1995

Finding Connected Components on a Scan Line Array Processor

Ronald I. Greenberg
Loyola University Chicago, Rgreen@luc.edu

Finding Connected Components on a Scan Line Array Processor

Ronald I. Greenberg*
Department of Electrical Engineering
and Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
rig@eng.umd.edu

(Preliminary Version)

Abstract

This paper provides a new approach to labeling the connected components of an $n \times n$ image on a scan line array processor (comprised of n processing elements). Variations of this approach yield an algorithm guaranteed to complete in $o(n \lg n)$ time as well as algorithms likely to approach $O(n)$ time for all or most images. The best previous solutions require using a more complicated architecture or require $\Omega(n \lg n)$ time. We also show that on a restricted version of the architecture, any algorithm requires $\Omega(n \lg n)$ time in the worst case.

1 Introduction

The scan line array processor (SLAP) (also referred to as the Princeton Engine or Sarnoff Engine) has been proposed by several authors as an efficient SIMD machine for low-level or intermediate-level image processing tasks [4, 9, 10, 13]. The basic structure of the SLAP for computations on an image of size up to $n \times n$ pixels is a linear array of n processors. The rows of the image are input to the SLAP one after another in the direction perpendicular to the array connections (as in Figure 1). Thus, each image row is input to the SLAP in constant time, one pixel per processor. For some low-level image processing tasks, such as median filtering with a small window size or convolution of an image with a small kernel (e.g., [12]), only a constant amount of memory per processor is required. But for intermediate-level tasks, it may be necessary to input the entire image to the SLAP before output is produced, so the SLAP is designed to have $\Theta(n)$ memory per processor. Thus, in $O(n)$ time, an entire image can be input to the SLAP, with each processor holding one column of the image. Communication can also be performed using the linear-array connections between the processors. On any given time step, one data item (i.e., $O(\lg n)$ bits) can be transferred on the link between each pair of adjacent processors.

*Supported in part by NSF grant CCR-9321388.

To appear in *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, July 1995, Santa Barbara, California.*

A fundamental intermediate-level image processing task is labeling of the connected components in a binary image. That is, each pixel is 0 or 1, two pixels are connected if there is a path of adjacent (horizontally or vertically) 1-valued pixels from one to the other, and the problem is to label each pixel so that any two 1-valued pixels are assigned the same label if and only if they are connected.

This paper gives a SLAP algorithm that labels connected components of an $n \times n$ image in $O(n \lg n / \lg \lg n)$ time. In addition, this algorithm or variants of it are likely to run in close to $O(n)$ time (which is a clear lower bound) on all or most images. Previous SLAP algorithms required $\Omega(n \lg n)$ time [2, 12], even with an unnatural "shuffled row-major" input ordering [2]. Various algorithms have been proposed to solve the problem in $O(n)$ time on a two-dimensional mesh of n^2 processors [6, 16, 18], but the drawbacks of such an approach are great. Even with $n = 128$, n^2 processors would greatly exceed the available resources on most existing parallel machines, whereas much larger images can easily be handled using a linear array configuration of processors. Other algorithms can yield even better than $O(n)$ time [5, 15, 17], but only with interconnection networks that are more complicated and, therefore, more costly. Related work has also been done by Schwartz, Sharir, and Siegel [19] and Dillencourt, Samet, and Tamminen [7]. They show that component labeling of a rectangular image can be performed in time linear in the number of pixels ($O(n^2)$ time for an $n \times n$ image) when the pixels of the image are read in scan line order. But they do not consider any type of multiprocessing solution.

Section 2 of this paper gives the high-level algorithm for the new approach to component labeling on the SLAP and proves it correct. Section 3 discusses final implementation details and resulting time bounds. Section 4 gives two additional results: (1) an extension to computing for each component a function of initial labels assigned to the component's pixels and (2) a lower bound for a restricted version of the SLAP. Section 5 provides concluding remarks.

2 The high-level algorithm

This section gives the high-level algorithm for labeling the connected components of an $n \times n$ image on the SLAP. It is assumed that the image has been input so that each processor holds one column of the image, as is natural on the SLAP. Also, the rows and columns are numbered from 0 to $n-1$, from top to bottom and left to right, respectively.

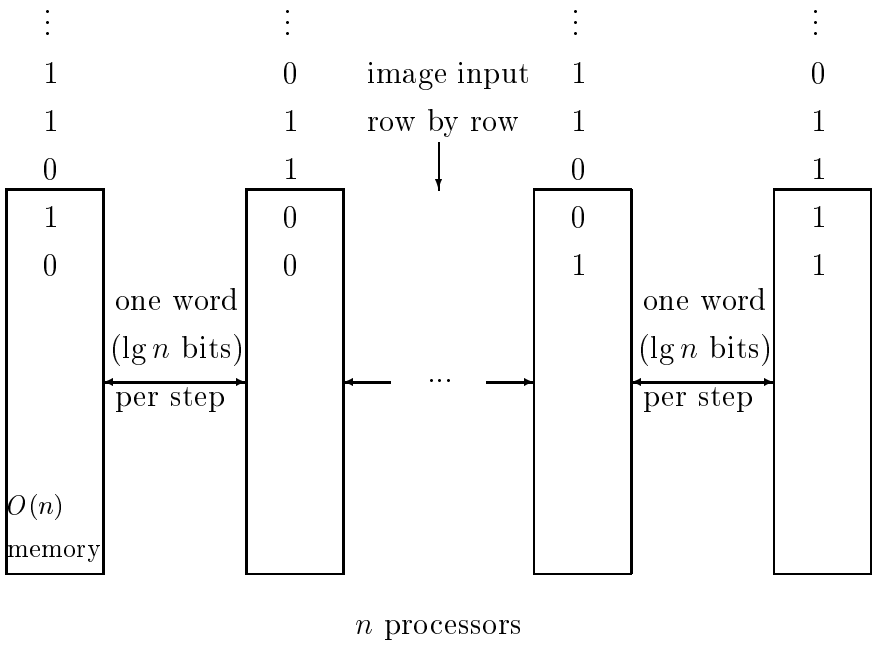


Figure 1: The SLAP architecture.

The top-level procedure involves computing what may be referred to as a *left-connected component labeling* and a *right-connected component labeling*. In a left-connected component labeling, two pixels in columns i and $i' > i$ receive the same label if they are in the same component of the subimage comprised of columns 0 through i'. In a right-connected component labeling, these pixels have the same label if they are in the same component of the subimage comprised of columns i through $n-1$. Once a left-connected component labeling and a right-connected component labeling (using different labels) have been found, it is easy to determine an overall connected component labeling as specified in Algorithm CC in Figure 2. Correctness follows from an observation also used in previous SLAP algorithms. That is, given a cut through the image and correct component labelings of the two subimages determined by the cut, the pixels bordering on the cut can be correctly labeled by performing component labeling on just the pixels bordering on the cut. The approach of Algorithm CC can be viewed as fitting into the same framework by doubling each column and considering cuts running between the two copies of each column. We can maintain consistency of the various "border labelings" produced in the last step of Algorithm CC by imposing the rule that each component gets labeled with the least label seen on its pixels at this stage. A convenient initial labeling for each pixel is the position in column-major order (i.e., $in + j$ for the pixel in column i and row j). With this initial labeling, the overall algorithm described in this section labels each component with the least initial label of its pixels.

The difficult part of performing component labeling is the implementation of steps 1 and 2 in Algorithm CC. The two steps are completely analogous, so the remaining discussion will consider only the left-component labeling of step 1.

Intuitively, we would like to do left-component labeling by performing some local work within processors while also performing a left to right sweep of information from processor to processor. A couple sample images should suggest the difficulty of performing the labeling correctly and efficiently. In Figure 3(a), processors seem to require a complicated organization of information about connections between components that occur in columns to the left. Figure 3(b) illustrates a pattern that if repeated over and over would cause excessive delay for a naive approach of passing labels to the right in a top to bottom fashion; it may be necessary to pass labels before they are final and then to patch up later.

The high-level structure of the left-component labeling procedure is specified in Algorithm LEFT-COMPONENTS in Figure 4.

The most difficult part of the left-component labeling algorithm is step 1, which groups the pixels in each column into sets, with one set for each left-component intersecting the column. It uses the operations that define the familiar union-find problem [1]. The basic union-find operations may be expressed as follows:

1. MAKE-SET(x) creates a new set containing the single element x.
2. FIND(x) finds the set that contains the element x.
3. UNION(S, T) combines the sets S and T.

Tarjan [20] showed that any sequence of union-find operations can be performed with only slightly more than constant amortized time per operation, and we will take up further details of these operations in Section 3. In this section, we will analyze the connected components algorithm under the assumption that each union-find operation can be performed in constant time.

To facilitate the operation of the LEFT-COMPONENTS procedure, we slightly augment the MAKE-SET and UNION operations to maintain two additional pieces of information for each set S, which we refer to as $adjnext[S]$ and $adjprev[S]$.

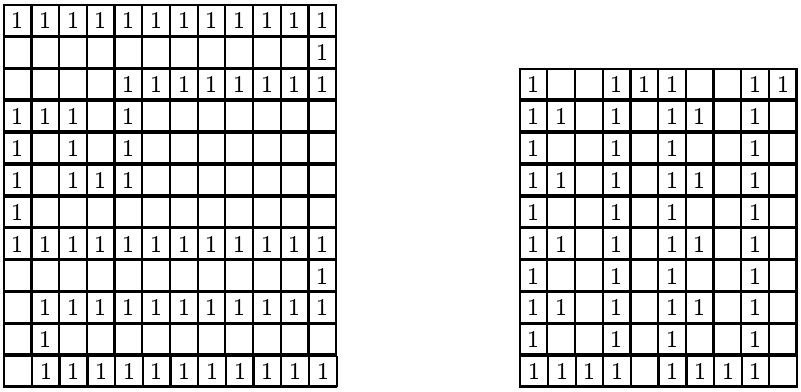
Algorithm CC

1 Find a left-connected component labeling of the image, placing the results for processor i in a local array *leftlabel*, such that *leftlabel*[j] is the label for the pixel in column i and row j.

2 Do the same for right-connected component labeling, with results going into the array *rightlabel*.

3 Within each processor, in parallel, perform component labeling on the graph with nodes $\{$ *leftlabel*[j] $\forall j \} \cup \{$ *rightlabel*[j] $\forall j \}$ and edges $\{ ($*leftlabel*[j], *rightlabel*[j]) $\forall j \}$.

Figure 2: The top-level procedure for the $O(n)$ component labeling algorithm.



(a) (b)

Figure 3: Images illustrating the difficulty of of left-component labeling. Empty boxes are 0-pixels.

Algorithm LEFT-COMPONENTS

1 Perform union-find operations on the pixels within each processor (column) to place all pixels belonging to the same left-component into the same set.

2 For each pixel, find which set it belongs to.

3 Assign the appropriate left-component label to each set.

4 Assign the appropriate left-component label to each pixel.

Figure 4: The high-level structure of the left-connected component labeling algorithm.

The value of *adjnext*[S] is some row index for which a 1-pixel of S is adjacent to a 1-pixel in the next column to the right; if no such index exists, *adjnext*[S] = NIL. Similarly, *adjprev*[S] is a row index for which a 1-pixel of S is adjacent to a 1-pixel in the *previous* column if such an index exists. It is easy to see that only a constant amount of local computation and a constant number of nearest neighbor communications are required to maintain the *adjnext* and *adjprev* information during any union-find operation.

We can now use the procedure UNION-FIND-PASS of Figure 5 to express step 1 of Algorithm LEFT-COMPONENTS. The basic idea for the correctness and efficiency of procedure UNION-FIND-PASS is embodied in the following lemma:

Lemma 1 UNION-FIND-PASS *produces the correct grouping of rows in each column, and completes in $O(n)$ time under the assumption that unions and finds are constant time.*

Proof. The basic idea of the procedure is as follows. In the first phase, it groups together the pixels that comprise vertical runs of 1's within the column (lines 1–7). That is, as we march down the column, whenever we find a continuation of a vertical run, we union the new pixel with the existing vertical run. In the second phase, we perform unions based on information about unions performed in the previous column. In both of these phases, we record in a queue for eventual transmission to the next column information about unions performed in the current column. The only unions for which information must be passed from one column to the next is what we may refer to as a *relevant* union. A union between two sets of pixels S_1 and S_2 in column i is relevant to column $i+1$ if S_1 and S_2 each contain at least one 1-pixel adjacent to a 1-pixel in column $i+1$.

We can prove the procedure yields the correct result by induction on the number of executions of line 12. To assist in the formalization, let us refer to the k-th such execution in processor i as $E_{i,k}$ and the grouping into components in column i produced by that call as $C_{i,k}$. The statement to be proved inductively is that $C_{i,k}$ is correct based on $C_{i-1,f(k)}$, where $E_{i-1,f(k)}$ is the latest execution in processor $i-1$ that does not enqueue data generating an execution later than $E_{i,k}$ in processor i. (This is sufficient to establish that column $n-1$ ends up with a correct grouping based on the final grouping in column $n-2$, which is correct based on the final grouping in column $n-3$, etc., so that the overall result is correct for grouping according to left-components.) The base case for the induction is established by arguing that the first phase of UNION-FIND-PASS (lines 1–7) produces the correct local grouping into components in column i given that no relevant unions have occurred in column $i-1$. For the induction step, consider $E_{i-1,k'}$, where this is the execution that unions sets S and T and enqueues the data that generates $E_{i,k}$. By the induction hypothesis, we know that $C_{i,k-1}$ is correct based on $C_{i-1,k'-1}$. Since S and T were already sets in $C_{i-1,k'-1}$, we know that $C_{i,k-1}$ already groups together within a set S' all pixels in column i that are adjacent to S and also groups together within a set T' all pixels in column i that are adjacent to T. Thus, to obtain a grouping in column i that is correct based on $C_{i-1,k'}$ we need only union S' and T'; this is achieved by $E_{i,k}$, which unions the sets containing a pixel adjacent to S and a pixel adjacent to T. Therefore, $C_{i,k}$ is correct based on $C_{i-1,k'}$. Finally, $C_{i,k}$ is also correct based on $C_{i-1,f(k)}$, since there are no unions strictly between $E_{i-1,k'}$ and $E_{i-1,f(k)+1}$ that are relevant to column i, and unions that are not relevant as defined above do not affect the grouping in column i.

For the bound on running time, note first that phase one of UNION-FIND-PASS (lines 1–7) is certainly $O(n)$ time. Some items may be enqueued during phase one, but this can only improve the speed relative to the situation in which all enqueues are viewed as occurring during phase two. Then we can argue inductively that the time for phase two is $O(n+i)$ in processor i. For the base case, we note that processor 0 has a trivial phase two, and the time for phase two is $O(n)$ even counting the time for enqueues that really occur in phase one. For the induction step, we note that with the enqueues counted as occurring in phase two, only a constant amount of time must pass after each enqueue until the corresponding dequeue in the next processor. ■

So far we have assumed without justification that each union-find operation can be completed in constant time, a matter to be revisited in Section 3. No such assumption is necessary to show that the remainder of procedure LEFT-COMPONENTS (steps 2–4) can be completed in $O(n)$ time. Steps 2 and 4 just involve a sequence of n finds in each processor independently in parallel. We will see in Section 3 that it is easy to ensure that these sequences can be completed in $O(n)$ time. In fact, by doing a find on every pixel in step 2, we can ensure that every later find is constant time, because no unions occur after step 1.

All that remains is to analyze step 3 of LEFT-COMPONENTS using the fact that we can make each find execute in constant time. It can be implemented with the procedure LABEL-PASS in Figure 6, that is somewhat similar to UNION-FIND-PASS. The basic idea here is to pass the label of each set to the right exactly once. A processor cannot wait for all incoming information to be received before sending information out, but it must wait to send out the label of a given set until it has received any incoming information for that particular set. The *adjprev* values of the sets are used to determine when to wait for incoming information. As in UNION-FIND-PASS, processor i receives all necessary information by the time of its $n+i$-th dequeue so that the total time for the pass is $O(n)$.

Incorporating the analysis of LEFT-COMPONENTS into the overall analysis of Algorithm CC yields:

Lemma 2 *Algorithm* CC *computes the component labeling in $O(n)$ time under the assumption that unions and finds are constant time.* ■

Proof. Putting together the analyses of all steps of Algorithm LEFT-COMPONENTS, we obtain correct operation in time $O(n)$ if all unions and finds are constant time. We can then find right components by an analogous right to left sweep. Finally, we put together these two labelings as specified by the last step of Algorithm CC in each processor independently; this step is also $O(n)$ time by the familiar sequential algorithm to find connected components on a graph of n edges. ■

3 The implementation details

The remaining algorithmic detail that we must consider is the implementation of the union-find operations, which we

```
      Algorithm UNION-FIND-PASS
 1    for j ← 0 to n − 1 do MAKE-SET(j) endfor
 2    Set outgoing[i] to an empty queue.
 3    for j ← 1 to n − 1 do
 4        if image[i, j − 1] = image[i, j] = 1
 5        then Call APPLY on the pair of rows (j − 1, j)
 6        endif
 7    endfor
 8    if i = 0 then incoming ← EOS else incoming ← NIL endif
 9    while incoming ≠ EOS do
10        incoming ← DEQUEUE(outgoing[i − 1])      ⟨returns NIL if empty queue⟩
11        if incoming does not equal NIL or EOS
12        then Call APPLY on incoming
13        endif
14    endwhile
15    Enqueue EOS onto outgoing[i].

      Algorithm APPLY(rowpair)
 1    Set topset to FIND-SET of the top row in rowpair.
 2    Set botset to FIND-SET of the bottom row in rowpair.
 3    if topset ≠ botset then
 4        if adjnext is non-NIL for topset and botset
 5        then Enqueue the pair of rows (adjnext[topset], adjnext[botset]) onto outgoing[i].
 6        endif
 7        UNION(topset, botset)
 8    endif
```

Figure 5: Pseudocode for processor i in the union-find pass of the left-component labeling algorithm. The dequeue on $outgoing[i-1]$ represents a constant number of communications with the processor to the left; other variables are local. The two-dimensional array *image* contains the pixel values.

```
      Algorithm LABEL-PASS
 1    for j ← 0 to n − 1 do
 2        S ← FIND(j)
 3        if adjprev[S] = NIL and label[S] is not set then
 4            label[S] ← in + j
 5            Enqueue (label[S], adjnext[S]) onto outgoing[i].
 6        endif
 7    endfor
 8    if i = 0 then incoming ← EOS else incoming ← NIL endif
 9    while incoming ≠ EOS do
10        incoming ← DEQUEUE(outgoing[i − 1])      ⟨returns NIL if empty queue⟩
11        if incoming does not equal NIL or EOS then
12            S ← FIND(the row specified by incoming)
13            label[S] ← the label specified by incoming
14            Enqueue (label[S], adjnext[S]) onto outgoing[i].
15        endif
16    endwhile
17    Enqueue EOS onto outgoing[i].
```

Figure 6: Pseudocode for processor i in the labeling pass corresponding to line 3 of Algorithm LEFT-COMPONENTS.

treated as constant time in the previous section; in actuality, the situation is more complicated. We will show that a simple implementation yields an $O(n \lg n)$ running time for Algorithm CC, that we can also obtain $O(n \lg n / \lg \lg n)$ running time, and that simple implementations are likely to do much better than the worst-case bound of $O(n \lg n)$.

Let us first consider the union-find approach that is probably most widely recognized as an efficient implementation. In this implementation, each set is represented by a tree, with each element other than the root having a pointer to its parent. The MAKE-SET operation creates a representation of an element as a lone tree node. A FIND involves walks up a path in the tree and returning the root, which serves as the name of the set. For a UNION, we are given the roots of two trees (generally obtained by executing two finds), and we make one root the parent of the other by creating one new parent pointer. Tarjan showed that by incorporating the two heuristics of *path compression* and *weighted union*, any sequence of union-find operations can be executed in very nearly constant amortized time per operation [20]. Path compression is applied during finds; after we walk up a path to find the root for some node, we make all nodes that were encountered on the find path point directly to that root. Weighted union affects the choice as to which of the two given roots becomes the root of the combined tree; we point the root of the smaller set to the root of the larger set.

Tarjan's analysis [20] shows that any sequence of n union-find operations on a set of n elements can be executed in time $O(n\alpha(n))$, where $\alpha(n)$ is a function that grows so slowly that it may be considered to have a value of at most 4 for all practical purposes. Thus, the average time of the operations is nearly constant.

Unfortunately, (nearly) constant average time per union-find operation is not enough to obtain an $O(n)$ SLAP algorithm for component labeling based on Lemma 1; individual operations could require more than constant time. Two observations, however, immediately follow from the union-find implementation analyzed by Tarjan. First, Algorithm CC can be implemented to run in $O(n \lg n)$ time. This follows from the fact that as long as we use weighted union, no node in any tree ever has depth greater than $\lg n$, so each individual union-find operation is $O(lgn)$ time. Second, all parts of the algorithm other than UNION-FIND-PASS can be completed in $O(n)$ time. Based on the arguments in Section 2, we just need to show that the n finds in step 2 of LEFT-COMPONENTS can be executed in time $O(n)$. This is true as long as we use path compression, because each pointer beyond one that is followed in a find leads to reduction of a node's depth to 1.

One way to achieve a better bound on the running time of Algorithm CC is to use a union-find implementation with a better worst-case time for each individual operation. A union-find implementation with $O(\lg n / \lg \lg n)$ time per operation is provided by Blum [3], which immediately yields the following result:

Theorem 3 *Algorithm* CC *can be implemented to run in time* $O(n \lg n / \lg \lg n)$. ■

Though the union-find implementation analyzed by Tarjan leads to a slightly poorer worst-case bound of $O(n \lg n)$ for component labeling on the SLAP, this implementation is likely to achieve better than worst-case performance in practice. We have noted that the bottleneck in the analysis is entirely in the UNION-FIND-PASS procedure. In fact, phase one of the procedure (lines 1–7) constitutes a very restricted sequence of union-find operations that can clearly be executed in $O(n)$ time in parallel in each processor. We need only worry about a sequence of much longer than average operations as we move across the processors tracing out the effect of an operation in one processor on succeeding processors in phase two. Several factors suggest that the $\Theta(n \lg n)$ worst-case behavior would be rare in practice. First, the bound of at most n executions of the procedure APPLY (containing two finds and a union) per processor in phase two is conservative; the number may be much less, depending on the image. Second, only certain sequences of n unions and finds will generate operations requiring $\Theta(\lg n)$ time. It may also be noted that the sequence of unions and finds in each processor is substantially restricted. Denote the sequence of row pairs on which the finds and unions (whenever the corresponding sets are unequal) occur in processor i based on the dequeues of information from the previous column as (t_1, b_1), then (t_2, b_2), etc with $t_k \leq b_k$. This sequence has the property that we never have t_k or b_k strictly between t_{k-1} and b_{k-1}. That is viewing the row pairs as intervals, the intervals do not intersect in more than one row, or the interval (t_k, b_k) contains the interval (t_{k-1}, b_{k-1}). Some encouragement is provided by various results on the complexity of a sequence of n operations given some advance knowledge about the structure of unions or a suitable probability distribution on the operations (though we must still have a concern regarding the complexity of individual operations) [8, 11, 14, 22].

It is also possible that improved performance can be obtained by having processors perform some path compression when they would otherwise just be waiting for union-find operations to be generated by the previous processor. If a processor can reduce the depth of its deepest nodes while waiting for a long operation in the previous processor, it will be able to execute the operation generated by the previous processor more quickly when it finally arrives. One possible approach is to have each processor execute the two finds specified by dequeued information in parallel and to enqueue a pair of finds for the next processor as soon as two pixels are found that are adjacent to 1-pixels in the next column. If the former processor later discovers that it was executing a pair of finds on two pixels that already belong to the same set, it could then quash the pair of finds it had previously passed to the next processor. In connection with the idea of replacing idle time with compression, it may be useful to apply a "one-pass" compression scheme such as the "halving" scheme shown by Tarjan and Van Leeuwen [21] to yield comparable performance to ordinary compression. With such a scheme progress is made on compression even if a find is aborted before reaching the root. (The "union by rank" variation on weighted union is also shown to be a good choice [21].)

The final version of this paper will report on experimental results of actual implementation of Algorithm CC.

4 An Extension and a Lower Bound for a Restricted Architecture

In this section, we extend the previous results to a more general type of component labeling, and we show that $\Omega(n \lg n)$

time is required on a weaker version of the SLAP architecture.

Recall that the algorithm of Section 2 labels each component with the position of its first pixel in a column-major ordering of all the pixels. Let us refer to this component labeling as the column-major labeling. The algorithm of Section 2 can be extended so that instead of producing the column-major labeling it will accept as input any initial labeling of the pixels and will then label each component with the minimum initial label of its pixels. In fact, we can generalize further to replace "minimum" with any binary operator that is associative and commutative, but we work with "minimum" here for simplicity:

Corollary 4 *There is a SLAP algorithm to solve the following problem in the same asymptotic time as to produce any component labeling: Given any set of initial labels of pixels, label the pixels of each component with the minimum initial label of its pixels.*

Sketch of proof. Begin by producing any component labeling, e.g., as specified in Section 2. Then computing locally within each processor, relabel each component with the minimum initial label of its pixels in the corresponding column of the image. Then use a process similar to Procedure LABEL-PASS to record for each component in processor i the minimum initial label of its pixels in the subimage composed of columns 0 through i; in the modified procedure, we have existing labels as specified above, and new labels are generated by taking the minimum of incoming and existing labels. Next, perform the same process in a right-to-left pass to record for each component in processor i the minimum initial label of its pixels in the subimage composed of columns i through $n-1$. Finally, within each processor, we just take the minimum of the two recorded labels for each component to obtain the overall minimum of initial labels of its pixels in the entire image. ■

For the result that $\Omega(n \lg n)$ time is required on a restricted SLAP, we consider a SLAP in which the amount of data that can be communicated between adjacent processors in one time step is just 1 bit instead of a word of $\lg n$ bits. We show that on such an architecture, $\Omega(n \lg n)$ time is required. We concentrate here on the problem of producing the column-major labeling as defined at the beginning of this section. The lower bound should hold even if we require only that pixels in the same component get the same label; this argument is deferred to the final paper.

Theorem 5 *A SLAP in which each pair of adjacent processors can exchange just one bit at any time step requires $\Omega(n \lg n)$ time to perform component labeling.*

Proof. As indicated above, we prove the result here for column-major labeling. Consider an image in which only the even-indexed rows contain 1-valued pixels. Then, to determine the labeling of pixels in the rightmost column, the rightmost processor must know the extent of the rightmost run of 1's in each row. That is, there are n possibilities for the correct label to assign to each 1-pixel in the rightmost column. Thus, there are $\Omega((n/2)^n)$ possible labelings of the rightmost column, implying that $\Omega(n \lg n)$ bits are required to describe the labeling. Since the rightmost processor begins with n bits of information and receives at most one additional bit of information on each time step, $\Omega(n \lg n)$ time is required before it can produce the correct labeling of the rightmost column. ■

5 Conclusion

This paper has shown that component labeling on an $n \times n$ image can be performed in $O(n \lg n / \lg \lg n)$ time on a SLAP of n processors, and also proposes an algorithm likely to perform better in practice. This paper has also shown that a SLAP allowing just one bit of communication between each pair of adjacent processors at a given time step requires $\Omega(n \lg n)$ time for the component labeling problem.

The most obvious open question is to narrow the gap between the upper bound of $O(n \lg n / \lg \lg n)$ and the lower bound of $\Omega(n)$ for the worst-case time of component labeling on the SLAP.

Acknowledgements

Thanks to David Helman and Joseph JáJá of the University of Maryland and Amihood Amir of Georgia Tech for helpful discussions.

References

[1] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] ALNUWEIRI, H. M., AND PRASANNA, V. K. Optimal geometric algorithms for digitized images on fixed-size linear arrays and scan-line arrays. *Distributed Computing 5* (1991), 55–65.

[3] BLUM, N. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing 15*, 4 (Nov. 1986), 1021–1024.

[4] CHIN, D., ET AL. The Princeton engine: A real-time video system simulator. *IEEE Trans. Consumer Electronics 34*, 2 (May 1988), 285–297.

[5] CYPHER, R., SANZ, J. L. C., AND SNYDER, L. Hypercube and shuffle-exchange algorithms for image component labeling. *Journal of Algorithms 10* (1989), 140–150.

[6] CYPHER, R., SANZ, J. L. C., AND SNYDER, L. Algorithms for image component labeling on SIMD mesh-connected computers. *IEEE Trans. Computers 39*, 2 (Feb. 1990), 276–181.

[7] DILLENCOURT, M. B., SAMET, H., AND TAMMINEN, M. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM 39*, 2 (Apr. 1992), 253–280.

[8] DOYLE, J., AND RIVEST, R. L. Linear expected time of a simple union-find algorithm. *Information Processing Letters 5*, 5 (Nov. 1976), 146–148.

[9] Fisher, A. L. Scan line array processors for image computation. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (1986), pp. 338–345.

[10] Fisher, A. L., and Highnam, P. T. Real-time image processing on scan line array processors. In *Proceedings of the IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management* (1985), pp. 484–489.

[11] Gabow, H. N., and Tarjan, R. E. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences 30*, 2 (1985), 209–221.

[12] Helman, D., and JáJá, J. Efficient image processing algorithms on the scan line array processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence 17*, 1 (Jan. 1995), 47–56. Earlier version in Proceedings of the 1993 International Conference on Parallel Processing.

[13] Knight, S., et al. The Sarnoff engine: A massively parallel computer for high definition system simulation. In *Proceedings of Application Specific Array Processors* (1992), pp. 342–357.

[14] Knuth, D. E., and Schönhage, A. The expected linearity of a simple equivalence algorithm. *Theoretical Computer Science 6* (1978), 281–315.

[15] Kumar, V. K. P., and Eshaghian, M. M. Parallel geometric algorithms for digitized pictures on mesh of trees. In *Proceedings of the 1986 International Conference on Parallel Processing* (1986), pp. 270–273.

[16] Levialdi, S. On shrinking binary picture patterns. *Communications of the ACM 15*, 1 (Jan. 1972), 7–10.

[17] Miller, R., and Stout, Q. Data movement techniques for the pyramid computer. *SIAM Journal on Computing 16*, 1 (1987), 38–60.

[18] Nassimi, D., and Sahni, S. Finding connected components and connected ones on a mesh-connected parallel computer. *SIAM Journal on Computing 9*, 4 (1980), 744–757.

[19] Schwartz, J. T., Sharir, M., and Siegel, A. An efficient algorithm for finding connected components in a binary image. Tech. Rep. 154, Department of Computer Science, NYU, Feb. 1985. Revised July, 1985.

[20] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22*, 2 (Apr. 1975), 215–225.

[21] Tarjan, R. E., and van Leeuwen, J. Worst-case analysis of set union algorithms. *Journal of the ACM 31*, 2 (Apr. 1984), 245–281.

[22] Yao, A. C. On the average behavior of set merging algorithms. In *Proceedings of the 8th ACM Symposium on Theory of Computing* (1976), ACM Press, pp. 192–195.