The graphs support the earlier claims about row/column replacement. For a 256K array, the addition of 12 redundant rows and 12 redundant columns provide a yield in excess of 99 percent for $p \leqslant 10^{-4}$. However, for the 16M array, the addition of 40 extra rows and 40 extra columns produces a negligible yield for $p > 2.7 \times 10^{-5}$.

III. Conclusion

In this correspondence we show how hard defects can corrupt a random access memory and describe the current technique for controlling these defects during manufacture: row/column replacement.

Row/column replacement is asymptotically ineffective as a means of controlling hard defects. As we let the memory array grow unbounded in each direction, the probability of reducing the fraction of defects in the array goes to zero regardless of the fraction of extra rows and columns available for spare switching.

Finally, the asymptotic failure described above may become a significant limitation for very large memory arrays.

IV. Future Considerations

As random access memories get larger, a more effective means of controlling hard defects must be incorporated into their design. One obvious method is the inclusion of on-chip error correction. From Shannon's theory we know that for any $R < C(p)$ there exists a (n_c, k_c) code with $k_c/n_c \geqslant R$ such that the probability of a decoding error can be made arbitrarily small. (Here, $C(p) = 1 - h(p)$ where p is the probability of a defective cell; if the location of the defects are made available to the encoder or decoder, then $C(p) = 1 - p$ [9].) Such a code could be implemented on the rows of a RAM and provide a high degree of protection. This contrasts vividly with the "zero yield" which row/column replacement offers for large RAM's.

Currently, on-chip error correction is being increasingly considered as a means of providing protection from so-called "soft errors" [10]–[13]. These errors are transient in that they can be "scrubbed" from the system by rewriting the contents of the affected memory cells. The advisability of using on-chip ECC's to control *both* hard and soft errors is something which should be considered.

Acknowledgment

The authors thank T. Berger for his helpful suggestions on the asymptotic results.

References

[1] T. Mano *et al.*, "A redundancy circuit for a fault-tolerant 256K MOS RAM," *IEEE J. Solid State Circuits,* vol. SC-17, pp. 726–730, Aug. 1982.

[2] S. E. Schuster, "Multiple word/bit line redundancy for semiconductor memories," *IEEE J. Solid State Circuits,* vol. SC-13, pp. 698–703, Oct. 1978.

[3] R. P. Cenker *et al.*, "A fault tolerant 64K dynamic random access memory," *IEEE Trans. Electron. Dev.*, vol. ED-26, pp. 853–860, June 1979.

[4] R. T. Smith, "Laser programmable redundancy and yield improvement in a 64K DRAM," *IEEE J. Solid State Circuits,* vol. SC-16, pp. 506–513, Oct. 1981.

[5] T. E. Mangir, "Sources of failures and yield improvement for VLSI and restructurable interconnects for RVLSI and WSI: Part I—Sources of failures and yield improvement for VLSI," *Proc. IEEE,* vol. 72, pp. 690–708, June 1984.

[6] C. H. Stapper *et al.*, "Yield model for productivity optimization of VLSI memory chips with redundancy and partially good product," *IBM J. Res. Develop.*, vol. 24, pp. 398–409, May 1980.

[7] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes.* Amsterdam, The Netherlands: North Holland, 1977.

[8] I. Csiszar and J. Korner, *Information Theory: Coding Theorems for Discrete Memoryless Systems.* New York: Academic, 1981.

[9] C. Heegard and A. A. El Gamal, "On the capacity of computer memory with defects," *IEEE Trans. Inform. Theory,* vol. IT-29, pp. 731–739, Sept. 1983.

[10] T. C. May and M. H. Woods, "Alpha particle induced soft errors in dynamic memories," *IEEE Trans. Electron. Dev.*, vol. ED-26, pp. 2–9, Jan. 1979.

[11] F. I. Osman, "Error-correction technique for random access memories," *IEEE J. Solid State Circuits,* vol. SC-17, pp. 877–881, Oct. 1982.

[12] T. Mano *et al.*, "Circuit techniques for a VLSI memory," *IEEE J. Solid State Circuits,* vol. SC-18, pp. 463–469, Oct. 1983.

[13] J. Yamada *et al.*, "A submicron 1 Mbit dynamic RAM with a 4-Bit-at-a-time built-in ECC circuit," *IEEE J. Solid State Circuits,* vol. SC-19, pp. 627–633, Oct. 1984.

A Parallel Algorithm to Compute the Shortest Paths and Diameter of a Graph and Its VLSI Implementation

BHABANI P. SINHA, BHARGAB B. BHATTACHARYA, SURANJAN GHOSE, AND PRADIP K. SRIMANI

***Abstract*—In this correspondence we develop a parallel algorithm to compute the all-pairs shortest paths and the diameter of a given graph. Next, this algorithm is mapped into a suitable VLSI systolic architecture and the performance of this proposed VLSI implementation is evaluated.**

Index Terms—**Diameter, parallel algorithms, pipelining, shortest paths, VLSI architecture.**

I. Introduction

Enumeration of shortest paths between all pairs of vertices and finding the diameter of a graph constitute an important problem in graph theory and have many practical applications involving some commodity flow, e.g., in a computer communication network. In a communication network, the diameter of the network graph is a deciding factor in choosing the system topology which defines the interprocessor communication architecture. Further, a knowledge of the shortest paths between every two processing nodes in a network is essential to determine dynamically the optimal feasible route from one processor to the other in order to minimize the communication delay.

Various algorithms [2]–[5] exist for this shortest path problem; they are sequential in nature, and the time complexity of the best known algorithm of this class to compute all-pairs shortest distances is $O(n^{5/2})$ [5] while that of all-pairs shortest paths is $O(n^3)$ where n is the number of vertices.

The availability of low-cost, high-speed processor arrays during the last decade gave an impetus for parallelization of programs [12]. With the steep decrease in hardware cost due to the recent VLSI technology, there is a growing trend toward parallelization of different existing algorithms and their VLSI implementation [7]–[10], [13]–[16] to improve upon the execution time at the cost of providing a larger number of processors. Guibas, Kung, and Thompson [9] have given algorithms for dynamic programming and transitive closure problems suitable for VLSI implementation and their ideas can be readily extended to solve the shortest path problem as well. In this correspondence we follow a different approach to design a

Manuscript received September 17, 1984; revised August 9, 1985.

B. P. Sinha and S. Ghose are with the Electronics Unit, Indian Statistical Institute, Calcutta, 700 035, India.

B. B. Bhattacharya is with the Department of Computer Science, University of Nebraska, Lincoln, NE 68588.

P. K. Srimani is with the Department of Computer Science, Southern Illinois University, Carbondale, IL 62901, on leave from the Indian Institute of Management, Calcutta, India.

IEEE Log Number 8610487.

parallel algorithm to compute the shortest paths between every pair of vertices, and subsequently find the diameter of a graph. The algorithm has then been mapped on a suitable VLSI architecture. The results we have obtained regarding the number of processors and computation time are, however, similar to those in [9].

The design of any algorithm for VLSI implementation is often guided by the requirement of minimum interprocessor data communication time in a VLSI hardware using identical processor cells [9], [11] so as to reduce both the cost of production and signal propagation delay. A formal approach to design an algorithm suitable for VLSI circuit and mapping that algorithm into an appropriate VLSI architecture has been described by Moldovan [1]. The method consists of the following steps.

i) The algorithm is developed in a form in which all variables are pipelined.

ii) The data dependence vectors [1] are then found.

iii) A suitable linear transformation of these data dependence vectors is identified to select an appropriate VLSI systolic array.

Following the approach in [1], we first describe a parallel algorithm for the shortest path problem with all variables suitably pipelined. This algorithm can be executed in $O(n)$ parallel steps, each step requiring $O(n)$ time of computation. We have indicated that each of these parallel steps can also be executed in $O(\log n)$ time so that the time complexity of the proposed parallel algorithm is $O(n \log n)$. Next we map this algorithm in one of many possible VLSI systolic architectures using n^2 processors which, incidentally, turns out to be similar to Illiac IV architecture [6]. By choosing a different linear transformation of data dependence vectors we could get a different systolic architecture. This is in contrast to the approach of Guibas, Kung, and Thompson [9] where the algorithms were developed on a previously fixed interconnection pattern among the processor cells.

II. Description of the Parallelizable Algorithm

In this section, we first briefly state the sequential algorithm of Hu [3] to compute the shortest distances. Next we identify the concurrently executable segments of Hu's algorithm and finally present an algorithm which is readily parallelizable and suitable for VLSI implementation.

Let us denote the edge from vertex v_i to v_j of a graph by (v_i, v_j), the set of vertices by V and the set of edges by E. Let $D = [d_{ij}]$ be the distance matrix of the given graph where

$$d_{ij} = \begin{cases} 0, & \text{if } i=j \\ \infty, & \text{if } (v_i, v_j) \notin E,\ i \neq j \\ \text{nonnegative weight of the edge } (v_i, v_j), & \\ \quad \text{otherwise.} & \end{cases}$$

Hu's algorithm to compute shortest distances between every pair of vertices essentially computes the product matrix $P = D \times D$ according to some well-defined rules, first through a forward multiplication and then through a backward multiplication process. This algorithm in Pidgin ALGOL is given in Fig. 1. The time complexity of this algorithm is clearly $O(n^3)$.

It is easy to see that the entries of the product matrix which lie on any line parallel to main diagonal as shown in Fig. 2, can all be computed in parallel in both forward and backward steps, satisfying the above ordering requirement of in-place computations. Fig. 2 shows the parallel steps with step i) in the forward multiplication process for $n = 4$.

Let us define two integers l and u as follows.

$$l = \begin{cases} 1, & \text{for } i \leq n \\ i-n+1, & \text{otherwise} \end{cases}$$

$$u = \begin{cases} i, & \text{for } i \leq n \\ n, & \text{otherwise.} \end{cases} \tag{1}$$

```
forward : begin
            for s=1 until n do
            begin
                for t=1 until n do
                begin
                    p_st ← d_st;
                    for k=1 until n do
                        p_st ← min (p_st, d_sk + d_kt);
                    d_st ← p_st;
                end
            end
          end;
backward : begin
            for s=n step -1 until 1 do
            begin
                for t=n step -1 until 1 do
                begin
                    p_st ← d_st;
                    for k=1 until n do
                        p_st ← min (p_st, d_sk + d_kt);
                    d_st ← p_st;
                end
            end
          end.
```

Fig. 1. Hu's algorithm.

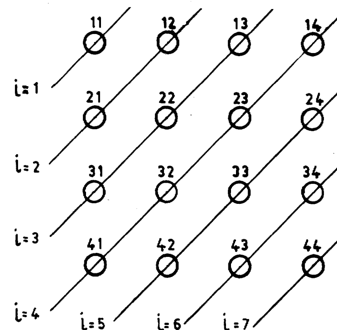

Fig. 2. Parallel steps in forward process.

l and u signify the lower and upper bounds of column numbers, respectively, that should be processed at the ith parallel step. Thus at the ith parallel step, the entries $p_{i-j+1,j}$, $l \leq j \leq u$ can all be computed in parallel. Referring to Fig. 1, we see that the most recently computed d_{sk}'s need to be broadcast [1] to different parallel steps in computing the entries $p_{i-j+1,j}$ for which $i - j + 1 = s$. Similarly, entries d_{kt} are to be broadcast to compute $p_{i-t+1,t}$ for all i. Elimination of this broadcasting of variables is desirable, whenever possible, to minimize data communication time in between two parallel steps and can be done through proper pipelining of variables. We develop below our algorithm to compute shortest distances between all pairs of vertices exploiting the above inherent parallelism in Hu's method. The algorithm, written in Pidgin ALGOL, has been described as Algorithm A, shown in Fig. 3. Here we have used three variables $-p^k_{i-j+1,j}$, r^i_{jk}, c^i_{kj}, for pipelining of data, i, j, and k being the three loop indexes. $p^k_{i-j+1,j}$ stands for the value of the $(i - j + 1, j)$th entry of the product matrix before the kth iteration of the innermost loops (innerloop 2 and innerloop 6), r^i_{jk} and c^i_{kj} stand for the respective row and column elements to compute $p^{k+1}_{i-j+1,j}$ in the kth iteration of the innermost loops for given values of i and j, respectively. The loop "forward" performs the forward multiplication and the loop "backward" does the backward multiplication. At the end of execution, c^1_{kj} contains the shortest distance from vertex v_k to v_j for all k, j, $1 \leq k, j \leq n$.

Theorem 1: Algorithm A computes shortest distances between every pair of vertices correctly.

Proof: We shall prove here only the correctness of the "forward" loop. The correctness of the "backward" loop can be established likewise.

```
Algorithm A :
1)                 begin
                     for j = 1 until n do
                       for k = 1 until n do
                         c^1_{kj} <- d_{kj};
2) forward :         for i = 1 until (2n - 1) do
                     begin
3)                     if i <= n then
                         for k = 1 until n do
                           r^i_{1k} <- d_{ik};
4)                     compute l and u;
5) innerloop 1 :       for j = 1 until u do
                       begin
6)                       p^1_{i-j+1,j} <- ∞;
7) innerloop 2 :         for k = 1 until n do
                           p^{k+1}_{i-j+1,j} <- min [p^k_{i-j+1,j}, (r^i_{jk} + c^i_{kj})];
8)                       r^i_{jj} <- p^{n+1}_{i-j+1,j}; c^i_{i-j+1,j} <- p^{n+1}_{i-j+1,j};
                       end;
9) innerloop 3 :       for j = 1 until n do
                         for k = 1 until n do
                           begin
                             r^{i+1}_{j+1(mod n),k} <- r^i_{jk}; c^{i+1}_{kj} <- c^i_{kj};
                           end
                     end;
10) backward :       for i = (2n - 1) step -1 until 1 do
                     begin
11) innerloop 4 :      for j = 1 until n do
                         for k = 1 until n do
                           begin
                             r^i_{jk} <- r^{i+1}_{j+1(mod n),k}; c^i_{kj} <- c^{i+1}_{kj};
                           end;
12)                    compute l and u;
13) innerloop 5 :      for j = u step -1 until l do
                       begin
14)                      p^1_{i-j+1,j} <- ∞;
15) innerloop 6 :        for k = 1 until n do
                           p^{k+1}_{i-j+1,j} <- min [p^k_{i-j+1,j}, (r^i_{jk} + c^i_{kj})];
16)                      r^i_{jj} <- p^{n+1}_{i-j+1,j}; c^i_{i-j+1,j} <- p^{n+1}_{i-j+1,j};
                       end
                     end
                   end.
```

Fig. 3. Parallel algorithm for all-pairs shortest distances.

Innerloop 2 computes the $(i - j + 1, j)$th entry of the product matrix. The sum of the row and column indexes of this entry is $(i + 1)$. Since i varies from 1 to $(2n - 1)$ in steps of unity, it implies that innerloop 1 computes the entries along lines parallel to the main diagonal starting from the left. From Statements 1), 3), and innerloop 3, it is easy to show by induction on i and j that in innerloop 2,

$$r^i_{jk} = c^i_{i-j+1,k}.$$

Similarly in Statement 8), $r^i_{jj} = c^i_{i-j+1,j}$, and both these are replaced here by the newly computed value of $(i - j + 1, j)$th entry of the product matrix $p^{n+1}_{i-j+1,j}$ in innerloop 2. Innerloop 2 uses r^i_{jk} and c^i_{kj}, and for all k, $k < j$ and $k < i - j + 1$, r^i_{jk} and c^i_{kj} have been replaced by $(i - j + 1, k)$th and (k, j)th entries, respectively, of the product matrix before $p^{k+1}_{i-j+1,j}$ is computed since the entries of the product matrix are all calculated along diagonals starting from top left. Hence, the loop "forward" correctly executes the forward multiplication of Hu's method. Q.E.D.

In the next section, we shall find a suitable VLSI architecture on which Algorithm A can be implemented and then we shall describe our parallel algorithm using the proposed architecture.

III. VLSI Systolic Implementation

To map Algorithm A into an appropriate VLSI array, we first define the notion of data dependence vector.

Consider a program having m loop variables $\alpha_1, \alpha_2, \cdots, \alpha_m$ such that for $1 \le p < q \le m$, the loop corresponding to α_p can never be embedded by the loop corresponding to α_q. We define the loop vector I as an m-tuple of integers $I = (a_1, a_2, \cdots, a_m)$ where a_i is the value assumed by the loop variable α_i at any stage of execution. Let $X(I_1)$ and $Y(I_2)$ be two variables generated when $I = I_1$ and $I = I_2$, respectively. Then $(X(I_1), Y(I_2))$ is called a generated-used pair of variables [1] if $X(I_1)$ is dependent on $Y(I_2)$ and the data dependence vector [1] corresponding to $(X(I_1), Y(I_2))$ is defined as $d = I_1 - I_2$.

In Algorithm A, we identify three pairs of generated-used variables: $(p^{k+1}_{i-j+1,j}, p^k_{i-j+1,j})$, $(r^{i+1}_{j+1,k}, r^i_{jk})$, and (c^{i+1}_{kj}, c^i_{kj}) in the forward process. To discriminate between the generated and used variables, let us use i, j, and k for the indexes of the used variables and i', j', and k' for those of the generated variables. The data dependence vector d_l corresponding to a generated-used pair of variables then comes out to be $d_l = (i - i', j - j', k - k')^T$, obtained by equating the corresponding indexes of the generated and used variables and then taking their differences. Hence, here we get three data dependence vectors $d_1 = (0\ 0\ 1)^T$, $d_2 = (1\ 1\ 0)^T$, and $d_3 = (1\ 0\ 0)^T$ corresponding to the above three pairs of generated-used variables. Let

$$D = [d_1\ d_2\ d_3] = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

We now seek a linear transformation $\mathfrak{I} = \left[\begin{smallmatrix}\pi \\ S\end{smallmatrix}\right]$ on D to get $\mathfrak{I}D = \Delta$, the transformed data dependence vector such that:

i) $\mathfrak{I}$ is a bijection and consists of integers.

ii) Subfunction π can be related to the processing time and S can be related to the geometrical properties of the algorithm concerning data communication in the VLSI array.

iii) $\pi d_l > 0$ for all l, $1 \le 1 \le 3$.

iv) Elements of Δ are smallest possible integers to optimize the processing time and data communication requirement of the transformed algorithm.

In our case, let

$$\mathfrak{I} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix}$$

with

$$\pi = [t_{11}\ \ t_{12}\ \ t_{13}] \text{ and } S = \begin{bmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix}.$$

Now $\pi d_1 = t_{13} > 0$, $\pi d_2 = t_{11} + t_{12} > 0$, and $\pi d_3 = t_{11} > 0$. To get maximum concurrency we shall choose the smallest possible integers for t_{11}, t_{12}, and t_{13}. So let $t_{11} = 1$, $t_{12} = 0$, $t_{13} = 1$. With this choice of π, there are many possible choices for S satisfying bijection property of $\mathfrak{I}$. One of these leads to

$$\mathfrak{I} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which results in

$$\Delta = \mathfrak{I}D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

with smallest possible integral elements. The original i, j, and k are transformed to $\hat{i}$, $\hat{j}$, and $\hat{k}$ given by $(\hat{i}\ \hat{j}\ \hat{k})^T = \mathfrak{I}(i\ j\ k)^T$. Fig. 4 shows the network geometry for $n = 4$ which follows from this choice of $\mathfrak{I}$. It is evident from Δ that the variables $p^k_{i-j+1,j}$ travel in the direction of k, r^i_{jk}'s travel in the direction of j. Variables c^i_{kj}'s do not move in space, they are simply updated in time. Since $p^k_{i-j+1,j}$'s move at a speed n times that of r^i_{jk} as evident from algorithm A, the processing time is clearly $O(2(2n - 1)n) = O(4n^2 - 2n)$ considering both the forward and backward processes. The end-around connections of the processor cells are given in conformity with the requirements of operations in Statements 9) and 11) of Algorithm A.

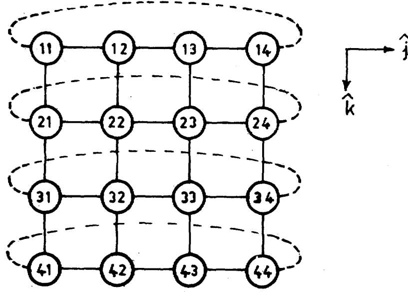

Fig. 4. VLSI array for $n = 4$.

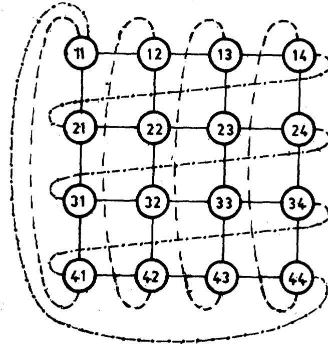

Fig. 5. VLSI array with modified end-around connections.

It is however easy to establish that the end-around connections can be made similar to Illiac IV architecture and still the algorithm would work. Moreover, the innerloop 2 and innerloop 6 can be a little bit modified so that the minimum of $(r^i_{jk} + c^i_{kj})$ for all k can be found out in $O(\log n)$ time instead of $O(n)$ time, resulting into overall processing time to be $O(n \log n)$, in the same way as done in [7]. Before describing the hardware algorithm with this modification, let us first fix the processor architecture as follows.

i) We shall use an $n \times n$ mesh connected SIMD processor array consisting of n^2 identical processors with bidirectional interprocessor communication link as in Fig. 5, which is similar to Illiac IV architecture.

ii) Each processor P_{ij} of the array has four internal registers $-R_{ij}$, C_{ij}, A_{ij}, and B_{ij}. R_{ij}'s of different processors are connected by the horizontal links and B_{ij}'s are connected by vertical links.

iii) We consider the following instruction set for the processor cells:

a) MOVE B(k)—Content of B_{ij} is transferred to $B_{i+k \,(\mathrm{mod}\, n),j}$, $1 \le i \le n$, $1 \le j \le n$
b) MOVE $R(\pm 1)$—Content of R_{ij} is transferred one position to the right $(+1)$ or left (-1), $1 \le i \le n$, $1 \le j \le n$
c) EXCHANGE—$B_{ij} \leftarrow A_{ij}$, $1 \le i \le n$, $1 \le j \le n$
d) EXCHANGE(x, y)—$B_{ij} \leftarrow A_{ij}$, for all i such that $i = x + k \pmod n$, $0 < k \le y$ and for all j, $1 \le j \le n$
e) COMPARE-EXCHANGE—$A_{ij} \leftarrow \min(A_{ij}, B_{ij})$, $1 \le i \le n$, $1 \le j \le n$
f) TRANSFER $R(x)$—$R_{xj} \leftarrow A_{xj}$, $1 \le j \le n$
g) TRANSFER $C(x)$—$C_{xj} \leftarrow B_{xj}$, $1 \le j \le n$
h) ADD(x, y)—$A_{ij} \leftarrow R_{ij} + C_{ij}$, $1 \le i \le n$, $x \le j \le y$
i) LOAD COLUMN(x)—$C_{ix} \leftarrow d_{ix}$, $1 \le i \le n$
j) LOAD ROW(x)—$R_{i1} \leftarrow d_{xi}$, $1 \le i \le n$
k) STORE COLUMN(x)—$d_{ix} \leftarrow C_{ix}$, $1 \le i \le n$.

The LOAD and STORE instructions above communicate with memory which stores initially d_{ij}'s. We now describe below the hardware algorithm using this processor architecture. Here we use a procedure "find minimum and update $RC(i)$" which computes the minimum of the contents of the A registers of the rth column of the processor array, $1 \le r \le u$, l and u being defined as in (1), and then updates the appropriate R and C registers of the rth column with this computed minimum value. This procedure will be described later. After the execution of this hardware algorithm, the memory contains the shortest distances between every pair of vertices in place of d_{ij}'s.

Hardware Algorithm B:

```
begin
    for i = 1 until n do
    begin
        LOAD COLUMN(i); LOAD ROW(i); ADD(1, i);
        find minimum and update RC(i); MOVE R(1);
    end;
    for i = 2 until n do
    begin
        ADD(i, n); find minimum and update RC(i);
        MOVE R(1);
    end;
    for i = n until 2 do
    begin
        MOVE R(-1); ADD(i-1, n);
        find minimum and update RC(i);
    end;
    STORE COLUMN(n);
    for i = n-1 until 1 do
    begin
        MOVE R(-1); ADD(1, i);
        find minimum and update RC(i);
        STORE COLUMN(i)
    end;
end.
```

The essence of the above algorithm is that at the ith parallel step, the $(i - r + 1, r)$th entry, $l \le r \le u$ of the product matrix, is computed by the rth processor column and it can be verified that both the registers R_{rr} and $C_{i-r+1,r}$ contain this entry at this step. Accordingly, the procedure "find minimum and update $RC(i)$," when executed by the processors P_{mr}'s $(1 \le m \le n)$ of the rth column $(l \le r \le u)$ of the processor array, update the registers R_{rr} and $C_{i-r+1,r}$ with the minimum of the contents of all A_{mr}'s. The procedure for all processors P_{mr}'s of the rth column is given as follows, assuming that n is a power of 2.

Procedure: find minimum and update $RC(i)$

```
            begin
1) loop:        for j = n/2, n/4, n/8, ..., 1
                begin
2)                  EXCHANGE (r+j (mod n), j); MOVE B(-j);
3)                  COMPARE-EXCHANGE
                end;
4)              TRANSFER R(r); EXCHANGE;
5)              if i-2r+1 (mod n) < n/2 then
                    MOVE B(i-2r+1) else MOVE B(2r-i-1);
6)              TRANSFER C(i-2r+1)
            end.
```

The loop in Statement 1) of the above procedure computes the minimum of the contents of A_{mr}'s $(1 \le m \le n)$ implementing the following algorithm which finds the minimum of n elements $e_1, e_2, \cdots, e_n$ and sets e_1 to this minimum value.

```
begin
    j <- n/2;
    while j > 1 do
    begin
        for i = 1 until j do
        begin
            e_i <- min (e_i, e_{i+j}); j <- j/2;
        end,
    end
end.
```

Statement 2) in the procedure is so written that this minimum value is stored in register A_{rr} after the execution of the loop. Statement 5) transfers this minimum value to $B_{i-r+1,r}$ through the shorter route.

Let t_c be the execution time of the COMPARE-EXCHANGE/ADD instructions, t_r be that for MOVE $B(\pm 1)$/MOVE $R(\pm 1)$ instructions (t_r = data communication time between two adjacent processors), and t_e be that for the EXCHANGE/TRANSFER R/TRANSFER C instructions. The execution time for the procedure "find minimum and update $RC(i)$" is then found to be equal to

$$(\log\ n)t_e+(n-1)t_r+(\log\ n)t_c+2t_e+\left(\frac{n}{2}-1\right)t_r+t_e$$

$$=(\log\ n+3)t_e+\left(\frac{3n}{2}-1\right)t_r+(\log\ n)t_c.$$

If the memory access time for LOAD/STORE instructions is t_s, then the time complexity of the algorithm B is

$$0(n\ \log\ n)t_c+0(n^2)t_r+0(n\ \log\ n)t_e+0(n)t_s.$$

To find out the shortest paths between every pair of vertices, we can employ one more register in each processor to store the value of k for which $(r^i_{jk} + c^i_{kj})$ is minimum. After the shortest distances are calculated, the shortest paths between every pair of vertices can then be calculated from these k values [3] in $O(n)$ time.

IV. CONCLUSIONS

This correspondence describes the parallel version of Hu's algorithm to compute shortest distances between all pairs of vertices in a graph. Out of many possibilities a suitable VLSI architecture is chosen on which the proposed algorithm is mapped with a time complexity of $O(n \log n)$ in t_c and t_e, but $O(n^2)$ in t_r where t_c is the time for one compare-exchange or addition operations, t_e is that for one register transfer operation within a processor and t_r is that for data communication time between two adjacent processors. This algorithm will be particularly suitable for calculating the diameter and shortest paths in a computer communication network.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous referees for their constructive criticism and valuable comments.

REFERENCES

[1] D. I. Molodovan, "On the design of algorithms for VLSI systolic arrays," *Proc. IEEE*, vol. 71, pp. 113-120, Jan. 1983.
[2] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numer. Math.*, vol. 1, pp. 269-271, 1959.
[3] T. C. Hu, "Revised matrix algorithms for shortest paths," *SIAM J. Appl. Math.*, vol. 15, pp. 207-218, Jan. 1967.
[4] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. Ass. Comput. Mach.*, vol. 5, p. 345, 1962.
[5] M. L. Fredman, "New bounds on the complexity of the shortest path problem," *SIAM J. Comput.*, vol. 5, pp. 83-89, Mar. 1976.
[6] G. H. Barnes, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746-757, Aug. 1968.
[7] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.*, vol. C-27, pp. 2-7, Jan. 1979.
[8] M. Kumar and D. S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," *IEEE Trans. Comput.*, vol. C-32, pp. 254-264, Mar. 1983.
[9] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinational algorithms," in *Proc. Conf. Very Large Scale Integration: Architect., Des., Fabrication*, California Inst. Technol., Pasadena, Jan. 1979, pp. 509-525.
[10] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. Caltech Conf. on VLSI*, pp. 65-90, Jan. 1979.
[11] ——, "Why systolic architectures?" *Computer*, vol. 15, pp. 37-46, Jan. 1982.
[12] L. Lamport, "The parallel execution of DO loops," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 83-93, Feb. 1974.
[13] A. Mukhopadhyay, "WEAVESORT—a net sorting algorithm for VLSI," Univ. Central Florida, Orlando, Tech. Rep., pp. 53-81, 1981.
[14] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 263-271, Apr. 1977.
[15] S. Todd, "Algorithm and hardware for a merge sort using multiple processors," *IBM J. Res. Develop.*, vol. 22, pp. 509-517, Sept. 1978.
[16] C. D. Thompson, "The VLSI complexity of sorting," *IEEE Trans. Comput.*, vol. C-32, pp. 1171-1184, Dec. 1983.
[17] D. J. Kuck, *The Structure of Computers and Computations, Vol. 1.* New York: Wiley, 1978.

An Algorithm for Determining the Fault Diagnosability of a System

JAGANNATHAN NARASIMHAN AND KAZUO NAKAJIMA

Abstract—**The fault diagnosability problem is the problem of computing the maximum number of faulty units which a system can tolerate without losing its capability of identifying all such faulty units. We study this problem for the model introduced by Barsi, Grandoni, and Maestrini [2]. We present a new characterization of the model, and develop an efficient diagnosability algorithm for a system in this model.**

Index Terms—**Connection assignment, diagnosable systems, fault diagnosis, self-diagnosis, system diagnosability, test links.**

I. INTRODUCTION

With the advent of inexpensive processing elements, it is now possible to design and build large-scale computing networks. Thus, fault diagnosis at the system level gains increasing importance. In this area, Preparata, Metze, and Chien [13] first introduced a formal graph-theoretic model. In this so-called *PMC model*, a system S is decomposed into n independent subsystems or units, and for the purpose of fault diagnosis, a *connection assignment* of test links is established so that each unit is tested by a subset of the other units. Let $U = \{u_1, u_2, \cdots, u_n\}$ be the set of n units. Then, the connection assignment of S may be represented by a digraph $G = (U, T)$ where each unit $u_i \in U$ is represented by a vertex and each edge $(u_i, u_j) \in T$ represents a test link by which a test is carried out by unit $u_i \in U$ on unit $u_j \in U$. The outcome of test (u_i, u_j) is represented by the weight $w(u_i, u_j)$ of the edge, where $w(u_i, u_j) = 0$ (1) if u_i evaluates u_j to be fault-free (faulty). The set of all test outcomes of S is called the *syndrome* of S. In the PMC model, it is assumed that the test outcome $w(u_i, u_j)$ is reliable if u_i is a fault-free unit; while if u_i is faulty, $w(u_i, u_j)$ may or may not be correct, regardless of the conditions of the units involved in the test.

Manuscript received September 4, 1986; revised January 15, 1986. This work was supported in part by the Joint Services Electronics Program at Texas Tech University under ONR Contract N00014-76-C-1136, and the Office of Naval Research under Contract N00014-84-C-0104.

J. Narasimhan was with the Department of Electrical Engineering/Computer Science, Texas Tech University, Lubbock, TX 79409. He is now with Burroughs Corporation, Mission Viejo, CA 92691.

K. Nakajima was with the Department of Electrical Engineering/Computer Science, Texas Tech University, Lubbock, TX 79409. He is now with the Department of Electrical Engineering, University of Maryland, College Park, MD 20742.

IEEE Log Number 8610488.