A New Parallel Algorithm for Two-Pass Connected Component Labeling

***Abstract*—Connected Component Labeling(CCL) is one of the most important step in pattern recognition and image processing. Connected component labeling assigns labels to a pixel such that adjacent pixels sharing the same features are assigned the same label. Typically, CCL requires several passes over the data. For example, in a two-pass technique, the first pass, each pixel is given a provisional label and label equivalence information is stored. In the second pass,an actual label is given to each pixel. Suzuki et al have proposed two algorithms for CCL with two-pass technique called Link by Rank and Path Compression(LRPC), and ARun. The *LRPC* algorithm uses a decision tree to assign provisional labels and an array-based union-find datastructure to store label equivalence information. The *ARun* algorithm employs a special scan order over the data and three linear arrays instead of the conventional union-find datastructure. To the best of our knowledge, there has not been any effort yet on parallelizing two-pass CCL for shared memory architecture.**

We present a scalable parallel two-pass CCL algorithm called *PARemSP*, which employs scan strategy of *ARun* algorithm and the best union-find technique called *RemSP* 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 strategy of *ARun* algorithm along with *RemSP* simultaneously. 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 for an image of size 22822×20384. 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 [4]. Therefore, connected-component labeling continues to be an active area of research [5]–[12].

There exist many algorithms for computing connected components in a given image. These algorithms are categorized into mainly four groups [13] : 1) repeated pass algorithms, 2) two-pass algorithms 3) Algorithms with hierarchical tree equivalent representations of the data, 4) parallel algorithms. 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. As the algorithm traverses image twice that's why 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. Hoverver all these algorithms shares 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. [1], and [2] are two developed techniques for two-pass CCL algorithms. The algorithm in [1], which we refer to as LRPC, uses a decision tree to assign provisional labels and an array-based union-find datastructure to store label equivalence information. However, the technique for employed for union-find, Link by Rank and Path Compression is not the best technique available [15]. The algorithm in [2], which we refer to as ARUN, employs a special scan order over the data and three linear arrays instead of the conventional union-find datastructure. However, this unconventional implementation is not suited for parallel implementation.

We propose 2 two-pass algorithm for labeling the connected components called AREMSP and REMSP, which are based on REM union-find algorithm [3] and the scan strategy of ARUN and LRPC algorithms. Since REM union-find is an interleaved algorithm which implements immediate parent check test and compression technique called *Splicing* [3], our proposed sequential two-pass algorithm AREMSP is 39% faster than LRPC and 4% faster than ARUN. Another advantage of using REM union-find approach is that its parallel implementation is shown to scale better with increasing number of processor [15]. Parallel REM union-find implementation thus allows us to process the pixels of the image in any order. Therefore, we propose a parallel implementation of our proposed sequential algorithm two-pass CCL algorithm called PAREMSP. For scalability, our algorithm in the first pass, divides the image into equal proportions and executes the scan strategy of ARUN algorithm along with REMSP concurrently on each portion of the image. To merge the provisional labels on the image boundary, we use the parallel version of REMSP

[15]. Our experiments show the scalability of PAREMSP achieving speedups up to 20.1 using 24 cores on shared memory architecture for an image of size $22,822 \times 20,384$. Additionally, the parallel algorithm does not make use of any hardware specific routines, and thus is highly portable.

The remainder of this paper is organized as follows. In section II, we provided related work on connected component labeling. In section III, we propose our sequential two-pass CCL algorithm AREMSP and it's parallel version in section IV. We present our experimental methodology and results in section V. We conclude our work and propose future work in section VI.

II. Related Work

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 [1], which we refer to as LRPC, provides two strategies to improve the running time of the algorithm. First strategy reduces the average number of neighbors accessed by factor of 2 by employing a decision tree. Second strategy replaces the conventional pointer based union-find algorithm, which is used for storing label equivalence, by 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 [14] 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 $rtable$ 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 image pixels two by two [2]. This algorithm uses the same data structure given in [14] 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 REMSP, we have used the decision tree suggested by the LRPC algorithm for the first part of scanning step but for the second part we have used REM union-find approach instead of Link by Rank and Path Compression technique. [3] compares all of the different variations of union-find algorithms over different graph data sets and found that REM 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 [2] but for the second step we use REMSP instead of the data structure used by [2].

We have compared both of our proposed implementations with LRPC, RUN, and ARUN algorithms and found out that AREMSP performs best among all the algorithms. Finally we have also provided a shared memory parallel implementation of REMSP called AREMSP using OpenMP. We use the parallel implementation of REMSP given in [15].

III. Proposed Algorithm

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 containing of 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 the 8-connectedness of the pixel.

A. REMSP Algorithm

In the first scan step of REMSP, we process image lines one by one using the forward scan mask as shown in Figure 1b. We have used the decision tree proposed by [1] 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 desicion tree as shown in Figure 2. Let $label$ denote the $2D$ array storing the labels and let p denote equivalence array then according to LRPC 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.

a	b	c
d	e	
f	g	

(a) Forward Scan Mask for *ARemSP*

a	b	c
d	e	

(b) Forward Scan Mask for *RemSP*

Fig. 1: Forward Scan Mask

However, the implementation of MERGE operation in our proporsed algorithm REMSP is different from that of in LRPC. We have used the implementation of union-find proposed by *Rem* [3] for merge operation. *Rem* integrates the Union operation with a compression technique known as Splicing (sp). In the case when $rootx$ is to be moved to $p(rootx)$ it works as follows: just before this operation, $rootx$ is stored in a temporary variable z and then, just before moving $rootx$ up to its parent $p(z)$, $p(rootx)$ is set to $p(rooty)$, making the subtree rooted at $rootx$ a sibling of $rooty$. This neither compromises the increasing parent property (because $p(rootx) < p(rooty)$) nor invalidates the set structures (because the two sets will have been merged when the operation ends.) The effect of sp is that each new parent has a higher value than the value of the old

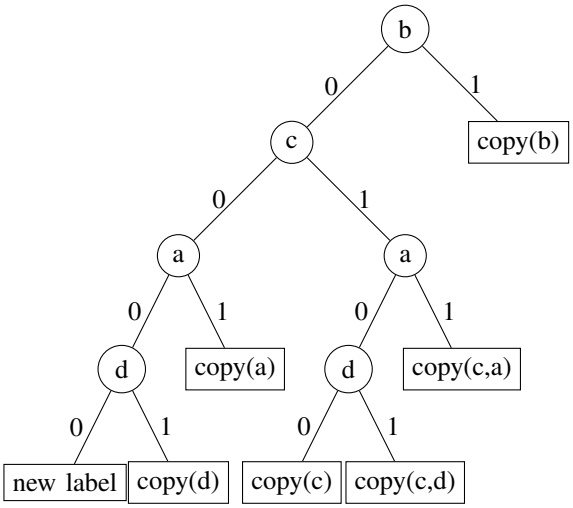


Fig. 2: Decision tree for *RemSP*

parent, thus compressing the tree. The algorithm for MERGE is given as Algorithm 4. After the first step, we carry out the analysis phase using FLATTEN algorithm. FLATTEN algorithm also generates consecutive labels. The algorithm for FLATTEN is given as Algorithm 2. The full algorithm for REMSP is given as Algorithm 1. The implementation of $RemSP - I$ is given in Appendix A

Algorithm 1 Pseudo-code for RemSP

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

```
1: function REMSP(image)
2:     RemSP − I(image)                    ▷ Scan Phase of RemSP
3:     flatten(p, count)                   ▷ Analysis Phase of RemSP
4:     for row in image do                 ▷ Labeling Phase of RemSP
5:         for col in row do
6:             label(e) ← p[label(e)]
7:         end for
8:     end for
9: end function
```

Algorithm 2 Pseudo-code for flatten

InOut: $1D$ array p containing the equivalance info
Input: Max value of provisional label $count$

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

B. AREMSP Algorithm

In the first scan step of AREMSP, we process image two lines at a time and processes pixels two by two using the mask shown in figure 1a suggested in [2]. We will give 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 give 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 equivalance. Otherwise, if there are more than one connected component in the mask and as they are connected to e so all the labels of the connected components are equivalent labels and needs to be merged. For all the cases, one can refer [2]. However, our implementation of the union-find is different from [2]. We use the implementation of union-find proposed by *Rem* [3] for merge operation in AREMSP. We use FLATTEN for analysis phase and generating consequtive labels. The full algorithm for AREMSP is given as Algorithm 3. The implementation of $RemSP - I$ is given in Appendix A

Algorithm 3 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:     ARemSP − I(image)                   ▷ Scan Phase of RemSP
3:     flatten(p, count)                   ▷ Analysis Phase of RemSP
4:     for row in image do                 ▷ Labeling Phase of RemSP
5:         for col in row do
6:             label(e) ← p[label(e)]
7:         end for
8:     end for
9: end function
```

Algorithm 4 Pseudo-code for merge

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

```
1: function MERGE(p,x,y)
2:     rootx ← x, rooty ← y
3:     while p[rootx] ≠ p[rooty] do
4:         if p[rootx] > p[rooty] then
5:             if rootx = p[rootx] then
6:                 p[rootx] ← p[rooty]
7:                 return p[rootx]
8:             end if
9:             z ← p[rootx], p[rootx] ← p[rooty], rootx ← z
10:        else
11:            if rooty = p[rooty] then
12:                p[rooty] ← p[rootx]
13:                return p[rootx]
14:            end if
15:            z ← p[rooty], p[rooty] ← p[rootx], rooty ← z
16:        end if
17:    end while
18:    return p[rootx]
19: end function
```

IV. Parallelizing AREMSP Algorithm

In the following we describe how one can run AREMSP algorithm in parallel on a shared memory system. In doing so we make the assumption that memory read/write operations are atomic and any operations issued concurrently by different processors will be executed in some unknown sequential order

unless specific constructs are used to ensure an ordering. However, two dependent operations issued by the same processor will always be applied in the same order as they are issued. This is in accordance with the memory model when using the atomic directive in *OpenMP*.

In PAREMSP, we divide the image among threads row-wise. The image is divided into chunks of equal size and given to the threads. In the first step, each thread run Phase-I 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. Now 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 [15]. We implement the parallel algorithm using OpenMP directives *pragma omp parallel* and *pragma omp for*. The pseudo code for parallel implementation of Rem's Algorithm is given as Algorithm 6. The pseudo code of PAREMSP is given as Algorithm 5.

Algorithm 5 Pseudo-code for PARemSP

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

function PAREMSP($image$)
  $numiter \leftarrow row/2$ ▷ As we are processing 2 rows at a time
  # pragma omp parallel
  $chunk \leftarrow numiter/numberofthreads$
  $size \leftarrow 2 \times chunk$
  $start \leftarrow$ start index of the thread
  $count \leftarrow start \times col$
  # pragma omp for
  $ARemSP - I(image)$
  # pragma omp for
  for $i = size$ to $row - 1$ **do**
    for col in row **do**
      if $label(e) \neq 0$ **then**
        if $label(b) \neq 0$ **then**
          $merger(p, label(e), label(b))$
        else
          if $label(a) \neq 0$ **then**
            $merger(p, label(e), label(a))$
          end if
          if $label(c) \neq 0$ **then**
            $merger(p, label(e), label(c))$
          end if
        end if
      end if
    end for
    $i \leftarrow i + size$
  end for
  $flatten(p, count)$
  for row in $image$ **do**
    for col in row **do**
      $label(e) \leftarrow p[label(e)]$
    end for
  end for
end function

Algorithm 6 Pseudo-code for merger

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

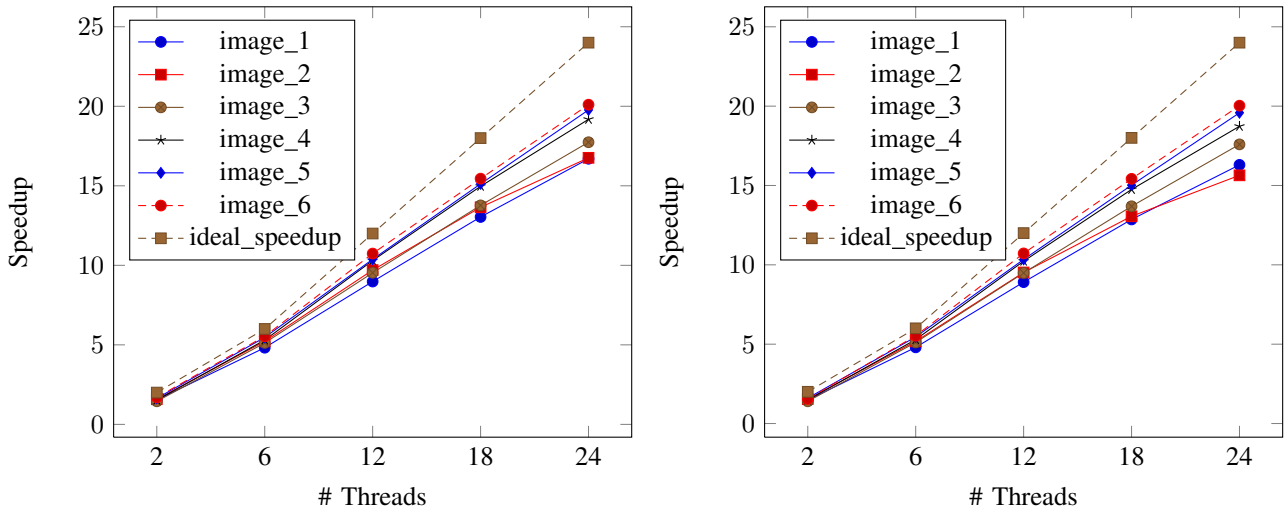
function MERGER(p,x,y)
  $rootx \leftarrow x, rooty \leftarrow y$
  while $p[rootx] \neq p[rooty]$ **do**
    if $p[rootx] > p[rooty]$ **then**
      if $rootx = p[rootx]$ **then**
        $omp\_set\_lock(\&(lock\_array[rootx]))$
        $success \leftarrow 0$
        if $rootx = p[rootx]$ **then**
          $p[rootx] \leftarrow p[rooty]$
          $success \leftarrow 1$
        end if
        $omp\_unset\_lock(\&(lock\_array[rootx]))$
        if $success = 1$ **then**
          break
        end if
      end if
      $z \leftarrow p[rootx], p[rootx] \leftarrow p[rooty], rootx \leftarrow z$
    else
      if $rooty = p[rooty]$ **then**
        $omp\_set\_lock(\&(lock\_array[rooty]))$
        $success \leftarrow 0$
        if $root = p[rooty]$ **then**
          $p[rooty] \leftarrow p[rootx]$
          $success \leftarrow 1$
        end if
        $omp\_unset\_lock(\&(lock\_array[rooty]))$
        if $success = 1$ **then**
          break
        end if
      end if
      $z \leftarrow p[rooty], p[rooty] \leftarrow p[rootx], rooty \leftarrow z$
    end if
  end while
  return $p[rootx]$
end function

V. Experiments

For the experiments we used Hopper. Hopper is NERSC's first peta-flop system, a Cray XE6, with a peak performance of 1.28 Petaflops/sec, 153, 216 compute cores for running scientific applications, 217 terabytes of memory, and 2 petabytes of online disk storage. All algorithms were implemented in C using OpenMP and compiled with gcc.

Our test dataset consists of 4 types of image dataset: Texture, Arial, Miscellaneous and NLCD. First three datasets are taken from the image database of the University of Southern California []. The fourth dataset is taken from US National Cover Database 2006 []. All of the images are converted to binary images by means of MATLAB. Texture, Arial and Miscellaneous dataset contain images of size 1024×1024 or less. NCLD dataset contains images of size bigger than 3000×4000. The biggest image in the dataset is $22,822 \times 20,384$.

Firstly, we did the experiment over all the sequential algorithms. The experimental results are shown in Table I. In the table, we have shown the minimum, maximum and average execution time of all the 4 datasets. As we can see that execution time of *ARemSP* is lowest among all the sequential algorithms thus *ARemSP* is best among all the sequential algorithms. Then we tested the parallel algorithm *PARemSP* over all the images. Fig 3a-3b shows the speedup of the algorithm for NCLD image dataset.The images are labeled



(a) local **(b)** local + merge

Fig. 3: Speedup for different images and different numbers of threads for NLCD dataset

TABLE I: Comparison of various execution times[*msec*] for sequential algorithms

Image type		LRPC	RemSP	ARun	ARemSP
Arial	Min	2.5	2.48	1.98	1.95
	Average	13.68	13.25	11.90	11.86
	Max	86.64	80.90	72.92	70.17
Texture	Min	2.07	2.06	1.58	1.53
	Average	8.42	8.20	7.32	7.27
	Max	16.86	16.18	14.81	14.47
Miscellaneous	Min	0.50	0.49	0.36	0.36
	Average	3.28	3.21	2.75	2.74
	Max	12.96	12.81	11.30	11.20
NLCD	Min	4.61	4.46	3.77	3.75
	Average	307.66	299.55	244.88	242.59
	Max	130.72	127.38	103.65	102.14

in the increasing order of their sizes. We get a maximum speedup of 20.1 for image of size 22822×20384. Fig 3a shows the speedup for *Phase-I* of PAREMSP i.e. the local computation and fig 3b shows the overall speedup (i.e. local + merge). We can see that there is not significant difference between both the speedups, implying that merge operation does not have a significant overhead. We can also see from the graph that as the image size increases, speedup increases so we can conclude that we will get linear speedup as the image size increases. We have also shown the speedup for all the other datasets in fig 4. We get a maximum seedup of 10 in this case as the images are small in size. The speedup also decreases in some cases as the number of threads increases. This is because the image size is small so as the number of threads increases, the threads will have less work to perform and the overhead due to thread creation will increase.

VI. Conclusion

In this paper, we presented 2 sequential CCL algorithms REMSP and AREMSP which are based on union-find technique of REM algorithm and scan strategies of ARUN and LRPC algorithms. REMSP algorithm uses the scan strategy of LRPC algorithm whereas ARUN uses the scan strategy of ARUN

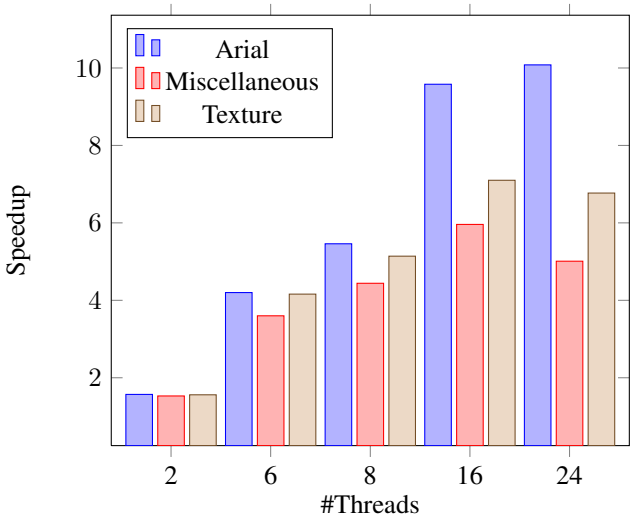


Fig. 4: Speedup for different images and different numbers of threads for Arial, Texture & Miscellaneous dataset

algorithm. Based on the experiments, we found out that AREMSP outperforms over all the other sequential algorithms. Then we present the parallel version of AREMSPwhich gives a maximum speedup of 20.1 on 24 threads. Fig 3 indicates that the parallel algorithm is scalable. As the algorithm just uses the standard *OpenMP* directives thus is easily portable.

Appendix

The scan phase of REMSP and AREMSP are given as Algorithm 7 and 8 respectively. The scan algorithms are based on the forward scan masks as shown in fig 1. The scan algorithms also use the MERGE algorithm which is given as Algorithm 4.

Algorithm 7 Pseudo-code for RemSP Phase-I

Input: $2D$ array $image$ containing the pixel values
InOut: $2D$ array $label$ containing the privisonal labels and $1D$ araay p containing the equivalance info
Output: maximum value of provisional label in $count$

```
1: function REMSP-I(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:                     label(e) ← label(b)
7:                 else
8:                     if image(c) = 1 then
9:                         if image(a) = 1 then
10:                            label(e) ← merge(p, label(c), label(a))
11:                        else
12:                            if image(d) = 1 then
13:                                label(e) ← merge(p, label(c), label(d))
14:                            else
15:                                label(e) ← label(c)
16:                            end if
17:                        end if
18:                    else
19:                        if image(a) = 1 then
20:                            label(e) ← label(a)
21:                        else
22:                            if image(d) = 1 then
23:                                label(e) ← label(d)
24:                            else
25:                                label(e) ← count,
26:                                p[count] ← count,
27:                                count + +
28:                            end if
29:                        end if
30:                    end if
31:                end if
32:            end if
33:        end for
34:    end for
35:    return count
36: end function
```

Algorithm 8 Pseudo-code for ARemSP Phase-I

Input: $2D$ array $image$ containing the pixel values
InOut: $2D$ array $label$ containing the privisonal labels and $1D$ araay p containing the equivalance info
Output: maximum value of provisional label in $count$

```
1: function AREMSP-I(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:                        end if
11:                    else
12:                        if image(f) = 1 then
13:                            label(e) ← label(f)
14:                            if image(a) = 1 then
15:                                merge(p, label(a))
16:                            end if
17:                            if image(c) = 1 then
18:                                merge(p, label(e), label(c))
19:                            end if
20:                        else
21:                            if image(a) = 1 then
22:                                label(e) ← label(a)
23:                                if image(c) = 1 then
24:                                    merge(p, label(e), label(c))
25:                                end if
26:                            else
27:                                if image(c) = 1 then
28:                                    label(e) ← label(c)
29:                                else
30:                                    label(e) ← count,
31:                                    p[count] ← count,
32:                                    count + +
33:                                end if
34:                            end if
35:                        end if
36:                    end if
37:                else
38:                    label(e) = label(d)
39:                    if image(b) = 0 then
40:                        if image(c) = 1 then
41:                            merge(p, label(e), label(c))
42:                        end if
43:                    end if
44:                end if
45:                if image(g) = 1 then
46:                    label(g) ← label(e)
47:                end if
48:            else
49:                if image(g) = 1 then
50:                    if image(d) = 1 then
51:                        label(g) ← label(d)
52:                    else
53:                        if image(f) = 1 then
54:                            label(g) ← label(f)
55:                        else
56:                            label(e) ← count,
57:                            p[count] ← count,
58:                            count + +
59:                        end if
60:                    end if
61:                end if
62:            end if
63:        end for
64:    end for
65:    return count
66: end function
```

References

[1] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis and Applications*, 12(2):117–135, 2009.

[2] Lifeng He, Yuyan Chao, and Kenji Suzuki. A new two-scan algorithm for labeling connected components in binary images. In *Proceedings of the World Congress on Engineering*, volume 2, 2012.

[3] Md Mostofa Ali Patwary, Jean Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In *Experimental Algorithms*, pages 411–423. Springer, 2010.

[4] Hussein M Alnuweiri and Viktor K Prasanna. Parallel architectures and algorithms for image component labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(10):1014–1034, 1992.

[5] Rafael C Gonzales and RE Woods. Digital image processing, 1993.

[6] Pankaj K Agarwal, Lars Arge, and Ke Yi. I/o-efficient batched union-find and its applications to terrain analysis. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 167–176. ACM, 2006.

[7] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004.

[8] Hiroki Hayashi, Mineichi Kudo, Jun Toyama, and Masaru Shimbo. Fast labelling of natural scenes using enhanced knowledge. *Pattern Analysis & Applications*, 4(1):20–27, 2001.

[9] Qingmao Hu, Guoyu Qian, and Wieslaw L Nowinski. Fast connected-component labelling in three-dimensional binary images based on iterative recursion. *Computer Vision and Image Understanding*, 99(3):414–434, 2005.

[10] Felipe Knop and Vernon Rego. Parallel labeling of three-dimensional clusters on networks of workstations. *Journal of Parallel and Distributed Computing*, 49(2):182–203, 1998.

[11] Alina N Moga and Moncef Gabbouj. Parallel image component labelling with watershed transformation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(5):441–450, 1997.

[12] Kuang-Bor Wang, Tsorng-Lin Chia, Zen Chen, and Der-Chyuan Lou. Parallel execution of a connected component labeling operation on a linear array architecture. *J. Inf. Sci. Eng.*, 19(2):353–370, 2003.

[13] Kenji Suzuki, Isao Horiba, and Noboru Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1–23, 2003.

[14] Lifeng He, Yuyan Chao, and Kenji Suzuki. A run-based two-scan labeling algorithm. *Image Processing, IEEE Transactions on*, 17(5):749–756, 2008.

[15] Md Patwary, Mostofa Ali, Peder Refsnes, and Fredrik Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 827–835. IEEE, 2012.