Parallel image component labeling for target acquisition

Vipin Chaudhary
Wayne State University
Department of Electrical and Computer Engineering
Parallel and Distributed Computing Laboratory
Detroit, Michigan 48202
E-mail: vipin@eng.wayne.edu

Jake K. Aggarwal
University of Texas at Austin
Department of Electrical and Computer Engineering
Computer Vision Research Center
Austin, Texas 78712-1084

Abstract. An important step in target acquisition is to be able to label various unconnected objects (components) in the scene. We present new algorithms for labeling connected components in a binary image. Eight connectivity for object and background pixels is assumed. The sequential algorithm is described first. The computational complexity of the algorithm is linear (optimal) in the number of object pixels in the image. The memory requirement is also linear in the number of object pixels. The representation of the connected components used in the algorithm makes it easy to calculate certain properties of regions, i.e., area, perimeter, etc. The algorithm is then parallelized and implemented on a shared memory computer. The computational complexity of this parallel algorithm is $O(\lceil \log n \rceil)$ for an image with n object pixels using n processors, and is the best achieved yet. The implementations of the algorithm on several distributed memory architectures, i.e., a binary tree connected computer $[O(\log n)]$, a unidirectional and bidirectional mesh connected computer $[O(n^{1/2})]$, and a hypercube computer $[O(\log n)]$ are discussed. The computational and communicational complexities of these implementations are computed, and are the best yet achieved. The algorithm is easily extended for gray-level images without affecting the complexities. Results of the implementation on the sequent balance multiprocessor are presented. [S0091-3286(98)00307-9]

Subject terms: parallel; component labeling; PRAM; run-length encoding; shared memory multicomputer; mesh-connected computer; hypercube.

Paper TAM-03 received Feb. 3, 1998; revised manuscript received Feb. 15, 1998; accepted for publication Feb. 19, 1998.

1 Introduction

To analyze a picture of a scene, we must identify each object to describe the complete scene. This identification involves assigning unique labels to disjoint objects. Labeling the objects is a fundamental task in computer vision, since it forms a bridge between low-level image processing and high-level symbolic processing.[1]

The problem has been studied for over three decades. One of the first known algorithms[2] performed two passes over the image. In the first pass, labels were generated for object pixels and stored in an equivalence table. During the second pass, these labels were replaced by the smallest equivalent labels. The memory requirement and computational complexity for evaluating the equivalent label were very high for images with complex regions. Lumia et al.[3] reduced the size of the equivalence table by reinitializing the table for each scan line. This algorithm also required two passes. Haralick[4] proposed an algorithm that did not require an equivalence table but, instead, required the repeated propagation of labels in the forward and backward directions. This technique improved on the memory requirement but required a large number of passes through the image for complex objects. Mandler and Oberlander[5] presented a one-pass algorithm to generate border line chain codes for each component and also provided a list of adjacent regions for each region. The drawback of this algorithm is the computationally expensive corner detection. Samet and Tamminen[6] gave an algorithm applicable to hierarchical data structures, such as quadtrees, octrees, and in general bintrees. The authors used a very small equivalence table and the processing cost was linear in the size of the image. Several other techniques to accomplish related problems with varying degrees of accuracy and computation time involved were presented.[7,8]

Most computer vision tasks require an enormous amount of computation, demanding high-performance computers for practical real-time applications. Parallelism appears to be the only economical way to achieve this level of computation. Hirschberg et al.[9] gave one of the earlier formulations of a parallel approach to connected component labeling. Given a graph representation with n nodes, this technique required $n^2/\log n$ processors with shared memory in an $O(\log^2 n)$ algorithm. Nassimi and Sahni[10] presented an algorithm on an $n \times n$ mesh connected computer that labeled the components in an $n \times n$ image in $O(n)$ time. Tucker[11] presented a parallel divide-and-conquer algorithm on a simple tree connected single instruction multiple data (SIMD) computer, which required $O(n^{1/2})$ time for an image of size n. Cypher et al.[12] suggested an exclusive-read exclusive-write parallel random access machine (EREW PRAM) algorithm with a complexity of $O(\log n)$ and hypercube and shuffle-exchange algo-

 0091-3286/98/$10.00

rithms with a complexity of $O(\log^2 n)$ for an image of size n (Refs. 13 and 14). These algorithms apply only to binary images since the concept of boundary would otherwise be imprecise. They also assume four connectivity for pixels. The number of processors used for the EREW PRAM algorithm is $3n+2n^{1/2}$ for an image of size n, thus, putting a very low bound on the processor efficiency. Sunwoo et al.[15] also implemented a parallel algorithm on a hypercube whose complexity was linear in the size of the image. Agarwal and Nekludova[16] presented an algorithm on an EREW PRAM with a complexity of $O(\log n)$. This algorithm is described for hex-connected images. Little et al.[17] gave parallel algorithms on the connection machine, which took $O(\log n)$ router cycles for an image of size n. Maresca and Li[18] presented an algorithm that labeled components in an $n\times n$ image in $O(n)$ time on a polymorphic torus. Manohar and Ramapriyan[19] gave another algorithm on a mesh connected computer with a complexity of $O(n \log n^2)$ for an $n\times n$ image. Woo and Sahni[20] compare the performances of the connected components algorithm for various partitions and mappings of the data. Their algorithms have a complexity of $O(n^2)$ for an image of size n. Alnuweiri and Kumar[21] survey several parallel algorithms and architectures for image component labeling. Bekhale and Banerjee[22] implement their algorithm on a hypercube machine. They look at the problem as occurring in very large scale integration (VLSI) circuit extraction. Hsu et al.[23] present theoretical computational complexities of various algorithms.

Most of the preceding algorithms are not practical. We look at the algorithm from a computer vision point of view. In most computer vision tasks, the input is either directly from a camera (accessible usually to one processor) or is an image stored in a computer. In both cases, the image is usually compressed in run length encoding (our input specification). Since the input image is accessible to only one processor, it needs to be split into subimages and distributed to the various processors. Almost all other algorithms assume that the initial image resides in all processors (or the subimages reside in the processors). Furthermore, in a typical computer vision application, the image, with the connected components marked, is further processed to recognize these components. This calls for merging the subimages to one processor. This overhead is also ignored by most other work (the final connected components are distributed among the processors). The output representation of the components we use is very desirable and effective for the next level of computation.

We present a new sequential algorithm for connected component labeling with a worst-case computational complexity that is linear in the number of objects (or more precisely, the number of object pixels) in the image. Most previous algorithms have the disadvantage that their complexity is a function of both the object and the background pixels. The memory requirement of our algorithm is also linear in the number of objects. The algorithm is then parallelized and implemented on the sequent balance shared memory multicomputer. The computational complexity of our new parallel algorithm on a shared memory computer is $O(\lceil \log n \rceil)$ for an image of size n. Unlike other parallel algorithms for connected component labeling, which are very specific to the architecture, the algorithm described in this paper applies to a very general class of architectures.

For implementation of the algorithm on distributed memory architectures, we compute the communicational complexity, along with the computational complexity. We evaluate the complexities for a binary tree connected computer, a unidirectional and a bidirectional mesh connected computer, and a hypercube connected computer. Complexities obtained for these architectures are the best yet achieved. The communication overhead for the distributed memory computers is well illustrated by the communicational and computational complexities. Because of the dominance of the communicational complexity, the efficiency of processors decreases with an increase in the number of processors.

This paper is organized as follows. We first introduce the concept of connected component labeling and the notation and definitions used in our description. This is followed by a description of our data structure, the sequential algorithm, the complexity of the sequential algorithm, and the parallel algorithm and its complexity evaluation. Next, we discuss parallelizing and implementing the preceding algorithm on shared memory architectures and present a complexity evaluation. The implementations of the algorithm on various distributed memory computers and their computational and communicational complexities are discussed next. Then we present the results of implementing the algorithm on the sequent balance computer and conclude the article. Appendix A lists the sequential algorithm in detail, and Appendix B provides evaluations of certain communicational complexities.

2 Connected Component Labeling

This section describes the sequential and parallel algorithms for connected component labeling and their complexity evaluation. We use certain assumptions about the input image for our algorithm. First, it is a binary image with object pixels represented as runs.[1] This representation is very economical and often used for image compression. We assume an eight connectivity for all the pixels.

2.1 *Notation*

Henceforth, we denote object pixels by pixels, unless they are ambiguous. A conexon (also path in Ref. 1) of length n from pixel P to pixel Q is a sequence of pixel points $P=P_0,P_1,\ldots,P_n=Q$ such that P_i is a neighbor (an eight connectivity) of P_{i-1}, $1\leqslant i\leqslant n$. We say that pixel P is connected to pixel Q if a conexon exists from P to Q. This is a conexon^k if it lies within rows $1\ldots k$. A maximal set of connected pixels within rows $1\ldots k$ (at least one of which is in row k) forms the object^k. The maximal set of connected pixels in row k forms a rep^k. The set of rep^k belonging to the same object^k forms the part^k. A touch^k is a rep^k that connects two or more rep^{k-1}, at least two of which are not in the same part^{k-1}. These definitions are illustrated by an example in Fig. 1.

2.2 *Input/Output*

For the sake of simplicity, we assume the input to be in the run length representation[1] from left to right and top to bot-

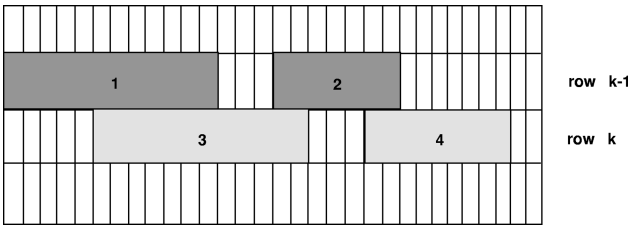


Fig. 1 Example to illustrate the notation. The reps 3 and 4 are rep^k. Reps 1, 2, 3, and 4 form an object^k. Reps 3 and 4 collectively form the part^k. Rep 3 is a touch^k that connects two rep^{k-1} (reps 1 and 2), which are not in the same part^{k-1}.

tom. This does not affect the generality of the algorithm, since it is trivial to convert image data (a 2-D array) to the run length representation. The format of the input to the algorithm is as follows:

1. If a row has at least one rep in it, then it is a sequence of integers $(r,n,l_1,r_1,\ldots,l_n,r_n]$, where r is the row number; n is the number of reps in this row; and l_i,r_i for $1\leqslant i\leqslant n$ are the left and right column numbers of the ith reps in row r, respectively.
2. If there are no reps in the row, then the sequence for that row is null.

The output of the algorithm is a linked list of the various connected components in the input image. Each connected component is, in turn, a linked list of the reps it comprises.

2.3 *Data Structure*

There are two basic structures in this algorithm, namely reps and objects. Linked lists are used to represent both these concepts. A third linked list, component, is used to represent the output—the list of connected components in the image. The reps in a row are assumed to be in order from left to right, i.e., the leftmost rep occurs first and the rightmost rep occurs last. Each rep^k is represented as a structure consisting of the following fields:

- *left*—the starting column number of rep^k
- *right*—the column number where rep^k ends
- *row*—the row number of this rep^k
- *repnext*—a pointer to the neighboring rep^k
- *repcomp*—a pointer to the neighboring rep^k (to be used in the parallel algorithm)
- *object*—a pointer to the object^k connected to this rep^k.

Each object^k is also a linked list of structures having the following fields:

- *leftmost*—a pointer to the leftmost rep^k associated with this object^k
- *end*—a pointer to the end of the linked list of reps from previous rows that are connected to this object^k
- start—a pointer to the start of the linked list of reps in previous rows that are connected to this object^k
- *label*—a positive integer assigned to this object^k
- *objnext*—a pointer to the next object^k in the linked list
- touch—a Boolean set to true if a rep^k is found to be connected to object^{k-1}.

''Component'' is a linked list of structures with the following fields:

- *compo*—a pointer to the first rep belonging to this component
- *compnext*—a pointer to the next component in the linked list.

2.4 *Properties*

1. A rep^k L is connected to a rep^{k-1}M if $L\rightarrow$right $\geqslant M\rightarrow$left-1 AND $L\rightarrow$left$\leqslant M\rightarrow$right$+1$.
2. A rep^k is connected to a part^{k-1} if it is connected to one of the rep^{k-1} in that part^{k-1}.
3. A rep^k is connected to an object^{k-1} if it is connected to the part^{k-1} of object^{k-1}.
4. If a conexon is found between objects A and B in row k, and if $A\rightarrow$leftmost$<B\rightarrow$leftmost then $B\rightarrow$label$=A\rightarrow$label.
5. Two rep^k A and B are neighbors if no rep^k C exists such that $A\rightarrow$left$<C\rightarrow$left$<B\rightarrow$left.

2.5 *Sequential Algorithm*

The sequential algorithm is conceptually very simple. Every rep is associated with an object. If two reps are connected, they both point to the same object, i.e., the object with the smaller label. All the reps pointing to the same object form a component. We now describe the major procedures of the algorithm. For the detailed algorithm, see Appendix A.

```
procedure COMPONENT()
begin
      INITIALIZE();
      while not(EOF) do
            MAKELISTS();
            CONNECTIONS();
            REGIONS();
            CHANGELABEL();
            JOIN();
      od
      PRINTOUT(comps);
end COMPONENT;
```

The procedure COMPONENT is the main routine of this algorithm and invokes all other procedures, directly or indirectly. The procedure MAKELISTS reads the sequence of integers in each rep and forms a linked list of structures which make up the rep^k. The left and right ends of column numbers of the rep and the row number are inserted into the corresponding fields. The rep^k in the input are ordered according to their position in the row, with the leftmost appearing first and the rightmost appearing last. The repnext field points to the next rep (structure). The object^{k-1} are stored in a separate linked list and the objnext field points to the next object.

The conexons between rep^k and rep^{k-1} are found by the procedure CONNECTIONS. When a conexon is found, the rep^{k-1} has the object^{k-1} marked as current and the rep^k is said (touch=true) to be connected to the object^{k-1} of rep^{k-1}. The procedure CONNECTED takes two reps as input parameters and checks to see if they are connected. When a rep^k is found to be connected to two neighboring rep^{k-1}, these neighboring object^{k-1} must be joined. Their labels are updated by the procedure LABELLING.

Procedure LABELLING takes two labels as input parameters and assigns them the label of the object^{k-1} that is leftmost and connected to this object^{k-1} by a conexon^k.

The procedure REGIONS, as the name suggests, looks for regions in the input image. It traverses the object^{k-1} linked list, looking for any object^{k-1} that were not found to be connected to any rep^k. Every object^{k-1} satisfying the preceding forms a connected component. The linked list of components compos points to this connected component.

Procedure CHANGELABEL computes the label for the object^{k-1}. Then, it traverses the linked list rep^k, assigning new label to all the rep^k that are not connected to object^{k-1}, and putting label to the rep^k that are connected to object^{k-1}. The leftmost field of each object is assigned the leftmost rep^k that touches the object^k.

JOIN traverses through the linked list of objects looking for object^{k-1} to be joined with other object^{k-1}. The linked lists of rep^k are then combined.

2.6 *Complexity Analysis of the Sequential Algorithm*

Let t be the total number of reps of object pixels and r be the number of rows in the image.

Lemma 1. $\forall t,\ r \leqslant t$.

Proof. In the worst case, there will be, at most, one rep in any row. In our representation, if there are no reps in a row, then that row does not exist. Thus, the total number of rows cannot exceed the number of reps of object pixels in the image.

Theorem 1. The time complexity of the procedure COMPONENT is $O(t)$.

Proof. The while loop in COMPONENT is executed once for each row—r times and $r \leqslant t$. For every row, most procedures are executed once. We find a bound on the number of times each statement is executed in each of these procedures.

The procedure INITIALIZE is executed once and it invokes procedures CREATOBJ and CREATREP a constant number of times. Thus, the number of executions in this procedure is constant.

Consider the k'th iteration of COMPONENT. In MAKELISTS, the iterations due to the for loop equal the number of rep^k.

In CONNECTIONS, the outer while loop is executed rep^{k-1} times. So, all the instructions, except those within the inner while loop, are executed rep^{k-1} times. The variable nrep is set before the outer while loop and the condition of the inner while loop is satisfied at most rep^k times. Hence, the number of executions of statements in CONNECTIONS is bound by rep^{k-1}+rep^k.

The while loop in REGIONS is executed object^{k-1} times. Since the number of object^{k-1} is at most rep^{k-1}, the executions are bound by rep^{k-1}.

There are two loops in CHANGELABEL. The variable joint is incremented each time rep^{k-1} and its right neighbor are found to be connected to the same rep^k in CONNECTIONS. Hence, the value of joint is at most rep^{k-1}. This sets a bound on the execution of the for loop. The while loop is executed rep^k times and, hence, the bound on the number of executions in this procedure is the maximum of rep^k and rep^{k-1}.

The while loop in JOIN is executed once for each object^{k-1} and object^k. Hence, the number of executions are bound by the sum of rep^k and rep^{k-1}.

The statements in LABELLING and CONNECTED are executed only once. Thus, we have an upper bound on the number of executions of statements in the algorithm on the k'th iteration of COMPONENT, that is, the sum of rep^k and rep^{k-1}. This gives us an upper bound on the number of executions for COMPONENT: $2r-1$. Hence, by Lemma 1, the complexity of COMPONENT is $O(t)$.

2.7 *Parallel Algorithm*

The parallel algorithm is conceptually very simple. The procedure PARALLEL_LABEL is the main routine of this algorithm and invokes all other procedures, directly or indirectly. The input image is split into subimages, one for each processor. This division of the image is done statically such that the subimages are approximately the same size. The subimages are then distributed to the various processors. Then, each processor labels the components of its corresponding subimage using the sequential image component labeling procedure (i.e., COMPONENT). Next, the components of the subimages are merged to obtain the components of the entire image.

```
procedure PARALLEL_LABEL()
begin
    SPLIT_IMAGE();
    DISTRIBUTE_IMAGE();
    do in parallel for all processors
        COMPONENT();
    od
    MERGE_COMPONENTS();
end PARALLEL_LABEL;
```

The input image is divided into approximately equal size subimages by the procedure SPLIT_IMAGE. Then, these subimages are distributed to various processors by the procedure DISTRIBUTE_IMAGE. The subimage distribution scheme depends on the interconnection between the processors.

After labeling the components in each subimage using the sequential labeling procedure COMPONENT, the procedure MERGE_COMPONENTS is invoked. This procedure involves merging components that reside in separate subimages. An efficient merging scheme depends on the interconnection between the processors. In most cases, the communication pattern among the processors during the distribution of subimages is the inverse of the communica-

tion pattern during the merging of subimages. The difference lies in the computation involved within these procedures.

Consider two adjacent subimages S_1 and S_2. Without a loss of generality, we further assume that S_1 extends from rows i to $k-1$ and that S_2 extends from rows k to j. The merger of these subimages involves only rows $k-1$ and k. If rep^{k-1} is connected to rep^{k}, then the objects corresponding to these reps are assigned the same label—the smaller one. Also, if a touch^{k} is found, then the labels of the part^{k-1} and rep^{k} are given the same label—the one associated with part^{k-1}.

If $\text{rep}^{k-1}\ A$ is connected to $\text{rep}^{k}\ B$, then the following condition holds:

$$(A\rightarrow\text{left}\leqslant B\rightarrow\text{right}+1)\wedge(A\rightarrow\text{right}\geqslant B\rightarrow\text{left}-1).$$

The preceding condition is the same as that used in joining two reps in the sequential algorithm. To combine two components corresponding to $\text{rep}^{k-1}\ A$ and $\text{rep}^{k}\ B$, we only need to make the following changes:

$$B\rightarrow\text{object}\rightarrow\text{leftmost}\rightarrow\text{object}\rightarrow\text{end}\rightarrow\text{repnext}$$

$$=A\rightarrow\text{object}\rightarrow\text{leftmost}\rightarrow\text{object}\rightarrow\text{start},$$

$$A\rightarrow\text{object}\rightarrow\text{leftmost}\rightarrow\text{object}\ \rightarrow\text{start}$$

$$=B\rightarrow\text{object}\rightarrow\text{leftmost}\rightarrow\text{object}\rightarrow\text{start},$$

$$A\rightarrow\text{object}\ \rightarrow\text{leftmost}=B\rightarrow\text{object}\rightarrow\text{leftmost}.$$

Thus, combining two components requires a constant amount of computation.

2.8 *Complexity Analysis of the Parallel Algorithm*

The parallel algorithm involves computation and communication of data between the processors. Since the relationship between these operations vary with the parallel computers, we choose to evaluate the computational and communicational complexity separately.

Consider the labeling of components of an image of size n using p processors. The splitting of the image takes constant time. The distribution of the image involves constant computation at each processor. But, the number of times these computations are performed depends on the distribution scheme. We assume that to be L. Since the merging process is inverse of the distribution process with an additional constant computation (i.e., 3), the complexity of the merging process is $O(L)$. Thus, the computational complexity of the parallel algorithm is $O(n/p+L)$, where $O(n/p)$ is the complexity of labeling components of each subimage.

The evaluation of the communicational complexity is much more involved since it depends entirely on the distribution scheme used for the particular multicomputer. We evaluate the communicational complexities of the parallel algorithm for several multicomputers in the following section.

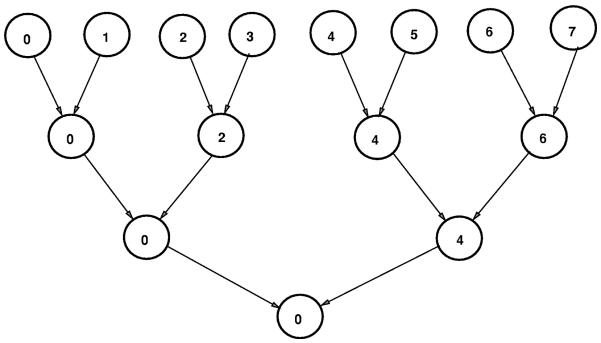


Fig. 2 Merger of components in subimages is done in a binary tree fashion.

3 Implementation on Shared Memory Architectures

Having described the sequential algorithm for labeling connected components, we now present a strategy to implement the algorithm on a shared memory computer.

The image is partitioned horizontally into a set of subimages, preferably into as many as the number of processors. Each of these subimages has its connected components labeled concurrently by multiple processors. These subimages are then merged at several levels, as are their labels. Next we discuss the hierarchical merging process, as shown in Fig. 2.

The subimages are numbered as consecutive integers from 0 upward. Subimages with consecutive numbers are adjacent. Note that the binary representation of the subimage numbers for adjacent subimages differs at the least significant digit. At the first level, adjacent subimages (0 and 1,2,3, . . .) are merged to form new subimages. The subimage obtained by merging two subimages inherits the lower subimage number. For cases with an odd number of subimages, the last subimage is merged with an empty subimage (or, passed on as is to the next level). Thus, the adjacent images at the second level differ at the second most significant digit in their binary representation. This procedure of merging is continued until all the subimages are merged into a single image. In the above discussion, we assume that a subimage is comprised of one or more rows of reps. However, this technique can easily be extended to include cases where the subimages are comprised of a part of the row (as described in the previous section).

3.1 *Complexity*

Consider an image with n object pixels labeled by p processors. On an average, each subimage has $\lceil n/p\rceil$ object pixels. Thus, labeling components for all the subimages takes $O(\lceil n/p\rceil)$ time. The maximum number of components that need to be merged at each level is $O(\lceil n/p\rceil)$. Since merging two components requires a constant amount of computation, the components can be merged in constant time at each level. The number of levels of merging is $\lceil\log p\rceil$. Hence, the computational complexity of the entire labeling process is $O(\lceil n/p\rceil+k'\lceil\log p\rceil)$, where k' is a constant. If we have n processors, then the complexity is $O(\lceil\log n\rceil)$.

We now evaluate the theoretical speedups and efficiencies achievable. We follow the notation of Lakshmivarahan and Dhall.[24] Here $T_p(n)$ denotes the time required by a parallel algorithm using p processors for problem size n, and $S_p(n)$ denotes the speedup, and is defined as $S_p(n) = T_1(n)/T_p(n)$. The efficiency $E_p(n)$ is defined as $E_p(n) = S_p(n)/p$.

Clearly $T_1(n) = kn$, where k is a constant; $T_p(n) = k\lceil n/p \rceil + k'\lceil \log p \rceil$. Thus,

$$S_p(n) = T_1(n)/T_p(n) = kn/(k\lceil n/p \rceil + k'\lceil \log p \rceil),$$

$$E_p(n) = S_p(n)/p = [kn/(k\lceil n/p \rceil + k'\lceil \log p \rceil)]/p.$$

Thus, $E_p(N) \to 0$ as $n \to \infty$. In this case, we do not get linear speedup and the efficiency of the processors is very low.

The algorithm is said to have optimal speedup if the speedup is linear and the efficiency is unity. In our case, we get very poor speedup and efficiency. The key to increasing the speedup and efficiency is to require that the problem size be much larger than the processor size.

Let $n = Lp \log p$, where L is a large constant. Thus,

$$S_p(n) = kLp \log p/(k\lceil L \log p \rceil + k'\lceil \log p \rceil)$$
$$= pkL/(kL + k') \to p \quad \text{as} \quad L \to \infty,$$

$$E_p(n) \to 1 \quad \text{as} \quad L \to \infty.$$

Thus, if we have a large image and few processors, then we must get near linear speedups and unit efficiency.

Obviously the communicational complexity of the algorithm is zero for shared memory architectures.

4 Implementation on Distributed Memory Architectures

This section discusses the parallel implementation of the connected component labeling algorithm on various distributed memory architectures. Unlike shared memory computers, distributed memory computers have a communication overhead. We present the computational and communicational complexity of implementing the algorithm on various distributed memory architectures. Most previous algorithms for distributed memory architectures did not consider the communicational complexity. As we show, communicational complexity dominates.

For the complexity evaluation, we assume the size of the image to be n pixels and the number of processors to be p. The unit of communication is a pixel and it takes unit time for unit distance. We shall assume that the distance between two processors directly connected to each other is unity. All the complexity evaluations are worst case evaluations. Note that the complexities hold for both SIMD and multiple-item, multiple-data (MIMD) distributed memory architectures.

4.1 *Binary Tree Connected Computer*

We are given a binary tree connected computer with $p = 2q - 1$ processors. Thus, we have q leaf processors. We first split the image equally such that each leaf processor has a subimage of size n/q. For now, we assume that each leaf processor has its own subimage associated with it. We describe the distribution of the subimages later.

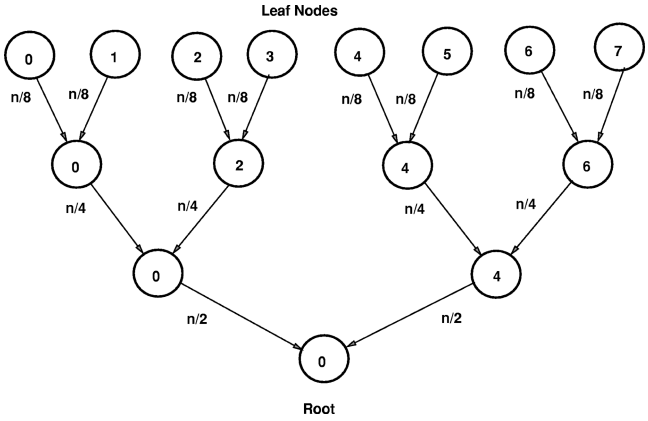


Fig. 3 Merging process proceeds from the leaf processors to the root. The arrows indicate the direction of data movement and the superscript on the arrow indicates the communication cost.

The algorithm is implemented as follows. During the first phase, each of the leaf processors computes the connected components of its associated subimage. These connected components are then merged in a binary tree fashion, as in shared memory architectures. The difference in the two is that in the case of a tree connected binary computer, the entire subimage is communicated between processors, along with the connected component information. Finally, the connected components of the entire image are in the root processor.

4.1.1 *Computational complexity*

The connected component labeling process in the leaf processors takes $O(n/q)$ time. After this, there are $\log q$ merging steps, each of which takes a constant amount of computation. Thus, the total computation time is $O(n/q + \log q) = O[2n/(p+1) + \log (p+1)/2]$. If we have a binary tree connected computer with $2n-1$ processors, then the computational complexity of the algorithm is $O(\log n)$.

4.1.2 *Communicational complexity*

The merging process occurs in phases in a binary tree fashion, as Fig. 3 shows. The total communicational time required is $O(n/q + 2n/q + \ldots + n/2) = O[n(q-1)/q] = O[n(p-1)/(p+1)]$. Thus, if we have a binary tree connected computer with $2n-1$ processors, then the communicational complexity of the algorithm is $O(n)$.

The distribution of the subimages is the inverse process of merging performed at different levels of the binary tree connected computer. Thus, the distribution process has the same communicational complexity as merging, i.e., $O(n)$. Hence, the communicational complexity of the algorithm is $O(n)$.

4.2 *Mesh Connected Computer*

The interconnections between the processors in the mesh connected computer can either be unidirectional or bidirectional. For example, the AT&T Pixel Machine has unidi-

rectional interconnection links, i.e., at any instance all the links can transmit data in only one direction. The computational complexities and the communicational complexities for both these types of mesh connected computers are dealt with separately. For the sake of simplicity, we consider a mesh connected computer with $p=q^2$ processors. We first split the image equally such that each processor has a subimage of size n/q^2. The distribution of the subimages is done as an inverse process of merging the subimage connected components.

The algorithm for both unidirectional and bidirectional mesh connected computers is implemented as follows. During the first phase, each of the processors computes the connected components of its associated subimage. These connected components are then merged in a binary tree fashion as in the binary tree connected computer. The exact merging scheme can be split into two operations: row and column merging. We first merge the subimage connected components of all the rows. The resulting merged subimages of row merging are then merged as a column merge. Note that each of these mergings is similar to the mergings of the binary tree connected computer. The difference is that the distances between subsequent merges do not remain equal.

4.2.1 *Unidirectional mesh*

For a unidirectional mesh connected computer, it can be easily shown by induction that the number of mergings required is $2(q-1)$, i.e., $q-1$ for row merging and $q-1$ for column merging. Figure 4 presents an example of a merging process with four processors. The arrows indicate the direction of data movement and the terms by the arrows indicate the relative amounts of data transfer.

Computational complexity. The connected component labeling process in the processors takes $O(n/p)$ time. After this process, there are $2(q-1)$ merging steps, each of which takes a constant amount of computation. Thus, the total computation time is $O(n/p+2p^{1/2}-2)$. If we have a mesh connected computer with n processors, then the computational complexity of the algorithm is $O(n^{1/2})$.

Communicational complexity. We denote the communication time for row merging by U_r (see Appendix B for detailed derivation) and the communication time for column merging by U_c:

$$U_r = O\{n/p[1+1+(2+2)+(4+4+4+4)+\cdots(q-1)\text{terms}]\}$$
$$= O(n/3-n/2p^{1/2}+2n/3p),$$

$$U_c = O\{n/q[1+1+(2+2)+(4+4+4+4)+\cdots(q-1)\text{terms}]\}$$
$$= O(np^{1/2}/3-n/2+2n/3p^{1/2}).$$

Thus, the total communicational complexity U is given as follows:

$$U = U_r + U_c$$

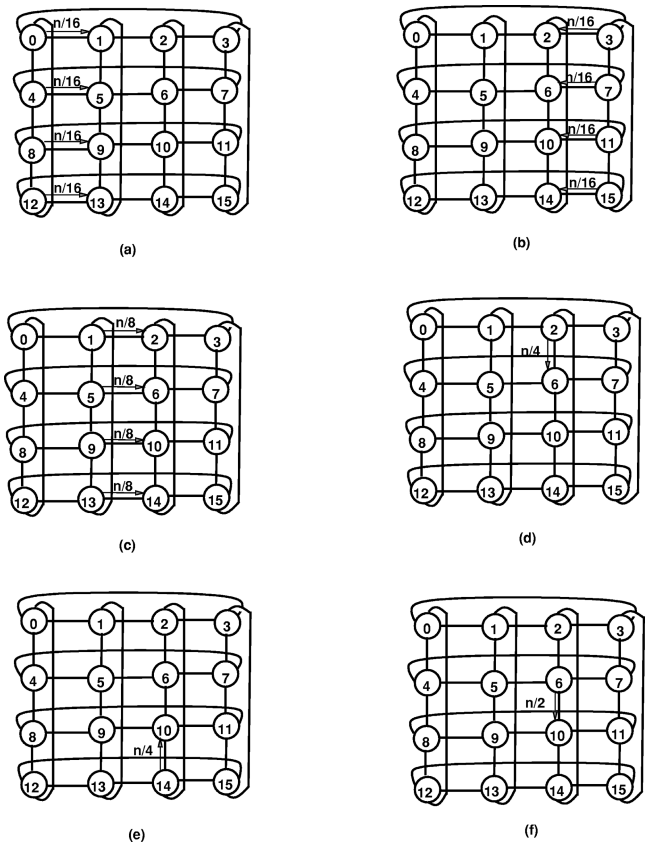


Fig. 4 Phases illustrating the merging process for a unidirectional mesh with 16 processors. The arrows indicate the direction of data movement and the terms by the arrows indicate the communicational costs.

$$= O(n/3-n/2p^{1/2}+2n/3p+np^{1/2}/3-n/2+2n/3p^{1/2})$$
$$= O[1/3(np^{1/2}-n/2+n/2p^{1/2}+2n/p)].$$

If we have a mesh connected computer with n processors, then the communicational complexity of the algorithm is $O[1/3[n^{3/2}-n/2+n^{1/2}/2+2)]=O(n^{3/2})$.

4.2.2 *Bidirectional mesh*

For a bidirectional mesh connected computer, it can be easily shown by induction that the number of mergings required is q, i.e., $q/2$ for row merging and $q/2$ for column merging. Figure 5 gives an example of a merging process with four processors. The arrows indicate the direction of data movement and the terms by the arrows indicate the relative amounts of data transfer.

Computational complexity. The connected component labeling process in the processors takes $O(n/p)$ time. After this process, there are q merging steps, each of which takes a constant amount of computation. Thus, the total computation time is $O(n/p+p^{1/2})$. If we have a mesh connected computer with n processors, then the computational complexity of the algorithm is $O(n^{1/2})$.

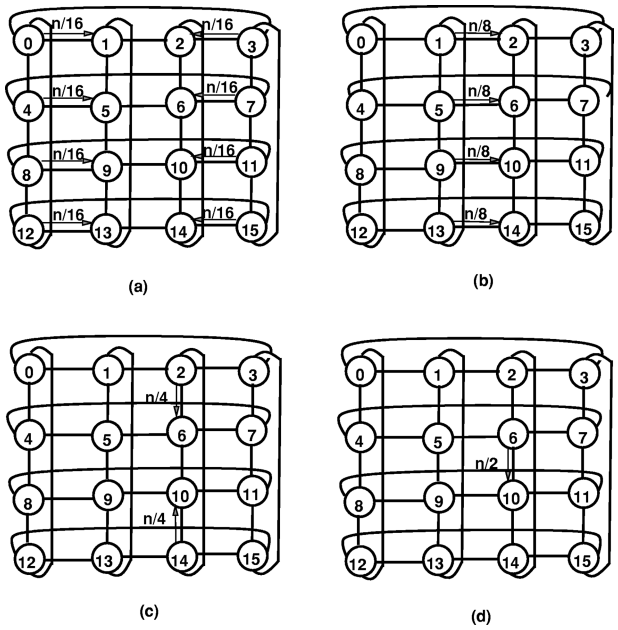


Fig. 5 Phases illustrating the merging process for a bidirectional mesh with 16 processors. The arrows indicate the direction of data movement and the terms by the arrows indicate the communicational costs.

Communicational complexity. We denote the communication time for row merging by B_r (see Appendix B for detailed derivation) and the communication time for column merging by B_c:

$$B_r = O\{n/p[1+2+(4+4)+(8+8+8+8) + \ldots q/2\text{terms}]\} = O(n/3+2n/3p),$$

$$B_c = O\{n/q[1+2+(4+4)+(8+8+8+8) + \ldots q/2\text{terms}]\} = O(np^{1/2}/3+2n/3p^{1/2}).$$

Thus, the total communicational complexity B is given as follows:

$$B = B_r + B_c = O(n/3+2n/3p+np^{1/2}/3+2n/3p^{1/2}) = O[1/3(np^{1/2}+n+2n/p^{1/2}+2n/p)].$$

If we have a mesh connected computer with n processors, then the communicational complexity of the algorithm is $O[1/3(n^{3/2}+n+2n^{1/2}+2)] = O(n^{3/2})$.

4.3 *Hypercubes*

A hypercube of degree d has 2^d nodes and each node has exactly d neighbors. The distance between any two nodes is less than or equal to d. We first discuss the embedding of binary trees in hypercubes. Wu[25] presents three results that we use in our implementation of the algorithm on the hypercube.

Proposition 1. A complete binary tree of height $d>2$ cannot be embedded in a hypercube of degree $\leq d$ such that adjacency is preserved. In other words, a complete binary tree cannot be embedded in a hypercube with a dilation cost of 1 and an expansion cost of less than 2.

Proposition 2. A complete binary tree of height $d>0$ can be embedded in a hypercube of degree $d+1$ in such a way that the adjacencies of nodes of the binary tree are preserved.

Proposition 3. A complete binary tree of height $d>0$ can be embedded in a hypercube of degree d with cost $=2$; i.e., neighbors in the binary tree are mapped into nodes of, at most, distance 2 away in the hypercube.

A complete binary tree of height d has 2^d-1 nodes. The smallest hypercube large enough to house a binary tree of height d is of degree d. The algorithm is implemented on the hypercube as follows. For the sake of simplicity, we consider a hypercube with $p=2^d$ processors. We first split the image equally such that each processor has a subimage of size n/p. During the first phase, each of the processors computes the connected components of its associated subimage. These connected components are then merged in a binary tree fashion, as in the binary tree connected computer. The distribution of the subimages is an inverse process of merging the subimage connected components.

By proposition 1, it is clear that we cannot embed a complete binary tree (to be used in our merging process) in a hypercube with a dilation cost of 1 and an expansion cost of less than 2. Propositions 2 and 3 give us two solutions to embed the complete binary tree in a hypercube. In the first, we can use a hypercube with twice the number of processors. In the second, the neighboring nodes in the binary tree will have a distance of 2 between them. Using twice the number of processors is undesirable, since it reduces the efficiency of processors. In addition, being unable to preserve the adjacency of the tree nodes increases our communicational cost. These two drawbacks are alleviated using a pseudobinary tree.

A pseudobinary tree is a binary tree structure that can easily be embedded into the hypercube topology such that a node in the hypercube can represent more than one node in the corresponding pseudobinary tree.[26] A pseudobinary tree is an efficient topology for distributing and merging subimages. The modified singlecast scheme[26] in which the controller (one of the processing elements) distributes a set of images is used to distribute the subimages. The merging process is exactly the inverse. Figure 6 illustrates a pseudobinary tree of a hypercube of degree 3. Figure 7 shows the phases of merging in the hypercube.

4.3.1 *Computational complexity*

The connected component labeling process in the processors takes $O(n/p)$ time. After this process there are $\log p$ merging steps, each of which takes a constant amount of computation. Thus, the total computation time is $O(n/p+\log p)$. If we have a mesh connected computer with n processors, then the computational complexity of the algorithm is $O(\log n)$.

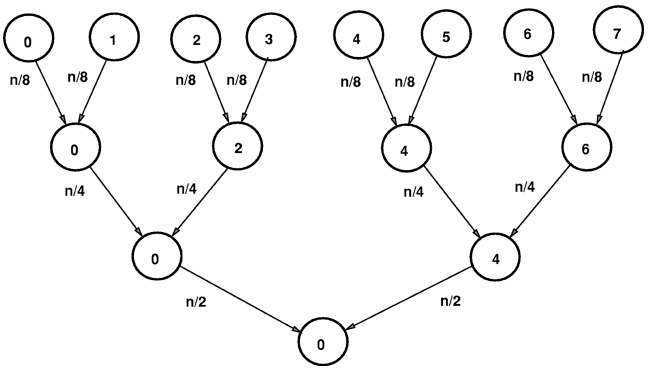


Fig. 6 Pseudobinary tree of a hypercube of degree 3. The arrows indicate the direction of data movement and the terms by the arrows indicate the communicational costs.

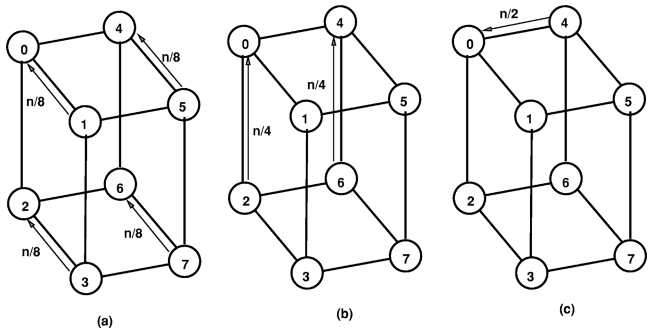


Fig. 7 Phases illustrating the merging process for a hypercube of degree 3 using the pseudobinary tree. The arrows indicate the direction of data movement and the terms by the arrows indicate the communicational costs.

4.3.2 *Communicational complexity*

As Fig. 6 shows, the merging process occurs in phases in a binary tree fashion. The total communicational time required is $O(n/p+2n/p+\cdots+n/2)=O[n(p-1)/p]$. Thus, if we have a hypercube with n processors, the communicational complexity of the algorithm is $O(n)$.

The distribution of the subimages is the inverse process of merging performed at different levels of the pseudobinary tree. Thus, the distribution process has the same communicational complexity as merging, i.e., $O(n)$. Hence, the communicational complexity of the algorithm is $O(n)$.

5 Implementation Results

The sequential algorithm presented is optimal, since accessing and storing the input itself has a complexity that is linear in the number of object pixels in the image. Both the sequential and the parallel algorithms described in this paper were tested on several images with all sorts of complex regions. The images contain convex, concave, simply connected, and multiply connected regions with holes. For a 256×256 image with 172 components, the sequential algorithm takes 351 ms. Figures 8 and 9, respectively, give the speedups obtained by the parallel algorithm for two images. Efficiency (which is the ratio of the speedup to the number of processors used) for the two images is also shown. We obtain a maximum speedup of 10.53 using 12 processors for the 256×256 binary image, and the efficiency varies around 0.75 on the sequent balance multiprocessor. The maximum speedup obtained for the binary text image is 8.07 using 12 processors.

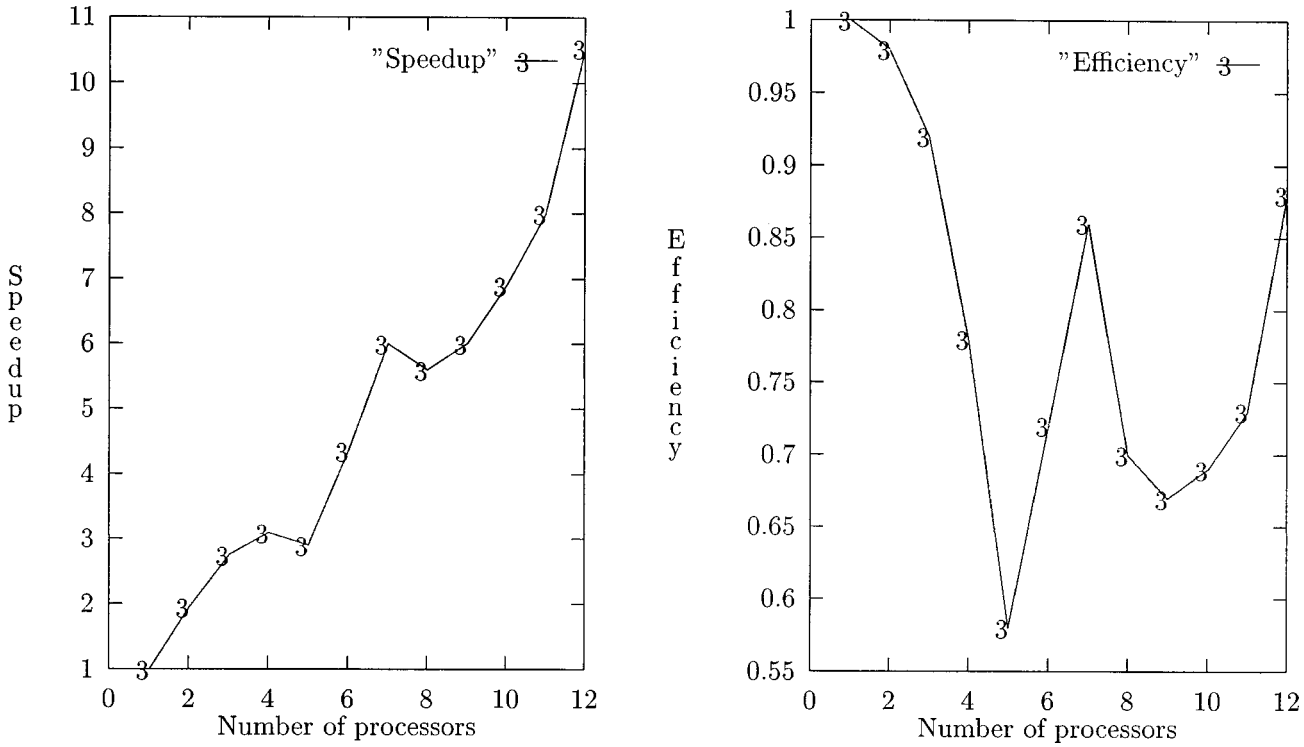


Fig. 8 Speedup and efficiency of a 256×256 binary image with 172 components.

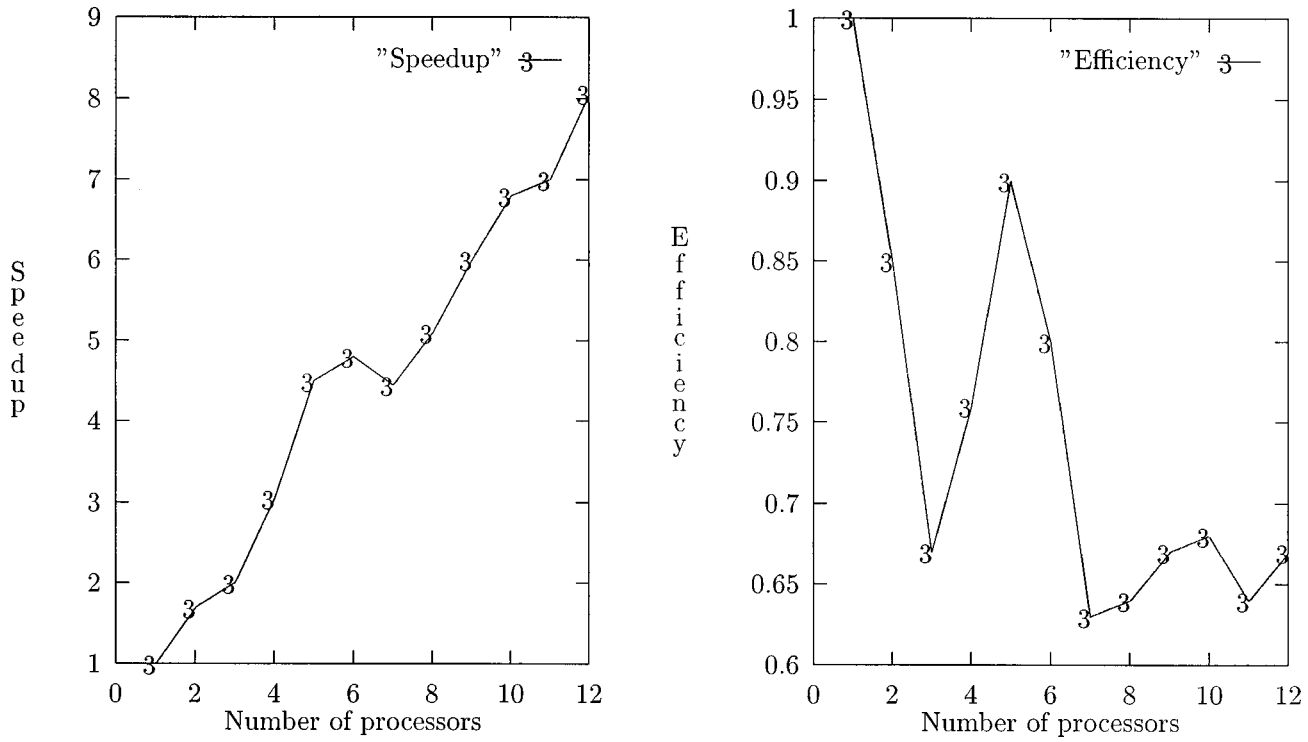


Fig. 9 Speedup and efficiency of a 512×512 binary text image with 793 components.

The variation in speedups and efficiencies is because of load imbalance among the processors. This imbalance results from statically splitting the input image into subimages with approximately the same number of object pixels. The execution time of labeling an object varies with the type of the objects. But the major overhead appears to be the spread of an object over several subimages. This was verified by experimentation. We split an image into subimages (approximately equal in size) in several different ways. The execution time of the parallel algorithm was consistently higher when an object was spread over several subimages.

6 Conclusion

We present an optimal and practical sequential algorithm for labeling connected components whose memory requirement and computational complexity is linear in the number of object pixels in the binary image. Since no assumptions have been made on the type of input, the algorithm works for all types of binary images. The representation of the connected components as a linked list of reps makes it easy to calculate certain features of regions, i.e., area, perimeter, etc. The input and output representations of the image make it possible for the algorithm to be used in real computer vision applications. The algorithm is easily extended to gray-level images by including another field indicating the gray level in the rep. The only other change is an addition to the connectivity check for two reps. Two reps are connected only if they have the same gray levels.

The new parallel algorithm implemented on a shared memory computer presented in this paper has a computational complexity of $O(\lceil \log n \rceil)$ for an image of size n. We discuss the results of implementing the algorithm on the sequent balance multiprocessor. The speedup graphs show an almost linear speedup. The computational and communicational complexities of the algorithm implemented on various distributed memory architectures, i.e., a binary tree connected computer, a unidirectional and bidirectional mesh connected computer, and a hypercube computer are computed. It is trivial to see that the algorithm can be implemented on the polymorphic torus architecture with complexities no worse than those of the mesh connected computers. These complexities are the best yet achieved. The communicational complexities are greater than the computational complexities for all the distributed memory architectures. This explains the decrease in the efficiency of processors with the increase in the number of processors.[15] The theoretical formulation of these complexities gives a better idea as to which architecture will have a better efficiency. In other words, we have an analytic expression for the communication overhead. We also verified that the speedup and efficiency obtained are nearly linear and unity, respectively, when the image size (as we have chosen) is much larger than the number of processors used.

7 Appendix A: The Pseudo Code for the Sequential Algorithm

This appendix describes the sequential algorithm for connected component labeling. We use an easy to understand, pseudo PASCAL/C for representation.

Global Declarations

```
integer labelnum=0;
integer joint=0;
rep *newrep, *oldrep, *end;
objects *oldobj, *stack[int][int];
```

```
 component *compos;
end Global Declarations
 procedure COMPONENT()
 begin
      INITIALIZE();
      while not(EOF) do
           MAKELISTS();
           CONNECTIONS();
           REGIONS();
           CHANGELABEL();
           JOIN();
           oldrep→repnext=newrep→repnext;
           newrep→repnext=end;
      od
      PRINTOUT(comps);
 end COMPONENT;
 procedure INITIALIZE()
 begin
      CREATOBJ(oldobj);
      CREATREP(oldrep);
      CREATREP(end);
      end→left=end→right=MAXCOLS+2;
      oldrep→repnext=end;
      CREATREP(newrep);
      newrep→repnext=end;
 end INITIALIZE;
 procedure CREATREP(temp)
 begin
      temp→object=NULL;
      temp→repnext=NULL;
 end CREATREP;
 procedure CREATOBJ(temp)
 begin
      temp→start=temp→end=NULL;
      temp→label=++labelnum;
      temp→touch=false;
      temp→objnext=NULL;
 end CREATOBJ;
 procedure MAKELISTS()
 begin
      read(row, numrep);
      nrep=newrep;
      for(i=1; i≤numrep; i++) do
           CREATREP(temp);
           nrep→repnext=temp;
           nrep=temp;
           read(left, right);
           nrep→left=left;
           nrep→right=right;
           nrep→row=row;
      od
      nrep→repnext=end;
 end MAKELISTS;
 procedure CONNECTIONS()
 begin
      joint=0;
      nrep=newrep→repnext;
      while(oldrep→repnext≠end) do
           orep=oldrep→repnext;
           while(nrep→right≤orep→right) do
                if CONNECTED(nrep, orep)
                then
                     orep→object→touch=true;
                     nrep→object→label
                     =orep→object→label;
                nrep=nrep→repnext;
           od
           if CONNECTED(nrep, orep) then
                orep→object→touch=true;
                nrep→object→label=
                     orep→object→label;
                if
                CONNECTED(nrep, orep→repnext)
                then
                     stack[++joint][1]=
                     orep→object→label;
                     stack[joint][2]=
                     orep→repnext→object→label;
                     LABELLING(orep→object,
                       orep→repnext→object);
           oldrep→repnext=orep→repnext;
           orep→repnext=orep→object→end;
           orep→object→end=orep;
           if (orep→object→start==NULL) then
                orep→object→start=orep;
      od
 end CONNECTIONS;
 procedure REGIONS()
 begin
      pobj=oldobj;
      while(pobj→objnext≠NULL) do
           obj=pobj→objnext;
           if (obj→touch) then
                pobj=obj;
                obj→touch=false;
           else
                dummy=compos;
                while(dummy→compnext≠NULL) do
                     dummy=dummy→compnext;
                od
                new→comp=obj→end;
                dummy→compnext=new;
                pobj→objnext=obj→objnext;
      od
 end REGIONS;
 procedure CHANGELABEL()
 begin
      for(p=joint; p≥1; joint--) do
           LABELLING(stack[p][1], stack[p][2]);
      od
      nrep=newrep→repnext;
      while(nrep≠end) do
           if (nrep→object==NULL) then
                CREATOBJ(obj);
                nrep→object=obj;
                obj→leftmost=nrep;
                obj→objnext=oldobj→objnext;
                oldobj→objnext=obj;
```

```
            else
                nrep→object=
                nrep→object→leftmost→object
                if (nrep→object→leftmost==NULL)
                    then nrep→leftmost=nrep;
            nrep=nrep→repnext
        od
end CHANGELABEL;
procedure JOIN()
begin
        obj=oldobj→objnext;
        pobj=oldobj;
        while (obj≠NULL) do
            obj→touch=false;
            if (obj==obj→leftmost→object) then
                pobj=obj;
            else
                obj→start→repnext=obj→leftmost
                →object→end;
                obj→leftmost→object→end=obj→end;
                pobj→objnext=obj→objnext;
            obj=pobj→objnext;
        od
end JOIN;
procedure LABELLING(p, q)
begin
        if (p→leftmost→object→label
        <q→leftmost→object→label)
        then
            q→leftmost=p→leftmost;
            q→label=p→leftmost→object→label;
        else
            p→leftmost=q→leftmost;
            p→label=q→leftmost→object→label;
end LABELLING;
procedure CONNECTED(p, q)
begin
        if ((p→left≤q→right+1)AND
        (p→right≥q→left−1))
        then
            return(true);
        else
            return(false);
end CONNECTED;
```

8 Appendix B: Evaluating Communicational Complexities

8.1 *Unidirectional Mesh*

For the sake of simplicity, we assume that $q=2^x$, for some $x \in N$.

$$U_r = O\{n/p[1+1+(2+2)+(4+4+4+4) + \cdots (q-1)\text{terms}]\}$$
$$= O\{n/q^2[1+2^0\cdot 2^0+2^1\cdot 2^1+2^2\cdot 2^2 + \cdots + 2^r(2^r-1)]\}.$$

From these two expressions we can evaluate the value of r since

$$1+2+2^2+2^3+\cdots+2^r-1=q-1-1 \Rightarrow r=\log(q/2).$$

Thus,

$$U_r = O\{n/q^2[1+(1+2^2+2^4+\cdots+2^{2r})-2^r]\}$$
$$= O\{n/q^2[1+(2^{2r+2}-1)/3-2^r]\}$$
$$= O\{n/q^2[1+(q^2-1)/3-q/2]\}$$
$$= O\{n/p[1+(p-1)/3-p^{1/2}/2]\}$$
$$= O(n/3-n/2p^{1/2}+2n/3p).$$

It is easy to see that $U_c = p^{1/2}U_r$.

8.2 *Bidirectional Mesh*

For the sake of simplicity, we assume that $q=2^x$, for some $x \in N$.

$$B_r = O\{n/p[1+2+(4+4)+(8+8+8+8) + \cdots q/2\text{terms}]\}$$
$$= O[n/q^2(1+2^1\cdot 2^0+2^2\cdot 2^1+2^3\cdot 2^2+\cdots+2^r\cdot 2^{r-1})].$$

From these two expressions we can evaluate the value of r since

$$1+2+2^2+2^3+\cdots+2^{r-1}=q/2-1 \Rightarrow r=\log(q/2).$$

Thus,

$$B_r = O[n/q^2(1+2^1+2^3+2^5+\cdots+2^{2r-1})]$$
$$= O\{n/q^2[1+(2^{2r+2}-1)/3]\}$$
$$= O\{n/q^2[1+(q^2-1)/3]\}$$
$$= O\{n/p[(p+2)/3]\}$$
$$= O(n/3+2n/3p).$$

It is easy to see that $B_c = p^{1/2}B_r$.

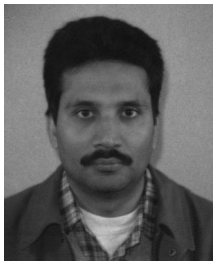
Acknowledgments

This research was supported in part by IBM.

References

1. A. Rosenfeld and A. C. Kak, *Digital Picture Processing*, Academic Press, New York (1982).
2. A. Rosenfeld and J. L. Pfaltz, ''Sequential operations in digital signal processing,'' *J. ACM*, **13**(4), 471–494 (1966).
3. R. Lumia, L. Shapiro, and O. Zuniga, ''A new connected components algorithm for virtual memory computers,'' *Comput. Vis. Graph. Image Process.* **22**(2), 287–300 (1983).
4. R. M. Haralick, ''Some neighborhood operations,'' in *Some Neighborhood Operations in Real Time/Parallel Computing Image Analysis*, M. Onoe, K. Preston Jr., and A. Rosenfeld, Eds., Plenum Press, New York (1981).
5. E. Mandler and M. F. Oberlander, ''One-pass encoding of connected components in multi-valued images,'' in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 64–69 (1990).
6. H. Samet and M. Tamminen, ''An improved approach to connected component labeling of images,'' in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 312–318 (1986).

7. S. W. Zucker, ''Region growing: childhood and adolescence,'' *Comput. Graph. Image Process.* **5**(3), pp. 382–399 (1976).
8. V. Chaudhary and J. K. Aggarwal, ''Parallelism in computer vision—a review,'' in *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P. S. Gopalakrishnan, and L. Kanal, Eds., pp. 270–309, Springer-Verlag, New York (1990).
9. D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, ''Computing connected components on parallel computers,'' *Commun. ACM* **22**(8), 461–464 (1979).
10. D. Nassimi and S. Sahni, ''Finding connected components and connected ones on a mesh connected parallel computer,'' *SIAM J. Comput.* **9**(4), 744–757 (1980).
11. L. W. Tucker, ''Labeling connected components on a massively parallel tree machine,'' in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 124–129 (1986).
12. R. Cypher, J. L. C. Sanz, and L. Snyder, ''EREW PRAM and mesh connected computer algorithms for image component labeling,'' in *Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, pp. 122–128 (1987).
13. R. Cypher, J. L. C. Sanz, and L. Snyder, ''Hypercube and shuffle-exchange algorithms for image component labeling,'' in *Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, pp. 5–9 (1987).
14. R. Cypher, J. L. C. Sanz, and L. Snyder, ''Practical algorithms for image component labeling on SIMD mesh connected computers,'' in *Proc. Int. Conf. on Parallel Processing*, pp. 772–779, Pennsylvania State University Press (1987).
15. M. H. Sunwoo, B. S. Baroody, and J. K. Aggarwal, ''A parallel algorithm for region labeling,'' in *Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, pp. 27–34 (1987).
16. A. Aggarwal and L. Nekludova, ''A parallel $O(\log N)$ algorithm for finding connected components in planar images,'' in *Proc. Int. Conf. Parallel Processing*, pp. 783–786, Pennsylvania State University Press (1987).
17. J. J. Little, G. Blellock, and T. Class, ''Parallel algorithms for computer vision on the connection machine,'' in *Proc. IEEE Int. Conf. on Computer Vision*, pp. 587–591 (1987).
18. M. Maresca, H. Li, and M. Lavin, ''Connected component labeling on polymorphic torus architecture,'' in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 951–956 (1988).
19. M. Manohar and H. K. Ramapriyan, ''Connected component labeling of binary images on a mesh connected massively parallel processor,'' *Comput. Vis. Graph. Image Process.* **45**, 133–149 (1989).
20. J. Woo and S. Sahni, ''Hypercube computing: connected components,'' *J. Supercomput.* **3**(3), 209–234 (1990).
21. H. Alnuweiri and V. K. P. Kumar, ''Parallel architectures and algorithms for image component labeling,'' Technical Report IRIS 253 University of Southern California (1992).
22. K. P. Bekhale and P. Banerjee, ''Parallel algorithms for geometric connceted component labeling on a hypercube multiprocessor,'' *IEEE Trans. Comput.* **41**(6), 699–709 (1992).
23. W. J. Hsu, L. R. Wu, and X. Lin, ''Optimal algorithms for labeling image components,'' in *Proc. Int. Conf. on Parallel Processing*, Vol. III, pp. 75–82, Pennsylvania State University Press (1990).
24. S. Lakshmivarahan and S. K. Dhall, in *Analysis and Design of Parallel Algorithms*, pp. 18–24, McGraw-Hill (1990).
25. A. Y. Wu, ''Embedding of tree networks into hypercubes,'' *J. Parallel Distrib. Comput.* **2**, 238–249 (1985).
26. S. Y. Lee and J. K. Aggarwal, ''Exploitation of image parallelism via the hypercube,'' in *Proc. 2nd Conf. on Hypercube Multiprocessors*, (1986).



Vipin Chaudhary received his BTech (Hons.) degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1986, where he was awarded the President of India Gold Medal for academic excellence, and his MS degree in computer science and his PhD degree in electrical and computer engineering from the University of Texas at Austin, in 1989 and 1992, respectively. He is currently an assistant professor of electrical and computer engineering and computer science at Wayne State University. From January 1992 to May 1992, he was a postdoctoral researcher with the Computer and Vision Research Center, the University of Texas at Austin. His research interests are parallel and distributed computing, computer vision, and biomedical engineering. Currently his research is focused on automatic parallelization of programs, network security issues, and treatment planning of cancer. He has published over 35 papers in these areas. His research sponsors include the National Science Foundation, the U.S. Army, Ford Motor Company, General Motors, IBM, and Cray Research.

Jake K. Aggarwal has served on the faculty of the University of Texas at Austin College of Engineering since 1964 and is currently the Cullen Professor of Electrical and Computer Engineering and directs the Computer and Vision Research Center. His research interests include computer vision, parallel processing of images, and pattern recognition. He has been a fellow of IEEE since 1976. He received the Senior Research Award of the American Society of Engineering Education in 1992 and 1996 Technical Achievement Award of the IEEE Computer Society. He is author or editor of 7 books and 31 book chapters and over 170 journal papers, as well as numerous proceedings papers and technical reports. He chaired the IEEE Computer Society Technical Committee on Pattern Analysis and Machine Intelligence from 1987 to 1989, directed the North Atlantic Treaty Organization Advanced Research Workshop on Multisensor Fusion for Computer Vision, Grenoble, France, in 1989, chaired the IEEE Computer Society Conference on Computer Vision and Pattern Recognition in 1993, and was president of the International Association for Pattern Recognition in 1992 to 1994. He currently is the IEEE Computer Society representative to the International Association for Pattern Recognition (IAPR).