

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 [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, 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. However 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. [11], and [12] are two developed techniques for two-pass CCL algorithms. The algorithm in [11], 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 [1]. The algorithm in [12], 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 a two-pass algorithm for labeling the connected components called AREMSP, which is based on REM union-find algorithm [13] and the scan strategy of ARUN algorithm. Since REM union-find is an interleaved algorithm which implements immediate parent check test and compression technique called *Splicing* [13], 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 [14]. 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 among threads and each thread runs the scan strategy of ARUN algorithm along with REMSP simultaneously. To merge the provisional labels on the image boundary, we use the parallel version of REMSP [14]. 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 its 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 ??.

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 [11], 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 [15] 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 [12]. This algorithm uses the same data structure given in [15] 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. [13] 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 [12] but for the second step we use REMSP instead of the data structure used by [12].

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 [14].

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 [11] 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. 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.

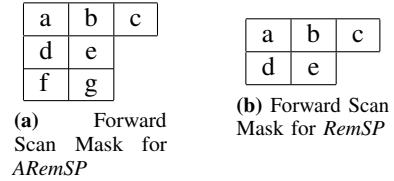


Fig. 1: Forward Scan Mask

However, the implementation of MERGE operation in our proposed algorithm REMSP is different from that of in LRPC. We have used the implementation of union-find proposed by Rem [13] 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 parent, thus compressing the tree. The algorithm

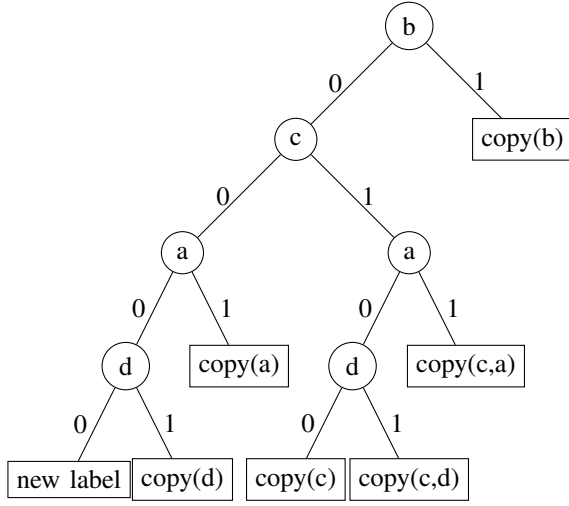


Fig. 2: Decision tree for RemSP

for MERGE is given as Algorithm 4. The full algorithm for REMSP is given as Algorithm 1.

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)                                ▷ Phase-I of RemSP
3:   flatten(p, count)
4:   for row in image do                                ▷ Phase-II 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 equivalence 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 [12]. 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 equivalence. 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 [12]. However, our implementation of the union-find is different from [12]. We use the implementation of union-find proposed by Rem [13] for merge operation in AREMSP. The full algorithm for AREMSP is given as Algorithm 3.

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)                                ▷ Phase-I of AREMSP
3:   flatten(p, count)
4:   for row in image do                                ▷ Phase-II of AREMSP
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 [14]. 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

```

1: function PAREMSP(image)
2:   numiter  $\leftarrow$  row/2 ▷ As we are processing 2 rows at a time
3:   # pragma omp parallel
4:   chunk  $\leftarrow$  numiter/numberofthreads
5:   size  $\leftarrow$  2  $\times$  chunk
6:   start  $\leftarrow$  start index of the thread
7:   count  $\leftarrow$  start  $\times$  col
8:   # pragma omp for
9:   ARemSP  $\leftarrow$  I(image)
10:  # pragma omp for
11:  for i = size to row - 1 do
12:    for col in row do
13:      if label(e)  $\neq$  0 then
14:        if label(b)  $\neq$  0 then
15:          merger(p, label(e), label(b))
16:        else
17:          if label(a)  $\neq$  0 then
18:            merger(p, label(e), label(a))
19:          end if
20:          if label(c)  $\neq$  0 then
21:            merger(p, label(e), label(c))
22:          end if
23:        end if
24:      end if
25:    end for
26:    i  $\leftarrow$  i + size
27:  end for
28:  flatten(p, count)
29:  for row in image do
30:    for col in row do
31:      label(e)  $\leftarrow$  p[label(e)]
32:    end for
33:  end for
34: 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

Algorithm 6 Pseudo-code for merger

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

```

1: function MERGER(p, x, y)
2:   rootx  $\leftarrow$  x, rooty  $\leftarrow$  y
3:   while p[rootx]  $\neq$  p[rooty] do
4:     if p[rootx] > p[rooty] then
5:       if rootx = p[rootx] then
6:         omp_set_lock(&(lock_array[rootx]))
7:         success  $\leftarrow$  0
8:         if rootx = p[rootx] then
9:           p[rootx]  $\leftarrow$  p[rooty]
10:          success  $\leftarrow$  1
11:        end if
12:        omp_unset_lock(&(lock_array[rootx]))
13:        if success = 1 then
14:          break
15:        end if
16:      end if
17:      z  $\leftarrow$  p[rootx], p[rootx]  $\leftarrow$  p[rooty], rootx  $\leftarrow$  z
18:    else
19:      if rooty = p[rooty] then
20:        omp_set_lock(&(lock_array[rooty]))
21:        success  $\leftarrow$  0
22:        if rooty = p[rooty] then
23:          p[rooty]  $\leftarrow$  p[rootx]
24:          success  $\leftarrow$  1
25:        end if
26:        omp_unset_lock(&(lock_array[rooty]))
27:        if success = 1 then
28:          break
29:        end if
30:      end if
31:      z  $\leftarrow$  p[rooty], p[rooty]  $\leftarrow$  p[rootx], rooty  $\leftarrow$  z
32:    end if
33:  end while
34:  return p[rootx]
35: end function

```

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 \times 1024 or less. NCLD dataset contains images of size bigger than 3000 \times 4000. The biggest image in the dataset is 22,822 \times 20,384.

Firstly, we did the experiment over all the sequential algorithms. The result is given in Table I. We can see that

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

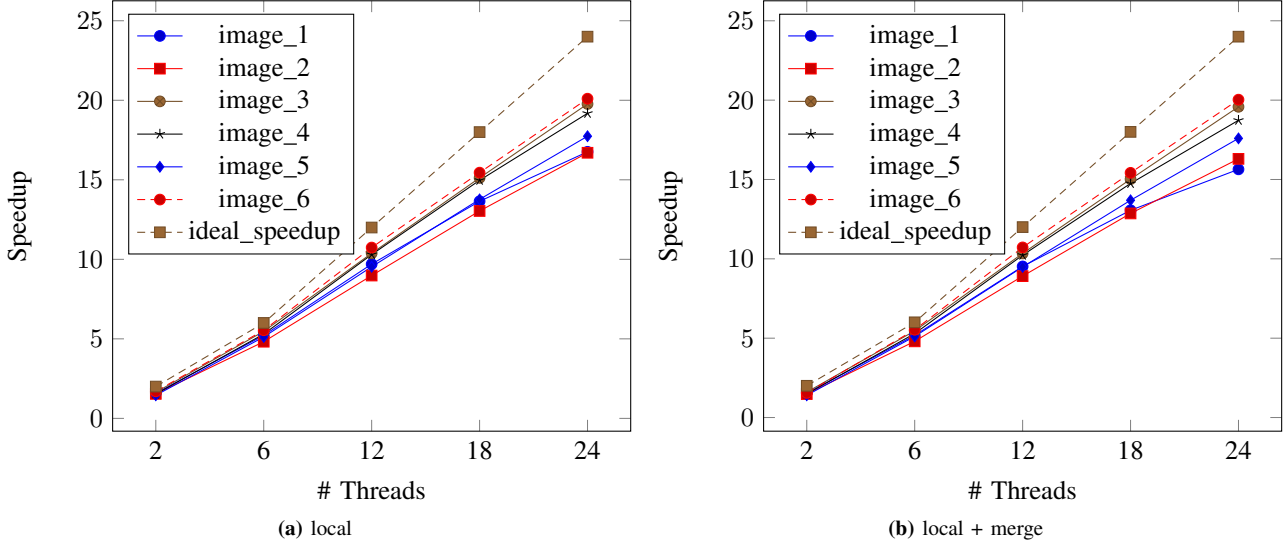


Fig. 3: Speedup for different images and different numbers of threads for NLCD dataset

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 NLCD image dataset. 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 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.

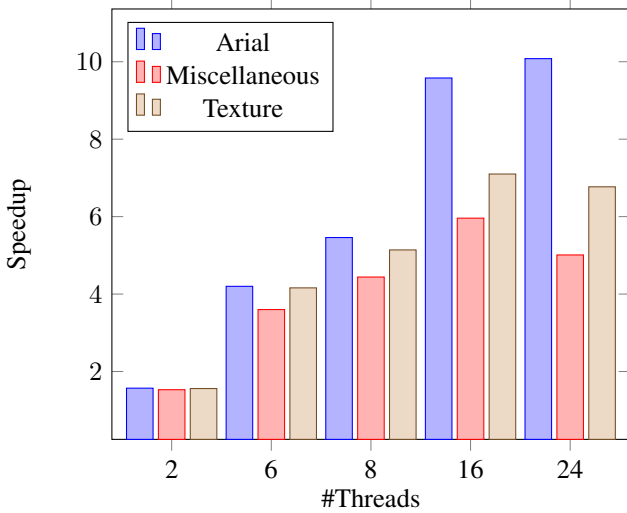


Fig. 4: Speedup for different images and different numbers of threads for Arial, Texture & Miscellaneous dataset

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 array *p* containing the equivalence 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)  $\leftarrow$  label(b)
7:         else
8:           if image(c) = 1 then
9:             if image(a) = 1 then
10:              label(e)  $\leftarrow$  label(a)
11:              merge(p, label(c), label(a))
12:            else
13:              if image(d) = 1 then
14:                label(e)  $\leftarrow$  label(d)
15:              else
16:                label(e)  $\leftarrow$  label(c)
17:              end if
18:            end if
19:          else
20:            if image(a) = 1 then
21:              label(e)  $\leftarrow$  label(a)
22:            else
23:              if image(d) = 1 then
24:                label(e)  $\leftarrow$  label(d)
25:              else
26:                label(e)  $\leftarrow$  count,
27:                p[count]  $\leftarrow$  count,
28:                count ++
29:              end if
30:            end if
31:          end if
32:        end if
33:      end if
34:    end for
35:  end for
36: return count
37: 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 array *p* containing the equivalence 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)  $\leftarrow$  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)  $\leftarrow$  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)  $\leftarrow$  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)  $\leftarrow$  label(c)
29:          else
30:            label(e)  $\leftarrow$  count,
31:            p[count]  $\leftarrow$  count,
32:            count ++
33:          end if
34:        end if
35:      end if
36:    end if
37:  end for
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:  if image(g) = 1 then
45:    label(g)  $\leftarrow$  label(e)
46:  end if
47:  if image(g) = 1 then
48:    if image(d) = 1 then
49:      label(g)  $\leftarrow$  label(d)
50:    else
51:      if image(f) = 1 then
52:        label(g)  $\leftarrow$  label(f)
53:      else
54:        label(e)  $\leftarrow$  count,
55:        p[count]  $\leftarrow$  count,
56:        count ++
57:      end if
58:    end if
59:  end if
60:  if image(g) = 1 then
61:    if image(d) = 1 then
62:      label(g)  $\leftarrow$  label(d)
63:    else
64:      if image(f) = 1 then
65:        label(g)  $\leftarrow$  label(f)
66:      else
67:        label(e)  $\leftarrow$  count,
68:        p[count]  $\leftarrow$  count,
69:        count ++
70:      end if
71:    end if
72:  end if
73:  if image(g) = 1 then
74:    if image(d) = 1 then
75:      label(g)  $\leftarrow$  label(d)
76:    else
77:      if image(f) = 1 then
78:        label(g)  $\leftarrow$  label(f)
79:      else
80:        label(e)  $\leftarrow$  count,
81:        p[count]  $\leftarrow$  count,
82:        count ++
83:      end if
84:    end if
85:  end if
86:  if image(g) = 1 then
87:    if image(d) = 1 then
88:      label(g)  $\leftarrow$  label(d)
89:    else
90:      if image(f) = 1 then
91:        label(g)  $\leftarrow$  label(f)
92:      else
93:        label(e)  $\leftarrow$  count,
94:        p[count]  $\leftarrow$  count,
95:        count ++
96:      end if
97:    end if
98:  end if
99:  if image(g) = 1 then
100:    if image(d) = 1 then
101:      label(g)  $\leftarrow$  label(d)
102:    else
103:      if image(f) = 1 then
104:        label(g)  $\leftarrow$  label(f)
105:      else
106:        label(e)  $\leftarrow$  count,
107:        p[count]  $\leftarrow$  count,
108:        count ++
109:      end if
110:    end if
111:  end if
112:  if image(g) = 1 then
113:    if image(d) = 1 then
114:      label(g)  $\leftarrow$  label(d)
115:    else
116:      if image(f) = 1 then
117:        label(g)  $\leftarrow$  label(f)
118:      else
119:        label(e)  $\leftarrow$  count,
120:        p[count]  $\leftarrow$  count,
121:        count ++
122:      end if
123:    end if
124:  end if
125:  if image(g) = 1 then
126:    if image(d) = 1 then
127:      label(g)  $\leftarrow$  label(d)
128:    else
129:      if image(f) = 1 then
130:        label(g)  $\leftarrow$  label(f)
131:      else
132:        label(e)  $\leftarrow$  count,
133:        p[count]  $\leftarrow$  count,
134:        count ++
135:      end if
136:    end if
137:  end if
138:  if image(g) = 1 then
139:    if image(d) = 1 then
140:      label(g)  $\leftarrow$  label(d)
141:    else
142:      if image(f) = 1 then
143:        label(g)  $\leftarrow$  label(f)
144:      else
145:        label(e)  $\leftarrow$  count,
146:        p[count]  $\leftarrow$  count,
147:        count ++
148:      end if
149:    end if
150:  end if
151:  if image(g) = 1 then
152:    if image(d) = 1 then
153:      label(g)  $\leftarrow$  label(d)
154:    else
155:      if image(f) = 1 then
156:        label(g)  $\leftarrow$  label(f)
157:      else
158:        label(e)  $\leftarrow$  count,
159:        p[count]  $\leftarrow$  count,
160:        count ++
161:      end if
162:    end if
163:  end if
164:  if image(g) = 1 then
165:    if image(d) = 1 then
166:      label(g)  $\leftarrow$  label(d)
167:    else
168:      if image(f) = 1 then
169:        label(g)  $\leftarrow$  label(f)
170:      else
171:        label(e)  $\leftarrow$  count,
172:        p[count]  $\leftarrow$  count,
173:        count ++
174:      end if
175:    end if
176:  end if
177:  if image(g) = 1 then
178:    if image(d) = 1 then
179:      label(g)  $\leftarrow$  label(d)
180:    else
181:      if image(f) = 1 then
182:        label(g)  $\leftarrow$  label(f)
183:      else
184:        label(e)  $\leftarrow$  count,
185:        p[count]  $\leftarrow$  count,
186:        count ++
187:      end if
188:    end if
189:  end if
190:  if image(g) = 1 then
191:    if image(d) = 1 then
192:      label(g)  $\leftarrow$  label(d)
193:    else
194:      if image(f) = 1 then
195:        label(g)  $\leftarrow$  label(f)
196:      else
197:        label(e)  $\leftarrow$  count,
198:        p[count]  $\leftarrow$  count,
199:        count ++
200:      end if
201:    end if
202:  end if
203:  if image(g) = 1 then
204:    if image(d) = 1 then
205:      label(g)  $\leftarrow$  label(d)
206:    else
207:      if image(f) = 1 then
208:        label(g)  $\leftarrow$  label(f)
209:      else
210:        label(e)  $\leftarrow$  count,
211:        p[count]  $\leftarrow$  count,
212:        count ++
213:      end if
214:    end if
215:  end if
216:  if image(g) = 1 then
217:    if image(d) = 1 then
218:      label(g)  $\leftarrow$  label(d)
219:    else
220:      if image(f) = 1 then
221:        label(g)  $\leftarrow$  label(f)
222:      else
223:        label(e)  $\leftarrow$  count,
224:        p[count]  $\leftarrow$  count,
225:        count ++
226:      end if
227:    end if
228:  end if
229:  if image(g) = 1 then
230:    if image(d) = 1 then
231:      label(g)  $\leftarrow$  label(d)
232:    else
233:      if image(f) = 1 then
234:        label(g)  $\leftarrow$  label(f)
235:      else
236:        label(e)  $\leftarrow$  count,
237:        p[count]  $\leftarrow$  count,
238:        count ++
239:      end if
240:    end if
241:  end if
242:  if image(g) = 1 then
243:    if image(d) = 1 then
244:      label(g)  $\leftarrow$  label(d)
245:    else
246:      if image(f) = 1 then
247:        label(g)  $\leftarrow$  label(f)
248:      else
249:        label(e)  $\leftarrow$  count,
250:        p[count]  $\leftarrow$  count,
251:        count ++
252:      end if
253:    end if
254:  end if
255:  if image(g) = 1 then
256:    if image(d) = 1 then
257:      label(g)  $\leftarrow$  label(d)
258:    else
259:      if image(f) = 1 then
260:        label(g)  $\leftarrow$  label(f)
261:      else
262:        label(e)  $\leftarrow$  count,
263:        p[count]  $\leftarrow$  count,
264:        count ++
265:      end if
266:    end if
267:  end if
268:  if image(g) = 1 then
269:    if image(d) = 1 then
270:      label(g)  $\leftarrow$  label(d)
271:    else
272:      if image(f) = 1 then
273:        label(g)  $\leftarrow$  label(f)
274:      else
275:        label(e)  $\leftarrow$  count,
276:        p[count]  $\leftarrow$  count,
277:        count ++
278:      end if
279:    end if
280:  end if
281:  if image(g) = 1 then
282:    if image(d) = 1 then
283:      label(g)  $\leftarrow$  label(d)
284:    else
285:      if image(f) = 1 then
286:        label(g)  $\leftarrow$  label(f)
287:      else
288:        label(e)  $\leftarrow$  count,
289:        p[count]  $\leftarrow$  count,
290:        count ++
291:      end if
292:    end if
293:  end if
294:  if image(g) = 1 then
295:    if image(d) = 1 then
296:      label(g)  $\leftarrow$  label(d)
297:    else
298:      if image(f) = 1 then
299:        label(g)  $\leftarrow$  label(f)
300:      else
301:        label(e)  $\leftarrow$  count,
302:        p[count]  $\leftarrow$  count,
303:        count ++
304:      end if
305:    end if
306:  end if
307:  if image(g) = 1 then
308:    if image(d) = 1 then
309:      label(g)  $\leftarrow$  label(d)
310:    else
311:      if image(f) = 1 then
312:        label(g)  $\leftarrow$  label(f)
313:      else
314:        label(e)  $\leftarrow$  count,
315:        p[count]  $\leftarrow$  count,
316:        count ++
317:      end if
318:    end if
319:  end if
320:  if image(g) = 1 then
321:    if image(d) = 1 then
322:      label(g)  $\leftarrow$  label(d)
323:    else
324:      if image(f) = 1 then
325:        label(g)  $\leftarrow$  label(f)
326:      else
327:        label(e)  $\leftarrow$  count,
328:        p[count]  $\leftarrow$  count,
329:        count ++
330:      end if
331:    end if
332:  end if
333:  if image(g) = 1 then
334:    if image(d) = 1 then
335:      label(g)  $\leftarrow$  label(d)
336:    else
337:      if image(f) = 1 then
338:        label(g)  $\leftarrow$  label(f)
339:      else
340:        label(e)  $\leftarrow$  count,
341:        p[count]  $\leftarrow$  count,
342:        count ++
343:      end if
344:    end if
345:  end if
346:  if image(g) = 1 then
347:    if image(d) = 1 then
348:      label(g)  $\leftarrow$  label(d)
349:    else
350:      if image(f) = 1 then
351:        label(g)  $\leftarrow$  label(f)
352:      else
353:        label(e)  $\leftarrow$  count,
354:        p[count]  $\leftarrow$  count,
355:        count ++
356:      end if
357:    end if
358:  end if
359:  if image(g) = 1 then
360:    if image(d) = 1 then
361:      label(g)  $\leftarrow$  label(d)
362:    else
363:      if image(f) = 1 then
364:        label(g)  $\leftarrow$  label(f)
365:      else
366:        label(e)  $\leftarrow$  count,
367:        p[count]  $\leftarrow$  count,
368:        count ++
369:      end if
370:    end if
371:  end if
372:  if image(g) = 1 then
373:    if image(d) = 1 then
374:      label(g)  $\leftarrow$  label(d)
375:    else
376:      if image(f) = 1 then
377:        label(g)  $\leftarrow$  label(f)
378:      else
379:        label(e)  $\leftarrow$  count,
380:        p[count]  $\leftarrow$  count,
381:        count ++
382:      end if
383:    end if
384:  end if
385:  if image(g) = 1 then
386:    if image(d) = 1 then
387:      label(g)  $\leftarrow$  label(d)
388:    else
389:      if image(f) = 1 then
390:        label(g)  $\leftarrow$  label(f)
391:      else
392:        label(e)  $\leftarrow$  count,
393:        p[count]  $\leftarrow$  count,
394:        count ++
395:      end if
396:    end if
397:  end if
398:  if image(g) = 1 then
399:    if image(d) = 1 then
400:      label(g)  $\leftarrow$  label(d)
401:    else
402:      if image(f) = 1 then
403:        label(g)  $\leftarrow$  label(f)
404:      else
405:        label(e)  $\leftarrow$  count,
406:        p[count]  $\leftarrow$  count,
407:        count ++
408:      end if
409:    end if
410:  end if
411:  if image(g) = 1 then
412:    if image(d) = 1 then
413:      label(g)  $\leftarrow$  label(d)
414:    else
415:      if image(f) = 1 then
416:        label(g)  $\leftarrow$  label(f)
417:      else
418:        label(e)  $\leftarrow$  count,
419:        p[count]  $\leftarrow$  count,
420:        count ++
421:      end if
422:    end if
423:  end if
424:  if image(g) = 1 then
425:    if image(d) = 1 then
426:      label(g)  $\leftarrow$  label(d)
427:    else
428:      if image(f) = 1 then
429:        label(g)  $\leftarrow$  label(f)
430:      else
431:        label(e)  $\leftarrow$  count,
432:        p[count]  $\leftarrow$  count,
433:        count ++
434:      end if
435:    end if
436:  end if
437:  if image(g) = 1 then
438:    if image(d) = 1 then
439:      label(g)  $\leftarrow$  label(d)
440:    else
441:      if image(f) = 1 then
442:        label(g)  $\leftarrow$  label(f)
443:      else
444:        label(e)  $\leftarrow$  count,
445:        p[count]  $\leftarrow$  count,
446:        count ++
447:      end if
448:    end if
449:  end if
450:  if image(g) = 1 then
451:    if image(d) = 1 then
452:      label(g)  $\leftarrow$  label(d)
453:    else
454:      if image(f) = 1 then
455:        label(g)  $\leftarrow$  label(f)
456:      else
457:        label(e)  $\leftarrow$  count,
458:        p[count]  $\leftarrow$  count,
459:        count ++
460:      end if
461:    end if
462:  end if
463:  if image(g) = 1 then
464:    if image(d) = 1 then
465:      label(g)  $\leftarrow$  label(d)
466:    else
467:      if image(f) = 1 then
468:        label(g)  $\leftarrow$  label(f)
469:      else
470:        label(e)  $\leftarrow$  count,
471:        p[count]  $\leftarrow$  count,
472:        count ++
473:      end if
474:    end if
475:  end if
476:  if image(g) = 1 then
477:    if image(d) = 1 then
478:      label(g)  $\leftarrow$  label(d)
479:    else
480:      if image(f) = 1 then
481:        label(g)  $\leftarrow$  label(f)
482:      else
483:        label(e)  $\leftarrow$  count,
484:        p[count]  $\leftarrow$  count,
485:        count ++
486:      end if
487:    end if
488:  end if
489:  if image(g) = 1 then
490:    if image(d) = 1 then
491:      label(g)  $\leftarrow$  label(d)
492:    else
493:      if image(f) = 1 then
494:        label(g)  $\leftarrow$  label(f)
495:      else
496:        label(e)  $\leftarrow$  count,
497:        p[count]  $\leftarrow$  count,
498:        count ++
499:      end if
500:    end if
501:  end if
502:  if image(g) = 1 then
503:    if image(d) = 1 then
504:      label(g)  $\leftarrow$  label(d)
505:    else
506:      if image(f) = 1 then
507:        label(g)  $\leftarrow$  label(f)
508:      else
509:        label(e)  $\leftarrow$  count,
510:        p[count]  $\leftarrow$  count,
511:        count ++
512:      end if
513:    end if
514:  end if
515:  if image(g) = 1 then
516:    if image(d) = 1 then
517:      label(g)  $\leftarrow$  label(d)
518:    else
519:      if image(f) = 1 then
520:        label(g)  $\leftarrow$  label(f)
521:      else
522:        label(e)  $\leftarrow$  count,
523:        p[count]  $\leftarrow$  count,
524:        count ++
525:      end if
526:    end if
527:  end if
528:  if image(g) = 1 then
529:    if image(d) = 1 then
530:      label(g)  $\leftarrow$  label(d)
531:    else
532:      if image(f) = 1 then
533:        label(g)  $\leftarrow$  label(f)
534:      else
535:        label(e)  $\leftarrow$  count,
536:        p[count]  $\leftarrow$  count,
537:        count ++
538
```

References

- [1] 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.
- [2] Rafael C Gonzales and RE Woods. Digital image processing, 1993.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis and Applications*, 12(2):117–135, 2009.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] Lifeng He, Yuyan Chao, and Kenji Suzuki. A run-based two-scan labeling algorithm. *Image Processing, IEEE Transactions on*, 17(5):749–756, 2008.