SIAM J. COMPUT.
Vol. 16, No. 1, February 1987

# DATA MOVEMENT TECHNIQUES FOR THE PYRAMID COMPUTER*

RUSS MILLER†‡ AND QUENTIN F. STOUT†§

**Abstract.** The pyramid computer was initially proposed for performing high-speed low-level image processing. However, its regular geometry can be adapted naturally to many other problems, providing effective solutions to problems more complex than those previously considered. We illustrate this by presenting pyramid computer solutions to problems involving component labeling, minimal spanning forests, nearest neighbors, transitive closure, articulation points, bridge edges, etc. Central to these algorithms is our collection of data movement techniques which exploit the pyramid's mix of tree and mesh connections. Our pyramid algorithms are significantly faster than their mesh-connected computer counterparts. For example, given a black/white square picture with $n$ pixels, we can label the connected components in $\theta(n^{1/4})$ time, as compared with the $\Omega(n^{1/2})$ time required on the mesh-connected computer.

**Key words.** pyramid computer, graph-theoretic algorithms, image processing, component labeling, mesh-connected computer

**AMS(MOS) subject classifications.** 68Q25, 68Q20, 68U10

## 1. Introduction.

Pyramid-like parallel computers have long been proposed for performing high-speed low-level image processing [4], [17], [24], [32], [34]. The pyramid has a simple geometry which adapts naturally to many types of problems, and which may have ties to human vision processing. The pyramid can be projected into a regular pattern in the plane, which makes it ideal for VLSI implementation, providing thousands or millions of processing elements. At least three pyramid computers for image processing are currently being constructed [12], [23], [30].

There is no reason to limit pyramid computers to low-level image processing. They can be adapted to many other problems, and should be considered as alternatives to machines such as the mesh-connected computer. To show this, we present several new fundamental pyramid computer algorithms which are significantly faster than their mesh-connected computer counterparts. These algorithms solve problems in graph theory, image processing, and digital geometry.

The pyramid computer we consider is a combination of tree and mesh structures. Complete definitions appear in § 2, with the essentials being that a pyramid of size $n$ has an $n^{1/2} \times n^{1/2}$ mesh-connected computer as its base, and $\log_4(n)$ levels of mesh-connected computers above. A generic processing element (PE) at level $k$ is connected to 4 siblings at level $k$, 4 children at level $k-1$, and a parent at level $k+1$. (See Fig. 1.)

To date, the literature on pyramids primarily consists of two classes of algorithms. The first concentrates on the tree structure, using child-parent links. Examples of this are the component labeling in [6], [29], the feature extraction in [20], the median filtering in [31], the selection in [25], the single-figure convexity in [15], and the polygon construction in [21]. These algorithms work efficiently only when the amount of data

* Received by the editors August 23, 1983; accepted for publication (in revised form) Janaury 20, 1986. This research was supported by the National Science Foundation under grant MCS-83-01019.

† Mathematical Sciences, State University of New York, Binghamton, New York 13901.

‡Present address, Department of Computer Science, State University of New York, Buffalo, New York 14260.

§ Present address, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109.

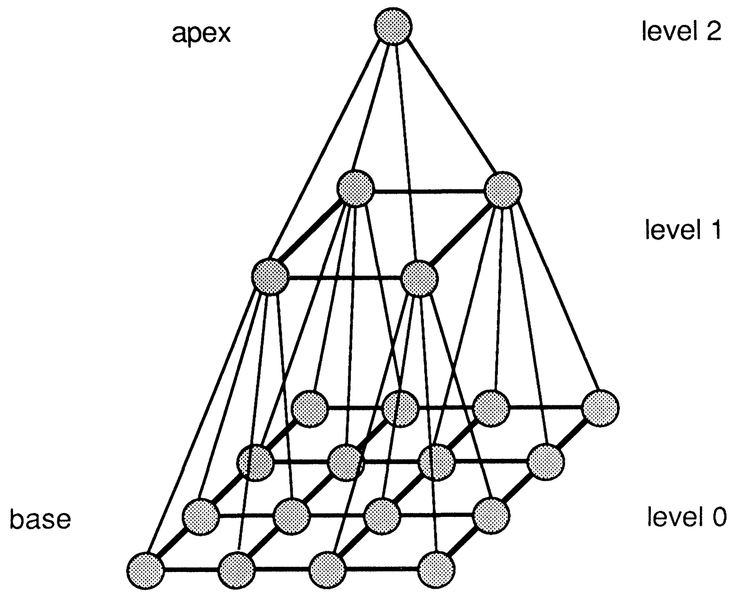


FIG. 1. *A pyramid computer of size* 16.

can be drastically reduced; otherwise too much data must pass through the apex, creating a bottleneck. The second class of algorithms concentrates on the mesh, essentially ignoring everything above the base. Examples of this are the sorting and median filtering in [25], matrix multiplication, and the multiple-label convexity in [15]. Reference [25] shows that the excessive data movement requirements of sorting force any pyramid algorithm to take $\Omega(n^{1/2})$ time. Since the base mesh can sort in $\theta(n^{1/2})$ time, the mesh oriented approach to sorting is within a multiplicative constant of being optimal.

In this paper, we consider a third class of algorithms which utilizes both types of connections. The basic approach is to reduce $O(n)$ pieces of intial data, stored one piece per base PE, down to $O(n^{1/2})$ pieces of data from which the desired result can be obtained. As has been noted for other models [10], [16], [18], this final information should be quickly moved to a region where interprocessor communication is as fast as possible, and once the answer has been obtained the results should be quickly moved to their final locations. For the pyramid this suggests moving the $O(n^{1/2})$ pieces to the middle level, which is an $n^{1/4} \times n^{1/4}$ mesh. The movement to and from the middle level is often the most time-consuming part of the algorithm, so we have developed a collection of efficient operations for performing these data movements, as well as techniques for reducing the amount of movement required.

These new data movement operations are presented for several algorithmic strategies, such as divide-and-conquer, and for various formats of the input data. They are used in several different algorithms, some of which solve various versions of the connected component labeling problem defined in § 2. In § 3, we use the pyramid read and pyramid write operations in an algorithm which labels the components of a graph of $\theta(n^{1/2})$ vertices in $\theta(n^{1/4} \log(n))$ time, where the graph is given as unsorted edges stored one per base PE. In § 4, we show that if the input is organized as an adjacency matrix, then the faster pyramid matrix read and pyramid matrix write operations reduce the time to $\theta(n^{1/4})$. In § 5, we consider input which is a digitized black/white picture, for which we wish to label the black figures. Since each black pixel is a vertex, there may be $\theta(n)$ vertices, but the geometry of the situation allows us to use the funneled read operation to complete the labeling in $\theta(n^{1/4})$ time. These times are far better than the $\Omega(n^{1/2})$ required on a mesh-connected computer of size $n$ [2], [18], [37].

Section 5 also introduces the operation of reducing a function over a cross-product. This is used to solve a nearest neighbor problem in which for each black component we wish to determine the label of and distance to the nearest black component. This operation is somewhat unusual in that once the relevant data has been collected at the proper level of the pyramid, it is then spread downward to finish the calculations.

In § 6, we give the detailed implementations of the data movement operations and also consider the optimality of our algorithms. In § 7, we extend the operations to pyramids of other dimensions. Throughout the paper we also solve related problems, such as marking minimal weight spanning forests, finding the transitive closure of a symmetric boolean matrix, marking articulation points, and deciding if a graph is bipartite.

**2. Definitions.** The *mesh-connected computer* (MCC) *of size n* is a single instruction stream-multiple data stream (SIMD) machine in which $n$ processing elements (PEs) are arranged in a square lattice. (We assume that $n$ is a perfect square.) PE $(i,j)$, $1 \leqq i,j \leqq n^{1/2}$, is connected via unit-time communication links to PEs $(i \pm 1, j)$ and PEs $(i, j \pm 1)$, assuming they exist. See [7], [14], [16], [18], [33] for an overview of the MCC.

A *pyramid computer* (PC) *of size n* is an SIMD machine that can be viewed as a full, rooted, 4-ary tree of height $\log_4(n)$, with additional horizontal links so that each horizontal *level* is an MCC. A PC of size $n$ has at its base an MCC of size $n$, and a total of $(4/3)n - (1/3)$ PEs. The levels are numbered so that the base is level 0 and the apex is level $\log_4(n)$. A PE at level $i$ is connected via bidirectional unit-time communication links to its 9 neighbors (assuming they exist): 4 siblings at level $i$, 4 children at level $i-1$, and a parent at level $i+1$. (See Fig. 1.) We make the standard assumptions that each PE has a fixed number of words (registers), each of length $\theta(\log(n))$, and that all operations take unit time. Each PE contains registers with its row, column and level coordinates, the concatenation of which provides a unique label for the PE. (These registers can be initialized in $\theta(\log(n))$ time if necessary.)

We will illustrate the use of our data movement techniques by giving solutions to a variety of problems. Each problem involves a graph $G = (V, E)$ where $V$ is the set of vertices and $E$ is the set of edges. The graph can be expressed in various forms:

(a) *Unordered edge input.* The edges of the graph are initially distributed in a random fashion throughout the base of the PC, no more than one edge per PE.

(b) *Adjacency matrix input.* PE$(i,j)$ at the base of the PC contains entry $A(i,j)$ of the adjacency matrix $A$ of the graph.

(c) *Digitized picture input.* A digitized black/white picture is initially stored one pixel per PE in the base of the PC. The vertex set consists of the black pixels, where neighboring black pixels have an edge between them.

The problems we solve are the following:

(1) *Component labeling.* The input to the problem is an undirected graph $G = (V, E)$, given in any of the three input formats. We assume that the elements of $V$ have a linear order. The component labeling algorithm assigns to each vertex a component label, where two vertices receive the same component label if and only if there is a connected path between them. In this paper, the component label will be chosen to be the minimum label of any vertex in the component.

(2) *Minimal spanning forest.* Given a weighted undirected graph, mark the edges of a minimal weight spanning tree for each component of the graph. The input format can be unordered edges or a weight matrix.

(3) *Nearest neighbor.* Given a digitized picture at the base of the PC with its components already labeled, calculate for each black component the label of the component nearest to it and its corresponding distance. Any $l_p$ metric can be used to measure the distance.

(4) *Transitive closure.* Compute the transitive closure of a symmetric boolean matrix initially given at the base of the PC.

(5) *Bipartite graphs.* Given an undirected graph $G=(V, E)$, decide if $G$ is bipartite. That is, decide if $V$ can be partitioned into sets $V1$ and $V2$ so that each edge of $G$ joins a member of $V1$ to a member of $V2$.

(6) *Cyclic index.* Compute the cyclic index of an undirected graph $G=(V, E)$, where the cyclic index of $G$ is the largest number $s$ so that $V$ can be partitioned into sets $V(0), \ldots, V(s-1)$, such that for any edge $(x, y)$, if $x$ is in $V(i)$ then $y$ is in $V((i \pm 1) \bmod s)$.

(7) *Bridge edges.* Given an undirected graph, decide which edges are bridge edges, where a bridge edge is an edge whose removal increases the number of components.

(8) *Articulation points.* Given an undirected graph, decide which vertices are articulation points, where a vertex is called an articulation point if its removal (along with its incident edges) increases the number of components.

(9) *Biconnectivity.* Given an undirected graph, decide if all components are biconnected, where a component is said to be biconnected if, for any two points in the component, there are two disjoint paths between them.

**3. Graphs as unsorted edges.** In this section, we assume that the graphs are given as unsorted edges stored one per PE at the base of the pyramid, where edges may be represented more than once. This format is the most general, including the others as special cases.

**3.1. Data movement operations.** There are several well-known data movement operations for the MCC. Two of these, the random access read (RAR) and random access write (RAW), will be defined here for the MCC and then extended to the PC. These operations involve two sets of PEs, the *sources* and the *destinations.* Source PEs send zero, one or two records, each consisting of a key and one or more data parts. (The upper limit of two records simplifies our algorithms. Most authors allow only one, in which case the operation needs to be repeated.) Destination PEs receive zero, one or two records sent by the source PEs. We allow the possibility that a PE is both a source and a destination.

MCC *Random Access Write* (RAW): In a RAW the destination PEs specify the number of records they wish to receive. At the end of the RAW, the number of records received by a destination PE is between zero and the number requested. Each record sent by a source PE is received by a destination PE, with the exception that if two or more source PEs send records with the same key then only the minimum such record is received. (In other circumstances the minimum could be replaced by any other commutative, associative, binary operation computable in $\theta(1)$ time.)

MCC *Random Access Read* (RAR): In a RAR no two records sent by source PEs can have the same key. Each destination PE specifies the keys of the records it wishes to receive, or it specifies a null key, in which case it receives nothing. Several destination PEs can request the same key. A destination PE may specify a key for which there is no source record, in which case it receives a null message.

Both the RAW and the RAR can be completed in $\theta(n^{1/2})$ time on an MCC of size $n$ [19]. On the PC, the RAW and RAR extend to the pyramid write and pyramid read, respectively. We now describe the actions of these operations, deferring the details of their implementations to § 6.1.

A *pyramid write* has the source and destination PEs playing the same roles as in the MCC's RAW. Furthermore, all destination PEs must lie on one level and all source PEs must lie on the same level or some level below. (If both levels are the same then a given PE might be both a source and a destination.) As an example, consider the following "sample" call:

```
Pyramid write from level $L$ up to level $M$,
    For all PEs on level $L$,
        if test1 then send($A1$,$B1$,$C1$), send($A2$,$B2$,$C2$);
    For all PEs on level $M$,
        if test2 then receive($D$,$E$,$F$);
```

Since source PEs are descendents of destination PEs it must be that $L \leqq M$ in this example. *test1* and *test2* are arbitrary boolean tests. For a PE on level $L$, if *test1* is true then the PE creates and sends two records, one whose key is the value of $A1$, with the values of $B1$ and $C1$ as data, and a second with key $A2$, with data $B2$ and $C2$. (The key is always the first component.) If *test1* is false then the PE does not send any records. A PE on level $M$ will not try to receive any record if *test2* is false. If *test2* is true it will try to receive a single record, where the key goes into $D$ and data parts go into $E$ and $F$. If no record is received then the values of $D$, $E$, and $F$ become $\infty$.

A *pyramid read* parallels the RAR in the same way that the pyramid write parallels the RAW, except now the destination PEs are descendents of the source PEs. As an example, consider

```
Pyramid read at level $L$ from level $M$,
    For all PEs on level $M$,
        if test1 then send($A$,$B$)
    For all PEs on level $L$,
        if test2 then receive($C$,$D$);
```

If a PE on level $L$ requests a key $C$ which has not been sent then $D$ is set equal to $\infty$.

In § 6.1, we show how to implement the pyramid write and pyramid read. If the top level is an MCC of size $m$ and the bottom level is $i-1$ levels below, then the time for both operations is $\theta(i+(m*i)^{1/2})$.

**3.2. Component labeling.** Except for obvious differences in the computer model and the data movements operations, our component labeling algorithm is similar to those in [9], [11], [13], [18], to which the reader is referred for proofs and further details. The algorithm proceeds through a series of stages, where at each stage the vertices are partitioned into disjoint *clubs*. Vertices are in the same club only if they are in the same component of the graph. We say that a club is *unstable* if it is not an entire component.

Initially each vertex is its own club. During a single *stage* of the algorithm, unstable clubs are merged together to form larger clubs, and the number of unstable clubs decreases by at least half. This is repeated until no unstable clubs remain. Since each stage reduces the number of unstable clubs by at least half, at most $\lfloor \log_2(v) \rfloor$ stages are needed for a graph with $v$ vertices.

Each club has a unique label, which is the minimum label of any vertex in the club. During the algorithm, let $L(x)$ denote the current label of the club containing vertex $x$. Initially $L(x)=x$. During a stage, clubs are merged as follows: let $u$ be a label of an unstable club. Compute $M(u)=\min\{L(y)\colon (x,y)\in E, L(x)=u\}$. The graph whose vertices are the labels of unstable clubs and whose edges are of the form $(u, M(u))$, $u$ an unstable club, is called a *min-tree forest.* In this forest each unstable club is connected to at least one other, which guarantees that the number of unstable clubs is at least halved after each stage. For each tree in the min-tree forest, we form a new club which is the union of all the clubs in the tree. For each unstable club $u$, let $N(u)$ denote the resulting label of the tree containing $u$. Since $N(u)$ is the minimum label of any club in $u$'s tree, it is the minimum label of any vertex in the new larger club which contains all the vertices originally in $u$. For each vertex $x$, the new value of $L(x)$ is $N(L(x))$, and the stage is completed.

Our component labeling algorithm for a PC is given in Fig. 2. It incorporates an integer function count_keys which counts the number of distinct keys in the base. The operation of count_keys is similar to that of the pyramid write, and is given in detail in § 6.1.

An important point of the implementation is moving the data to a place where the min-tree forest relabeling can be done quickly. The forest is essentially upward directed, in that $M(u)\leqq u$, and this makes it easier to label. Reference [18] showed that if the forest has $f$ vertices then the relabeling can be done in $\theta(f^{1/2})$ time on an MCC of size $\theta(f)$. If the forest data remained at the base, then the relabeling would take $\theta(n^{1/2})$ time. However, by first moving the data up the pyramid and then relabeling it, this step will take only $\theta(v^{1/2})$ time. We use forest_level to indicate the level of the PC on which the forest is formed. Initially, this level must have at least $v$ PEs. Each stage reduces the size of the forest by at least half, so after 2 stages forest_level can be increased by 1. Without this upward movement, the time would increase by a factor of $\log(n)$.

THEOREM 1. *Given a pyramid computer of size n, if the base contains the unsorted edges of an undirected graph with v vertices, then the above algorithm labels the components in* $\theta\ (\log(n)+v^{1/2}[1+\log(n/v)]^{1/2})$ *time.*

*Proof.* Proposition 3 of § 6.1 shows that count_keys finishes in the time claimed. Within the loop, at the start of an iteration, let $k$ be the number of PEs at level forest_level. The pyramid read and write take $\theta(\text{forest\_level}+k^{1/2}[1+\text{forest\_level}]^{1/2})$ time and the min-tree forest relabeling takes $\theta(k^{1/2})$ time. Since $k=n/4^{\text{forest\_level}}$, the time for this iteration of the loop is

$$\theta(\text{forest\_level}+n^{1/2}[1+\text{forest\_level}]^{1/2}/2^{\text{forest\_level}}).$$

The initial value of forest_level is $\lfloor \log_4(n/v)\rfloor$ and forest_level increases every 2 iterations, so the total time of the algorithm is

$$\theta\left(\sum_{i=\lfloor \log_4(n/v)\rfloor}^{\log_4(n)} \frac{i+n^{1/2}[1+i]^{1/2}}{2^i}\right)=\theta\left(\log(n)+v^{1/2}\left[1+\log\left(\frac{n}{v}\right)\right]^{1/2}\right). \qquad \square$$

In the worst case, when $v=\theta(n)$, our PC algorithm takes $\theta(n^{1/2})$ time, which is better than the $\theta(n^{1/2}\log(n))$ MCC algorithm of [18]. This situation arises, for example, when considering planar graphs, for which $v\leqq 3e-6$ edges. For smaller values of $v$ our improvement over MCC algorithms becomes even more pronounced. All MCC algorithms must take $\Omega(n^{1/2})$ time, but for a dense graph with $v=\theta(n^{1/2})$ our pyramid algorithm requires only $\theta(n^{1/4}\log^{1/2}(n))$ time.

```
For all base PEs,
    Label1:=Vertex1; Label2:=Vertex2;
    If Vertex1<∞ then  create a record with key Vertex1,
                       create a record with key Vertex2;

v:=count_keys;    {v is the number of vertices}

forest_level:=⌊log_4(n/v)⌋;

for stage:=1 to ⌊log_2(v)⌋ do

    Pyramid write from the base upto level forest_level,
        For all base PEs,
            send(Label1, Label2), send(Label2, Label1);
        For all PEs at level forest_level,
            receive(Vertex1, Vertex2);

    Relabel the min-tree forest, so that each PE on level forest_level has
        Label1:=N(Vertex1).

    Pyramid read at the base from level forest_level,
        For all PEs on level forest_level,
            if Vertex1<∞ then send(Vertex1, Label1);
        For all base PEs,
            receive(Label1, temp_label1),  receive(Label2, temp_label2);

    For all base PEs,
        if temp_label1<∞ then Label1:=temp_label1,
        if temp_label2<∞ then Label2:=temp_Label2;

    If (stage mod 2)=0  then forest_level:=forest_level + 1;

end {for};
```

FIG. 2. *Component labeling algorithm.*

**3.3. Minimal spanning forests.** The strong similarities between component labeling algorithms and minimal spanning forest algorithms are well known. In particular, others have noted that small changes to a component labeling algorithm for a parallel computer can give a minimal spanning forest algorithm for the same computer [3], [10], [22]. There are two changes that must be made to our component labeling algorithm. First, one must keep track of which edges are used. Second, when clubs are being merged each club must use an edge of minimal weight, rather than an edge to a club of minimal index. Furthermore, a club may have more than one minimal edge, which may introduce cycles. We prevent this by ordering the edges. We say that weighted edge $(w1, x1, y1)$ is less than weighted edge $(w2, x2, y2)$ if $w1 < w2$, or if

$w1 = w2$ and $\min(x1, y1) < \min(x2, y2)$, or if $w1 = w2$, $\min(x1, y1) = \min(x2, y2)$, and $\max(x1, y1) < \max(x2, y2)$.

Incorporating these changes is quite straightforward, giving the following result:

THEOREM 2. *Given a pyramid computer of size n, if the base contains the unsorted weighted edges of an undirected graph with v vertices, then the above algorithm finds a minimal spanning forest in* $\theta(\log(n) + v^{1/2}[1+\log(n/v)]^{1/2})$ *time.*

Even if the edges are unweighted, spanning forests can be quite useful. We illustrate this with an example. To decide if an undirected graph $G = (V, E)$ is bipartite, let each edge have weight 1 and use Theorem 2 to select a spanning forest. Using a pyramid write, write the edges of the forest to level $\lfloor \log_4(n/v) \rfloor$. In each tree of the forest, select the vertex of minimum label as the root, and use the MCC algorithm in [27] to determine the depth of each vertex in its rooted tree. (This algorithm takes $\theta(v^{1/2})$ time.) Say that a node is in $V1$ if its depth is even, and is in $V2$ if its depth is odd. It is easy to show that $G$ is bipartite if and only if this particlar choice of $V1$ and $V2$ is such that every edge of $E$ joins a member of $V1$ and a member of $V2$. To check whether this property is true, have the base PEs use pyramid reads to determine the depths of the vertices of the edges they contain. Finally, pass these results to the apex, combining them along the way.

The above algorithm takes $\theta(\log(n) + v^{1/2}[1+\log(n/v)]^{1/2})$ time. Furthermore, we can solve several graph-theoretic problems by using Theorem 2 to pick a spanning forest, moving the forest to level $\lfloor \log_4(n/v) \rfloor$, using an MCC algorithm at that level, and using pyramid reads and writes to move data up and down. MCC algorithms for several graph-theoretic problems are given in [27], and these can be incorporated in a PC algorithm as described, yielding:

COROLLARY 1. *Given a pyramid computer of size n, if the base contains the unsorted edges of an undirected graph G with v vertices, then in* $\theta(\log(n) + v^{1/2}[1+\log(n/v)]^{1/2})$ *time one can*

(a) *decide if G is bipartite,*

(b) *determine the cyclic index of G,*

(c) *find all bridge edges of G,*

(d) *find all articulation points of G,*

(e) *decide if G is biconnected.*

We should mention that some of the MCC algorithms of [27] are patterned after MCC algorithms in [2], with the difference that the algorithms in [2] require matrix input while those in [27] use only edge input. The algorithms of [2] are unsuitable because there may not be $v^2$ PEs to hold the adjacency matrix. More important, the algorithms of [2] are too slow because they use matrix calculations that take $\theta(v)$ time on a PC.

**4. Graphs as adjacency matrices.** In this section, we consider undirected graphs with $n^{1/2}$ vertices, where the graph is given as an adjacency matrix or weight matrix. We assume that the $(i, j)$ entry of the matrix is stored in base PE $(i, j)$. Because the input is now more structured, we are able to give algorithms which are slightly faster than those of § 3.

**4.1. Data movement operations.** The algorithms of this section require two new data movement operations, *pyramid matrix write* and *pyramid matrix read.* A pyramid matrix write performs the same basic action as a pyramid write and comes in two versions, one for rows and one for columns. In the row (column) version source PEs lie in the base, and those in the same row (column) send the same key. The pyramid matrix read performs the same basic action as a pyramid read, and also comes in two versions. For the row (column) version, all destination PEs lie in the base, and those in the same row (column) request the same key.

Detailed implementations of these operations appear in § 6.2, where it is shown that if a pyramid matrix write (read) has its destination (source) PEs at a level which is an MCC of size $k$, $k \leqq 2n^{1/2}$, then the time used is $\theta(\log(n) + k^{1/2}[1+\log(n/k^2)]^{1/2})$. (Though we never have more than $n^{1/2}$ keys, we must allow $k = 2n^{1/2}$ since the highest level holding $n^{1/2}$ PEs actually has $2n^{1/2}$ PEs when $n > 256$ is an odd power of 4.)

**4.2. Matrix algorithms.** Our algorithms for graphs given as adjacency or weight matrices are simple adaptations of our algorithms for unsorted edges. We merely start with the previous algorithms, remove the call to count_keys, replace pyramid read with pyramid matrix read, and replace pyramid write with pyramid matrix write. The resulting algorithms are faster than those of § 3 by a factor of $\log^{1/2}(n)$.

THEOREM 3. *Suppose the adjacency matrix of an undirected graph with $n^{1/2}$ vertices is stored in the base of a pyramid computer of size n. Then the above algorithm labels the connected components in $\theta(n^{1/4})$ time.*

THEOREM 4. *Suppose the weight matrix of a weighted undirected graph with $n^{1/2}$ vertices is stored in the base of a pyramid computer of size n. Then the above algorithm marks a minimal spanning forest in $\theta(n^{1/4})$ time.*

COROLLARY 2. *Suppose the adjacency matrix of an undirected graph G with $n^{1/2}$ vertices is stored in the base of a pyramid computer of size n. Then in $\theta(n^{1/4})$ time one can*

(a) *decide if G is bipartite,*
(b) *determine the cyclic index of G,*
(c) *find all bridge edges of G,*
(d) *find all articulation points of G,*
(e) *decide if G is biconnected.*

Determining the transitive closure of a symmetric boolean matrix stored in the base of a PC is a simple adaptation of component labeling. First perform component labeling for matrix input. For PEs which are storing off-diagonal entries (i.e., for which the row and column are different), the new entry is 1 if the row label equals the column label, while otherwise it remains 0. For PEs on the diagonal, if the original entry was 1 it remains so, while if it was 0 then it becomes 1 only if some other entry in the row is 1. Pyramid matrix reads and writes can be used to determine the proper diagonal entries, giving the following result:

COROLLARY 3. *Suppose an $n^{1/2} \times n^{1/2}$ symmetric boolean matrix is stored in the base of a pyramid computer of size n. Then the transitive closure can be determined in $\theta(n^{1/4})$ time.*

**5. Divide-and-conquer algorithms.** In this section, we use a divide-and-conquer approach to solve a variety of geometric problems involving black/white pictures stored one pixel per PE at the base of the PC. The use of divide-and-conquer for geometric problems is well known, but a naive use of this strategy on the pyramid computer does not necessarily produce good results. We demonstrate some efficient implementations of this strategy on the PC.

Throughout this section we will often divide the MCC at some level into squares of some size $S$. What we mean by this is that we will completely partition the MCC into disjoint squares of size $S$, where $S$ is a power of 4. Using this partitioning, the concept of *the square of size S at level l containing* PE $P$ is well defined (assuming that level 1 is of size $S$ or greater). The term *picture square* will be used to refer to a square in the base.

The computations for our divide-and-conquer solutions will proceed in a bottom-up fashion. The first *stage* will involve analyzing picture squares of size $4^c$, for some small constant $c$ which depends upon the particular problem. In general, stage $i$ has analyzed picture squares of size $4^{c+i-1}$, and stage $i+1$ combines the results together to analyze picture squares of size $4^{c+i}$. An important point of our solution strategy is that at the end of stage $i$, each picture square of size $4^{c+i-1}$ has been reduced to $O(2^i)$ records of data, from which stage $i+1$ can produce the analysis for picture squares of size $4^{c+i}$. Our algorithms proceed rapidly by moving the data that represents a picture square up through the subpyramid whose base is the picture square.

For a picture square of size $4^{c+i-1}$, the square of size $4^{c+\lceil i/2\rceil-1}$ at level $\lfloor \frac{i}{2} \rfloor$ which contains all level $\lfloor \frac{i}{2} \rfloor$ ancestors of the PEs in the picture square is called the *data square corresponding to the picture square.* Notice that when stage $i$ is working on picture squares of size $4^{c+i-1}$, the corresponding data squares have enough PEs to contain the results from stage $i-1$, so all the work is performed in the data squares. Further, the data square corresponding to a picture square is either the union of the data squares corresponding to the picture's quadrants, or else it is the union of the parents of the quadrants' data squares. This means that the data from one stage is either already in place, or must move up only one level, for the next stage.

The last stage of the divide-and-conquer algorithm is stage $\log_4(n)-c+1$, during which the entire picture is analyzed. Since the intermediate results are scattered in data squares throughout the pyramid, a final step is needed to move these results back down to the base. This final data movement is accomplished via a funnel read, described in § 5.1. Section 5.1 also introduces a data movement operation called reducing a function. This operation allows data squares to perform some calculations (such as computing the nearest neighbor for each point from a set of points) in time linear in the edgelength of the square, even though we do not know of an MCC algorithm which finishes in this time. The operation of reducing a function uses PEs below the data square to help perform the calculations in the desired time.

**5.1. Data movement operations.** We now describe data movement operations that will be used to implement divide-and-conquer algorithms on the PC.

*Funnel read.* Assume each base PE knows the key for data it wishes to read from its stage 1 data square. For a stage $i$ data square which is responsible for supplying the data for a given key, there are three possibilities: either one of its PEs has the data, or it must read the data from its stage $i+1$ data square (where by its stage $i+1$ data square we mean the data square it supplies data to), or one of its PEs has an alias for the key and must read the data for the alias from its stage $i+1$ data square. (If $i$ is the last stage, then the square must have the data.) Further, a data square of size $S$ never receives more than $S$ requests. The funnel read ultimately obtains the data for all of the base PEs in $\theta(S^{1/2})$ time, where $S$ is the size of the data squares at the final stage. Figure 3 is a picture of a funnel read, and its detailed implementation is in § 6.3.

*Reducing a function.* Given sets $Q$, $R$, and $S$, let $g$ be a function mapping $Q\times R$ into $S$, and let $*$ be a commutative, associative, binary operation over $S$. Define $f$, a map from $Q$ into $S$, by $f(q)=*\{g(q,r)\colon r\in R\}$. We say $f$ is the *reduction of* $g$. For example, if $Q$ and $R$ are sets of points in some metric space, if $S$ is the real numbers, if $g(q,r)$ is the distance from $q$ to $r$, and if $*$ is the minimum, then $f(q)$ is the distance from $q$ to the nearest point in $R$.

Suppose the elements of $Q$ are stored one per PE in a square of size $m$ at level $i$, and the elements of $R$ are also stored one per PE in the square. (A PE may contain an element of $Q$ and an element of $R$.) Suppose $g$ and $*$ can both be computed in

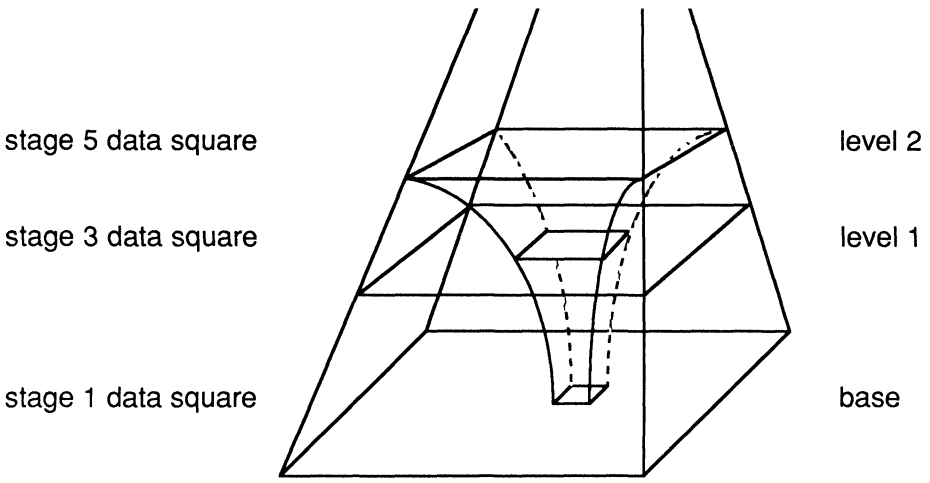


FIG. 3. *A single* PE*'s view of a funnel read.*

$\theta(1)$ time. Then the operation of reducing a function will compute $f(q)$ and store it in the PE containing $q$, for all $q \in Q$, in $\theta(m^{1/2} + m/4^i)$ time. The detailed explanation of this operation appears in § 6.3.

**5.2. Component labeling for digitized pictures.** As was noted in § 2, a digitized picture can be viewed as being an undirected graph, where the black pixels are the vertices, and adjacent black pixels have an edge between them. Upon termination of our component labeling algorithm for digitized pictures, every base PE containing a black pixel will also contain the label of the pixel's component.

ALGORITHM. We follow the basic divide-and-conquer strategy outlined above. Our algorithm is similar to the MCC algorithm of [18], but is significantly faster.

First, each base PE $P$ containing a black pixel generates a record $(p, q, \infty)$ for each neighbor $Q$ containing a black pixel, where $p$ is the index of $P$, $q$ is the index of $Q$, and the third component of the record will be used to store the component label of $p$ and $q$ generated at this stage. Thus, each PE may generate as many as four records, one for each of its neighbors.

At stage 1, picture squares of size 256 are labeled. (That is, $c = 4$ in the generic divide-and-conquer strategy.) This is accomplished by performing the component labeling algorithm for unsorted edge data, as described in § 3.2, simultaneously for every picture square of size 256 by using the edge data just generated in each picture square. That is, for each picture square we can apply the unsorted edge data labeling algorithm only to those records $(x, y, \infty)$ for which both $x$ and $y$ are in the picture square, i.e., we omit those records for which $x$ is on the border of the square and $y$ is in an adjacent picture square. (Since $x$ and $y$ are concatenated coordinates of PEs, in $\theta(1)$ time a PE containing $(x, y, \infty)$ can decide whether or not $x$ and $y$ are in the same picture square.)

When the edge data labeling algorithm is completed, those PEs with a record $(x, y, \infty)$ use a RAR in their picture square to determine the (possibly) new label of $x$. The label is stored in the third field of this record. (If the third field was $\infty$ at the end of the edge data labeling algorithm, it must be that $y$ lies outside of $x$'s stage 1 picture square.) These records have all of the information needed for the next stage of the algorithm, since the only components for which a single label has not yet been assigned are those which lie in two or more stage 1 picture squares. All records generated originally are kept in their stage 1 data squares, and each PE containing a record $(x, y, z)$ with $y$ outside the picture square generates a record $(z, y, \infty)$ for use in the next stage. There are at most 64 such records generated within a single picture square of size 256. (Each corner pixel may generate 2 such records, and all other border pixels may generate 1.) These records are now spread out in their stage 1 data squares so that no PE holds more than one such record. This concludes stage 1.

At stage i, the algorithm labels picture squares of size $4^{3+i}$, using data squares of size $4^{3+\lceil i/2 \rceil}$ at level $\lfloor \frac{i}{2} \rfloor$. If $i$ is odd, then all of the data needed (the records generated at the end of the previous stage) is already present, while if $i$ is even, then the data is in the four data squares one level below. In the latter case, the data is moved up and distributed so that no PE has more than one record. (This can be done in $\theta(2^{i/2})$ time.) Next, component labeling is performed for this edge input, again using only edges both of whose vertices lie in the picture squares. When finished, a PE containing a record with a vertex outside of the picture square generates a record for the next stage. Since we are working on picture squares of size $4^{3+i}$, at most $2^{5+i}$ records can be generated for the next stage.

After stage $\log_4(n)-3$, all of the labels have been decided. Notice that if $P$'s component extends outside of $P$'s stage 1 picture square, then the labeling information in the stage 1 data square may be incorrect, and $P$ would need to consult later stages. The component may extend outside of $P$'s picture square for many stages, so in advance $P$ does not know which data square has the needed labeling information. This is where a funnel read is used, moving labels from the data squares of the last stage back down towards the base, taking $\theta(n^{1/4})$ time. This gives the following result:

THEOREM 5. *The component labeling problem for digitized picture input on a pyramid computer of size n can be solved in $\theta(n^{1/4})$ time.*

This represents a substantial improvement over the $\theta(n^{1/2})$ MCC algorithm in [18]. Reference [29] recently presented a different PC algorithm for labeling components in a digitized picture. This algorithm is designed to label "convex blobs," but because it uses only child-parent links it takes $\theta(n^{1/2})$ time to label a $D \times n^{1/2}$ rectangle, for any constant $D$. In contrast, our algorithm will label any digitized picture, and hence all "convex blobs," in $\theta(n^{1/4})$ time.

**5.3. Nearest neighbors.** The solution to the nearest neighbor problem is quite similar to the solution just presented for the component labeling problem. Therefore, we will describe in detail only those aspects of the algorithm that change.

In the nearest neighbor problem, we wish to find the kin of each component, where the *kin* of a component is the label/distance pair representing the nearest labeled component with a different label. (In case of ties, the component of smallest label is chosen.) The input to the nearest neighbor problem is a digitized picture with its components already labeled, and at the conclusion of the algorithm each black pixel will have the kin information for its component.

Our divide-and-conquer algorithm is based on the following observation: assume that the 4 quadrants within a picture square have been analyzed. When combining the 4 quadrants, the only components whose kin could lie in a quadrant other than their own are those components that have at least one pixel that is an extreme point. An *extreme point* is a black pixel that is, relative to its component, either the northernmost or southernmost black pixel in its column, or the easternmost or westernmost black pixel in its row. Within a quadrant, components with no extreme points must have determined their kin in earlier stages since they are totally surrounded by other components within their quadrant.

We again analyze squares of size 256 at stage 1. Within each picture square, for each component $C$ we determine the closest component within the square, and store this in a record ($C$, kin ($C$)). (This kin information may be incorrect, but the final funnel read will bring the correct information down from a data square above.) For each column $i$ in the picture square, form the records (1, $i$, tr ($i$), tl ($i$)), (2, $i$, br ($i$), bl ($i$)), where tr($i$) (br($i$)) is the row of the topmost (bottommost) black pixel in the column restricted to the square, and tl ($i$) (bl ($i$)) is the label of this pixel. (If the column has no black pixel then set the coordinates to $\infty$.) Similarly, for each row $j$ we form records (3, $j$, lc ($j$), ll ($j$)), (4, $j$, rc ($j$), rl ($j$)) for the leftmost and rightmost black pixels in the row. These are the records needed for the next stage.

In general, at stage $i+1$ we first find, for each black pixel represented in one of the records passed on from stage $i$, the nearest black pixel (represented in a record) of a different label. We use the operation of reducing a function to do this, where $Q$ and $R$ are the records, $S$ is the real numbers, $*$ is the minimum, and $g$ is the distance, with the exception that $g$ gives an infinite distance if the two points have the same label. When the operation is finished, we use an RAR to form a record ($C$, kin ($C$)) for each component $C$ represented by one or more pixels. To generate the records for the next stage, notice that for each column in the stage $i+1$ picture square there are two type 1 records. The one representing the topmost pixel is passed to the next stage, and similar reductions occur for records of types 2, 3 and 4.

Finally, after the last stage of the algorithm, a funnel read brings the correct kin information back to the base, giving the following result:

THEOREM 6. *The nearest neighbor problem for digitized picture input on a pyramid computer of size n can be solved in* $\theta(n^{1/4})$ *time.*

We note that if one is interested only in determining, for each black pixel, the location of the nearest black pixel, then the PC needs only $\theta(\log(n))$ time [26].

**6. Data movement operations.** In this section, we describe how to perform the data movement operations used in earlier sections. We also discuss the optimality of our algorithms.

**6.1 Pyramid read, pyramid write, and count-keys.** A pyramid read starts with records stored at some level $i$, each with a different key, and moves them down to level $j$ where they can be read. Let $m = n/4^i$ and $S = 4 * \lfloor \log_4 {}^{[m/((i-j+1)]} \rfloor$. In this algorithm, we use the term *square* to mean "square of size $S$", and we divide levels $j \cdots i$ into squares. The squares on level $i$ are numbered from 1 to $m/S$ using a snake-like ordering. (See Fig. 4.) All of the data starting in square $k$ on level $i$ is called *packet k*.

By a *cycle* we mean $cS^{1/2}$ time units, where the constant $c$, independent of $n$ and $S$, is chosen so that in one cycle a square can perform all of the following functions:

1. Exchange packets with the next square on the same level (where next is with respect to the snake-like ordering).
2. Copy a packet to the four squares on the level below.
3. Perform an MCC RAR.

We now describe the pyramid read algorithm. Packets are first passed backwards along level $i$ towards square 1, using the snake-like ordering, one square per cycle. Once at square 1, a packet is moved forwards along level $i$, again using the snake-like ordering. Each time a square at level $i$ receives a packet moving forwards, it first copies

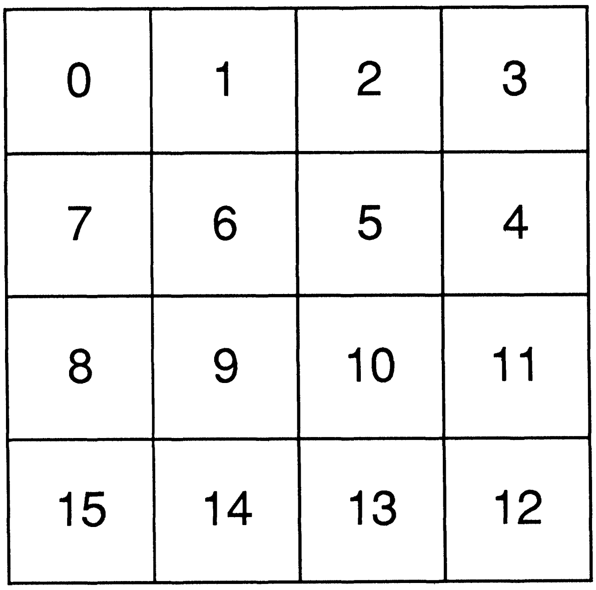


FIG. 4 *Snake-like ordering.*

it to the four squares below before passing it along, all in one cycle. Each square at level $(j+1)\ldots(i-1)$ which receives a packet just copies it to the four squares on the level below. Finally, each time a square on level $j$ receives a packet it does an MCC RAR.

PROPOSITION 1. *In a pyramid computer of size $n$, a pyramid read from level $i$ to level $j$ takes $\theta(i-j+1+[m * (i-j+1)]^{1/2})$ time, where $m = n/4^i$.*

*Proof.* The operation is finished when packet $m/S$ has moved backwards to square 1, forwards to square $m/S$, down to level $j$, and all level $j$ squares beneath square $m/S$ have done a RAR. This takes $2 * m/S - 1 + i - j$ cycles, or $\theta(i-j+1+[m * (i-j+1)]^{1/2}$ time. □

For the pyramid write, assume that the destination PEs are on level $i$, the source PEs are on level $j$, and $m$ and $S$ are defined as above. The pyramid write is basically performed by running the pyramid read in reverse. Slight differences arise because several base PEs can send records with the same key, but perhaps different data parts, in which case we need to take a minimum. Also, it is not initially known which packet a given record will end up in.

To accommodate these problems, in general a square $Z$ will have a packet's worth of data from each square feeding into it (either the four squares below, or, for squares at level $i$, the four squares below and the preceding square in the snake-like ordering). From this, $Z$ has enough to make at least one packet's worth of data. However, since the square to which it is feeding data may have some leftover data from the previous cycle, the square it is feeding informs $Z$ how many records are needed. In one cycle, $Z$ supplies the necessary data and informs each square feeding into it how many of that square's records need to be replaced. Since it takes one cycle to receive the data, and one cycle for $Z$ to pass on data after the new data is received, each step of the pyramid write takes two cycles.

Making these minor changes to the pyramid read, we obtain:

PROPOSITION 2. *In a pyramid computer of size $n$, a pyramid write from level $j$ to level $i$ can be performed in $\theta(i-j+1+[m * (i-j+1)]^{1/2})$ time, where $m = n/4^i$.*

Recall that count_keys is a function responsible for counting the number of distinct keys present in the base of a PC. If each key were represented only once, then count_keys could finish in $\theta(\log(n))$ time. However, keys in general are duplicated, so count_keys uses the pyramid write to eliminate duplicates. It first tries to determine if the number of keys is $\leqq K$, where $K = 4^{\lceil \log_4(\log_4(n)) \rceil}$, by doing a pyramid write to level $L = \log_4(n/K)$, where each destination PE requests one record. When finished, all PEs below level $L$ use a pyramid read to check whether their key was passed all the way up to level $L$. If this is so for all PEs, then the number of keys is the number of records at level $L$. Otherwise, count_keys sees if the number of keys is $\leqq 4K$ by doing a new pyramid write to level $L-1$. It continues multiplying the number of keys by 4 at each stage until it reaches a stage where the pyramid write succeeds in moving all the keys to level $L$. This gives us the following result:

PROPOSITION 3. *If there are k different keys present in the base of a pyramid computer of size n, then in* $\theta(\log(n) + k^{1/2}[1+\log(n/k)]^{1/2})$ *time count_keys will count them.*

**6.2. Pyramid matrix read and pyramid matrix write.** Assume that a pyramid matrix write has its destination PEs at level $i$, and let $m = n/4^i$. (Recall that $m \leqq 2n^{1/2}$.) The pyramid matrix write has two steps: the first moves the data to level $j = \log_4(m)$, and the second moves it to level $i$. (Note: if $m = 2n^{1/2}$ then set $j = i$ instead of $i+1$.)

To perform the first step of the row version of the pyramid matrix write (the column version is similar), we partition the PEs at level $j$ into strings of $k = 2^j$ PEs all in the same row, and call such a string and all PEs beneath it a *prism*. Notice that a prism includes parts of $k$ rows in the base, and hence sits over no more than $k$ different keys. In each prism, at time $j$ the first string PE receives the minimum record sent from any base PE beneath it in the first row of its prism. This PE passes the record on to the next PE in its string. In general, the computations are pipelined so that at time $j + r - 1 + p - 1$ the $p$th PE in the string of each prism receives the minimum record sent from any PE beneath it in the $r$th row of the prism, and also receives from the preceding string PE the minimum record sent from any base PE in the $r$th row beneath any of the preceding string PEs. The $p$th PE in the string takes the minimum of these two values and passes it to the $p+1$st PE in its string.

At time $j + k - 1$ the last PE in each string forms the minimum sent by any base PE in the first row of its prism, and this value is sent back towards the first PE of its string. These reverse messages are passed simultaneously with the previously mentioned ones. Finally, at time $j + 2 * k - 2$ the last string PE (the $k$th one) finds the minimum record sent by any base PE in the $k$th row of the prism. Simultaneously, the minimum record sent by any base PE in the first row of a prism has moved back to the first PE of its string, and the first step of the algorithm is finished.

The second step is just a pyramid write from level $j$ to level $i$. This gives us the following result:

PROPOSITION 4. *In a pyramid computer of size n, a pyramid matrix write to level i,* $i \geqq \lfloor \log_4(n)/2 \rfloor$, *takes* $\theta(\log(n) + m^{1/2}[2+\log(n/m^2)]^{1/2})$ *time, where* $m = 4^i$.

*Proof.* If $m \leqq n^{1/2}$, then the time for the first step is $\theta(m^{1/2})$, and the time for the second step is $\theta(i - j + 1 + m^{1/2}[i - j + 1]^{1/2})$. Since $j = \log_4(m)$ and $i = \log_4(n/m)$, we have the desired result. Otherwise, if $m = 2n^{1/2}$, then the time is $\theta(m^{1/2})$. In this case, $\log_4(n/m^2) = -1$, which is why there is a 2 instead of the usual 1 inside the brackets. □

For a pyramid matrix read, assume that the source PEs are at level $i$, and let $m = n/4^i$. Again we describe just the row version, which takes 3 steps. The first step uses prisms of height $j$, where $j$ is as above. By using the first step of the matrix write, in $\theta(m^{1/2})$ time the top row (string) of each prism contains the keys needed by the rows beneath. The second step is a pyramid read from level $i$ to level $j$. The third step reverses the first one, taking the data to the base.

PROPOSITION 5. *In a pyramid computer of size n, a pyramid matrix read from level i, $i \geqq \lfloor \log_4(n)/2 \rfloor$, takes $\theta(\log(n)+m^{1/2}[2+\log(n/m^2)]^{1/2})$ time, where $m=4^i$.*

**6.3. Funnel read and reducing a function.** The funnel read is straightforward. If the final stage of an algorithm is stage i, then we first have each stage $i-1$ data square use a pyramid read to obtain the data from its stage $i$ data square. (Notice that this runs in time linear in the size of the stage i data square.) We continue downwards, each stage $j-1$ data square using a pyramid read to obtain its data from its stage $j$ data square. When going down, the squares get smaller by a factor of 4 (see Fig. 3), so we obtain the following:

PROPOSITION 6. *Assume that the final stage of an algorithm is stage* i, *and that the stage* i *data square is an* MCC *of size S. Then a funnel read runs in $\theta(S^{1/2})$ time.*

Let $g$, $Q$, $R$, $S$, $*$, $f$, $m$, and $l$ be as in the description of reducing a function in § 5.1. We use the notation $G(A, B)$ to denote the function defined on a subset $A$ of $Q$ whose value at $a \in A$ is given by $G(A, B)\ (a) = *\{g(a, b)\colon\ b \in B\}$. Notice that $f$ is $G(Q, R)$.

For a set of PEs $S$, by *computing $G(A, B)$ in $S$* we mean that for each element $a$ in $A$ there is a PE in $S$ which computes and stores the value of $G(A,\ B)(a)$. Notice that if a set $A \subset Q$ is partitioned into subsets $A1$, $A2$, $A3$, and $A4$, then

$$G(A, B) = G(A1,\ B) \cup G(A2, B) \cup G(A3, B) \cup G(A4, B),$$

where we view a function as a set of ordered pairs. Also, if a set $B \subset R$ is partitioned into subsets $B1$, $B2$, $B3$, and $B4$, then

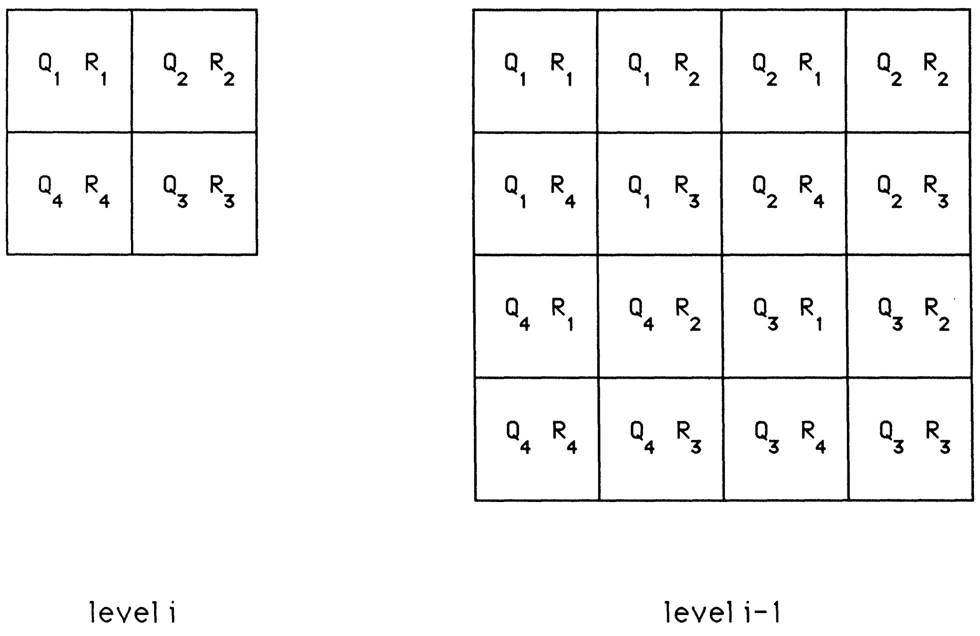
$$G(A, B)(a) = G(A, B1)(a) * G(A, B2)(a) * G(A, B3)(a) * G(A, B4)(a)$$

for any $a \in A$.

Using these observations, our operation of reducing a function is also straightforward. If $m=1$, then obviously the single PE just computes its value in $\theta(1)$ time. If $l=0$ (i.e., if all of the data is at the base), then we merely circulate all values of $R$ among all PEs holding members of $Q$, and each such PE calculates the associated $f$ value. This takes time proportional to $m$, the number of PEs. Otherwise, $Q$ is partitioned into 4 subsets (as equal as possible), $Q1$, $Q2$, $Q3$ and $Q4$, and these are arranged in the four quadrants of level $l$, as in the left-hand side of Fig. 5. The quadrant holding $Qi$ is responsible for computing $G(Qi, R)$. It does this by first having each PE copy its element of $Q$ into the PE's four children. The quadrant also copies all of $R$ to the square beneath it, creating the situation as in the right-hand side of Fig. 5. A square of size $m/4$ on level $l-1$ holding $Qi$ and $Rj$ is now responsible for computing $G(Qi, Rj)$, which it does recursively. When this is finished, beneath each quadrant of level $l$ the four squares of size $m/4$ at level $l-1$ send up their results. A PE at level $l$ holding an element $q \in Q$ receives $G(Q, R1)(q)$, $G(Q, R2)\ (q)$, $G(Q, R3)\ (q)$ and $G(Q, R4)\ (q)$ from its children, and by taking the $*$ of these computes $G(Q, R)\ (q)$.

PROPOSITION 7. *Suppose the elements of Q are stored one per* PE *in a square of size m at level l, and the elements of R are also stored one per* PE *in this square, and suppose g and $*$ can be computed in $\theta(1)$ time. Then the reduction of g can be computed in $\theta(m^{1/2}+m/4^l)$ time.*

*Proof.* It takes $\theta(m)$ time to copy the values of $Q$ and $R$ from level $l$ to the appropriate places on level $l-1$. Since the size of the squares reduces by a factor of 4 at each level, it takes $\theta(m)$ time to copy the values all the way down to the base.



$Q_1\ R_1$ $Q_2\ R_2$
$Q_4\ R_4$ $Q_3\ R_3$
$Q_1\ R_1$ $Q_1\ R_2$ $Q_2\ R_1$ $Q_2\ R_2$
$Q_1\ R_4$ $Q_1\ R_3$ $Q_2\ R_4$ $Q_2\ R_3$
$Q_4\ R_1$ $Q_4\ R_2$ $Q_3\ R_1$ $Q_3\ R_2$
$Q_4\ R_4$ $Q_4\ R_3$ $Q_3\ R_4$ $Q_3\ R_3$
level i
level i-1

FIG. 5. *Reducing a function.*

(If $m < n^{1/2}$, then the data does not even reach the base, instead only moving down $\log_4(m)$ levels until the problem has been broken into squares of size 1.) The squares on the base have size $m/4^l$, so they take $\theta(m/4^l)$ time to compute their values. It now takes only $\theta(1)$ time per level to combine results and move them up. □

**6.4. Optimality.** In this section we show that some of our results are quite close to being optimal.

PROPOSITION 8. *In a pyramid computer of size n, the time needed to move $B \geqq 1$ bits of data from the first column of the base to the last column of the base is $\Omega(\log(n) + (B/\log(n))^{1/2})$.*

*Proof.* Assume $B \geqq \log_2(n)$, and let $L = \lfloor \log_4(n * \log_2(n)/B) \rfloor$ and $E = n^{1/2}/2^L$. For each column of PEs at level $L$, call the entire column and all of its descendents a *prism.* The data initially resides in the leftmost prism and must move to the rightmost one. If a bit only moves along communication links involving PEs at level $L$ or below, then at least $E-1$ communication links must be traversed, since there are $E$ prisms, and each communication link either keeps the bit in the same prism or moves it to an adjacent one.

Figure 6 shows a side view of the PEs at level $L$ and above. The usual way of drawing the pyramid has been slightly altered so that all PEs in the same column and level are represented by a single PE. The time spent traversing any vertically drawn wires (communication links) will be ignored. The weights along all other wires indicate the number of steps that could be saved by using the wire. There are $(E/2) * (E/2-1)$ horizontal wires labeled 1, $(E/4)^2 * 2$ slanted wires labeled 1, $(E/4) * (E/4-1)$ horizontal wires labeled 3, and so forth. Since each wire can carry at most $C * \log_2(n)$ bits per unit time, for some constant $C$, in 1 unit of time the maximum number of bits moved by the nonvertical wires above level $L$ is

$$C * \log(n) * \left[ \sum_{i=1}^{\log_4(n)-L} (2^i - 1)(E^2/4^i - E/2^i) + \sum_{i=1}^{\log_4(n)-L-1} 2(2^i - 1)(E^2/4^{i+1}) \right],$$

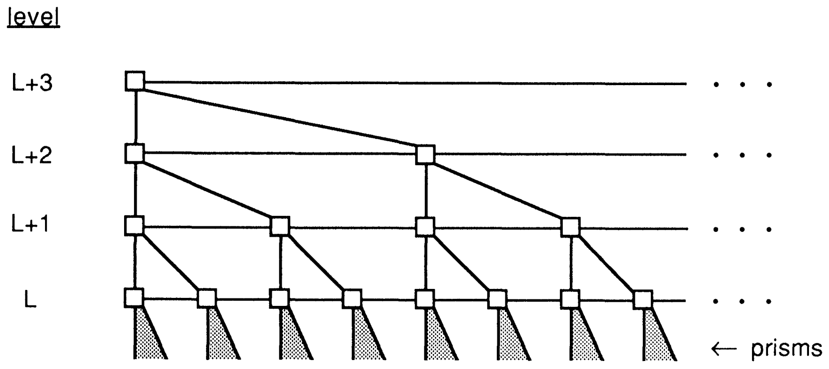


FIG. 6. *Another view of the pyramid computer.*

which is less than $C * E^2 * \log_2(n)$. Therefore, in $t$ units of time the total savings is less than $C * E^2 * t * \log_2(n)$.

On the other hand, a bit of data that reaches the rightmost prism in $t$ units of time must have crossed wires with a total weight of at least $E-1-t$. Furthermore, if all $B$ bits of data reach the rightmost prism in $t$ units of time, the total savings must be at least $B*(E-1-t)$. Therefore, $t$ must be such that

$$B*(E-1-t) < C * E^2 * t * \log_2(n),$$

or

$$t > B*(E-1)/(C * E^2 \log_2(n) + B) = \theta((B/\log(n))^{1/2}).$$

Since the pyramid computer of size $n$ has a diameter (maximum distance between any two vertices) of $2 * \log_4(n)$, we also have $t \geqq \log(n)$. Hence we have the desired result. □

For each of the problems considered in this paper, it is easy to devise inputs to which Proposition 8 applies. For example, suppose one is performing component labeling of digitized pictures, and the input is known to be of the form in Fig. 7, where an $X$ indicates a pixel which may or may not be black and a $Y$ indicates a pixel that

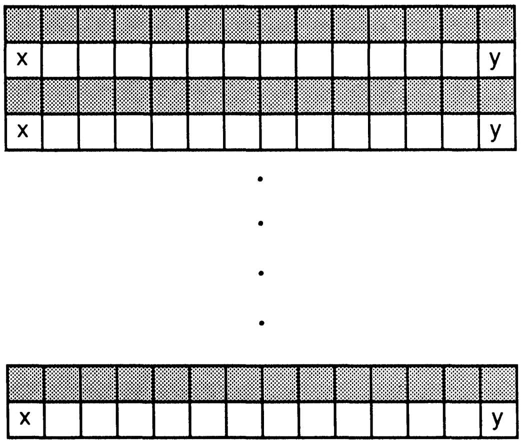


FIG. 7. *An image requiring extensive data movement.*

is always white. Notice that if the two black PEs neighboring a PE marked $Y$ end up with the same label, then the PE marked $X$ which is in $Y$'s row must be black. Since a $Y$ can determine if its black neighbors have the same label in $\theta(1)$ time after the component labeling algorithm is finished, the component labeling algorithm requires at least as much time as it takes to transmit $\theta(n^{1/2})$ bits from one edge to the opposite edge of the pyramid. By Proposition 8 this is $\Omega(n^{1/2}/\log^{1/2}(n))$. This lower bound is a factor of $\log^{1/2}(n)$ smaller than the time achieved in Theorem 5, showing that Theorem 5 is at most $\log^{1/2}(n)$ times optimal. We believe that Theorem 5, as well as Theorems 3, 4 and 6, and Corollary 2, are all optimal, and leave the proof of this as an open problem.

**7. Extensions to other dimensions.** Many of the data movement techniques that have been presented can be extended to pyramid computers of dimensions other than 2. Define a $j$-MCC *of size n* to be an SIMD machine in which $n$ PEs are arranged in a $j$-dimensional cubic lattice. PE$(i(1),\cdots,i(j))$ is connected to PE$(k(1),\cdots,k(j))$, provided that $\sum_{l=1}^{j}|i(l)-k(l)|=1$.

A $j$-PC *of size n* is an SIMD machine viewed as a full $2^j$-ary tree with additional horizontal links. The base of the $j$-PC of size $n$ is a $j$-MCC of size $n$. Each level of the pyramid is a $j$-MCC with $1/2^j$ as many PEs as the previous level. A PE at level $i$ is connected to its neighbors (assuming they exist): $2*j$ adjacent PEs at level $i$, $2^j$ children at level $i-1$, and a parent at level $i+1$.

In our analyses of algorithms for $j$-MCCs and $j$-PCs, our times consider $j$ fixed and $n$, $m$, $i$ and S as the parameters. We do not determine the full dependence on $j$, e.g., whether each step really needs $2^j$ comparisons instead of a fixed number, since a PE in a $j-$MCC is fundamentally different from one in a $k$-MCC for $j\neq k$. In this convention, sorting can be performed on a $j$-MCC of size $n$ in $\theta(n^{1/j})$ time [33]. Since the MCC RAR and the MCC RAW are implemented via sorting, we can

(1) Perform an RAR in $\theta(n^{1/j})$ time on a $j$-MCC of size $n$,

(2) Perform an RAW in $\theta(n^{1/j})$ time on a $j$-MCC of size $n$.

We can also extend some of the PC data movement techniques to the $j$-PC. We first consider the case $j>1$, and discuss the case $j=1$ separately at the end.

(1) $j$-PC *write*: Given that the destination level is a $j$-MCC of size $k$ and the source PEs are $i-1$ levels below, a $j$-PC write $(j>1)$ can be performed in $\theta(\log(n)+[k*i^{j-1}]^{1/j})$ time.

(2) $j$-PC *read*: Given that the source level is a $j$-MCC of size $k$ and the destination PEs are $i-1$ levels below, a $j$-PC read $(j>1)$ can be performed in $\theta(\log(n)+[k*i^{j-1}]^{1/j})$ time.

(3) *Count keys*: In a $j$-PC of size $n$ $(j>1)$, if there are $t$ different keys in the base then count_keys will finish in $\theta(\log(n)+[t+t*\log(n/t)^{j-1}]^{1/j})$ time.

(4) *Funnel read*: Assume that the last stage ($j$-dimensional) data cubes are of size $S$. (See the comments preceding Theorem 9 below for a discussion of the sizes of data cubes in a $j$-PC.) Then a $j$-PC funnel read $(j>1)$ can be performed in $\theta(S^{1/j})$ time.

(5) *Reducing a function*: Given sets $Q$ and $R$, stored 1 per PE in a $j$-MCC of size $m$ at level $i$, and given that $g$ and $*$ can be computed in $\theta(1)$ time, in $\theta(m^{1/j}+m/2^{ij})$ time a $j$-PC $(j>1)$ can compute the reduction of $g$.

We have omitted a discussion of the pyramid matrix read and write since the mapping of a matrix into a $j$-MCC is not natural. Given the data movement techniques that do extend, we are able to adapt several of our algorithms to the $j$-PC:

THEOREM 7. *Given a j-dimensional pyramid computer of size n $(j>1)$, if the base contains the unsorted edges of an undirected graph with v vertices, then the extension of* *the component labeling algorithm in* § 3.2 *labels the components in* $\theta(\log(n)+v^{1/j}[1+\log(n/v)^{j-1}]^{1/j})$ *time.*

THEOREM 8. *Given a j-dimensional pyramid computer of size n* $(j>1)$, *if the base contains the unsorted weighted edges of an undirected graph with v vertices, then the extension of the algorithm in* § 3.3 *finds a minimal spanning forest in* $\theta(\log(n)+v^{1/j}[1+\log(n/v)^{j-1}]^{1/j})$ *time.*

COROLLARY 4. *Given a j-dimensional pyramid computer of size n* $(j>1)$, *if the base contains the unsorted edges of an undirected graph G with v vertices, then in* $\theta(\log(n)+v^{1/j}[1+\log(n/v)^{j-1}]^{1/j})$ *time one can*

(a) *decide if G is bipartite,*
(b) *determine the cyclic index of G,*
(c) *find all bridge edges of G,*
(d) *find all articulation points of G,*
(e) *decide if G is biconnected.*

In the component label problem with digitized $j$-dimensional picture input, we again use stages of ($j$-dimensional) picture cubes and their associated data cubes, where the picture cubes increase in size by a factor of $2^j$ at each stage. We reduce a picture cube to an amount of data proportional to the cube's surface area, so the data squares at the final stage have $\theta(n^{(j-1)/j})$ PEs, and are on level $\theta((j-1)*\log_2(n))$. The extension of component labeling for such data is now straightforward.

THEOREM 9. *Given a digitized j-dimensional picture stored one pixel per* PE *at the base of a j*-PC *of size n* $(j>1)$, *the components can be labeled in* $\theta(n^{(j-1)/j^2})$ *time.*

Despite our success in extending component labeling to the $j$-PC, we cannot do as well for the nearest neighbor problem. The difficulty is that, since the final data cubes have size $\theta(n^{(j-1)/j})$, when we try reducing a function it takes $\theta(n^{(j-2)/j})$ time for $j>2$, which is no better than the edgelength of the base $(n^{1/j})$. Therefore our extension to a $j$-PC does not seem to be able to do any better than a $j$-MCC algorithm for the nearest neighbor problem.

Finally, a 1-PC behaves differently than a $j$-PC for $j>1$. One important difference is that a 1-PC of size $n$ can sort in $\theta(n/\log(n))$ time, versus $\theta(n)$ time for a 1-MCC of size $n$, while for $j>1$ a $j$-PC and a $j$-MCC sort in the same time [25]. This sorting difference is most apparent when considering problems such as component labeling, finding a minimal spanning forest, deciding if a graph is bipartite, etc., for a graph with $\theta(n)$ vertices, given as unsorted edge input. For a $j$-PC, $j>1$, these problems take $\theta(n^{1/j})$ time, but on a 1-PC they can be solved in $\theta(n/\log(n))$ time.

Another important difference is that one-dimensional digitized pictures are simplistic. Using the special properties of such input, there are easy 1-PC algorithms to do component labeling and finding nearest neighbors in $\theta(\log(n))$ time.

## 8. Conclusion.

Because of its similarity to some animal optic systems, its similarity to the (region) quadtree structure, and its natural use in multiresolution image processing, the pyramid computer has long been suggested for low-level image processing [5], [6], [25], [29], [30], [32], [34], [35]. Due to advances in technology, some pyramid computers are currently under construction [8], [12], [23], [30], leading us to believe that they deserve further algorithm study. This paper begins that study by showing that the pyramid computer can be used for more complex tasks than originally considered. For instance, we have shown that the pyramid computer can be used to efficiently solve problems in higher-level image processing, graph theory, and digital geometry.

A major contribution of this paper is the introduction of fundamental data movement operations for the pyramid computer to be used with a variety of standard input formats. These data movement operations are quite different from those used by earlier authors, in that most previous pyramid computer algorithms either concentrated soley on the child-parent links (adapting quadtree algorithms to the pyramid), or solely on the mesh-connected links of the base. In contrast, the data movement operations that we have presented intermingle the use of both types of links. They also make extensive use of intermediate levels of the pyramid to do calculations, store results and communicate data. Furthermore, we have shown how to use the base of the pyramid to aid in the computation of functions being performed at higher levels.

We have used our data movement operations to efficiently solve several geometric and graph-theoretic problems. Since there are numerous other problems in these and related fields which have divide-and-conquer solutions, the problem-solving techniques and data movement operations presented here should have a wide range of applications. For example, [15], [17] contain algorithms which use the pyramid read, pyramid write, and reducing a function operations to compute geometric properties such as convexity, diameter and smallest enclosing box.

It is interesting to compare the pyramid computer with other parallel architectures. Using the standard VLSI model in which processing elements are separated by at least unit distance and a wire has unit width, [5] has shown that a pyramid computer with a base of $n$ processing elements can be laid out in $\theta(n)$ area by a simple modification of the standard "*H tree*" layout scheme. The space of a layout for an interconnection scheme is one measure of its cost, as is the regularity of the layout. A mesh-connected computer of $n$ processing elements also requires only $\theta(n)$ with an extremely regular layout, but because it has a communication diameter of $\theta(n^{1/2})$ it requires $\Omega(n^{1/2})$ time to solve all of the problems considered here, compared with, say, the $\theta(n^{1/4})$ time needed by the pyramid computer to label the connected components of an image. (Mesh-connected computer algorithms taking $\theta(n^{1/2})$ time to solve problems presented in this paper appear in [19], [16].)

Another simple model that can be easily laid out in $\theta(n)$ area is the quadtree machine, which is simply a pyramid computer without the nearest neighbor links. Like the pyramid, the quadtree has a logarithmic communication diameter, but unlike the pyramid, the apex often acts as a bottleneck. For example, it is easy to show that the quadtree needs $\Omega(n^{1/2})$ time to label components or find nearest neighbors of an image, even if higher PEs have additional memory (as suggested in [1]). On the pyramid, we have used nearest neighbor connections at the intermediate levels to circumvent this bottleneck.

General-purpose interconnection schemes such as the shuffle-exchange, butterfly and cube-connected cycles can be used to provide poly-log time solutions to all the problems considered herein. (An algorithm is poly-log if it finishes in $P(\log^k(n))$ time for some constant $k$.) Unfortunately, these interconnection schemes require area that is nearly proportional to the square of that required to lay out the pyramid computer [36]. Although this extra area and complexity provide the capability of poly-log sorting, it is more than is needed for the problems considered here.

A more interesting model is the orthogonal trees or mesh of trees [36]. This model has a mesh-connected base of size $n$ augmented so that each row and column of the base mesh has a binary tree over it, with these trees being disjoint except at their leaves. In this model $\theta(n^{1/2}\log^2(n))$ bits can be moved from the leftmost $\log(n)$ columns to the rightmost $\log(n)$ columns in $\theta(\log(n))$ time. This is a significant improvement over the pyramid computer bound in Proposition 8, though not enough to provide poly-log sorting. This machine model has not received much consideration as an image processing machine, but for all of the problems considered herein involving images or adjacency matrices, orthogonal trees can solve them in poly-log time.

Orthogonal trees do have some drawbacks, however. While the pyramid computer can be laid out in linear area, orthogonal trees need a factor of $\log^2(n)$ more area [36]. Further, orthogonal trees seem to have few ties to other objects of interest for researchers in image processing, as opposed to the neural, data structure, and multi-resolution ties mentioned above for the pyramid computer. It is because of these ties that the image processing community is building pyramids and not orthogonal trees. Additional models which are closer to the pyramids, and which solve all of the image processing problems considered herein in poly-log time, have recently been suggested [28]. These models were designed by starting with the pyramid computer and modifying it to be much faster on the algorithms presented here.

**Acknowledgment.** The authors would like to express their appreciation to the referees for their constructive comments.

## REFERENCES

[1] N. AHUJA AND S. SWAMY, *Multiprocessor pyramid architecture for bottom-up image analysis*, IEEE Trans. PAMI, PAMI-6 (1984), pp. 463-474.

[2] M. J. ATALLAH AND S. R. KOSARAJU, *Graph problems on a mesh-connected processor array*, J. Assoc. Comput. Mach. 3 (1984), pp. 649-667.

[3] F. Y. CHIN, J. LAM AND I.-N. CHEN, *Efficient parallel algorithms for some graph problems*, Comm. Assoc. Comput. Mach., 25 (1982), pp. 659-665.

[4] C. R. DYER, *A quadtree machine for parallel image processing*, Tech. Report KSL 51, U. of Illinois at Chicago Circle, Chicago, IL, 1981.

[5] ———, *A* VLSI *pyramid machine for hierarchical parallel image processing*, Proc. PRIP, 1981, pp. 381-386.

[6] ———, *Pyramid algorithms and machines*, in Multicomputers and Image Processing Algorithms and Programs, K. Preston and L. Uhr, eds., Academic Press, New York, 1982, pp. 409-420.

[7] C. R. DYER AND A. ROSENFELD, *Parallel image processing by memory augmented cellular automata*, IEEE Trans. PAMI, PAMI-3 (1981), pp. 29-41.

[8] G. FRITSCH, W. KLEINOEDER, C. U. LINSTER AND J. VOLKERT, EMSY85—*The Erlanger multiprocessor system for a broad spectrum of applications*, Proc. 1983 Internat. Conf. on Parallel Processing, 1983, pp. 325-330.

[9] S. E. HAMBRUSCH, VLSI *algorithms for the connected component problem*, this Journal, 12 (1983), pp. 354-365.

[10] S. E. HAMBRUSCH AND J. SIMON, *Solving undirected graph problems on* VLSI, Tech. Rep. CS-81-23, Computer Science, Penn. State Univ., State College, PA, 1981.

[11] D. S. HIRSCHBERG, A. K. CHANDRA AND D. V. SARWATE, *Computing connected components on parallel computers*, Comm. Assoc. Comput. Mach., 22 (1979), pp. 461-464.

[12] S. LEVIALDI, *A pyramid project using integrated technology*, in Integrated Technology for Parallel Image Processing, S. Levialdi, ed., Academic Press, New York, 1985, to appear.

[13] R. MILLER, *Writing* SIMD *algorithms*, Proc. 1985 Internat. Conference on Computer Design: VLSI in Computers, 1985, pp. 122-125.

[14] R. MILLER AND Q. F. STOUT, *Computational geometry on a mesh-connected computer*, Proc. 1984 Internat. Conf. on Parallel Processing, 1984, pp. 66-74.

[15] ———, *Convexity algorithms for pyramid computers*, Proc. 1984 Internat. Conf. on Parallel Processing, 1984, pp. 177-184.

[16] ———, *Geometric algorithms for digitized pictures on a mesh-connected computer*, IEEE Trans. PAMI, PAMI-7 (1985), pp. 216-228.

[17] ———, *Pyramid computer algorithms for determining geometric properties of images*, Proc. 1985 ACM Symposium on Computational Geometry, 1985, pp. 263-277.

[18] D. NASSIMI AND S. SAHNI, *Finding connected components and connected ones on a mesh-connected parallel computer*, this Journal, 9 (1980), pp. 744-757.

[19] ———, *Data broadcasting in* SIMD *computers*, IEEE Trans. Comput. C-30 (1981), pp. 101-107.

[20] A. P. REEVES, *On efficient global feature extraction methods for parallel processing*, Comput. Graphics Image Processing, 14 (1980), pp. 159-169.

[21] B. SAKODA, *Parallel construction of polygonal boundaries from given vertices on a raster*, Tech. Rep. CS81 1-21, Computer Science, Penn. State Univ., State College, PA, 1981.

[22] C. SAVAGE AND J. JA'JA', *Fast, efficient parallel algorithms for some graph problems*, this Journal, 10 (1981), pp. 682-691.

[23] D. H. SCHAEFER et al., *The* PMMP—*a pyramid of* MPP *processing elements*, Proc. of the 18th Annual Hawaiian Internat. Conf. on Systems Science, 1, 1985, pp. 178-184.

[24] Q. F. STOUT, *Drawing straight lines with a pyramid cellular automaton*, Inform. Proc. Lett., 15 (1982), pp. 233-237.

[25] ———, *Sorting, merging, selecting, and filtering on tree and pyramid machines*, Proc. 1983 Internat. Conf. on Parallel Processing, 1983, pp. 214-221.

[26] ———, *Pyramid computer solutions of the closest pair problem*, J. Algorithms, 6 (1985), pp. 200-212.

[27] ———, *Tree-based graph algorithms for some parallel computers*, Proc. 1985 Internat. Conf. on Parallel Processing, 1985, pp. 727-730.

[28] ———, *Mesh and pyramid computers inspired by geometric algorithms*, Proc. Workshop on Algorithm-Guided Parallel Architectures for Automatic Target Recognition, 1985, pp. 293-315.

[29] S. L. TANIMOTO, *Programming techniques for hierarchical parallel image processors*, in Multicomputers and Image Processing Algorithms and Programs, K. Preston and L. Uhr, eds., Academic Press, New York, 1982, pp. 421-429.

[30] ———, *Towards hierarchical cellular logic: Design considerations for pyramid machines*, Tech. Rep. 81-02-01, Computer Science, Univ. Washington, Seattle, WA, 1981.

[31] ———, *Sorting, histogramming, and other statistical operations on a pyramid machine*, Tech. Rep. 82-08-02, Computer Science, Univ. Washington, Seattle, WA, 1982.

[32] S. L. TANIMOTO AND A. KLINGER, *Structured Computer Vision: Machine Perception through Hierarchical Computation Structures*, Academic Press, New York, 1980.

[33] C. D. THOMPSON AND H. T. KUNG, *Sorting on a mesh-connected parallel computer*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 263-271.

[34] L. UHR, *Layered 'Recognition Cone' networks that preprocess, classify, and describe*, IEEE Trans. Comput., C-21 (1972), pp. 758-768.

[35] ———, *Algorithm-Structured Computer Arrays and Networks*, Academic Press, New York, 1984.

[36] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Baltimore, MD, 1984.

[37] F. L. VAN SOY, *The parallel recognition of classes of graphs*, IEEE Trans. Comput., C-29 (1980), pp. 563-570.