of $n*n$ identical cells, and linear arrays of n cells each around the periphery of the design punctuated by simple controllers (see Fig. 6).

Remark. Notice that the wafer is sectioned into squares assumed roughly proportional to a single chip area. Hence using VLSI techniques each square itself could be a sizeable portion of the array. If we consider the block form of the reduced-cost tableau, then if each chip has enough cells for a $k*k$ grid, the wafer can implement a k block-partitioned reduced-cost matrix.

In our example there are 11^2 squares in the tableau portion of the API; if we assume a $4*4$ portion on each chip this implements a $44*44$ reduced-cost matrix. In all probability we will be able to increase the number of array cells in each wafer square. One final point about the wafer idea is that the row sorting, column sorting and other boundary cells absorb very little area, less than a single chip, hence we can increase the size of the tableau by using this extra space.

Notice that as only one input–output line has been used we may consider extending this to one line for each boundary to reduce the loading and unloading times.

Remark. We have ignored the problems of re-routeing if a chip fails; this can be solved by consulting the current literature on the subject but is beyond the scope of this paper.

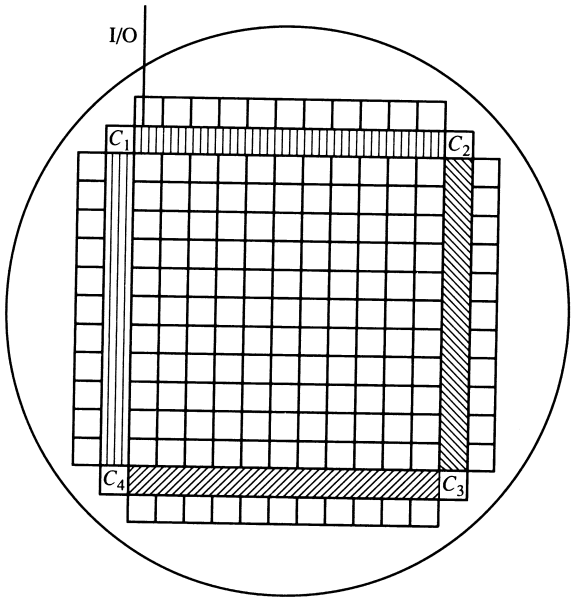

Figure 6. Mapping of API on to a wafer.

We can consider increasing the problem solved by adding some memory and control to the design. The principle is to develop a scheme similar to the LISA concept.[5] Here it would fabricate a processor with a single cell of the tableau array and simulate it with the control program built in as part of the new cell – a virtual grid of array processors storing the results in an auxiliary memory. This approach suffers from a reduction in speed of computation but could produce much larger grids (see Fig. 7).

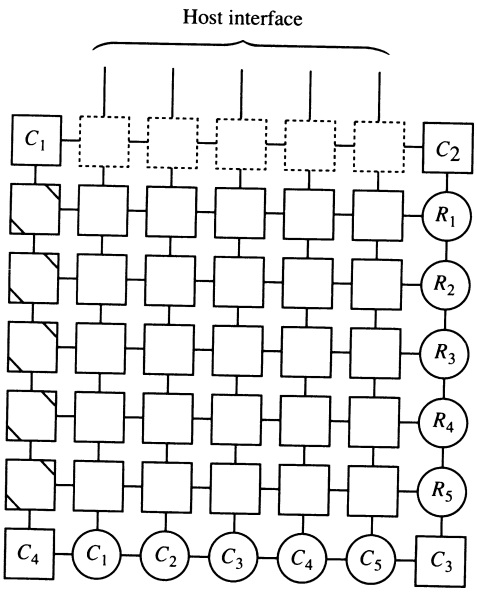

Figure 7. API systolic array ($n = 5$).

5. CONCLUSIONS

We have shown that the computational procedure known as the Hungarian method for solving the assignment problem can be implemented in a systolic manner.

The seemingly difficult problem of systolically drawing the minimal lines through zeros in the reduced-cost matrix and its later modifications proved simple to implement. We also introduced the notion of the systolic control ring (SCR) to control systolic movement within the systolic array, and discovered that the control exhibits more systolic properties than the actual data, although this could be the result of assigning one cost matrix element to each processor in the orthogonally connected tableau.

The time of the final algorithm is bounded as

$$2n^2+(16+k)n+8 \leqslant T \leqslant 10n^2+(16+k)n$$

and the design requires $O(n^2)$ basic cells of approximately equivalent area to an adder/subtractor with associated control and registers for recording line states of each variable table element. This is compared with $O(n^3)$ if the algorithm was implemented sequentially; however, a sequential machine would be able to take advantage of the smaller weight lists as lines are drawn, which the systolic array presented does not. The current array can however be modified by restructuring the control flow in the sorter cells to perform this.

We also showed that the design would be suitable for solving the large problems encountered in practice by using Wafer Scale Integration (WAPI) arrays, which would free any host (sequential machine in particular) for other tasks during the computation of the API algorithm. Smaller problems can be solved on a larger array by adding dummy jobs and individuals to pad up to full array size. The wafer form of the algorithm using a snake loading pattern requires a time bounded by

$$3n^2+(9+k)n+8 \leqslant T \leqslant 11n^2+(9+k)n.$$

Finally we note that a commonly used and often large problem can be solved simply and quickly using remarkably simple cells using systolic principles, especially if we use the wafer scale integration techniques currently under development.

REFERENCES

1. D. J. Evans and G. M. Megson, *A Systolic Implementation of the Simplex Algorithm*, Computer Studies Report 288, LUT. *Int. Jour. Comp. Math.* 1991 (in press).
2. G. M. Megson and D. J. Evans, *A Systolic Cylinder for the Revised Simplex Algorithm*, Computer Studies Report 304, LUT. *Int. Jour. Comp. Math.* 1991 (in press).
3. Nesa Wu & Richard Coppins, *Linear Programming and Extensions*. McGraw-Hill, Maidenhead (1981).
4. *Silicon Design* **3** (1) (1986).
5. G. M. Megson and D. J. Evans, Lisa: a parallel processing architecture. In *Conpar '86*, edited W. Handler *et al.*, pp. 361–375. Lecture Notes in Computer Science 237. Springer Verlag, Heidelberg.
6. R. P. Stallard, *OCCAM – The Loughborough Implementation*. Private communication (Nov. 1985).
7. G. M. Megson and D. J. Evans, *The Systolic Control Ring Instruction Processor (SCRIP)*. Int. Conf. on Parallel Processing for Computer Vision & Display, Leeds (12–15 Jan. 1988) *Integration, the VLSI journal.* 9 (1990) pp. 287–302.
8. D. J. Evans and G. M. Megson, Matrix inversion by systolic rank annihilation. *Int. Jour. Comp. Math.* **21**, 319–357 (1987).