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 ???. 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. Link by Rank and Path Compression(LRPC) [], and ARUN [] are two developed techniques for two-pass CCL algorithms. 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. We propose a two-pass algorithm for labeling the connected components called AREMSP, which is based on REM union-find algorithm [] and the scan strategy of ARUN algorithm. Since REM union-find uses compression technique called *Splicing* [], our proposed sequential two-pass algorithm AREMSP is xxx times faster than LRPC and xxx times 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 [**?**]. 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 in the first pass, divides the image among threads and each thread runs the scan strategy of ARUNalgorithm along with REMSPsimultaneously. To merge the provisional labels on the image boundary, we use the parallel version of REMSP []. Our experiments show the scalability of PAREMSP achieving speedups up to 19.5 using 24 cores on shared memory architecture for an image of size $11,411 \times 10,192$. 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 **??**, we provided related work on connected component labeling. In section **??**, we propose our sequential two-pass CCL algorithm ARᴇᴍSPand it's parallel version in section **??**. We present our experimental methodology and results in section **??**. We conclude our work and propose future work in section **??**.

II. Related Work

In any 2-pass algorithm,there are two parts in scanning step: 1) examining neighbors of current pixel which already got label to determine label for the current pixel. 2) storing label equivalence information to speed up the algorithm. In [11],they have given two strategies to speedup the algorithm. First strategy reduces the average number of neighbors accessed by factor of 2 through the use of decision tree. Second strategy replaces the conventional pointer based union-find algorithm used for storing label equivalence by array based union-find algorithm. They proves that array based approach will take less memory than the pointer based approach so that will speedup the algorithm. They implemented union-find algorithm using Link by Rank and Path Compression. We will call this algorithm as LRPC for our reference.

In [12], they replaced the union-find data structure by a different data structure to process label equivalence information.In their algorithm, at any point, all provisional labels that are assigned to a connected component found so 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.They used $rtable[]$ 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.We have implemented an algorithm with decision tree along with the above data structure. We will call it as Run for our reference.

In [13], they replaced the first part of scanning step by a new algorithm which process image lines two by two and process image pixels two by two. They used the same data structure given in [12] for processing label equivalence information.They also proved that as the number of lines to be scanned will be halved so it will speedup the algorithm. We will call this algorithm as Arun for our reference.

In [14], they compared all the different variations of union-find algorithms over different graph data sets and found that RemSP, the implementation given by Rem is the best alomg all the variations of union-find algorithm.

In this paper, we have given two different variations for CCL algorithm. In the first variation, 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 RemSP instead of Link by Rank and Path Compression. We will call this RemSP for our reference.

In the second one, we processed the image lines two by two as suggested by [13] but for the second part, we used RemSP instead of data structure used by [13]. We will call this ARemSP for our reference.Then we have compared both of our veriations with LRPC, Run and Arun algorithms and found out that ARemSP performs best among all the algorithms.

Finally we have also given a parallel implementation of ARemSP using OpenMP.We used the parallel implementation of RemSP given in [15] Then we have compared it with ARemSP.

III. Proposed Algorithm

For an $M \times N$ image, we denote $image(a)$ to denote the pixel value of pixel a. As we are working with binary images, we suppose the value of foregroud pixel is 1 and that of background pixel is 0. All pixels in the edge of an image are considered to be background pixels. We have proposed two 2-pass sequential algorithms $RemSP$ and $ARemSP$ in this paper. These two algorithms are different in their first scan.

A. *RemSP Algorithm*

In the first scan of RemSP, we process image lines one by one using the forward scan mask as shown in figure 1. 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 e, i.e., a,b,c and d, they proposed to examine the neighbors according to a desicion tree as shown in figure 3. 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.

However, the implementation of $merge$ operation in our proprosed algorithm $RemSP$ is different from that of in $LRPC$. We have used the implementation of union-find proposed by Rem [14] 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 for $merge$ is given as Algorithm 6. The full algorithm for $RemSP$ is given as Algorithm 2.

B. *ARemSP Algorithm*

In the first scan of $ARemSP$, we process image lines two by two and processes pixels two by two using the mask shown in figure 2 suggested in [13]. We will give the label to both e and g simultaneously. If both e and g are background pixels, then 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 [13]. In [13], they have used three arrays for merge operation. However, we have used the implementation of union-find proposed by Rem [14] for merge operation in $ARemSP$. The full algorithm for $ARemSP$ is given as Algorithm 3.

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 8. The pseudo code of *PARemSP* is given as Algorithm 1.

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 22822×20384.

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

Fig. 1: Forward Scan Mask for *ARemSP*

Fig. 2: Forward Scan Mask for *RemSP*

tested the parallel algorithm *PARemSP* over all the images. Fig **??** shows the speedup of the algorithm for NCLD image dataset. We get a maximum speedup of 20.1 for image of size 22822×20384. Fig 4 shows the speedup for *Phase-I of PARemSP* i.e. the local computation and fig 5 shows the overall speedup. We can see that there is not significant difference between both the speedups which shows that *merge* operation doesn't take much time. We have also shown the speedup for all the other datasets in fig 6. We get a maximum seedup of 10 in this case as the images are small in size.

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.
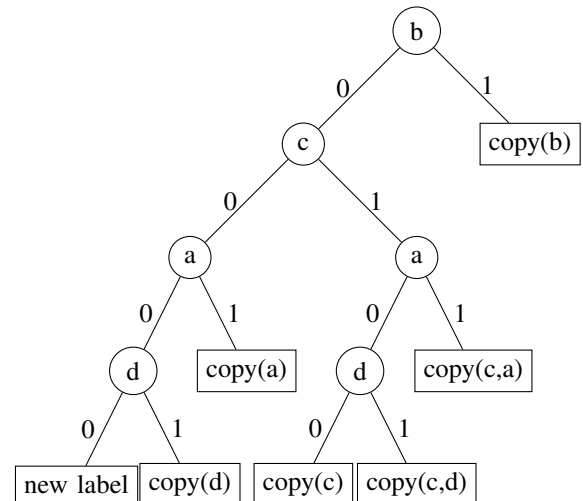
Fig. 3: Decision tree for *RemSP*

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

Image type		LRPC	RemSP	ARun	ARemSP
Arial	Min	2527.25	2481.50	1980.50	1951.00
	Average	13686.50	13258.34	11901.51	11861.39
	Max	86641.25	80903.25	72921.75	70176.50
Texture	Min	2075.00	2065.50	1586.75	1531.50
	Average	8425.19	8209.44	7323.11	7278.27
	Max	16868.25	16180.00	14816.25	14470.50
Miscellaneous	Min	504.75	497.00	368.50	362.75
	Average	3287.25	3215.57	2745.19	2742.44
	Max	12960.25	12812.25	11302.25	11200.00
NLCD	Min	4617.00	4460.50	3762.50	3756.75
	Average	307667.75	299554.33	244888.42	242597.46
	Max	1307274.50	1273826.50	1036528.50	1021456.75

Algorithm 1 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:    $ARemSP - I(image)$
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**

Algorithm 2 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:       $label(e) \leftarrow p[label(e)]$
7:     **end for**
8:   **end for**
9: **end function**

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:       $label(e) \leftarrow p[label(e)]$
7:     **end for**
8:   **end for**
9: **end function**

algorithm. *Image Processing, IEEE Transactions on*, 17(5):749–756, 2008.

[13] 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.

[14] 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.

[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.

[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 run-based two-scan labeling

---

Algorithm 4 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
```
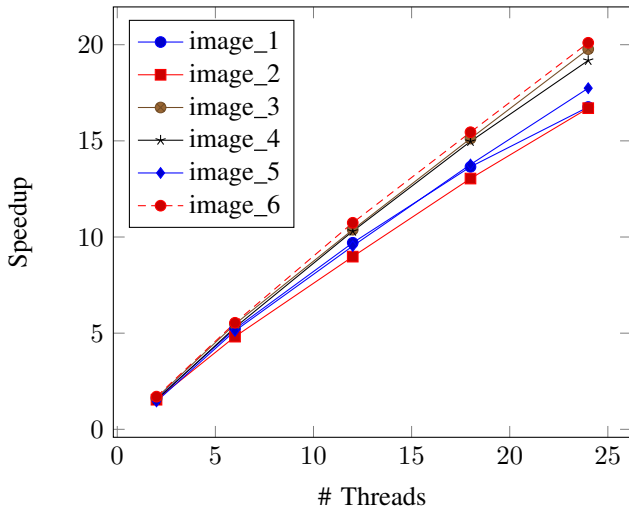

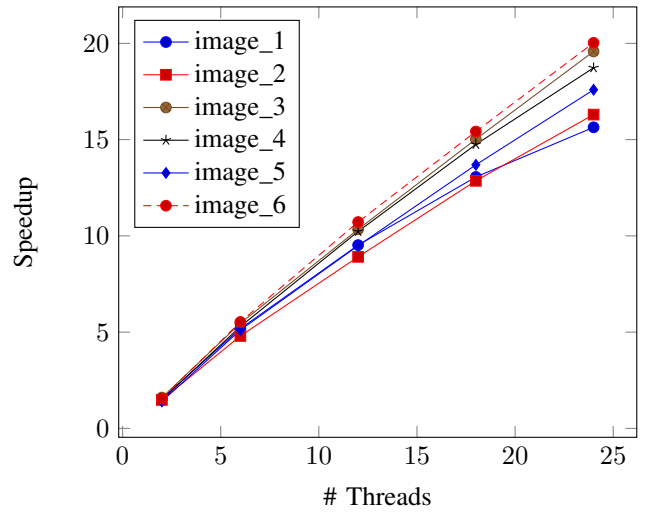Fig. 4: **Speedup for different images and different numbers of threads for local operation**

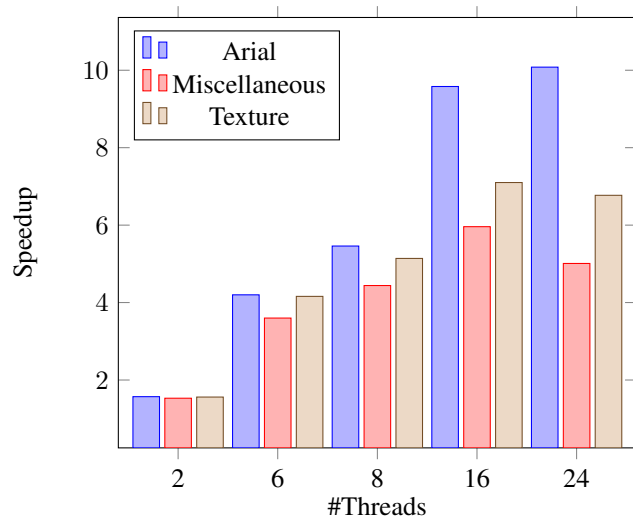Fig. 5: **Speedup for different images and different numbers of threads for local + merge**

Fig. 6: Speedup for different images and different numbers of threads

---

Algorithm 5 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
```

Algorithm 6 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
```

Algorithm 8 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 ← x, rooty ← y
 3:     while  p[rootx] ≠ 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 ← 0
 8:                 if rootx = p[rootx] then
 9:                     p[rootx] ← p[rooty]
10:                     success ← 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 ← p[rootx], p[rootx] ← p[rooty], rootx ← z
18:         else
19:             if rooty = p[rooty] then
20:                 omp_set_lock(&(lock_array[rooty]))
21:                 success ← 0
22:                 if root = p[rooty] then
23:                     p[rooty] ← p[rootx]
24:                     success ← 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 ← p[rooty], p[rooty] ← p[rootx], rooty ← z
32:         end if
33:     end while
34:     return p[rootx]
35: end function
```

Algorithm 7 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
```