A New Parallel Algorithm for Two-Pass Connected Component Labeling

Siddharth Gupta, Diana Palsetia, Md. Mostofa Ali Patwary, Ankit Agrawal, Alok Choudhary
Department of Electrical Engineering & Computer Science, Northwestern University, Evanston, IL 60208, USA
siddharth.bitsian@gmail.com, {drp925, mpatwary, ankitag, choudhar}@eecs.northwestern.edu

Abstract—Connected Component Labeling (CCL) is an important step in pattern recognition and image processing. It assigns labels to the pixels such that adjacent pixels sharing the same features are assigned the same label. Typically, CCL requires several passes over the data. We focus on two-pass technique where each pixel is given a provisional label in the first pass whereas an actual label is assigned in the second pass.

We present a scalable parallel two-pass CCL algorithm, called PAREMSP, which employs a scan strategy and the best union-find technique called REMSP, which uses REM’s algorithm for storing label equivalence information of pixels in a 2-D image. In the first pass, we divide the image among threads and each thread runs the scan phase along with REMSP simultaneously. In the second phase, we assign the final labels to the pixels. As REMSP is easily parallelizable, we use the parallel version of REMSP for merging the pixels on the boundary. Our experiments show the scalability of PAREMSP achieving speedups up to 20.1 using 24 cores on shared memory architecture using OpenMP for an image of size 405,20 MB. We find that our proposed parallel algorithm achieves linear scaling for a large resolution fixed problem size as the number of processing elements are increased. Additionally, the parallel algorithm does not make use of any hardware specific routines, and thus is highly portable.

I. Introduction

One of the most fundamental operations in pattern recognition is the labeling of connected components in a binary image. Connected component labeling (CCL) is a procedure for assigning a unique label to each object (or a connected component) in an image. Because these labels are key for other analytical procedures, connected component labeling is an indispensable part of most applications in pattern recognition and computer vision, such as fingerprint identification, character recognition, automated inspection, target recognition, face identification, medical image analysis, and computer-aided diagnosis. In many cases, it is also one of the most time-consuming tasks among other pattern-recognition algorithms [1]. Therefore, connected component labeling continues to be an active area of research [2]–[9].

There exist many algorithms for computing connected components in a given image. These algorithms are categorized into mainly four groups [10] : 1) repeated pass algorithms [11], [12], 2) two-pass algorithms [13]–[21] 3) Algorithms with hierarchical tree equivalent representations of the data [22]–[29], 4) parallel algorithms [30]–[35]. The repeated pass algorithms perform repeated passes over an image in forward and backward raster directions alternately to propagate the label equivalences until no labels change. In two-pass algorithms, during the first pass, provisional labels are assigned to connected components; the label equivalences are stored in a one-dimensional or a two-dimensional table array. After the first pass, the label equivalences are resolved by some search. This step is often performed by using a search algorithm such as the union-find algorithm. The results of resolving are generally stored in a one-dimensional table. During the second pass, the provisional labels are replaced by the smallest equivalent label using the table. Since the algorithm traverses image twice, these algorithms are called two-pass algorithms. In algorithms that employ hierarchical tree structures i.e., n-ary tree such as binary-tree, quad-tree, octree, etc., the label equivalences are resolved by using a search algorithm such as the union-find algorithm. Lastly, the parallel algorithms have been developed for parallel machine models such as a mesh connected massively parallel processor. However all these algorithms share one common step, known as scanning step in which provisional label is given to each of the pixel depending on its neighbors.

In this paper we focus on two-pass CCL algorithms. The algorithm in [36] and [37] are two developed techniques for two-pass Connected Component Labeling. The algorithm in [36], which we refer to as CCLLRPC, uses a decision tree to assign provisional labels and an array-based union-find data structure to store label equivalence information. However, the technique employed for union-find, Link by Rank and Path Compression is not the best technique available [38]. The algorithm in [37], which we refer to as ARUN, employs a special scan order over the data and three linear arrays instead of the conventional union-find data structure. There exists a parallel implementation of ARUN on TILE64 many core platform [39]. According to the experimental results given in [39], the parallel implementation is able to achieve a speedup of 10 on 32 processor units. This implementation is also not portable due to its implementation for specific hardware architecture.

We propose two two-pass algorithms for labeling the connected components, AREMSP and CCLREMSP, which are based on REM’s union-find algorithm REMSP [40], [41] and the scan strategy of ARUN and CCLLRPC algorithms. Since REM’s union-find is an algorithm which implements immediate parent check test and compression technique called Splicing [40], [41], our proposed sequential two-pass algorithm AREMSP is 39% faster than CCLLRPC and 4% faster than ARUN. Another advantage of using REM’s union-find approach is that its parallel implementation is shown to scale better with increasing number of processor [38]. Parallel REM’s
Additionally, the parallel algorithm does not make use of any version of first pass, divides the image into equal proportions and executes equivalence, by adopting array based union-find algorithm that based union-find algorithm, which is used for storing label of two. Second strategy replaces the conventional pointer reduces the average number of neighbors accessed by a factor of two. The parallel implementation gets maximum speedup of 2.5 on 4 threads.

As mentioned in [10], there exist different types of CCL algorithms. Repeated pass or multi pass algorithm repeatedly scans the image forward and backward alternatively to give labels until no further changes can be made to the assigned pixels [11], [12]. The algorithm in [10], which we call Suzuki’s algorithm modifies the conventional multi pass algorithm using one-dimensional table. There exists a parallel implementation of Suzuki’s algorithm using OpenMP in [42]. According to experimental results in [42], the parallel implementation gets maximum speedup of 2.5 on 4 threads.

In any two-pass algorithm, there are two steps in scanning step: 1) examining neighbors of current pixel which already assigned labels to determine label for the current pixel, 2) storing label equivalence information to speed up the algorithm.

The algorithm in [36], which we refer to as CCLL RPC, provides two strategies to improve the running time of the algorithm. First strategy employs a decision tree, which reduces the average number of neighbors accessed by a factor of two. Second strategy replaces the conventional pointer based union-find algorithm, which is used for storing label equivalence, by adopting array based union-find algorithm that uses less memory. The union-find algorithm is implemented using Link by Rank and Path Compression technique.

The union-find data structure in [43] is replaced by a different data structure to process label equivalence information. In this algorithm, at any point, all provisional labels that are assigned to a connected component found thus far during the first scan are combined in a set \( S(r) \), where \( r \) is the smallest label and is referred to as the representative label. The algorithm employs \( r \) table for storing representative label of a set, \( next \) to find the next element in the set and \( tail \) to find the last element of the set.

In another strategy, which we call ARUN, the first part of scanning step employs a scanning technique, which processes image two lines at a time and process two image pixels at a time [37]. This algorithm uses the same data structure given in [43] for processing label equivalence information. The scanning technique reduces the number lines to be processed by half thereby improving the speed of the two-pass CCL method.

In this paper, we provide two different implementations of two-pass CCL algorithm. These two algorithms are different in their first scan step. In the first implementation called CCLREMSP, we have used the decision tree suggested by the CCLL RPC algorithm for the first part of scanning step but for the second part we have used REM’s union-find approach instead of Link by Rank and Path Compression technique. The union-find algorithm maintains a collection of disjoint sets where each set represents connected elements. [40] compares all of the different variations of union-find algorithms over different graph data sets and found that REM’s implementation is best among all the variations. Thus in our second implementation, called AREMSP, we process the image lines two by two as suggested by [37] but for the second step we use REMSP instead of the data structure used by [37].

We have compared both of our proposed implementations with CCLL RPC, RUN, and ARUN algorithms and find that AREMSP performs best among all the algorithms. Finally we have also provided a shared memory parallel implementation of AREMSP called PAREMSP using OpenMP. We use the parallel implementation of REMSP given in [38].

## II. Related Work

In section II, we provided related work on connected component labeling. In section III, we propose our sequential two-pass CCL algorithms CCLREMSP and AREMSP and the parallel algorithm PAREMSP in section IV. We present our experimental methodology and results in section V. We conclude our work in section VI. The abbreviations used in the paper and their brief description is given in Table I.

### Table I: Abbreviations used in the paper and their brief description

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCL</td>
<td>Connected Component Labeling</td>
</tr>
<tr>
<td>ARUN</td>
<td>CCL algorithm suggested by [37]</td>
</tr>
<tr>
<td>REMP SP</td>
<td>union-find technique proposed by Rem [40]</td>
</tr>
<tr>
<td>AREMSP</td>
<td>CCL algorithm proposed in our paper using scan strategy of ARUN and REMP SP</td>
</tr>
<tr>
<td>PAREMSP</td>
<td>Parallel implementation of AREMSP proposed in our paper</td>
</tr>
<tr>
<td>CCLL RPC</td>
<td>CCL algorithm suggested by [36]</td>
</tr>
<tr>
<td>CCLREMSP</td>
<td>CCL algorithm proposed in our paper using scan strategy of CCLL RPC and REMP SP</td>
</tr>
</tbody>
</table>

Throughout the paper, for an \( M \times N \) image, we denote \( image(a) \) to denote the pixel value of pixel \( a \). We consider binary images i.e. an image containing two types of pixels: object pixel and background pixel. Generally, we consider value of object pixel as 1 and value of background pixel as 0. The connected component labeling problem is to assign a label to each object pixel so that connected object pixels have the same label. In \( 2D \) images, there are two ways of defining connectedness: 4-connectedness and 8-connectedness. In this paper, we have only used 8-connectedness of a pixel.

### III. Proposed Algorithm

In CCLREMSP, we have used the decision tree suggested in CCLL RPC (Figure 2) for scanning and REM’s union-find algorithm REMSP for storing label equivalence. The full algorithm for CCLREMSP is given as Algorithm 1.
Algorithm 1 Pseudo-code for CCLREMSP

Input: 2D array image containing the pixel values
Output: 2D array label containing the final labels

1: function CCLREMSP(image)
2:    Scan_CCLRemSP(image) \triangleright Scan Phase of CCLREMSP
3:    \triangleright count is the max label assigned during Scan Phase
4:    flatten(p, count) \triangleright Analysis Phase of CCLREMSP
5: for row in image do \triangleright Labeling Phase of CCLREMSP
6:    for col in row do \triangleright e is the current pixel to be labeled
7:        label(e) \leftarrow p[label(e)]
8: end function

In the scan step of CCLREMSP, we process image lines one by one using the forward scan mask as shown in Figure 1a. We have used the decision tree proposed by [36] for determining the provisional label of current pixel e as we can reduce the number of neighbors using decision tree. Instead of examining all four neighbors of pixel, say \(e\), i.e. \(a, b, c\) and \(d\), we only examine the neighbors according to a decision tree as shown in Figure 2 [36]. Let \(label\) denote the 2D array storing the labels and let \(p\) denote equivalence array then according to CCLLRPC algorithm, three functions used by this decision tree are defined as follows:

1. The one-argument copy function, \(copy(a)\), contains one statement: \(label(e) = p[label(a)]\)
2. The two-argument copy function, \(copy(c,a)\), contains one statements: \(label(e) = merge(p[label(c)], label(a))\)
3. The new label function sets \(count\) as \(label(e)\), appends \(count\) to array \(p\), and increments \(count\) by 1.

![Forward Scan Mask](a) Forward Scan Mask for RemSP
(b) Forward Scan Mask for ARemSP

Fig. 1: Forward Scan Mask

The implementation of \(Scan\_CCLRemSP\) is given as Algorithm 4. However, the implementation of \(MERGE\) operation in our proposed algorithm REMSP is different from that of in CCLLRPC. We have used the implementation of union-find proposed by REM’s for merge operation [40], [41]. REM’s integrates the union operation with a compression technique known as Splicing (SP).

In the \(MERGE\) algorithm, if \(x\) and \(y\) are the nodes to be merged then we set \(root_x\) to \(x\) and \(root_y\) to \(y\). When \(root_x\) is to be moved to \(p[root_y]\), firstly \(p[root_x]\) is stored in a temporary variable \(z\) then \(p[root_x]\) is set to \(p[root_y]\), making the subtree rooted at \(root_x\) a sibling of \(root_y\) and finally \(root_x\) is set to \(z\). The algorithm for \(MERGE\) is given as Algorithm 2. After the first step, we carry out the analysis phase using the \(FLATTEN\) algorithm. In the \(FLATTEN\) algorithm, we give smallest equivalent label of every connected component to all the pixels belonging to that connected component. The algorithm also generates consecutive labels. The algorithm for \(FLATTEN\) is given as Algorithm 3.

Algorithm 2 Pseudo-code for merge [40]

Input: 1D array \(p\) and two nodes \(x\) and \(y\)
Output: The root of united tree

1: function MERGE(p,x,y)
2:    root_x \leftarrow x, root_y \leftarrow y
3:    while \(p[root_x] \neq p[root_y]\) do
4:        if \(p[root_x] > p[root_y]\) then
5:            if \(root_x = p[root_y]\) then
6:                \(p[root_x] \leftarrow p[root_y]\)
7:                return \(p[root_x]\)
8:                \(z \leftarrow p[root_x], p[root_x] \leftarrow p[root_y], root_x \leftarrow z\)
9:            else
10:               \(p[root_x] = p[root_y]\) then
11:               \(p[root_y] \leftarrow p[root_x]\)
12:               return \(p[root_y]\)
13:               \(z \leftarrow p[root_y], p[root_y] \leftarrow p[root_x], root_y \leftarrow z\)
14:               return \(p[root_y]\)
15: end function

Algorithm 3 Pseudo-code for flatten [36]

Input: 1D array \(p\) containing the equivalence info
Input: Max value of provisional label \(count\)

1: function FLATTEN(p,count)
2:    \(k \leftarrow 1\)
3: for \(i = 1\) to \(count\) do
4:    if \(p[i] < i\) then
5:        \(p[i] \leftarrow p[p[i]]\)
6:    else
7:        \(p[i] \leftarrow k\)
8:    \(k \leftarrow k + 1\)
9: end function

Algorithm 4 Pseudo-code for flatten [36]

Input: 1D array \(p\) containing the equivalence info
Input: Max value of provisional label \(count\)

1: function FLATTEN(p,count)
2:    \(k \leftarrow 1\)
3: for \(i = 1\) to \(count\) do
4:    if \(p[i] < i\) then
5:        \(p[i] \leftarrow p[p[i]]\)
6:    else
7:        \(p[i] \leftarrow k\)
8:    \(k \leftarrow k + 1\)
9: end function

Algorithm 5 Pseudo-code for flatten [36]

Input: 1D array \(p\) containing the equivalence info
Input: Max value of provisional label \(count\)

1: function FLATTEN(p,count)
2:    \(k \leftarrow 1\)
3: for \(i = 1\) to \(count\) do
4:    if \(p[i] < i\) then
5:        \(p[i] \leftarrow p[p[i]]\)
6:    else
7:        \(p[i] \leftarrow k\)
8:    \(k \leftarrow k + 1\)
9: end function

B. AREMSP Algorithm

In AREMSP, we have used the decision tree suggested in ARUN for scanning and REM’s union-find algorithm for storing label equivalence. The full algorithm for AREMSP is given as Algorithm 5.

In the first scan step of AREMSP, we process an image two lines at a time and two pixels at a time using the mask shown in Figure 1b, which is suggested in [37].
Algorithm 4 Pseudo-code for CCLRemSP Scan Phase

Input: 2D array \( image \) containing the pixel values

InOut: 2D array \( label \) containing the provisional labels and 1D array \( p \) containing the equivalence info

Output: maximum value of provisional label in \( count \)

1: function SCAN_CCLRemSP(image)
2:    for row in image do
3:        for col in row do
4:            if image(e) = 1 then
5:                if image(b) = 1 then
6:                    copy(b)
7:                else
8:                    if image(c) = 1 then
9:                        if image(a) = 1 then
10:                           copy(c, a)
11:                        else
12:                            if image(d) = 1 then
13:                               copy(c, d)
14:                            else
15:                               copy(c)
16:                        else
17:                            if image(a) = 1 then
18:                               copy(a)
19:                        else
20:                            if image(d) = 1 then
21:                               copy(d)
22:                            else
23:                               new label
24:                    return count
25:    end function

Algorithm 5 Pseudo-code for ARemSP

Input: 2D array \( image \) containing the pixel values

Output: 2D array \( label \) containing the final labels

1: function ARemSP(image)
2:    Scan_ARemSP(image) ▷ Scan Phase of RemSP
3:    flatten(p, count) ▷ count is the max label assigned during Scan Phase
4:    for row in image do ▷ Analysis Phase of RemSP
5:        for col in row do ▷ \( e \) is the current pixel to be labeled
6:            label(e) ← p[label(e)]
7:    end function

We assign the label to both \( e \) and \( g \) simultaneously. If both \( e \) and \( g \) are background pixels, then nothing needs to be done. If \( e \) is a foreground pixel and there is no foreground pixel in the mask, we assign a new provisional label to \( e \) and if \( g \) is a foreground pixel, we will assign the label of \( e \) to \( g \). If there are foreground pixels in the mask, then we assign \( e \) any label assigned to foreground pixels. In this case, if there is only one connected component in the mask then there is no need for label equivalence. Otherwise, if there are more than one connected components in the mask and as they are connected to \( e \), all the labels of the connected components are equivalent labels and hence need to be merged. For all the cases, one can refer to [37]. However, our implementation for the merge operation is different from [37]. We use the implementation of union-find proposed by Rem [40], [41] for the merge operation in ARemSP. Similar to CCLRemSP, we use FLATTEN for analysis phase and generating consecutive labels. The implementation of SCAN_ARemSP is given as Algorithm 6.

Algorithm 6 Pseudo-code for ARemSP Scan Phase

Input: 2D array \( image \) containing the pixel values

InOut: 2D array \( label \) containing the provisional labels and 1D array \( p \) containing the equivalence info

Output: maximum value of provisional label in \( count \)

1: function SCAN_ARemSP(image)
2:    for row in image do
3:        for col in row do
4:            if image(e) = 1 then
5:                if image(d) = 0 then
6:                    if image(b) = 1 then
7:                        label(e) ← label(b)
8:                    if image(f) = 1 then
9:                        merge(p, label(e), label(f))
10:                else
11:                    if image(f) = 1 then
12:                        label(e) ← label(f)
13:                    if image(a) = 1 then
14:                        merge(p, label(a))
15:                    if image(c) = 1 then
16:                        merge(p, label(e), label(c))
17:                else
18:                    if image(a) = 1 then
19:                        label(e) ← label(a)
20:                    if image(c) = 1 then
21:                        merge(p, label(e), label(c))
22:                else
23:                    if image(e) = 1 then
24:                        label(e) ← label(c)
25:                else
26:                    label(e) ← count,
27:                    p[count] ← count,
28:                    count + ++
29:            end function

IV. Parallelizing ARemSP Algorithm

We now describe the parallel implementation of ARemSP algorithm on a shared memory system. We make the assumption about memory model as stated in OpenMP regarding the atomic directive. We assume that memory read/write operations are atomic and any operations issued concurrently by different processors will be executed in some unknown sequential order if no ordering constructs are being used. However, two dependent operations issued by the same processor will always
be applied in the same order as they are issued.

Algorithm 7 Pseudo-code for PAREmSP

Input: 2D array image containing the pixel values
Output: 2D array label containing the final labels

1: function PAREmSP(image)
2:   numiter ← row/2
3:   chunk ← numiter/numberofthreads
4:   size ← 2 × chunk
5:   start ← index of the thread
6:   count ← start × col
7: end function

Pseudo-code for MERGER [38]

Input: 2D array p and two nodes x and y
Output: The root of united tree

1: function MERGER(p, x, y)
2:   root_x ← x, root_y ← y
3:   while p[root_x] ≠ p[root_y] do
4:     if p[root_x] > p[root_y] then
5:       root_x ← p[root_x]
6:       success ← 0
7:       omp_set_lock(&lock_array[root_x])
8:       if root_x = p[root_x] then
9:         p[root_x] ← p[root_y]
10:        success ← 1
11:       omp_unset_lock(&lock_array[root_x])
12:      if success = 1 then
13:        break
14:     else
15:     if root_y = p[root_y] then
16:        success ← 0
17:        omp_set_lock(&lock_array[root_y])
18:       if root_y = p[root_y] then
19:         p[root_y] ← p[root_x]
20:        success ← 1
21:       omp_unset_lock(&lock_array[root_y])
22:      if success = 1 then
23:        break
24:      z ← p[root_x], p[root_y] ← p[root_y], root_y ← z
25:   end function

In PAREmSP, the image is divided row-wise into chunks of equal size and given to the threads. In the first step, each thread runs Scan Phase of AREMSP on it's chunk simultaneously. We initialize the label to the start index of the thread for every thread so that no two pixels in the image have the same label after the first step. After the first step, each pixel is given a provisional label. Next, the pixels at the boundary of each chunk need to be merged to get the final labels. In the second step, we merge the boundary pixels using parallel implementation of Rem’s Algorithm [38] which we call as MERGER. In MERGER, if a thread wants to perform merging, it will first acquire the necessary lock. Once it gets the lock, it will check whether the node is still a root node. If yes, then the thread will set the parent pointer and release the lock. On the other hand if some other processor has altered the parent pointer so that the node is no longer a root, the processor will release the lock and continue executing the algorithm from its current position.

V. Experiments

For the experiments we used a computing node of Hopper, a Cray XE6 distributed memory parallel computer. The node has 2 twelve-core AMD ‘MagnyCours’ 2.1-GHz processors and 32 GB DDR3 1333-MHz memory. Each core has its own L1 and L2 caches, with 64 KB and 512 KB, respectively. One 6-MB L3 cache is shared between 6 cores on the MagnyCours processor. All algorithms were implemented in C using OpenMP and compiled with gcc.

Our test data set consists of four types of image data set: Texture, Aerial, Miscellaneous and NLCID. First three data sets are taken from the image database of the University of Southern California. The fourth data set is taken from US National Cover Database 2006. All of the images are converted to binary images by MATLAB using im2bw(level) function with level value as 0.5. The function converts the grayscale image to a binary image by replacing all pixels in the image input with luminance greater than 0.5 with the value 1 (white) and replaces all other pixels with the value 0 (black). If the input image is not a grayscale image, im2bw converts the input image to grayscale, and then converts this grayscale image to binary (Figure 3). However, note that our algorithm can be easily extended to gray scale images.

Texture, Aerial and Miscellaneous data set contain images of size 1 MB or less. NCLID data set contains images of size bigger than 12 MB. The biggest image in the data set is 405.20 MB.

Firstly, we performed the experiment over all the sequential algorithms. The experimental results are shown in Table II. In the table, we have shown the minimum, maximum and average execution time of all the four data sets. The execution time of AREMSP is lowest among all the sequential algorithms. Thus AREMSP is best among all the sequential algorithms.

Next, we show our results for the parallel algorithm.
PAREMSP over all the images. We have shown the speedup for data sets (except NCLD image data set) in Figure 4. The minimum, maximum and average execution time of PAREMSP for all the datasets is also shown in Table IV. We get a maximum speedup of 10 in this case as the images are 1 MB or less in size. The speedup also decreases in some cases as the number of threads increases. This case occurs when the image size is small. As the number of threads increase, each threads has less work, and therefore the thread creation and termination overhead will affect the performance. Figure 5a-5b shows the speedup of the algorithm for NCLD image data set. The size of the images are given in Table III. We get a maximum speedup of 20.

Therefore, our parallel implementation is able to achieve near linear speed for large data sets.

### VI. Conclusion

In this paper, we presented two sequential CCL algorithms CCLREMSP and AREMSP which are based on union-find technique of REM’s algorithm and scan strategies of ARUN and CCLRPC algorithms. CCLREMSP algorithm uses the scan strategy of CCLRPC algorithm whereas AREMSP uses the scan strategy of ARUN algorithm. Based on the experiments, we found out that AREMSP outperforms all the other sequential algorithms. We also implement a portable parallel implementation of AREMSP for shared memory computers with standard OpenMP directives. Our proposed algorithm, PAREMSP, divides the image into equal proportions and executes the scan. To merge the provisional labels on the image boundary, we use the parallel version of REM’s algorithm. Our experimental results conducted on a shared memory computer show scalable performance, achieving speedups up to a factor of 20.1 when using 24 cores on data set of size 465.20 MB. Thus, our parallel algorithm achieves linear scaling for large fixed problem size while the number of processing elements are increased.

### Acknowledgment

This work is supported in part by the following grants: NSF awards CCF-0833131, CNS-0830927, IIS-0905205, CCF-0938000, CCF-1029166, ACI-1144061, and IIS-1343639; DOE awards DE-FG02-08ER25848, DE-SC0001283.
DE-SC0005309, DESC0005340, and DESC0007456; AFOSR award FA9550-12-1-0458.

References


