V. Conclusion

A quadratic-polynomial modulo $T_n = 2^{2n} + 1$ residue-number system is proposed to compute a complex multiplication using only two integer multiplications. For extending the dynamic range, the primary advantage of the T_n's over the Fermat numbers is due to the small number of bits between moduli T_i and T_{i+1}. Therefore, the VLSI architecture designs for the residue number system using the T_n's for computing a complex multiplication are balanced and regular.

Appendix

Some Theorems for Determining Relatively Prime T_n's in Pairs

It is of some interest to develop efficient algorithms for determining relatively prime T_n's in pairs without requiring the factorization of the T_n's. Towards this end the following theorems are useful.

Lemma 1: Let $p_1, p_2, \cdots, p_r$ be odd primes. Then $2^{2^{k_0} p_1^{k_1} \cdots p_{j-1}^{k_{j-1}} p_j^{k_j - 1} p_{j+1}^{k_{j+1}} \cdots p_r^{k_r}} + 1$ is a factor of $2^{2^{k_0} p_1^{k_1} \cdots p_{j-1}^{k_{j-1}} p_j^{k_j} p_{j+1}^{k_{j+1}} \cdots p_r^{k_r}} + 1$ where k_i are positive integers.

Proof: Let $N = 2^{2^{k_0} p_1^{k_1} \cdots p_j^{k_j - 1} \cdots p_r^{k_r}} + 1$. Then

$$2^{2^{k_0} p_1^{k_1} \cdots p_j^{k_j - 1} \cdots p_r^{k_r}} \equiv -1 \bmod N. \tag{A.1}$$

Hence,

$$\begin{aligned} &2^{2^{k_0} p_1^{k_1} \cdots p_j^{k_j} \cdots p_r^{k_r}} + 1 \\ &= (2^{2^{k_0} p_1^{k_1} \cdots p_j^{k_j - 1} \cdots p_r^{k_r}})^{p_j} + 1 \\ &\equiv ((-1)^{p_j} + 1) \bmod N \equiv 0 \bmod N. \end{aligned}$$

Therefore,

$$N \mid 2^{2^{k_0} p_1^{k_1} \cdots p_j^{k_j} \cdots p_r^{k_r}} + 1. \tag{A.2}$$

Lemma 2: Let A and B be two integers. If $A \equiv C \bmod B$ then $(A, B) \mid C$ where (A, B) represents the greatest common divisor (gcd) of A and B.

Proof: Let $d = (A, B)$. If $A \equiv C \bmod B$ then $A = mB + C$ where m is an integer. Also, if $d = (A, B)$ then one has $A = di$ and $B = dj$ where $(i, j) = 1$. Thus, $C = mB - A = di - mdj = d(i - mj)$. Hence, $d \mid C$.

Theorem 1: Let $T_n = 2^{2n} + 1$ where n is a positive integer. Then T_n and T_{n+1} are relatively prime.

Proof: For $T_n = 2^{2n} + 1$, $2^{2n} = -1 \bmod T_n$. Thus,

$$\begin{aligned} T_{n+1} &= 2^{2(n+1)} + 1 = 2^2 \cdot 2^{2n} + 1 \\ &\equiv 4 \cdot (-1) + 1 \bmod T_n \equiv -3 \bmod T_n. \end{aligned} \tag{A.3}$$

By Lemma 2, the gcd of T_n and T_{n+1} is either 1 or 3. But by Lemma 1, digit 3 is not a divisor. Therefore, $(T_n, T_{n+1}) = 1$.

Theorem 2: Let n be a positive integer. T_{n-1}, T_n, T_{n+1} are relatively prime in pairs if, and only if, n is an odd number.

Proof: By Theorem 1, one has $(T_{n-1}, T_n) = 1$ and $(T_n, T_{n+1}) = 1$. Also

$$T_{n+1} \equiv 2^4 \cdot (-1) + 1 \equiv -15 \bmod T_{n-1}. \tag{A.4}$$

Thus, by Lemma 2, $(T_{n-1}, T_{n+1}) \mid 15$.

If n is an odd number, both $n - 1$ and $n + 1$ are even numbers. But by Lemma 1 the smallest factor of T_{n-1} and T_{n+1} must be $2^4 + 1 = 17$. Hence, $(T_{n-1}, T_{n+1}) = 1$.

If n is an even number, then both $n - 1$ and $n + 1$ are odd numbers. But by Lemma 1 the smallest factor of T_{n-1} and T_{n+1} is 5. Thus, $(T_{n-1}, T_{n+1}) \neq 1$.

Corollary 2.1: Four adjacent T_n's cannot be relatively prime in pairs.

Proof: Take T_m, T_{m+1}, T_{m+2}, and T_{m+3} as four adjacent T_n's. Now one of $m + 1$ and $m + 2$ must be even. By Theorem 2 (T_m, T_{m+2}) and (T_{m+1}, T_{m+3}) cannot both be one. Hence, T_m, T_{m+1}, T_{m+2}, and T_{m+3} are not relatively prime in pairs.

References

[1] J. V. Krogmeier and W. K. Jenkins, "Error detection and correction in quadratic residue number systems," in *Proc. 26th Midwest Symp. Circuit Syst.*, Pnebla, Mexico, Aug. 1983.

[2] M. A. Soderstrand and G. D. Poe, "Application of quadratic-like complex residue number system arithmetic to ultrasonics," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing,* vol. 2, Mar. 1984, pp. 282A.5.1–A.5.4.

[3] R. Krishnan, G. A. Jullien, and W. C. Miller, "Complex digital signal processing using quadratic residue number systems," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing,* Tampa, FL, vol. 2, Mar. 1985, pp. 764–767.

[4] A. M. Despain, A. M. Peterson, O. S. Rothans, and E. Wold, "Fast Fourier transform processors using complex residue arithmetic," to be published in, *J. Parallel Distributed Processing.*

[5] J. H. Cozzens and L. A. Finkelstein, "Computing the discrete Fourier transform using residue number systems in a ring of algebraic integers," Mitre Corp. Rep., Bedford, MA, Feb. 1984.

[6] T. K. Truong, J. J. Chang, I. S. Hsu, D. Y. Pei, and I. S. Reed, "Techniques for computing the discrete Fourier transform using the quadratic residue fermat number systems," TDA Progress, Jet Propulsion Lab., Pasadena, CA, Rep., 42–81, Jan.–Mar. 1985.

[7] I. S. Reed, T. K. Truong, J. J. Chang, H. M. Shao, and I. S. Hsu, "VLSI residue multiplier modulo a fermat number," in *Proc. 7th Symp. Comput. Arithmet.*, June 4–6, 1985, Urbana, IL, pp. 203–206.

[8] L. M. Leibowitz, "A simplified binary arithmetic for the fermat number transform," *IEEE Trans. Acoust., Speech, Signal Processing,* vol. ASSP-24, pp. 356–359, Oct. 1976.

[9] F. J. Taylor, "A single modulus complex ALU for signal processing," *IEEE Trans. Acoust., Speech, Signal Processing,* vol. ASSP-33, Oct. 1985.

[10] J. Brillhart, D. H. Lehmar, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ Up to High Powers.* Providence, RI: Contempory Mathematics, vol. 22, American Mathematical Society, 1983.

[11] H. L. Keng, *Introduction to Number Theory.* Berlin, Germany: Springer-Verlag, 1982.

New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM

B. Awerbuch and Y. Shiloach

***Abstract*—Parallel algorithms for finding the connected components (CC) and a minimum spanning *FOREST* (MSF) of an undirected graph**

Manuscript received February 18, 1985; revised August 6, 1987.

B. Awerbuch is with the Department of Mathematics and Laboratory for Computer Science, Massachusettes Institute of Technology, Cambridge, MA 02139.

Y. Shiloach is with Elbit Computers, Ltd., Haifa 31053, Israel.

IEEE Log Number 8714093.

are presented. The primary model of computation considered is that called "shuffle-exchange network" in which each processor has its own local memory, no memory is shared, and communication among processors is done via a fixed degree network. This model is very convenient for actual realization. Both algorithms have depth of $O(\log^2 n)$ while using n^2 processors. Here n is the number of vertices in the graph. The algorithms are first presented for the PRAM (parallel RAM) model, which is not realizable, but much more convenient for the design and presentation of algorithms. The CC and MSF algorithms are no exceptions. The CC PRAM algorithm is a simplification of the one appearing in [17]. A modification of this algorithm yields a simple and efficient MSF algorithm. Both have depth of $O(\log m)$ and they use m processors, where m is the number of edges in the graph.

Index Terms—**Connectivity, parallel algorithms, PRAM's shuffle-exchange network, spanning trees.**

I. Introduction

The recent advances in VLSI technology make parallel algorithms very attractive. In a parallel algorithm, a computational task is split between many processors; this allows us to speed up significantly the computation time. This paper is concerned with parallel algorithms for finding the connected components (CC) and a minimum spanning forest (MSF) in an undirected graph. Those are very basic graph algorithms; they are used as building blocks in a more complex algorithm.

A. The Models

The Shuffle-Exchange (SE) Network [9] and the Parallel RAM (PRAM) [18] are two models of parallel computation. The first consists of a set of processors, each having a local random-accessed memory, that communicates via the constant-degree "shuffle-exchange" network. A detailed description of this model and several basic algorithms in it can be found in [9] and [14]. Its simplicity makes it very attractive to machine designers. In the PRAM model, all the processors share a common memory. There are a number of variants of this model which differ in the capability of performing concurrent READ's and/or WRITE's from the same memory cell. (See [16], [17].)

The PRAM is a more convenient model in terms of algorithm design and description. However, the performance of the "idealized" PRAM cannot be realized precisely, mainly because of the unbounded fan-in and fan-out required for each processor and each memory cell. It is unlikely that one could build a hardware device with thousands, and perhaps millions, of processors, unless processors have constant fan-out, like in the SE. This model also usually assumes that synchronization and assignment of processors to subproblems can be accomplished in constant time. In the SE, those operations are performed explicitly, so that there are no hidden costs. Thus, the first smoothly working prototypes of fully parallel machines (e.g., cosmic cube [11], The Connection Machine [13]) resemble SE much more than a PRAM. Approximations for PRAM do exist (e.g., BBN butterfly [12], [8]). However, those designs actually use interconnection networks which are similar to the shuffle-exchange. Many serious problems encountered in trying to realize an approximate PRAM can be found in [8] where the NYU project is described. For the reasons above, the SE is considered as our main model and our primary goal is to obtain efficient SE algorithms. We adopt a top-down approach, and begin the description with the PRAM algorithms followed by their efficient SE implementations. To simplify the presentation of PRAM algorithms, we adopt the strongest PRAM model in which concurrent reading and writing is allowed. The processors are coupled with (fixed) numbers that determine their relative strength in case of concurrent *WRITE*. If a number of processors try to write concurrently to the same cell, the strongest one succeeds.

B. Problems and Existing Solutions

Throughout this paper, we denote by n and m the number of nodes and the edges in the graph, respectively. In the PRAM model, the best known CC algorithm [17], requires $O(\log n)$ depth (= parallel time) using $n + m$ processors, and the best known MSF algorithm requires depth of $O(\log^2 n)$ using $n^2/\log^2 n$ processors [3]. It is worth mentioning that the CC algorithm of [17] works in slightly weaker PRAM model, in which it does not matter which processors succeed in concurrent write; however, our CC algorithm works in this model as well. In the SE model there is an $O(\log^4 n)$ CC algorithm that is mentioned in [14]. This algorithm is an SE implementation of the CC algorithm of [4] and it uses $m + n$ processors.

Although no SE algorithm for the MSF problem was explicitly presented, such an algorithm can be easily designed from the PRAM algorithms of [3] and [15] using the technique mentioned in [14]. Such an algorithm will require $O(\log^4 n)$ depth using n^2 processors.

C. The Modest Contributions of This Paper

In the PRAM domain, a new $O(\log n)$ CC algorithm for $m + n$ processors is presented. It is slightly simpler than the [17] algorithm, and has a substantially simpler correctness proof. A simple modification of this algorithm leads to a new PRAM MSF algorithm with the same performance.

Immediate results in the SE model can be obtained from the two SE simulations of a PRAM that are given in [1]. The second simulation, requires $\log^2 p$ steps per one PRAM step and uses $p + S$ processors, where p is the number of PRAM processors and S is the size of the actual space used by the algorithm.

Since the CC and MSF PRAM algorithms above use $m + n$ processors as well as $O(m + n)$ space, they can be implemented in an SE in $O(\log^3 n)$ time by $m + n$ processors. We were not able to improve this time with just $m + n$ processors.

A better depth can be achieved by using the first simulation in [1]. This simulation has a "slow-down" factor of just $\log p$ over the PRAM but it uses p^2 processors. Consequently, we could have $\log^2 n$ SE algorithms for both problems but with a very large number, $(m + n)^2$, processors.

In this paper, we present a technique that enables us to reduce the processors number to n^2 while keeping the depth at $O(\log^2 n)$. The PRAM algorithms for the CC and MSF problems are presented in Sections II and III, respectively. Section IV contains their implementation on SE.

II. A Simple Pram CC Algorithm

Preview

A "rooted tree" is a directed tree, with a distinguished vertex called the "root" that can be reached from each other vertex through a directed path. The "height" of a tree is the maximal distance, in terms of number of edges, of a vertex from the root. A "rooted star" is a tree of height 1. The vertices of the input graph G are numbered from 1 to n and they are referred to as "vertex i." In the course of the algorithm, vertex i maintains a "father" $F(i)$ which is another vertex or itself. The identity of the father may change during the algorithm. The father–son relation forms a directed graph, so called the "father's graph" which is denoted as FG. (This graph is referred to as the "pointer graph" in [17].) The vertices of FG are those of the G. FG has a directed link from each vertex to its father. A vertex $k = F(F(i))$ is the "grandparent" of i, denoted as $GF(i)$, and i is a "grandchild" of k.

Throughout the whole algorithm, FG is a *FOREST* of rooted trees, plus self-loops, which occur only at the roots. The set of vertices, contained in each tree of this forest, is a subset of some connected component of the graph. The algorithm merges trees that belong to the same component by "hooking" one onto another. The hooking of a tree T_1 onto T_2 is done by assigning the root of T_1 a

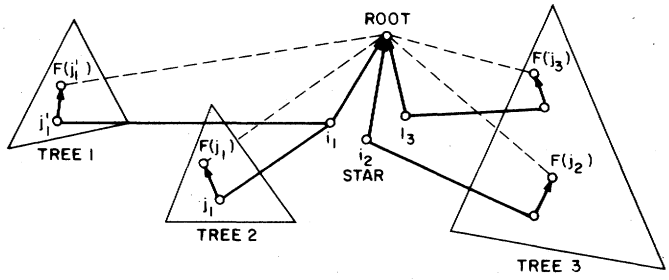

Fig. 1. Star hooking: $P(i_1, j_1)$ $P(i_1, j_1')$, $P(i_2, j_2)$, and $P(i_3, j_3)$ try simultaneously to hook the star on other trees by proposing the star's root new fathers, $F(j_1)$, $F(j_1')$, $F(j_2)$, and (Fj_3), respectively. Only one of them succeeds. The edges that cause these hooking attempts are marked by bold lines.

father in T_2, (see Fig. 1). Another step in the algorithm is to reduce the height of each tree by an operation called "shortcutting" in which each vertex adopts its grandparent as its new father. The algorithm terminates when the vertices of each connected component form a star in FG.

The root of each star is the "leader" of the whole component. Thus, queries like "are i and j in the same component?" can be answered in constant time by comparing their fathers. A processor $P(i, j)$ is attached to each edge (i, j). A sentence like "an edge (i, j) performs" should be interpreted as "$P(i, j)$ performs."

Sketch of Basic CC Algorithm:

The algorithm iterates on following three steps until all the processors have come to a STOP. Each step is executed in parallel by each processor $P(i, j)$.

Initialization: $F(i) \leftarrow i$ for $i = 1, \cdots, n$.

Step 1: (Conditional star hooking). If i belongs to a star and $F(i) > F(j)$ then $F(F(i)) \leftarrow F(j)$.

Step 2: (Unconditional star hooking). If i belongs to a star and $F(i) \neq F(j)$ then $F(F(i)) \leftarrow F(j)$.

Step 3: (Shortcutting). If i does not belong to a star then $F(i) \leftarrow GF(i)$, else—STOP.

Comments:

1) At Steps 1, and 2, a concurrent WRITE may occur when edges that connect vertices of a star with vertices that do not belong to it try simultaneously to update the father of the star's root. Only one update attempt will succeed and we do not care which. At Step 3, a concurrent WRITE of the same value is performed when all the edges that are incident with a vertex make the same shortcut.

2) The condition "i belongs to a star" is checked by the following STARCHECK procedure: a field ST is attached to each vertex i, indicating whether it belongs to a star or not. Upon termination of STARCHECK, ST is "TRUE" ("FALSE") if i belongs (does not belong) to a star. STARCHECK is based on the observation that a vertex i does not belong to a star iff it has a nontrivial grandparent or grandchild or nephew (= grandchild of father).

Starcheck:

1) $ST(i) \leftarrow TRUE$

2) If $F(i) \neq GF(i)$ then $ST(i) \leftarrow FALSE$ and $ST(GF(i)) \leftarrow FALSE$./* Excludes vertices that have nontrivial grandparent or grandchild.*/

3) $ST(i) \leftarrow ST(F(i))$/* Excludes vertices that have nontrivial nephews.*/

Remark: In the first iteration only isolated vertices of FG should be considered as stars and STARCHECK should be modified accordingly.

Theorem 1 (Conditional Correctness): If the algorithm terminates then

a) FG consists entirely of stars.

b) The vertices of each star form a connected component of the graph.

Proof:

a) Follows immediately from the termination condition in Step 3.

b) It is easy to see that the partition of the vertices determined by the connected components of FG is a refinement of the partition determined by the connected components of G. Moreover, a proper subset of a connected component cannot form a star in FG at the beginning of Step 3, since stars are unconditionally hooked on other trees in Step 2.

Theorem 2 (Logarithmic Depth):

a) Stars are not hooked on stars in Step 2.

b) FG is always a *FOREST* of rooted trees with self-loops at the roots.

c) If C is a connected component of G then the sum of the heights of the trees in FG that consist of vertices of C decreases by a factor of 3/2, at least, after each application of the loop, until these trees form a star.

Proof:

a) Hooking of a star on another tree always yields a nonstar. Thus, a star existing at the beginning of Step 2 has not been changed in Step 1. At the beginning of Step 2 there is no edge with endpoints belonging to different stars. Otherwise, the edge processors should have attempted to hook the root with the larger number on the one with the smaller number, at Step 1. (By the previous claim, they have not been changed since Step 1.)

b) This statement is true initially. Obviously, Step 3 cannot introduce new directed cycles if there were not any. In Steps 1 and 2, only stars are "hooked" onto other trees. If such a tree is not a star, it is not further hooked on another tree. Thus, the only possible way to create a directed cycle, either in Step 1 or in Step 2, is to "stitch" a bunch of stars via their roots in a circular manner as shown in Fig. 2. However, this cannot happen in Step 1 where each root is hung on a smaller father. It also cannot happen in Step 2, since, by the previous paragraph, stars are hooked only on nonstars in this step.

c) Hooking a star on another tree can add at most one to its height, i.e., the sum of heights of all the trees does not increase in steps 1 and 2, (excluding the case of Step 1 in the first iteration). Let C be a connected component of G. If the set of trees in FG consisting of the vertices of C has not shrunk yet to a single star, then at the beginning of Step 3 none of them is a star. Hence, at Step 3, the height of each tree as well as the sum of their heights, decrease by a factor of 3/2 at least.

III. A PRAM MSF Algorithm

A simple modification of the CC algorithm above yields a new PRAM MSF algorithm of the same depth and number of processors. As in the CC algorithm, each processor is assigned to an edge whose length is the number that determines the processor's "strength" where lower means stronger. It can be assumed that edge lengths are

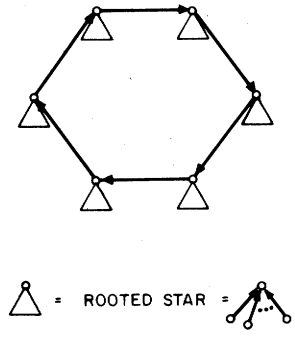

Fig. 2.

mutually distinct. This is achieved by considering each edge's "length" as a triple < actual length of (i, j), min (i, j), max (i, j) >, and taking the lexicographic order.

A. Informal Description of the MSF Algorithm

The MSF algorithm follows the scheme related to Sollin (see [2]) which uses the inherent local properties of the MSF. Namely—for any subset of vertices the shortest edge leaving it must belong to the MSF. The pairwise difference in the edges length guarantees the uniqueness of that edge which is crucial for the correctness of the algorithm below.

The algorithm maintains a set of undirected edges called *FOREST*, which is always a subforest of the MSF. In the course of the algorithm, *FOREST* grows until eventually it becomes the whole MSF. As in the CC algorithm above, we have a "fathers" graph *FG*. The connected components of its underlying undirected graph are identical to the connected components of *FOREST*, i.e., for each (maximal) directed tree in *FG* there exists a (maximal) tree in *FOREST*, spanning exactly the same set of vertices.

The algorithm iterates on three steps. Step 1 is essentially the same as Step 2 of the CC algorithm. All the processors that correspond to edges emanating from a star try to hook it on another tree. However, due to the stronger model used here, the winning processor is the one having highest priority, i.e., representing the shortest edge emanating from that star. Since this edge must belong to the MSF, it joins *FOREST*.

As a result of this step, directed cycles may be formed in *FG*. All of them are of length 2, as proved in Lemma 1 below. This happens if there exists an edge with two endpoints belonging to two different stars, and it is the shortest one emanating from both. At Step 2, these cycles are detected and "opened" by deleting the edge directed from the smaller vertex to the bigger one. Thus, *FG* is again a *FOREST* in the beginning of Step 3. Step 3 is the same shortcutting step as in the CC algorithm.

The MSF Algorithm:

FOREST(e) is a Boolean variable attached to each edge e of G. It is set to 1 if and when e joins *FOREST*. *WINNER(i)* contains the (shortest) edge whose processor is the winner in the contest of hooking the star rooted at i on another tree. This is a result of concurrent writing into *WINNER* in Step 1 below. The algorithm is executed in parallel by all edge processors $P(i, j)$.

Initialization: $FOREST(e) \leftarrow 0$ for all $e \in E$; $WINNER(i) \leftarrow NIL$ and $F(i) \leftarrow i$ for $i = 1, \cdots, n$.

Step 1: ((Unconditional) star hooking.)
If i belongs to a star and $F(i) \neq F(j)$
Then $F(F(i)) \leftarrow F(j)$ and $WINNER(F(i)) \leftarrow (i, j)$
If $WINNER\ (F(i)) = (i, j)$ then $FOREST(i, j) \leftarrow 1$.
Step 2: (Tie breaking).
If $i < F(i)$ and $i = GF(i)$
Then $F(i) \leftarrow i$
Step 3: (Shortcutting.)
If i does not belong to a star then $F(i) \leftarrow GF(i)$, else *STOP*. The algorithm loops on these three steps until everybody stops.

In order to prove the correctness and logarithmic depth of the algorithm, the following lemmas are needed.

Lemma 1: At each step of the algorithm
a) *FOREST* is a subset of the MSF.
b) the connected components of *FG* are always identical to the connected components of *FOREST* ■
c) *FG* is a directed *FOREST* with the exception of cycles of length 2 that exist only between steps 1 and 2.

Proof:
a) Initially, $FOREST(i, j) = 0$ for all i, j. Whenever $FOREST(i, j)$ is set to 1, the edge (i, j) is the shortest edge emanating from the set of vertices determined by the star rooted at $F(i)$. Thus, (i, j) belongs to the MSF.

b) This statement can be proved inductively since whenever two connected components of *FG* are connected, so do the corresponding components of *FOREST*.

c) The statement is true initially. The only way to produce any cycle in *FG* is by "stitching" a set of stars via their roots in Step 1 as shown in Fig. 2.

By b), the connected components of *FG* and *FOREST* are identical throughout the whole algorithm, and the connected components of *FOREST* that correspond to those stars can also be stitched by corresponding edges of *FOREST*. If a cycle of length 2 is formed in *FG* then it corresponds to just one (undirected) edge in *FOREST*. Such a cycle is opened at Step 2. If the cycle is longer, though, it corresponds to a nontrivial cycle in *FOREST*, a contradiction to a).

Theorem: Upon termination *FOREST* is the MSF of the graph.

Proof: It is easy to see that by the end of the algorithm the connected components of the graph coincide with those of *FG*. Thus, by statement b) above, *FOREST* spans the connected components of the graph and by a) it is the MSF. Q.E.D.

Theorem: The algorithm terminates after at most $O(\log n)$ iterations.

Proof: The proof is similar to that of the corresponding statement in the CC algorithm.

IV. SE Implementation of the CC and MSF Algorithms

Given below is a $O(\log^2 n)$ implementation of the CC and MSF algorithms in an SE that has n^2 processors. It is convenient to think of these processors as if they are placed in a $n \times n$ "matrix" where $P(i, j)$ is the jth processor in the ith row. The rows of this matrix correspond to the vertices of the graph. Moreover, $P(j, i)$ and $P(i, j)$ serve as mailboxes for the correspondence between the vertices i and j. A matrix transposition algorithm will serve as the mailman.

In terms of this matrix form, we now define three routines, *CHOOSEMIN, BROADCAST,* and *TRANSPOSE.* We shall express the various steps in the CC and MSF algorithms in terms of these routines and then show how they are implemented in the basic shuffle/exchange language of the SE, in logarithmic time.

Recall that each SE processor has a local memory. The phrase "$P(i, j)$ stores" should be read as "$P(i, j)$ stores in its local memory."

$X2 \leftarrow CHOOSEMIN(X1)$—Chooses a minimal value among all those that are stored at the field $X1$ in $P(i, j)$ for fixed i and all j and transmits it to all the processors in the ith row. They store it in another field named $X2$. *NIL* is considered as bigger than any non-*NIL* value.

Note that this operation is done independently within each row and simultaneously in all the rows. The same is true for the *BROADCAST* operation below.

$X2 \leftarrow BROADCAST(X1)$ from $P(i, F(i))$—Broadcasts the value stored at the field $X1$ in $P(i, F(i))$ to all the processors in row i. They store it at the field $X2$ (see Fig. 3).

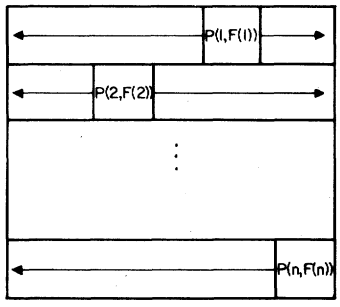

Fig. 3. The BROADCAST operation. Information is broadcasted from $P(i, j = F(i))$ to all its row members.

$X2 \leftarrow TRANSPOSE(X1)$—Transmits the value stored at the field $X1 \in P(i, j)$ to the field $X2$ in $P(j, i)$.

It is assumed in the following routines, that each processor $P(i, j)$ stores $F(i)$ in the field F. Thus, all the processors in the same row have the same value at F.

In the following we shall sometimes identify a name of a field with the value stored in it. The exact meaning will be clear from the context.

Shortcutting:

1) $X \leftarrow TRANSPOSE(F)$

2) $F \leftarrow BROADCAST(X)$ from $P(i, F(i))$.

/* The result of the first step is that $F(j)$ is stored at the field X in each $P(i, j)$. Thus, the value stored at the field X in $P(i, F(k))$ is $F(F(i)) = GF(i)$ and, therefore, after this step the values at F are updated correctly. This comment explains the "tie breaking" algorithm below as well (Fig. 4).*/

Tie Breaking:

1) $X \leftarrow TRANSPOSE(F)$.

2) $GF \leftarrow BROADCAST(X)$ from $P(i, F(i))$.

3) If $i = GF$ and $GF < F$ then $F \leftarrow i$.

Star Hooking:

1) $X1 \leftarrow TRANSPOSE(F)$.

/* The result of this step is that $F(j)$ is stored at the field $X1$ in each $P(i, j)$. */

2) If i does not belong to a star or if (i, j) is not an edge of G (or if $F \leq X1$ for conditional star hooking) then $X1 \leftarrow NIL$.

3) $X2 \leftarrow CHOOSEMIN(X1)$.

/* If i belongs to a star, then the value stored at $X2$ in row i represents the father of a vertex j such that $(i, j) \in E$ and, in case of conditional hooking, $F(j) < F(i)$. This is the proposal of vertex i to his father (the star's root) for a new father (of the root). In the following two steps, all the proposals from the root sons will be transferred to its row.*/

4) If $j \neq F$ then $X2 \leftarrow NIL$.

/*This statement makes sure that the proposal from vertex i will be transferred only to its father.*/

5) $X3 \leftarrow TRANSPOSE(X2)$.

/* In the next two steps, every root chooses the best proposal.*/

6) $X4 \leftarrow CHOOSEMIN(X3)$.

7) If $X4 \neq NIL$ then $F \leftarrow X4$.

The star hooking step in the MSF algorithm is a slight modification of this routine. The details are left to the reader.

Starcheck:

1) $ST \leftarrow TRUE$

/* The effect of the next two steps is that the grandparent of vertex i is stored at GF in all the processors of row i.*/

2) $X1 \leftarrow TRANSPOSE(F)$.

3) $GF \leftarrow BROADCAST(X1)$ from $P(i, F(i))$.

4) If $GF \neq F$ then $ST \leftarrow FALSE$.

/* In Steps 5, 6, and 7, grandparents are notified by their grandchilds that they are grandparents. By the end of Step 7, $X4$ is "FALSE" in each row that corresponds to a nontrivial grandparent.*/

5) $X2 \leftarrow \begin{cases} FALSE & \text{if } ST = FALSE \text{ and } j = GF \\ NIL & \text{otherwise.} \end{cases}$

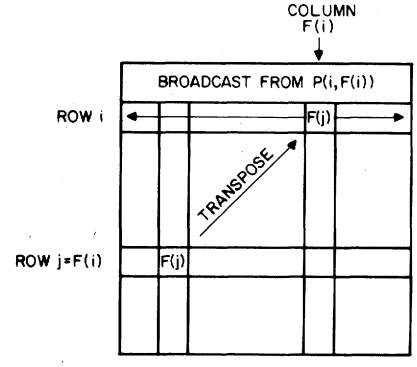

Fig. 4. The result of TRANSPOSE and BROADCAST of the field F from $P(i, j = F(i))$ is that $GF(i)$ is distributed to all the processors in row i.

6) $X3 \leftarrow TRANSPOSE(X2)$.

7) $X4 \leftarrow CHOOSEMIN(X3)$.

8) If $X4 = FALSE$ then $ST \leftarrow FALSE$.

9) $X5 \leftarrow TRANSPOSE(ST)$.

10) $ST \leftarrow BROADCAST(X5)$ from $P(i, F(i))$.

/* Steps 9 and 10 implement the PRAM statement $ST(i) \leftarrow ST(F(i))$.*/

A. SE Realization of BROADCAST, CHOOSEMIN, and TRANSPOSE

SUMMING and GROUPSUMMING: Let "*" denote any associative binary operation and let $A - a_1, \cdots, a_k$ be any k vector where a_i is stored in P_i for all i. The output of SUMMING is the vector of partial "sums" $a_1, a_1 * a_2, a_1 * a_2 * a_3 *, \cdots, a_1 * \cdots a_k$ such that $a_1 * \cdots a_k$ is stored in P_i for all i. GROUPSUMMING is similar to SUMMING. The difference is that here the vector $a_1, \cdots, a_k$ is partitioned into several segments and the result is that of SUMMING operating on each segment separately.

SUMMING and GROUPSUMMING of a vector of length k can be done in log k time by a k − SE as shown in [14].

Realization of BROADCAST and CHOOSEMIN by GROUPSUMMING. We are given an n^2 − SE and our GROUPSUMMING segments consist of all $P(i, j)$ for a fixed i, (i.e., each segment is a row in the processors "matrix").

If our binary operation is defined by: $a - b = a$, then we are able to broadcast a value from the leftmost processor in each segment $(P(i, 1))$ to all its row members. The operation $a * b = b$ enables right-to-left broadcasting. It is easy to obtain our broadcast from a combination of both.

Setting $a - b = \min(a, b)$ yields that the minimal value in each segment is stored in the rightmost processor of that segment. Combining this operation with right-to-left broadcasting yields our CHOOSEMIN operation.

TRANSPOSE is obtained by applying log n successive shuffles on a vector of length n^2 that represents the matrix elements row by row. It is exactly the matrix transposition algorithm appearing in [9].

REFERENCES

[1] B. Awerbuch, A. Israeli, and Y. Shiloach, "Efficient simulation of PRAM by ultracomputer," Tech. Rep. 120, IBM Israel Scientific Center, May 1983.

[2] C. Berge and A. Chouila-Houri, *Programming, Games and Transportation Networks.* New York: Wiley, 1965.

[3] F. Y. Chin, J. Lam, and I. Chen, "Efficient parallel algorithms for some graph problems," *Commun. ACM,* vol. 25, p. 659, Sept. 1982.

[4] D. S. Hirschberg, "Parallel algorithms for the transitive closure and the connected component problem," in *Proc. 8th Annu. ACM Symp. Theory Comput.,* Hershey, PA, 1976, p. 55.

[5] D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[6] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing

connected components on parallel computers," *Commun. ACM,* vol. 22, p. 461, 1979.

[7] D. Nassimi and S. Sahni, "Parallel permutation and sorting and a new generalized connection network," *J. Ass. Comput. Mach.*, vol. 29, p. 642, July 1982.

[8] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing a MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, p. 175, Feb. 1983.

[9] H. Stone, "Parallel processing with a perfect shuttle," *IEEE Trans. Comput.*, vol. C-20, p. 153, Feb. 1971.

[10] M. Snir, "On parallel search," in *Proc. ACM Symp. Distributed Comput.*, Aug. 1982.

[11] C. Seitz, "The cosmic cube," *Commun. ACM,* vol. 28, pp. 22–33, Jan. 1985.

[12] W. Growther, J. Goodhe, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurement on a 128-node butterfly parallel processor," in *Proc. 1985 IEEE Conf. Parallel Processing,* pp. 531–540.

[13] D. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[14] J. T. Schwartz, "Ultracomputers," *ACM TOPLAS,* vol. 2, p. 484, 1980.

[15] C. Savage and J. Ja'Ja, "Fast efficient parallel algorithms for some graph problems," *SIAM J. Comput.*, vol. 10, p. 682, Nov. 1981.

[16] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms,* vol. 2, p. 88, 1981.

[17] ——, "An (log n) parallel connectivity algorithm," *J. Algorithms*, vol. 3, p. 57, 1982.

[18] J. C. Willie, "The complexity of parallel computation," Ph.D. dissertation, Cornell Univ., Ithaca, NY, Aug. 1979.