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.gupta@northwestern.edu, {drp925, mpatwary, ankitag, choudhar}@eecs.northwestern.edu

***Abstract*—Connected Component Labeling (CCL) is one of the most 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. For example, in a two-pass technique, each pixel is given a provisional label in the first pass whereas an actual label is assigned in the second pass. Suzuki et al have proposed two algorithms for CCL with two-pass technique, which we refere to as** CCLLRPC **and** ARUN **[1], [2]. The** CCLLRPC **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. As the resolution of images becomes higher due to better digital camera technology and dynamic image matching has been common for surveillance, the connected component problem becomes a data-intensive application and hence the need for exploiting parallelism arises. The** ARUN **algorithm has been parallelized for Tile64 many-core platform, however the achieved speedup of 10 on 32 cores is sub-linear.**

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**, 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 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 using *OpenMP* for an image of size** $22,822 \times 20,384$**. We find that our proposed parallel algorithm achieves linear scaling for a large resolution fixed problem size while 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 [3]. Therefore, connected component labeling continues to be an active area of research [4]–[11].

There exist many algorithms for computing connected components in a given image. These algorithms are categorized into mainly four groups [12] : 1) repeated pass algorithms [13], [14], 2) two-pass algorithms [15]–[23] 3) Algorithms with hierarchical tree equivalent representations of the data [24]–[31], 4) parallel algorithms [32]–[37]. 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 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. [1], and [2] are two developed techniques for two-pass CCL algorithms. The algorithm in [1], which we refer to as CCLLRPC, uses a decision tree to assign provisional labels and an array-based union-find datastructure 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 [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. 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. As the parallel implementation is hardware specific and parallel efficiency is less than 33% , thus this unconventional implementation is not suited for parallel implementation.

TABLE I: Abbreviations used in the paper and their brief description

Abbreviation	Description
CCL	Connected Component Labeling
ARUN	CCL algorithm suggested by [2]
REMSP	union-find technique proposed by *Rem* [40]
AREMSP	CCL algorithm proposed in our paper using scan strategy of ARUN and REMSP
PAREMSP	Parallel implementation of AREMSP proposed in our paper
CCLLRPC	CCL algorithm suggested by [1]
CCLREMSP	CCL algorithm proposed in our paper using scan strategy of CCLLRPC and REMSP

We propose two two-pass algorithms for labeling the connected components that we call as 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 interleaved 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 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 two-pass CCL algorithm AREMSP which we call as 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 [38]. 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 algorithms CCLREMSP and AREMSP and parallel version of AREMSP 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.

II. Related Work

As mentioned in [12], 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 [13], [14]. The algorithm in [12], which we call as *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 [1], which we refer to as CCLLRPC, provides two strategies to improve the running time of the algorithm. First strategy reduces the average number of neighbors accessed by factor of two by employing a decision tree. 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 $rtable$ 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 two image pixels at a time [2]. 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 CCLLRPC 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. [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 [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 CCLLRPC, 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].

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 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 the 8-connectedness of the pixel.

A. CCLREMSP Algorithm

In CCLREMSP, we have used the decision tree suggested in CCLLRPC for scanning and REM'S union-find algorithm REMSP for storing label equivalance. 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

function CCLREMSP($image$)
  $Scan\_CCLRemSP(image)$ ▷ Scan Phase of CCLREMSP
  $flatten(p, count)$ ▷ Analysis Phase of CCLREMSP
  for row in $image$ **do** ▷ Labeling Phase of CCLREMSP
    for col in row **do**
      $label(e) \leftarrow p[label(e)]$
end function

In the first 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 [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 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.

a	b	c
d	e	

(a) Forward Scan Mask for *RemSP*

a	b	c
d	e	
f	g	

(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 proporsed algorithm REMSP is different from that of in CCLLRPC. We have used the implementation of union-find proposed by REM'S [40], [41] for merge operation. REM'S integrates the *union* operation with a compression technique known as Splicing (SP). In the MERGE algorithm when $root_x$ is to be moved to $p(root_x)$, 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 FLATTEN algorithm. In FLATTEN algorithm, we are giving smallest equivalent label of every connected component to all the pixels which belongs to that connected component. The algorithm also generates consecutive labels. The algorithm for FLATTEN is given as Algorithm 3.

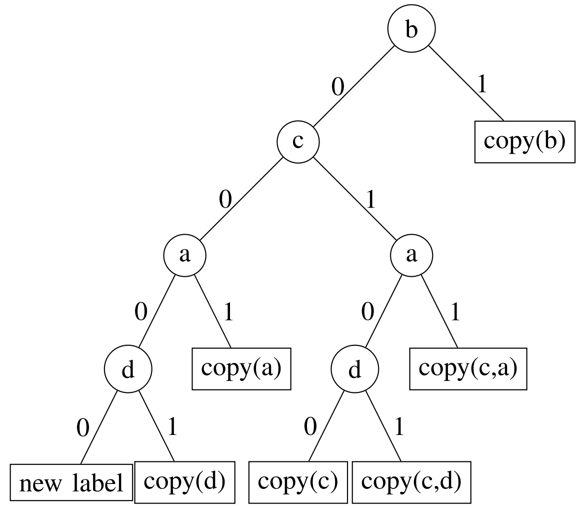


Fig. 2: Decision tree for *RemSP*

Algorithm 2 Pseudo-code for merge [40]

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

function MERGE(p,x,y)
  $root_x \leftarrow x, root_y \leftarrow y$
  while $p[root_x] \neq p[root_y]$ **do**
    if $p[root_x] > p[root_y]$ **then**
      if $root_x = p[root_x]$ **then**
        $p[root_x] \leftarrow p[root_y]$
        return $p[root_x]$
      $z \leftarrow p[root_x], p[root_x] \leftarrow p[root_y], root_x \leftarrow z$
    else
      if $root_y = p[root_y]$ **then**
        $p[root_y] \leftarrow p[root_x]$
        return $p[root_x]$
      $z \leftarrow p[root_y], p[root_y] \leftarrow p[root_x], root_y \leftarrow z$
  return $p[root_x]$
end function

Algorithm 3 Pseudo-code for flatten [1]

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

function FLATTEN(p,$count$)
  $k \leftarrow 1$
  for $i = 1$ to $count$ **do**
    if $p[i] < i$ **then**
      $p[i] \leftarrow p[p[i]]$
    else
      $p[i] \leftarrow k$
      $k++$
end function

Algorithm 4 Pseudo-code for CCLREMSP Scan Phase

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 $count$

```
function SCAN_CCLREMSP(image)
    for row in image do
        for col in row do
            if image(e) = 1 then
                if image(b) = 1 then
                    copy(b)
                else
                    if image(c) = 1 then
                        if image(a) = 1 then
                            copy(c, a)
                        else
                            if image(d) = 1 then
                                copy(c, d)
                            else
                                copy(c)
                    else
                        if image(a) = 1 then
                            copy(a)
                        else
                            if image(d) = 1 then
                                copy(d)
                            else
                                new label
    return count
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 equivalance. The full algorithm for AREMSP is given as Algorithm 5.

Algorithm 5 Pseudo-code for ARemSP

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

```
function AREMSP(image)
    Scan_ARemSP(image)              ▷ Scan Phase of RemSP
    flatten(p, count)               ▷ Analysis Phase of RemSP
    for row in image do             ▷ Labeling Phase of RemSP
        for col in row do
            label(e) ← p[label(e)]
end function
```

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 [2]. We asign 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 equivalance. Otherwise, if there are more than one connected component in the mask and as they are connected to e, all the labels of the connected components are equivalent labels hence need to be merged. For all the cases, one can refer [2]. However, our implementation of merge is different from [2]. We use the implementation of union-find proposed by *Rem* [40], [41] for merge operation in AREMSP. Similar to CCLREMSP, we use FLATTEN for analysis phase and generating consequtive labels. The implementation of *Scan_ARemSP* is given as Algorithm 8.

IV. Parallelizing AREMSP Algorithm

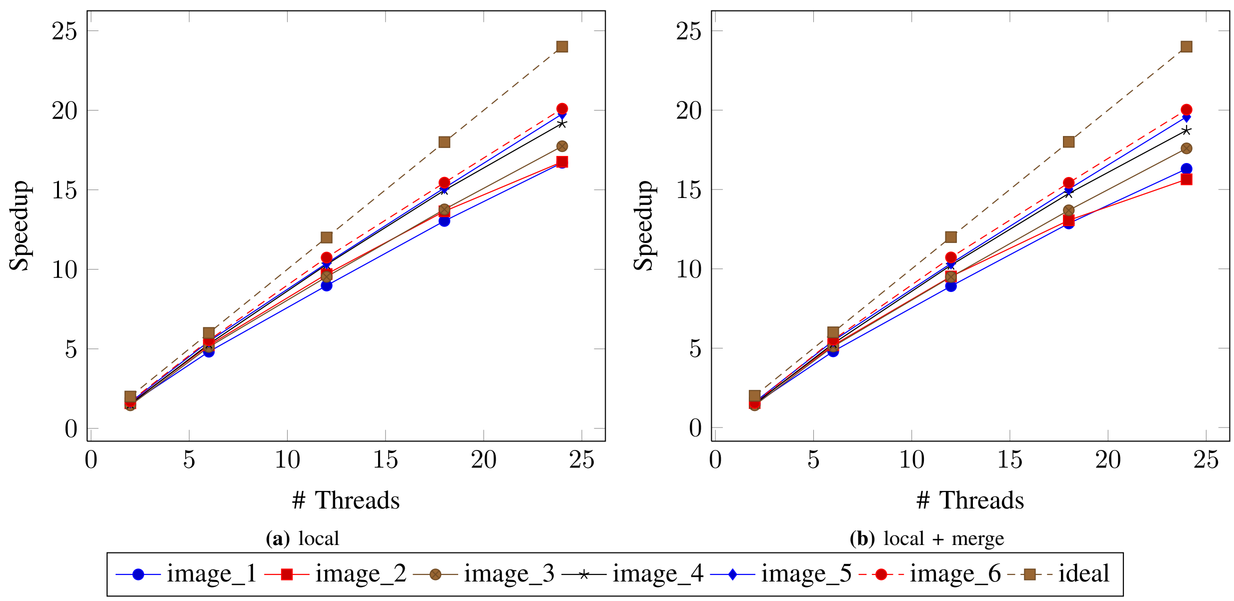
We now describe the parallel implementation of AREMSP algorithm on a shared memory system. We make the assumption about memory model as stated in *OpenMP* regarding the atomic directive. We assume that memory read/write operations are atomic and any operations issued concurrently by different processors 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. 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 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. For complete reference, one can refer [38]. We implement the parallel algorithm using OpenMP directives *pragma omp parallel* and *pragma omp for*. The pseudo code of MERGER is given as Algorithm 7. The pseudo code of PAREMSP is given as Algorithm 6.

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 dataset consists of four types of image dataset: Texture, Arial, Miscellaneous and NLCD. First three datasets are taken from the image database of the University of Southern California.[1] The fourth dataset is taken from US National Cover Database 2006.[2] All of the images are converted to binary images by means of MATLAB. However, note that

[1] http://sipi.usc.edu/database/
[2] http://dx.doi.org/10.1016/j.cageo.2013.05.014



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

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

Algorithm 6 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
  $Scan\_ARemSP(image)$
  # 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**
            $merger(p, label(e), label(a))$
          if $label(c) \neq 0$ **then**
            $merger(p, label(e), label(c))$
    $i \leftarrow i + size$
  $flatten(p, count)$
  for row in $image$ **do**
    for col in row **do**
      $label(e) \leftarrow p[label(e)]$
end function

Algorithm 7 Pseudo-code for merger [38]

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

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

our algorithm can be easily extended to gray scale images. 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 performed the experiment over all the sequential algorithms. The experimental results are shown in Table IV. In the table, we have shown the minimum, maximum and average execution time of all the four 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. Next, we show our results for the parallel algorithm PAREMSP over all the images. Figure 3a-3b shows the speedup of the algorithm for NCLD image dataset. The size of the images are given in Table III. We get

Algorithm 8 Pseudo-code for ARemSP Scan Phase

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 $count$

```
function SCAN_AREMSP(image)
    for row in image do
        for col in row do
            if image(e) = 1 then
                if image(d) = 0 then
                    if image(b) = 1 then
                        label(e) ← label(b)
                        if image(f) = 1 then
                            merge(p, label(e), label(f))
                    else
                        if image(f) = 1 then
                            label(e) ← label(f)
                            if image(a) = 1 then
                                merge(p, label(a))
                            if image(c) = 1 then
                                merge(p, label(e), label(c))
                        else
                            if image(a) = 1 then
                                label(e) ← label(a)
                                if image(c) = 1 then
                                    merge(p, label(e), label(c))
                            else
                                if image(c) = 1 then
                                    label(e) ← label(c)
                                else
                                    label(e) ← count,
                                    p[count] ← count,
                                    count + +
                else
                    label(e) = label(d)
                    if image(b) = 0 then
                        if image(c) = 1 then
                            merge(p, label(e), label(c))
                if image(g) = 1 then
                    label(g) ← label(e)
            else
                if image(g) = 1 then
                    if image(d) = 1 then
                        label(g) ← label(d)
                    else
                        if image(f) = 1 then
                            label(g) ← label(f)
                        else
                            label(e) ← count,
                            p[count] ← count,
                            count + +
    return count
end function
```

a maximum speedup of 20.1 on 24 cores for image of size $22,822 \times 20,384$.

Figure 3a shows the speedup for *Phase-I* of PAREMSP i.e. the local computation and Figure 3b shows the overall speedup (i.e. local + merge). We can see that there is not significant difference between both speedups, implying that merge operation does not have a significant overhead. Also as can be seen from the graph, as the image size increases, speedup also increases. Therefore, our parallel implementation is able to achieve near linear speed for large datasets. We have also shown the speedup for all the other datasets in Figure 4. We get a maximum seedup of 10 in this case as the images are $1,024 \times 1,024$ pixels or less 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.

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

Image type		CCLLRPC	CCLRemSP	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	1307.27	1273.82	1036.52	1021.45

TABLE III: Images and their sizes [in *pixels*]

Image name	Size
image_1	3232×2886
image_2	6079×5430
image_3	6464×5772
image_4	11411×10192
image_5	12158×10860
image_6	22822×20384

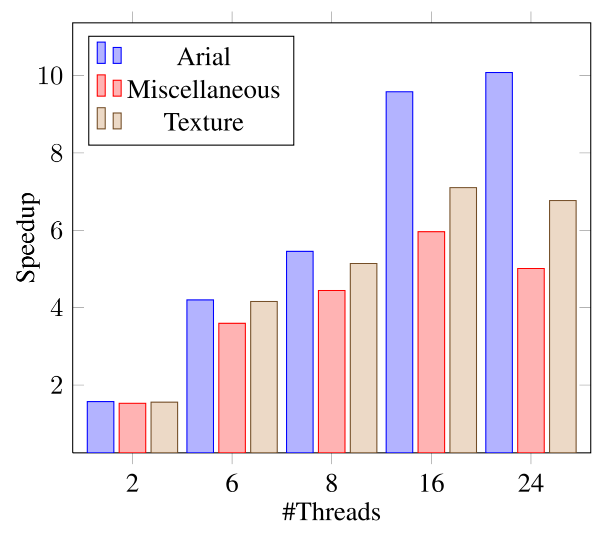


Fig. 4: Speedup for different images and different numbers of threads for Arial, Texture & Miscellaneous dataset

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 CCLLRPC algorithms. CCLREMSP algorithm uses the scan strategy of CCLLRPC algorithm whereas AREMSP uses the scan strategy of ARUN algorithm. Based on the

TABLE IV: Execution time [*msec*] of PAREMSP algorithm for various # threads

Image type		2	6	16	24
Arial	Min	1.39	0.84	1.02	1.38
	Average	7.92	3.03	1.87	2.15
	Max	46.86	16.72	7.32	6.97
Texture	Min	1.09	0.62	0.93	1.36
	Average	4.91	1.99	1.45	1.82
	Max	9.75	3.56	2.11	2.34
Miscellaneous	Min	0.36	0.36	0.79	1.18
	Average	1.99	0.97	1.05	1.46
	Max	7.96	3.24	1.91	2.27
NLCD	Min	2.52	1.16	1.32	1.67
	Average	162.86	58.50	20.20	13.47
	Max	676.41	184.71	78.33	51.00

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 pixel size $22,822 \times 20,384$. Thus, our parallel algorithm achieves linear scaling for large fixed problem size while the number of processing elements are increased.

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] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] RM Haralick. Some neighborhood operators. In *Real-Time Parallel Computing*, pages 11–35. Springer, 1981.

[14] A Hashizume, R Suzuki, H Yokouchi, H Horiuchi, and S Yamamato. An algorithm of automated rbc classification and its evaluation. *Bio Medical Engineering*, 28(1):25–32, 1990.

[15] Toshiyuki Gotoh, Yoshiyuki Ohta, Masumi Yoshida, and Yoshiaki Shirai. High-speed algorithm for component labeling. *Systems and Computers in Japan*, 21(5):74–84, 1990.

[16] Toshiyuki Gotoh, Yoshiyuki Ohta, Masumi Yoshida, and Yoshio Shirai. Component labeling algorithm for video rate processing. In *Hague International Symposium*, pages 217–224. International Society for Optics and Photonics, 1987.

[17] Masatoshi Komeichi, Yoshiyuki Ohta, Toshiyuki Gotoh, Toshiya Mima, and Masumi Yoshida. Video-rate labeling processor. In *1988 Intl Congress on Optical Science and Engineering*, pages 69–76. International Society for Optics and Photonics, 1989.

[18] Ronald Lumia. A new three-dimensional connected components algorithm. *Computer Vision, Graphics, and Image Processing*, 23(2):207–217, 1983.

[19] Ronald Lumia, Linda Shapiro, and Oscar Zuniga. A new connected components algorithm for virtual memory computers. *Computer Vision, Graphics, and Image Processing*, 22(2):287–300, 1983.

[20] Satoshi Naoi. High-speed labeling method using adaptive variable window size for character shape feature. In *IEEE Asian Conference on computer vision*, volume 1, pages 408–411, 1995.

[21] Azriel Rosenfeld. Connectivity in digital pictures. *Journal of the ACM (JACM)*, 17(1):146–160, 1970.

[22] Azriel Rosenfeld and John L Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966.

[23] Y Shirai. Labeling connected regions. *Three-Dimensional Computer Vision*, pages 86–89, 1987.

[24] Michael B Dillencourt, Hannan Samet, and Markku Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM (JACM)*, 39(2):253–280, 1992.

[25] Irene Gargantini and Zale Tabakman. Separation of connected component using linear quad-and oct-trees. In *Proc. 12th Conf. Numerical Mathematics and Computation*, volume 37, pages 257–276, 1982.

[26] Jean Hecquard and Raj Acharya. Connected component labeling with linear octree. *Pattern recognition*, 24(6):515–531, 1991.

[27] Hanan Samet. Connected component labeling using quadtrees. *Journal of the ACM (JACM)*, 28(3):487–501, 1981.

[28] Hanan Samet and Markku Tamminen. Computing geometric properties of images represented by linear quadtrees. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (2):229–240, 1985.

[29] Hanan Samet and Markku Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 10(4):579–586, 1988.

[30] Hanan Samet and Markku Tamminen. An improved approach to connected component labeling of images. In *International Conference on Computer Vision And Pattern Recognition*, pages 312–318, 1986.

[31] Markku Tamminen and Hanan Samet. Efficient octree conversion by connectivity labeling. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 43–51. ACM, 1984.

[32] Prabir Bhattacharya. Connected component labeling for binary images on a reconfigurable mesh architecture. *Journal of Systems Architecture*, 42(4):309–313, 1996.

[33] Alok Choudhary and Rajeev Thakur. Connected component labeling on coarse grain parallel computers: an experimental study. *Journal of Parallel and Distributed Computing*, 20(1):78–83, 1994.

[34] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[35] M Manohar and HK Ramapriyan. Connected component labeling of binary images on a mesh connected massively parallel processor. *Computer vision, graphics, and image processing*, 45(2):133–149, 1989.

[36] David Nassimi and Sartaj Sahni. Finding connected components and connected ones on a mesh-connected parallel computer. *SIAM Journal on computing*, 9(4):744–757, 1980.

[37] Stephan Olariu, James L Schwing, and Jingyuan Zhang. Fast component labelling and convex hull computation on reconfigurable meshes. *Image and vision computing*, 11(7):447–455, 1993.

[38] 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.

[39] Chien-Wei Chen, Yi-Ta Wu, Shau-Yin Tseng, and Wen-Shan Wang. Parallelization of connected-component labeling on tile64 many-core platform. *Journal of Signal Processing Systems*, pages 1–15, 2013.

[40] 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.

[41] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[42] Mehdi Niknam, Parimala Thulasiraman, and Sergio Camorlinga. A parallel algorithm for connected component labelling of gray-scale images on homogeneous multicore architectures. In *Journal of Physics: Conference Series*, volume 256, page 012010. IOP Publishing, 2010.

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