JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 29, 84–90 (1995)

A Processor Array with Bounded I/O Ports for Computing Transitive Closures

MOKHTAR A. ABOELAZE*,[1] AND BENJAMIN W. WAH†,[2]

*Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3; and †Coordinated Science Laboratory, University of Illinois, Urbana, Illinois 61801

In this paper, we present the design of a processor array (PA) with bounded number of input/output (I/O) ports (L-by-N PEs with $O(L)$ I/O ports, where $1 \leq L \leq N$) for computing transitive closures. Previous designs of PAs to solve this problem use either N-by-N PAs with $O(N)$ I/O ports or linear PAs with $O(1)$ I/O ports. These designs are restricted by their I/O bandwidth because designs using $O(N)$ I/O ports are limited by the number of PEs that can be supported by these ports, whereas designs with $O(1)$ ports are limited by the I/O time to load and drain data. In contrast, our proposed design uses $O(L)$ I/O ports, where L is a technology-dependent design parameter. In our proposed architecture, we have used more powerful processing elements (PEs), each with a microprogram-driven ALU, a limited number of I/O buffers, and $O(N/L)$ memory words. Our technique for mapping transitive-closure computations consists of four steps: (1) Starting from a mapping of a transitive-closure problem on an N-by-N PA, we map the computations into an L-by-N PA. (2) We then schedule the computations in the L-by-N PA in order to satisfy the dependencies. (3) Based on the new mapping, we derive speeds and directions of data flows and the number of buffers required in each PE. (4) Finally, we derive the microprogram in each PE for correct execution. We show that our design specializes to efficient designs in the boundary cases when $L = 1$ (linear PA) and $L = N$ (square mesh). © 1995 Academic Press, Inc.

1. INTRODUCTION

Advances in VLSI technology have made it possible to design specialized VLSI arrays of processing elements to solve specific application problems. These arrays, known as *systolic arrays,* are algorithm-specific and can be used in real-time applications. They were first proposed by Kung and Leiserson [6] and consist of multiple simple *processing elements* (PEs) connected together in a tightly coupled one-dimensional or two-dimensional topology. In recent years, systolic arrays have been extended to use more powerful processing elements. We refer to these as *processor arrays* (PAs) in this paper.

PAs are generally limited by their input/output (I/O) bandwidth. In a square-mesh configuration, the number of I/O ports is related to the square root of the number of PEs, thereby limiting the maximum throughput for a fixed number of I/O ports. To overcome this problem, linear PAs have been developed [10–12]. Although these require a constant number of I/O ports, there is a bottleneck in bringing data in and out of these PAs, again limiting the maximum throughput.

In view of these limitations, we present in this paper the design of a rectangular (L-by-N) PA with L I/O ports, where L, $1 \leq L \leq N$, is a parameter selected by the designer. We illustrate the design method by mapping the Floyd–Warshall algorithm for computing transitive closures on a rectangular PA. This design can compute transitive closures with $O(N^3)$ complexity in $O(N^2/L)$ time. In the boundary cases, a linear PA ($L = 1$) can solve the problem in $O(N^2)$ time, and a square mesh ($L = N$) can solve the problem in $O(N)$ time. Parameter L, therefore, represents a design trade-off between computational throughput and I/O complexity.

In our design, we assume that each PE has a simple ALU that performs basic algebraic and Boolean operations, I/O buffers for exchanging data with other PEs, a memory unit with $O(N/L)$ words (or buffers), and a control unit containing microinstructions to control the actions of the ALU.

In mapping the Floyd–Warshall algorithm on a rectangular PA, we start with a feasible mapping of the algorithm on a (logical) square mesh, followed by mapping the computations performed in this logical PA into an L-by-N physical PA. The correctness of the mapping is enforced by not mapping two operations to the same PE at the same time and by preserving dependence relations. We then compute the speeds and directions of data propagation and the timing of different operations. Finally, we derive the microprogram that controls the PEs.

This paper is organized as follows. In Section 2, we define the PA model and the motivations behind our proposed architecture. We then present the transitive-closure prob-

[1] Research supported by the National Science and Engineering Council of Canada Grant NSERC-OGP0043688. E-mail: aboelaze@cs.yorku.ca.

[2] Research supported by Joint Services Electronics Program Contract N00014-90-J-1270 and National Science Foundation Grant MIP 92-18715. E-mail: b-wah@uiuc.edu.

0743-7315/95 $12.00

lem in Section 3 and our mapping technique in Section 4. In Section 5, we compute the speeds and directions of data movement and the microprogram for controlling the PEs. Conclusions are drawn in Section 6.

2. PROCESSOR-ARRAY MODEL

The PAs studied in this paper are special cases of *control flow systolic arrays* (*CFSAs*) we have studied earlier [3]. Each PE in our PA consists of the following components:

(1) *Arithmetic and Logic Unit* (*ALU*). The ALU is capable of performing simple arithmetic and logic operations. It can also send/receive data to/from local storage unit or I/O registers.

(2) *Control Unit.* This is either a microprogram memory holding microinstructions to be executed, or a simple decoder that decodes control signals traveling with the data. A microprogram is usually in the form of a loop that iterates until the transitive closure is computed. On the other hand, a hard-wired control unit decodes control bits traveling with the data, and issues control signals to the ALU to perform specific functions. Since microprograms are more flexible, we assume in this paper that each PE has a microprogrammed control unit.

(3) *I/O Registers.* Four I/O registers are used in each PE to hold data while being transmitted to or received from the four neighbors (north, south, east and west) of this PE.

(4) *Storage Unit.* This is memory for holding intermediate results in each PE. In computing transitive closures, $O(N/L)$ memory words are required. Although this is problem-size dependent and renders our design not modularly expandable, we can always include additional memory in each PE so that the resulting chip can be used to solve problems up to a maximum size. More efficient sharing can be done by allowing multiple PEs to use the same memory.

The motivations behind our architecture are two folds. First, the number of I/O ports per chip is limited despite the fact that a large number of components can now be implemented in a chip. Hence, it is essential to design architectures that fully balance the use of I/O ports and computational elements. Second, it is expensive to design small quantities of special-purpose chips. To amortize the cost of designing, testing, and fabricating chips, it is necessary to design PAs that can be used to solve more than one application problems.

Our proposed PA presents a viable solution to these two issues. First, it relieves the I/O bottleneck in a square-mesh or a linear PA by using a rectangular configuration, thereby allowing better utilization of the available I/O bandwidth. Second, the control in each PE is driven by a microprogram [1, 2], hence allowing the same PA to be used to solve more than one application problems. The design we have proposed in this paper can be implemented on PAs with grain sizes ranging from a large number of simple PEs [2, 9] to a small number of powerful PEs.

3. TRANSITIVE-CLOSURE PROBLEM

The transitive-closure problem has been studied extensively in the literature for two reasons. First, many problems in signal and image processing can be solved by transforming to the transitive-closure problem. These include finding all-pair shortest paths, matrix triangulations, and algebraic-path problems. Second, the Floyd–Warshall algorithm, the fastest algorithm to find transitive closures, has linear dependencies, making it more difficult as well as more interesting to map to planar PAs.

Previous attempts to compute transitive closures have focused on either a square mesh with $O(N)$ I/O ports and $O(N)$ time, or a linear PA with $O(1)$ I/O ports and $O(N^2)$ time. In computing transitive closures in two-dimensional PAs, Kung *et al.* first introduced an algorithm that needs $(5N - 5)$ time units in an N-by-N mesh with $2N$ I/O ports [7, 8]. Subsequently, Benaini and Robert proposed a PA for solving the algebraic-path problem in $(5N - 2)$ time units using $[N(N + 2)/3 + 1]$ PEs [4]. Almost simultaneously, Cappello and Scheiman introduced a similar PA [5]. Later, Rajopadhye introduced an N-by-N PA that solves the problem in $(4N - 2)$ time steps with wraparound connections, and in $(5N - 4)$ steps without wraparound connections [13].

In computing transitive closures in one-dimensional PAs, Varman and Ramakrishnan first introduced a linear PA that takes $(9N^2 + N - 2)$ time units in $(2N - 1)$ PEs, each requiring $O(N)$ area [15]. Kumar and Tsai proposed two linear PAs: the first takes $(7N^2 - 3N + 1)$ time units using N^2 PEs of $O(1)$ area each, and the second takes $(4N^2 - N + 1)$ time units using N PEs of $O(N)$ area each [11]. Subsequently, Sarkar and Mukherjee developed two PAs for computing transitive closures. The first takes $(N^2 + N - 1)$ time units (plus I/O time) and runs in a ring of N PEs; the second takes $(2N^2 + 2N - 1)$ time units and runs in a linear PA of N PEs [14]. In both cases, a PE takes $O(N)$ area. Recently, we have developed optimal linear PAs that take the minimum time for a given maximum number of PEs, or uses the minimum number of PEs for a given maximum completion time [16, 17].

In this paper, we study a more general L-by-N PA architecture with $O(L)$ I/O ports that can find transitive closures in $(N^2/L + 2N + 2L - 5)$ time units. Our design specializes to a square mesh when $L = N$ and takes $(5N - 5)$ time units. This is the fastest design for computing transitive closures in a square mesh [7]. On the other hand, when $L = 1$, our design takes $(N^2 - 3)$ time units if the matrix is already preloaded in the PA, and $(2N^2 + 2N - 3)$ time units if matrix A needs to be loaded.

3.1. The Floyd–Warshall Algorithm

We define a directed graph G as $G(V, E)$, where V is a set of vertices, and E is a set of directed edges in the graph. The adjacency matrix of G is defined as A, where $a_{i,j} = 1$ if there is a directed edge from i to j. The *transitive closure*

of $G(V, E)$ is defined as $G^+(V, E^+)$, where E^+ represents the set of edges in G^+, and an edge between u and v in G^+ exists iff there is a path of length zero or more between u and v in G. In this case, the *adjacency matrix* of G^+ is A^+ where $a^+_{i,j} = 1$ iff there is a path between i and j in G.

The Floyd–Warshall algorithm is a fast and efficient sequential algorithm for finding transitive closures. The algorithm uses A as input and generates A^+ as output. It works as follows:

```
for k = 0 to N − 1 do
    for i = 0 to N − 1 do
        for j = 0 to N − 1 do
            a^k_{i,j} = a^{k-1}_{i,j} + a^{k-1}_{i,k} * a^{k-1}_{k,j}
        end
    end
end
```

The above algorithm has nonuniform dependencies, implying that a direct mapping to a planar PA will result in communications between nonneighboring PEs.

To eliminate nonuniform dependencies, one way is to pipeline data requiring nonadjacent communications along existing paths of a planar PA until they reach the required destinations. An example of pipelining computations with nonuniform dependencies is shown in Fig. 1. In this example, in computing $a^k_{i,j}$ in the kth iteration, $a^{k-1}_{i,k}$ and $a^{k-1}_{k,j}$ are needed. This means that during the kth iteration, results of the kth row and the kth column (marked by crossed-out circles) must be sent in a pipelined fashion to all PEs in the kth row and the kth column. The algorithm completes when data computed at location (k, k) have propagated to the four corners of the PA.

Kung *et al.* have applied pipelining to map linear dependence graphs on N-by-N PAs [7, 8]. In their design, a PE at location (k, k) computes $a^k_{k,k}$ in the kth iteration and propagates the result of the previous iteration towards the four corners of the PA, where they will be used to calculate $a^k_{i,j}$.

In this paper, we call element $a^{k-1}_{k,k}$ the *diagonal* element in the kth iteration, and the PE at which $a^{k-1}_{k,k}$ is computed, the *diagonal PE*. We also call $a^{k-1}_{i,k}$ the *pivot column*, e the PEs at which they are computed, the *pivot-col PEs*. Similarly, we call $a^{k-1}_{k,j}$ the *pivot row*, and the PE which they are computed, the *pivot-row PEs*.

4. MAPPING FLOYD–WARSHALL ALGORITHM INT A CONSTANT I/O PROCESSOR ARRAY

In this section, we present our scheme for mapping pipelined dependence graph as exemplified in Fig. 1 in an L-by-N PA. The idea is to use each PE in the rectangul PA to perform computations performed by s PEs in th square PA, where $s = \lceil N/L \rceil$.

We assume that the mapping function $\mathcal{Z}$ from computa tions to PEs is

$$\mathcal{Z}(i, j, k) = (\alpha, \beta), \tag{1}$$

where α and β are the coordinates of the PE that performs computation at $\langle i, j, k \rangle$. In this paper, we chose $\mathcal{Z}$ as

$$\mathcal{Z}(i, j, k) = \left(\left\lfloor \frac{i}{s} \right\rfloor, j \right). \tag{2}$$

Figure 2 shows the computation of a 6-by-6 transitive closure in a 2-by-6 PA. Consider operations performed in the 4th iteration that starts at time t_4.

time (t_4)
: $PE_{1,4}$ computes $a^4_{4,4}$ and send $a^3_{4,4}$ to its four neighbors. (Only three neighbors are shown in Fig. 2.)

time ($t_4 + 1$)
: $PE_{1,4}$ starts computing $a^4_{5,4}$, sends $a^3_{5,4}$ east and west, and $a^3_{4,4}$ north.
$PE_{0,4}$ receives $a^3_{4,4}$, starts computing $a^4_{1,4}$, and sends $a^3_{1,4}$ east and west.
$PE_{1,5}$ receives $a^3_{4,4}$, starts computing $a^4_{4,5}$, and sends $a^3_{4,5}$ north and south.
$PE_{1,3}$ receives $a^3_{4,4}$, starts computing $a^4_{4,3}$, sends $a^3_{4,4}$ east to $PE_{1,2}$, and $a^3_{4,3}$ north to $PE_{0,3}$.

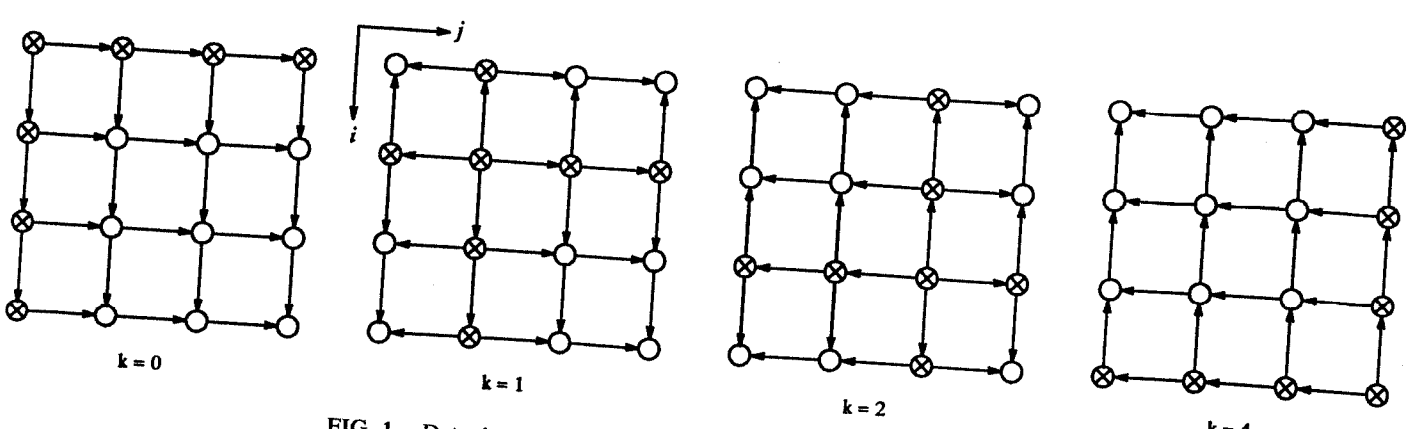

FIG. 1. Data dependencies in the Floyd–Warshall algorithm after pipelining.

time $(t_4 + 2)$ $PE_{1,4}$ starts computing $a^4_{3,4}$, sends $a^3_{3,4}$ east and west, and $a^3_{4,4}$ north.
$PE_{0,4}$ receives $a^3_{4,4}$, starts computing $a^4_{2,4}$, and sends $a^3_{2,4}$ east and west.
$PE_{1,5}$ receives $a^3_{5,4}$, starts computing $a^4_{5,5}$, and sends $a^3_{4,5}$ north.
$PE_{1,3}$ receives $a^3_{5,4}$, starts computing $a^4_{5,3}$, sends $a^3_{5,4}$ east, and $a^3_{4,3}$ north.
...

Although the mapping in Eq. (2) defines the computations performed in each PE, we still need to schedule the computations so that the following constraints are satisfied. (a) No two computations are scheduled at the same time in the same PE. (b) Data dependencies in a scheduled computation are satisfied, implying that all inputs needed by this computation have been generated, and that they have already propagated from where they were generated to where they are needed.

In scheduling the computations in each PE, we schedule only the first computation of each iteration. Since each PE performs s operations per iteration, a PE is considered busy for s time units after it starts computing. Let us denote by $P(i, j, k)$ the time at which $PE_{i,j}$ starts the kth iteration, and by $T(i, j, k)$ the time at which $a^k_{i,j}$ is computed.

Figure 3 shows the operations in the kth iteration. It starts by computing $a^k_{k,k}$ in $PE_{\lfloor k/s \rfloor,k}$. This is followed by computations in the kth row and the kth column and the rest of the elements.

(1) The first iteration can be scheduled as soon as data propagate from the first row and the first column to the rest of the PA; in this case, $P(i, j, 0) = i + j$.

(2) In scheduling computations in a diagonal PE (the intersection of the kth row and the kth column), we need

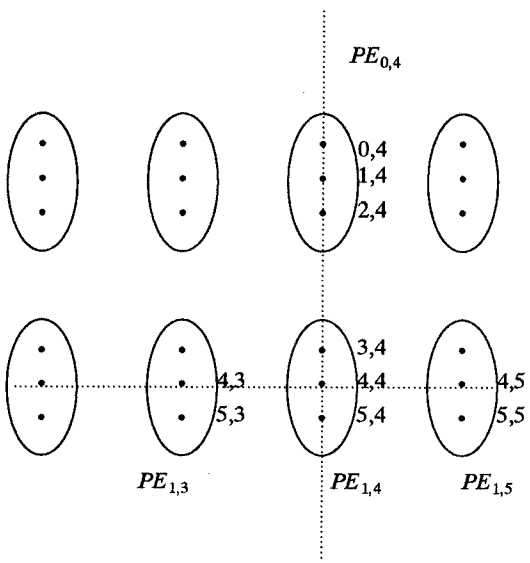

FIG. 2. Iteration 4 in computing a 6-by-6 transitive closure on a 2-by-6 PA.

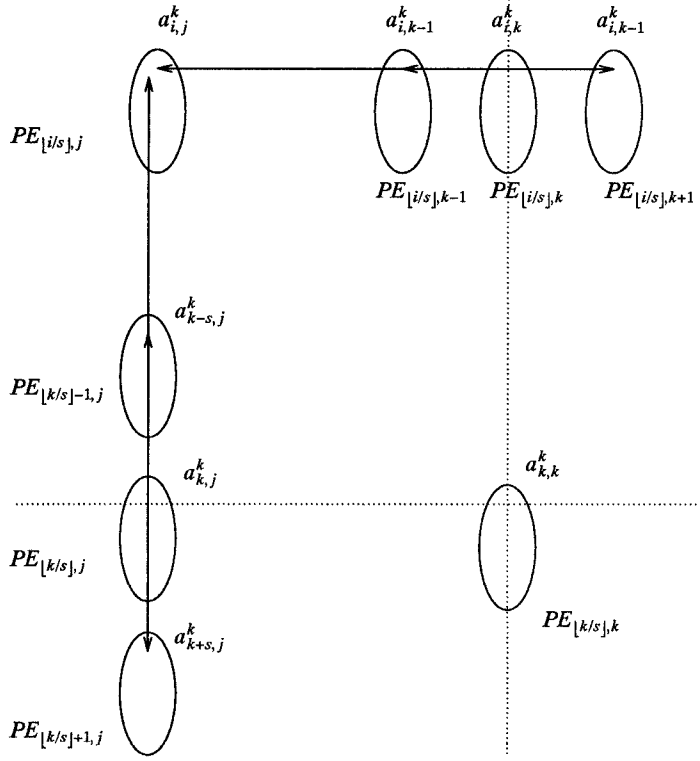

FIG. 3. Computations in the kth iteration.

to wait until the completion of the previous iteration by the same PE. In this case $P(i, j, k) = P(i, j, k - 1) + s$.

(3) The kth iteration for $PE_{i,j}$ can start after data elements needed by this PE have been generated in $PE_{i,k}$ and $PE_{\lfloor k/s \rfloor,j}$ and have propagated to $PE_{i,j}$; *i.e.*, $P(i, j, k) = \max\{P(i, k, k) + |k - j|, P(\lfloor k/s \rfloor, j, k) + |\lfloor k/s \rfloor - i|\}$. Combining the boundary cases, we have

$$P(i, j, k) = \begin{cases} i + j, & k = 0, \\ P(i, j, k - 1) + s, & j = k, i = \lfloor k/s \rfloor, \\ \max\{P(i, k, k) + |k - j|, \\ \quad P(\lfloor k/s \rfloor, j, k) + |\lfloor k/s \rfloor - i|\}, & \text{otherwise.} \end{cases} \tag{3}$$

Solving this recurrence, we get

$$P(i, j, k) = (s + 1)k + \lfloor k/s \rfloor + |i - \lfloor k/s \rfloor| + |k - j|. \tag{4}$$

Since every PE performs s computations per iteration, a sufficient condition to check that no two operations are scheduled at the same time in the same PE is

$$P(i, j, k) - P(i, j, k - 1) \geq s. \tag{5}$$

It is easy to show that the above condition is satisfied in Eq. (4). Another point to consider is that Eq. (4) only

guarantees correct execution of the first element in each PE in any iteration, but says nothing about the remaining $(s-1)$ elements. This does not violate any dependence relation since we assume that in iteration k, each PE starts with the $(k \bmod s)$th element (denoted as $(k)_s$) and proceeds to the $(k+1)_s$th element until the $(k-1)_s$th element has been generated. Hence, by adding a constant α, $0 \le \alpha < s$, to the lefthand and righthand sides of Eq. (4), the correct execution of the remaining $(s-1)$ elements in each PE is guaranteed.

As the last operation performed is the $(N-1)$th iteration in $PE_{0,0}$, the time taken to find a transitive closure is

$$T = P(0,0,N-1) + s - 1 = Ns + 2N + \frac{2N}{s} - 5. \quad (6)$$

Note that when $s = 1$, $T = 5N - 5$, which is the time taken by the mesh-structured PA proposed by Kung *et al* [7]. In the other extreme, when $s = N$ (for linear PAs), $T = N^2 + 2N - 3$, which is faster (by an additive constant) than the linear PA proposed by Sarkar and Mukherjee [14].

The mapping function shows that $PE_{i,j}$ performs operations at index $\{\lfloor i/s \rfloor + \alpha, j, k\}$, where $0 \le \alpha < s$ and $1 \le K \le N$. In iteration k, each PE performs s operations starting with the $(k)_s$th operation; i.e.,

$$T(i,j,k) = P(\lfloor i/s \rfloor, j, k) + (i-k)_s. \quad (7)$$

5. MICROPROGRAM GENERATION

In the previous section, we have described the scheduling of computations in such a way that no two computations are scheduled in the same PE at the same time, and that all dependence relations are preserved. Recall that a PE is controlled by a microprogram in its control unit. In this section, we introduce parameters that help generate the microprogram for realizing the proposed schedule.

The operations of the different PEs are as follows. During the kth iterations, PEs in the $\lfloor k/s \rfloor$th row and the kth column send the contents of the pivot elements to the rest of the PA and perform certain operations. PEs outside the $\lfloor k/s \rfloor$th row and the kth column receive a pivot element, perform some computations, and output the pivot element to the next PEs. We define parameter δ to be the time between performing two consecutive operations in the same PE in the same iteration. That is,

$$\delta = T(i+1,j,k) - T(i,j,k), \quad \text{where } \alpha s \le i < (\alpha+1)s, 0 \le \alpha < \lceil N/s \rceil. \quad (8)$$

By definition, $\delta = 1$. This means that once a PE starts the kth iteration, it performs s consecutive operations without waiting.

We further define τ_k to be the difference between starting two consecutive iterations in $PE_{i,j}$:

$$\begin{aligned}\tau_k(i,j,k) &= P(i,j,k+1) - P(i,j,k) \\ &= s + 1 + \lfloor (k+1)/s \rfloor - \lfloor k/s \rfloor \\ &\quad + |i - \lfloor (k+1)/s \rfloor| - |i - \lfloor k/s \rfloor| \\ &\quad + |k+1-j| - |k-j|. \end{aligned} \quad (9)$$

This is the difference in time between starting the kth iteration and the $(k+1)$st iteration. Since each iteration takes s time units, each PE idles for $[\tau_k(i,j,k) - s]$ time units after finishing the kth iteration.

By applying Eq. (9), we can compute $\tau_k(i,j,k)$ under various conditions that are enumerated in Table I. Note that since each PE performs s operations in an iteration, the time that any PE has to wait after finishing the kth operation and before starting the $(k+1)$th operation is 0, 2, or 4 time units ($\tau_k(i,j,k)$ in Table I).

To find the horizontal and vertical data movements, we define four additional parameters.

$\tau_i(i,j,k)$ This is the difference in time between generating a pivot-row element in the $(k-1)$th iteration and using it in the kth iteration. It can be shown to be the same as $\tau_k(\lfloor k/s \rfloor, j, k-1) - 1$. The term means that PEs in the kth row in the kth iteration use as a pivot the pivot-row element generated in the previous iteration. (A difference of one between τ_i and τ_k is due to element $(i-k)_s$ in Eq. (7).) This term also indicates that each PE should have at least s memory locations for storing the s elements it generated in an iteration.

$\tau_j(i,j,k)$ This is the difference in time between generating a pivot-column element in the $(k-1)$th iteration and using it in the kth iteration. It can be shown to be the same as $\tau_k(\lfloor i/s \rfloor, k, k-1) + 1$. The term means that PEs in the kth column in the kth iteration use as a pivot the pivot-column element generated in the previous iteration. (A difference of one between τ_j and τ_k is due to element $(i-k)_s$ in Eq. (7).) This term indicates that each PE should have at least s memory locations for storing the s elements it generated in an iteration.

$\tau_i'(i,j,k)$ This is the difference in time between using a pivot element in two neighboring PEs in direc-

TABLE I
$\tau_k(i,j,k)$ under Various Conditions in Eq. (9)

Conditions			$\tau_k(i,j,k)$
$k \ge j$	$(k)_s = s-1$	$i \ge \lfloor (k+1)/s \rfloor$	$s+2$
$k \ge j$	$(k)_s = s-1$	$i < \lfloor (k+1)/s \rfloor$	$s+4$
$k \ge j$	$(k)_s \ne s-1$	$i \ge \lfloor (k+1)/s \rfloor$	$s+2$
$k \ge j$	$(k)_s \ne s-1$	$i < \lfloor (k+1)/s \rfloor$	$s+2$
$k < j$	$(k)_s = s-1$	$i \ge \lfloor (k+1)/s \rfloor$	s
$k < j$	$(k)_s = s-1$	$i < \lfloor (k+1)/s \rfloor$	$s+2$
$k < j$	$(k)_s \ne s-1$	$i \ge \lfloor (k+1)/s \rfloor$	s
$k < j$	$(k)_s \ne s-1$	$i < \lfloor (k+1)/s \rfloor$	s

tion i (rows). Since the speed of propagation is one, each PE uses the pivot element and forwards it to the next PE in direction i. The term can be positive or negative, indicating that a pivot element can propagate in one of two directions.

$\tau'_j(i,j,k)$ This is the difference in time between using a pivot element in two neighboring PEs in direction j (columns). Since the speed of propagation is one, each PE uses the pivot element and forwards it to the next PE in direction j. The term can be positive or negative, indicating that a pivot element can propagate in one of two directions.

These definitions translate into the following equations:

$$\tau_i(k,j,k) = T(k,j,k) - T(k,j,k-1) \tag{10}$$

$$\tau_j(i,k,k) = T(i,k,k) - T(i,k,k-1) \tag{11}$$

$$\tau'_i(i,j,k) = T(i \pm s, j, k) - T(i,j,k) \tag{12}$$

$$\tau'_j(i,j,k) = T(i, j \pm 1, k) - T(i,j,k). \tag{13}$$

Solving these equations, we get

$$\tau_i(k,j,k) = \tau_k(\lfloor k/s \rfloor, j, k-1) + 1 \tag{14}$$

$$\tau_j(i,k,k) = \tau_k(\lfloor i/s \rfloor, k, k-1) - 1 \tag{15}$$

$$\tau_i(i,j,k) = \begin{cases} -1, & k > i, \\ 1, & \mathrm{k} < i, \end{cases} \tag{16}$$

$$\tau_j(i,j,k) = \begin{cases} -1, & k > j, \\ 1, & k < j. \end{cases} \tag{17}$$

By knowing parameters τ_k, τ_i, τ_j, τ'_i, τ'_j and δ, we can generate a microprogram to control the PEs. The program in shown in Fig. 4. It assumes that every PE knows its position in the PA (i, j). Each PE starts by waiting for $(i + j)$ time units. According to its position in the PA, it performs the following operations in an iteration. A diagonal PE computes the pivot elements using data from the previous iteration, and sends it to PEs in its row and column. Each of the nondiagonal PEs receives a pivot element, outputs the pivot element if it is in the main row or column (otherwise, transmits the pivot received to the next PE), and computes the required elements. At the end of the iteration, each PE waits for $(\tau(i, j, k) - s)$ time units before starting the next iteration.

```
wait(i + j)                          /* first iteration */
for k = 0 to N - 1                   /* Each PE knows its location in the PA (i, j) */
begin
    CASE ((k = j) & (i = ⌊k/s⌋))     /* Diagonal PEs */
        begin
        R_N = R_S = a ← (MEM[k])_s
        for ℓ = 0 to s - 1
            begin
            R_W = R_E = b ← (MEM[k + ℓ])_s
            (MEM[k + ℓ])_s ← (MEM[k + ℓ])_s + a * b
            end
        end
    CASE ((k = j) & (i ≠ ⌊k/s⌋))     /* Pivot-Column PEs */
        begin
        for ℓ = 0 to s - 1
            begin
            if (i > ⌊k/s⌋) then a ← R_N else a ← R_S
            if (i > ⌊k/s⌋) then a → R_S else a → R_N
            R_W = R_E = b ← (MEM[k + ℓ])_s
            (MEM[k + ℓ])_s ← (MEM[k + ℓ])_s + a * b
            end
        end
    CASE ((k ≠ j) & (k = ⌊i/s⌋))     /* Pivot-Row PEs */
        begin
        R_N = R_S = b ← (MEM[k])_s
        for ℓ = 0 to s - 1
            begin
            if (j > k) then a ← R_W else a ← R_E
            if (j > k) then a → R_E else a → R_W
            R_S = R_N = b ← (MEM[k])_s
            (MEM[k + ℓ])_s ← (MEM[k + ℓ])_s + a * b
            end
        end
    CASE ((k ≠ ⌊i/s⌋) & (k ≠ j))     /* Other PEs */
        begin
        for ℓ = 0 to s - 1
            begin
            if (i > ⌊k/s⌋) then a ← R_N else a ← R_S
            if (j > k) then b ← R_W else b ← R_E
            if (i > ⌊k/s⌋) then a → R_S else a → R_N
            if (j > k) then b → R_E else b → R_W
            (MEM[k + ℓ])_s ← (MEM[k + ℓ])_s + a * b
            end
        end
    wait(τ_k(i, j, k) - s)           /* wait to start the next iteration */
end
```

FIG. 4. Microprogram in each PE to compute transitive closures. (Note that the same program is executed by all the PEs; however, the action taken by each PE depends on the location of this PE in the PA. Further, note that $(x)_s$ means x mod s, and that the four I/O registers are R_N, R_S, R_E, and R_W.)

6. CONCLUSION

In this paper, we have proposed the design of a processor array with $O(L)$ IO ports for computing transitive closures, where L is a technology-dependent parameter that is related to the number of IO ports in a chip. Our mapping method starts with an N-by-N PA and maps the computations to an L-by-N PA. Based on the mapping function, we compute the directions and speeds of data. Finally, we show a microprogram for controlling the PEs. Our future work will focus on extending the optimal designs we have developed earlier [16, 17] to the I/O-limited processor arrays studied in this paper.

REFERENCES

1. Aboelaze, M. A., Lee, D.-L., and Wah, B. W. VLSI systolic arrays with constant I/O pins. In Y. Robert and P. Quinton (Eds.). *Algorithms and*

Architectures for VLSI. North-Holland, Amsterdam, 1992, pp. 205–210.

2. Aboelaze, M. A., Lee, D.-L., and Wah, B. W. Two-dimensional digital filtering using a constant I/O systolic array. *Proc. Int'l Symposium on Circuits and Systems,* Chicago, IL, May 1993, pp. 255–258.
3. Aboelaze, M. A. Systematic design of computational arrays, Ph.D. Thesis, Purdue University, West Lafayette, IN, 1988.
4. Benaini, A., and Robert, Y. Space-time minimal systolic arrays for gaussian elimination and the algebraic path problem. *Parallel Comput.* **15** (1990), 211–225.
5. Cappello, P. R., and Scheiman, C. J. A processor-time minimal systolic array for transitive closure. *Proc. Int'l Conf. on Application Specific Array Processors,* 1990, pp. 19–30.
6. Kung, H. T., and Leiserson, C. E. Highly concurrent systems introduction to VLSI System. In C. A. Mead and L. A. Conway (Eds.). *Introduction to VLSI Systems.* Addison–Wesley, Reading, MA, 1980.
7. Kung, S. Y., Lo, S. C., and Lewis, P. S. Optimal systolic design form the transitive closure and the shortest path problems. *IEEE Trans. Comput.* **C-36** (May 1987), 603–614.
8. Kung, S. Y. *VLSI Array Processors.* Prentice–Hall, Englewood Cliffs, NJ, 1988.
9. Lin, W. T., Yuan, C., and Ho, C. Y. Integrating systolic arrays into a supersystem. *IEEE Comput.* **20** (July 1987), 100–101.
10. Prasanna Kumar, V. K., and Tsai, Y. C. On mapping algorithms to linear and fault tolerant systolic arrays. *Proc. Int'l Conf. on Systolic Arrays,* 1988.
11. Prasanna Kumar, V. K., and Tsai, Y. C. Designing linear systolic arrays. *J. Parallel Distrib. Comput.* **7,** 3 (1989), 441–463.
12. Ramakrishnan, I. V., and Varman, P. J. Modular matrix multiplication on a linear array. *Proc. 11th Annual Symposium on Computer Architecture,* June 1984, pp. 232–238.
13. Rajopadhye, S. An improved systolic algorithm for the algebraic path problem. In Y. Robert and P. Quinton (Eds.). *Algorithms and Architectures for VLSI.* North-Holland, Amsterdam, 1992, pp. 187–198.
14. Sarkar, D., and Mukherjee, A. Design of optimal systolic algorithms for the transitive closure problem. *IEEE Trans. Comput.* **41,** 4 (Apr. 1992), 508–512.
15. Varman, P. J., and Ramakrishnan, I. V. Dynamic programming and transitive closure on linear pipelines. *Proc. Int'l Conf. on Parallel Processing,* 1984, pp. 359–364.
16. Ganapathy, K., and Wah, B. W. Synthesizing optimal lower dimensional processor arrays. *Proc. 1992 Int'l Conf. on Parallel Processing,* Vol. III, pp. 96–103.
17. Ganapathy, K., and Wah, B. W. Optimal synthesis of algorithm-specific lower-dimensional processor arrays. *IEEE Trans. Parallel Distrib. Systems,* to appear.

Received July 15, 1992; revised May 16, 1994; accepted July 28, 1994

BENJAMIN W. WAH received his Ph.D. degree in computer science from the University of California, Berkeley, California, in 1979. He is currently a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana–Champaign, Urbana, Illinois. Previously, he had served on the faculty of Purdue University (1979–85), as a program director at the National Science Foundation (1988–89), as Fujitsu Visiting Chair Professor of Intelligence Engineering, University of Tokyo (1992), and McKay Visiting Professor of Electrical Engineering and Computer Science, University of California, Berkeley (1994). In 1989, he was awarded a University Scholar of the University of Illinois. His current research interests are in the areas of parallel and distributed processing, knowledge engineering, and optimization. Dr. Wah is editor-in-chief of the IEEE Transactions on Knowledge and Data Engineering, and serves on the editorial boards of several other journals. He is a fellow of the IEEE.