CHAPTER 1

INTRODUCTION

Labelling of connected components in a binary image is a very important part of pattern recognition. This is in turn used in various fields such as machine learning, data mining and knowledge discovery in databases (KDD). This is because this labelling is key for other analytical procedures 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. Therefore, connected component labelling continues to be an active area of research and there exists a need for parallelising. Connected components labelling scans an image and groups its pixels into components based on pixel connectivity, *i.e.* all pixels in a connected component share similar pixel intensity values and are in some way connected with each other. Once all groups have been determined, each pixel is labelled with a gray level, number or a colour (colour labelling) according to the component it was assigned to. This project is a parallel implementation of 2 CCL algorithms, namely Suzuki's and Two pass algorithm and performs thread level parallelism.

CHAPTER 2

LITERATURE SURVEY

R. M Haralick[1] reviews a variety of neighbourhood operators in a way which emphasizes their common form. These operators are useful for image segmentation tasks as well as for the construction of primitives involved in structural image analysis. The 4 and 8 connectivity of graphs is discussed. This is used in the algorithms being implemented in the form of the masks used while scanning.

In [2], Kenji Suzuki, Isao Horiba and Noboru Sugie present a fast algorithm for labelling connected components in binary images based on sequential local operations. A one-dimensional table, which memorizes label equivalences, is used for uniting equivalent labels successively during the operations in forward and backward raster directions. The execution time of the algorithm is directly proportional to the number of pixels in connected components in an image.

The above algorithm has been parallelised by Mehdi Nikam [3]. The algorithm has been modified to enable labelling gray-scale images. There are several data dependencies in the algorithm therefore a mechanism similar to pipelining has been implemented to exploit parallelism. The parallel algorithm method achieved a speedup of 2.5 for image size of 256 × 256 pixels using 4 processing threads.

In [4], Lefeng He and Kenji Suzuki again present an efficient run-based two-scan algorithm for labelling connected components in a binary image. Unlike conventional label-equivalence-based algorithms, which resolve label equivalences between provisional labels, the algorithm resolves label equivalences between provisional label sets. The same authors in [5] present two optimization strategies to improve connected-component labelling algorithms. Taking together, they form an efficient two-pass labelling algorithm that is fast and theoretically optimal. The first optimization strategy reduces the number of neighbouring pixels accessed through the use of a decision tree, and the second one streamlines the union-find algorithms used to track equivalent labels.

CHAPTER 3

PROBLEM DEFINITION

The conventional Suzuki and Two pass algorithm are prominently used algorithms for connected components labelling. The process of labelling connected components is a significant step in any image processing and computer vision applications. This makes it important to reduce the execution time and increase computation speed up. Thus our project aims to optimize the Suzuki and two pass algorithm using thread level parallelism and pipeline architecture.

CHAPTER 4

REQUIREMENTS AND DESIGN

1) Operating system: 64bit Windows Linux environment with a C compiler.

2) RAM: 4.0 GB.

The algorithms are implemented in C/C++ with OpenMP directives. OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++.

CHAPTER 5

SEQUENTIAL CODE

5.1 Conventional and Suzuki's Algorithm

```
1.  Algorithm (data)
2.      First pass
3.      for row in data
4.          for column in row
5.              assign a label to data[row][col] using Forward Mask
6.      Next passes
7.      while there is a change in labels do
8.          for row in data
9.              for column in row
10.                 update the labels[row][column] using Backward Mask
11.
12.         for row in data
13.             for column in row
14.                 update the labels[row][column] using Forward Mask
15.
16. return labels
```

The above algorithm is the conventional algorithm for connected component labelling. In suzuki's algorithm an additional table is maintained that reduces the number of passes and promises linear time complexity.

5.1 Two pass Algorithm

The two pass algorithm consists of helper functions that scan the image, form and merge a decision tree and flatten the decision tree to give the final labels to the image pixels. Algorithms for each of the functions are given below.

5.2.1 Main function

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

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

5.2.2 Scan function

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

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

5.2.3 Merger function

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

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

5.2.4 Flatten function

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 function
```

CHAPTER 6

IMPLEMENTATION

In order to parallelise these sequential algorithms mentioned in the section above, pipeline architecture is used. Parallelism is attempted to be achieved at the thread level. This is done by creating multiple threads all of which execute the same block of code. However, the execution is done in a fixed pipelined architecture governed by locks that are set for every thread.

A label assigned to an object pixel before the final assignment is called a provisional label. For a 2D image, a forward scan assigns labels to pixels from left to right and top to bottom. A backward scan assigns labels to pixels from right to left and bottom to top. Each time a pixel is scanned, its neighbours in the scan mask are examined to determine an appropriate label for the current pixel. In the illustration shown in Figure 6.1, the current pixel being examined is marked as **e** and the four neighbours in the scan masks are designated as **a**, **b**, **c** and **d**. If there is no object pixel in the scan mask, the current pixel receives a new provisional label. On the other hand, if there are object pixels in the scan mask, the provisional labels of the neighbours are considered equivalent, a representative label is selected to represent all equivalent labels, and the current object pixel is assigned this representative label. A common strategy for selecting a representative is to use the smallest label. A more sophisticated labelling approach may have a separate data structure for storing the equivalence information or a different strategy to select a representative of the equivalent labels. Without considering the issues such as image formats or parallelization, we divide the labelling algorithms into three broad categories: multi-pass algorithms, two-pass algorithms and one-pass algorithms.

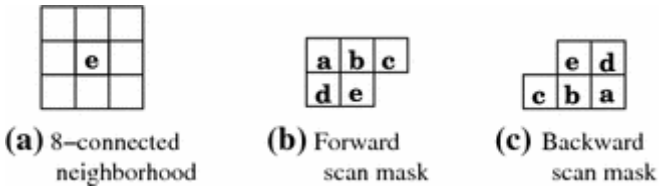

Figure 6.1: Description of an 8-connected neighbourhood and forward and backward scan masks

6.1 Multi-pass algorithms

The basic labelling algorithm described in the preceding paragraph is the best known example of this group. They may require a large number of passes before reaching the final labels. Given an image with *p* pixels, a labelling algorithm is said to be optimal if it uses $O(p)$ time. Because the number of passes over the image depends on the content of the image, multi-pass algorithms are not considered to be optimal.

To control the number of passes, one may alternate the direction of scans or directly manipulate the equivalence information. The most efficient multi-pass algorithm known is that of Suzuki et al. It uses a label connection table to reduce the number of scans.

Suzuki's algorithm promises a linear time complexity. Suzuki et al. show that maximum of four scans is sufficient for the images with complex geometrical shapes.

```
chunk = row / number of threads;
#pragma omp parallel
for row in data
    k = calculate the start index for this thread;
    while the condition is not 1 for this thread wait;
    for k in row to k + chunk
        label the pixel;
    set the condition of the next thread to 1;
```

The Suzuki algorithm is sequential in nature since the result of the previous iterations is used in the next one. Hence, to exploit parallelism irrespective of the data dependencies a method similar to pipelining is used. This implies that each thread continues iterations of the row of the previous thread while that thread moves on to the next row. For this synchronisation to take place, an additional 2 dimensional array is maintained which stores the result of the condition checks.

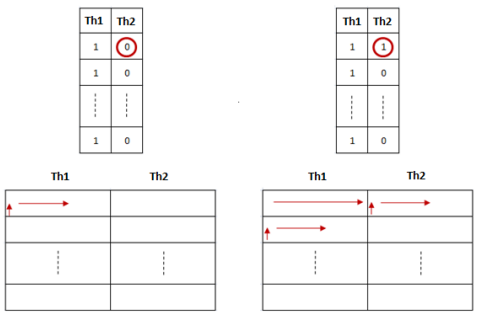

Figure 6.2: Example of the Suzuki's algorithm with 2 threads

6.2 Two-pass algorithms

Many algorithms in this group operate in three distinct phases.

6.2.1 Scanning phase: In this phase, the image is scanned once to assign provisional labels to object pixels, and to record the equivalence information among provisional labels.

6.2.2 Analysis phase: This phase analyses the label equivalence information to determine the final labels.

6.2.3 Labelling phase: This third phase assigns final labels to object pixels using a second pass through the image.

```
Input: 2D array image containing the pixel values
Output: 2D array label containing the final labels
 1: function PAREMSP(image)
 2:     numiter ← row/2 ▷ As we are processing 2 rows at a time
 3:     # pragma omp parallel
 4:     chunk ← numiter/numberofthreads
 5:     size ← 2 × chunk
 6:     start ← start index of the thread
 7:     count ← start × col
 8:     # pragma omp for
 9:     Scan_ARemSP(image)
10:     # pragma omp for
11:     for i = size to row − 1 do
12:         for col in row do
13:             if label(e) ≠ 0 then
14:                 if label(b) ≠ 0 then
15:                     merger(p, label(e), label(b))
16:                 else
17:                     if label(a) ≠ 0 then
18:                         merger(p, label(e), label(a))
19:                     if label(c) ≠ 0 then
20:                         merger(p, label(e), label(c))
21:         i ← i + size
22:     flatten(p, count)
23:     for row in image do
24:         for col in row do
25:             label(e) ← p[label(e)]
26: end function
```

Depending on the data structure used for representing the equivalence information, the analysis phase may be integrated into the scanning phase or the labelling phase. One of the most efficient data structures for representing the equivalence information is the *union-find* data structure. Because the operations on the union-find data structure are very simple, one expects the analysis phase and the labelling phase to take less time than the scanning phase. Indeed, many two-pass algorithms have the theoretically optimal time complexity of $O(p)$, where p is the number of pixels in an image.

The third algorithm is quite similar to the previous. However, it processes 2 lines at a time and 2 passes at a time, therefore uses a different mask. Hence, the pseudo code for the same does not vary. In the parallel implementation of the algorithm, the image is divided row-wise into chunks of equal size and given to the threads. A simple lock mechanism is used while merging to resolve dependencies.

CHAPTER 7

RESULTS AND DISCUSSIONS

The algorithms are parallelised using openMP directives embedded in a C code.

The execution time using the parallel architecture can be theoretically explained as follows:

$T_{sequential}$ =Number of row * Number of columns + $T_{startup}$

$T_{parallel}$ =Number of rows * (Number of columns / n) + n * $T_{startup}$

Where n is number of threads, $T_{sequential}$ is time for sequential execution, $T_{parallel}$ is time for parallel execution and $T_{startup}$ is startup time per thread.

The results obtained for each of the parallel algorithm versus its corresponding sequential code for a number of test matrices is given below.

Input matrix 1: The number of connected components is 1.

3 6

0 0 0 0 1 1

0 0 1 1 0 0

1 1 0 0 0 0

Sequential execution for Suzuki algorithm: 0.9122 ms

Parallel execution for Suzuki algorithm: 0.8841 ms

Sequential execution of Two pass algorithm: 0.3575 ms

Parallel execution of Two pass algorithm: 0.1902 ms

Input matrix 2: The numbers of connected components are 2.

4 4

0 0 1 1

0 0 1 1

1 0 0 0

1 1 0 0

Sequential execution for Suzuki algorithm: 0.9932 ms

Parallel execution for Suzuki algorithm: 0.8461 ms

Sequential execution of Two pass algorithm: 0.5092 ms

Parallel execution of Two pass algorithm: 0.5563 ms

Input matrix 3: The numbers of connected components are 4.

6 6

0 0 1 1 0 0

0 0 1 0 0 1

0 0 1 1 0 0

0 0 0 0 0 1

0 1 1 0 0 0

0 0 1 1 0 0

Time required for execution:

Sequential execution for Suzuki algorithm: 0.5832 ms

Parallel execution for Suzuki algorithm: 0.2000 ms

Sequential execution of Two pass algorithm: 0.0681 ms

Parallel execution of Two pass algorithm: 0.0332 ms

From the time stamps obtained by executing the sequential and parallel codes of the algorithms, it can be observed that there is an increase in the computation speed up obtained by thread level parallelism. The effect is prominent on larger data sets. For smaller data sets used above the decrease in execution time can be seen. Theoretically for larger data sets, the computation speed up obtained by increasing the number of threads used can be seen in the figure below.

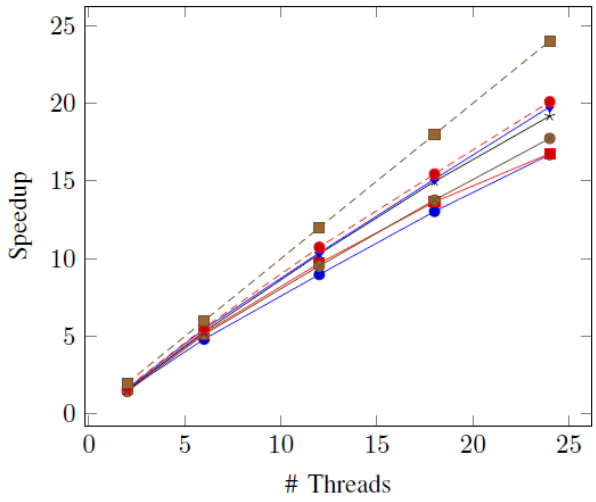

Figure 7.1: Execution speed up versus number of threads used for large images

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this work two algorithms have been implemented for optimizing the connected-component labelling process. The pipeline strategy minimizes the work in the sequential scanning phase of a labelling algorithm; and the second reduces the time needed for manipulating the equivalence information among the provisional labels. Our analyses show that a two-pass algorithm using these strategies has the optimal worst-case time complexity $O(p)$, where p is the number of pixels in an image. The optimization strategies are straightforward to implement and can be extended to higher dimensional images. It also produces consecutive labels, which are convenient for applications.

More work remains to be done for a better understanding of the performance features and trade-offs of these strategies. A derivation of a bound on the maximum number of scans needed by the algorithms would help us to understand them better. It should also be interesting to apply other parallelisation strategies such as divide and conquer and tree architectures to optimize the parallel architecture.