Parallel Connected Component Labeling Algorithm

EECE 528 Course Project
by Hongru Li
95318168

Connected Component Labeling

Connected Component Labeling (CCL) is an algorithm used in computer vision to detect connect regions in binary digital images. Rosenfeld et al.[1] define connected components labeling as the "creation of a labeled image in which the positions associated with the same connected component of the binary input image have a unique label." Serial Connected Component Labeling algorithm traverses the image pixels, labeling the vertices based on the connectivity and relatives values of their neighbors. The image graph connectivity can be 4-connected or 8-connected. The program I wrote are for 4-connected component labeling but can work for 8-connectivity with slight changes.

Fig1. The goal of CCL

The serial CCL algorithms are usually O(n) and processing a single image are quite fast. But the speedup of CCL are quite important because the call for real-time processing in target tracking. And sometimes one single image ,for example, the satellite image, can be vary big which need faster CCL algorithms.

Serial Two-Pass CCL Algorithm

The two-pass algorithm traverse the image twice to generate the label graph. The first pass gives every new connected component a unique label. However, along the way of traversing, some labeled connected components are found to be one connected region. So the algorithm will record the label of the two connected region as equal label. To summarize, the first pass gives pixels belonging to one connected region several labels which are different from labels of other connected regions and are asserted to be equal by equal label records.

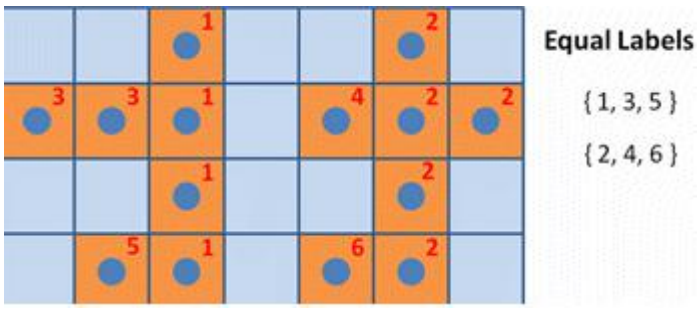

Fig2. Labeling Result After First Pass

In the second pass, the algorithm traverse the labeling graph and replace every label with its smallest equal alternative.

The data structure this algorithm used to record equal labels is an array. To check a label's smallest alternative, the label value is used as array index, and the value at that index is an equal label no bigger than the label being checked. The value in the array is the same as the index value if the label is the smallest one among equal labels.

Index	1	2	3	4	5	6	7	8
value	1	2	1	2	3	2	7	7

Fig3. Equal Label Array Expression

For example, in fig3, 8 labels are created. {1,3,5}, {2,4,6}, {7,8} are equal labels. I call 1,2,7 the root label of the three tuples respectively. You may find two separate connected region are linked to each other via a new pixel you meet. To assert the two separate connected region are one thing, you must find the root of the two region, for example, 1 and 2, and make A[2] = 1.

The work flow of two-pass algorithm can be described as:

First pass, for every white pixel:

1. Get the white neighboring elements of the current element
2. If there are no neighbors, uniquely label the current element and continue
3. Otherwise, find the neighbor with the smallest label and assign it to the current element
4. Store the equivalence between neighboring labels

Second pass, for every white pixel:

5. Relabel the element with the lowest equivalent label

OpenMPI Parallel CCL Algorithm

The OpemMPI version divides the origin image to many stripes and each process execute serial two-pass algorithm on one stripe. After that, pixels within every stripe are labeled with unique numbers. But labels from different stripes conflict to each other. Because all processes' label starts from 1. And some connected regions can be across the boundary of two stripes.

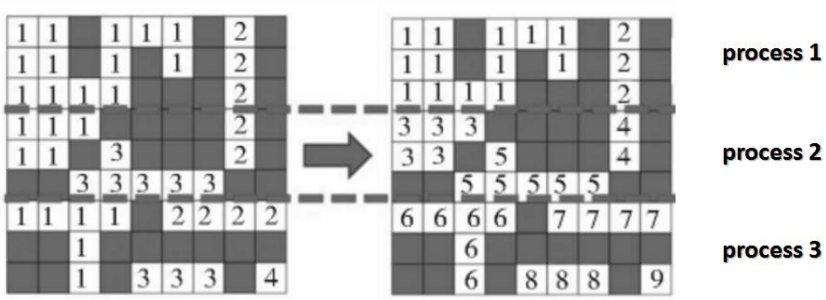

Fig4. Divide the Data to Many Processes

Therefore, to give a globally unique label to any connected region, I then let each process know how many labels all process with smaller PID used in total. The method to achieve this is to let every process tell process 1 the amount of labels it generated and process 1 sum up these numbers in order and gives a feedback. In the next stage, all process tell its upper neighbor process the boundary row and let its neighbor determine the equal labels generated by passing the partition lines. For example, in fig4, process 2 will detect the equal label tuple {5,6,7} and process 1 will detect tuple {1,3} and {2,4}. Process 1 gather the equal label info generated by all processes and merge them to a global equal labels array. Every process then change the label again according to the global equal label array.

The work flow of OpenMPI program is described below:

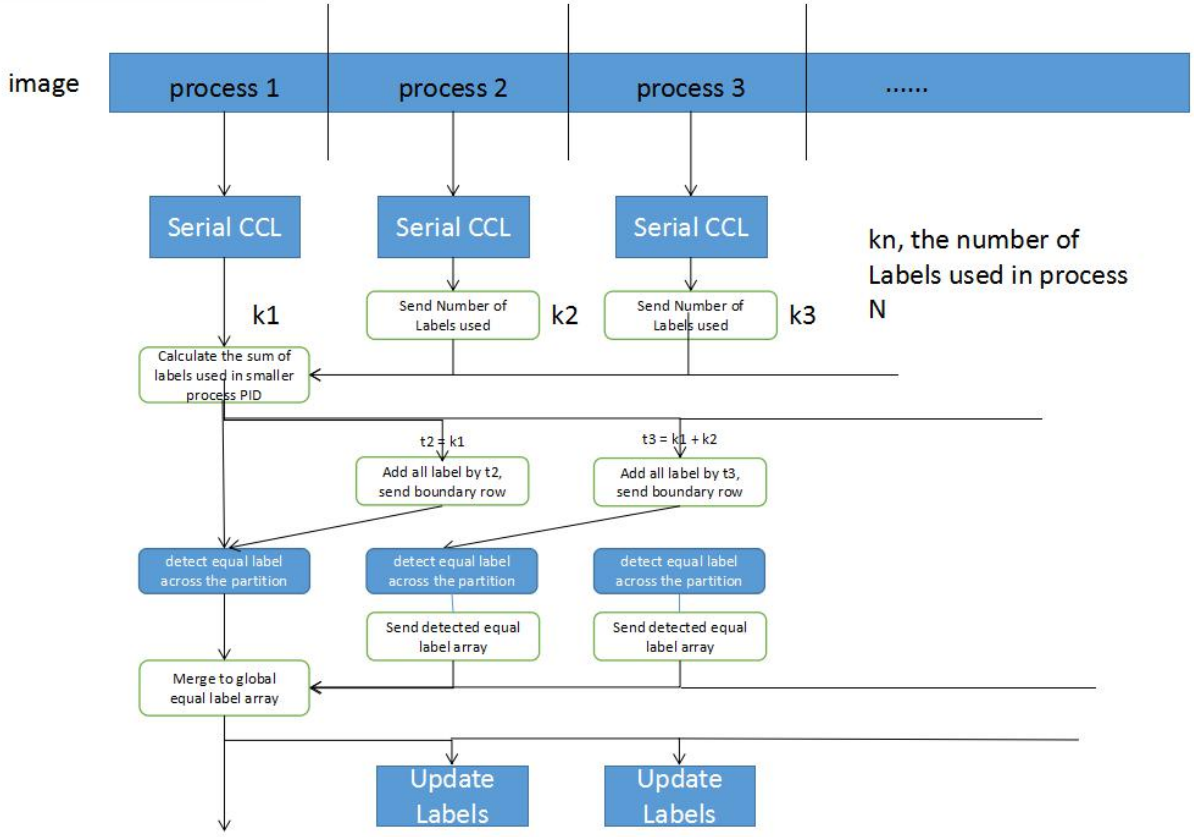

Fig5. the Work Flow of OpenMPI parallel CCL

CUDA SIMD Parallel CCL Algorithm

The two-pass algorithm is not suitable for SIMD machine because two-pass algorithm itself must be executed sequentially. But GPU can have much more threads than those provided by OpenMPI. To take the advantage of SIMD, I think the instruction executed in each processing unit should be alike and can be executed in parallel. So I use another algorithm to create CUDA CCL program.

The algorithm is shown in the graph below. The label array A is initialed with sequential numbers (A[i] = i). Firstly, each threads in GPU only process several pixels. The threads check the pixel, replacing the label with 0 if the pixel is background. Then, check the above and left

neighboring pixels and replace the label with the lowest neighboring white pixel label. The goal of this step is to make you find a root pixel whose label is equal to the index. For example in the graph below, 1 is a root pixel because A[1] = 1. The neighboring pixels' label is changing but threads do not need to synchronize. For example, the lowest label of pixel 41's neighbor is A[31], but the label 31 is changing to A[21] while A[41] changing to A[31]. The result of A[41] is unknown without synchronization. However, it is definite A[41] is smaller than 41 and lead you to the root pixel 21 afterwards.

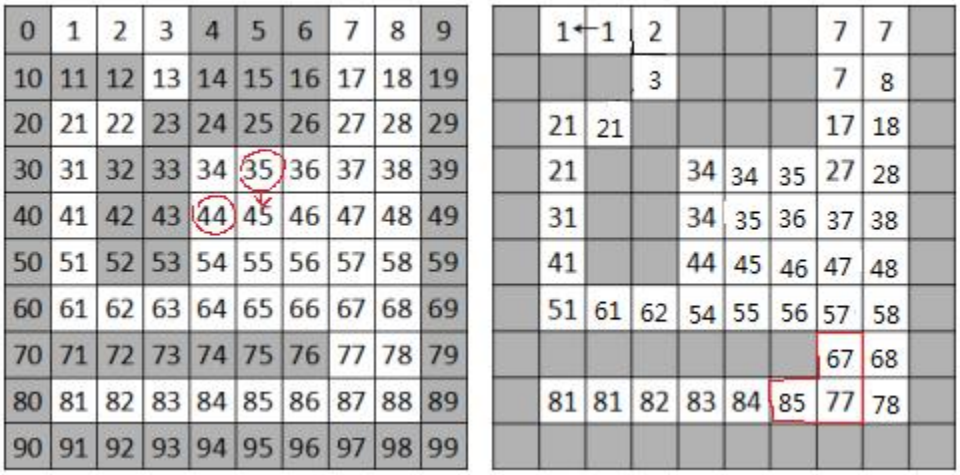

Fig6 .GPU CCL Algorithm

Now every connected region will have several root pixel like '81' and '7' in the above graph. If each connected region can keep only one root pixel, then I can finish CCL by replacing all labels with its root label. To remove any duplicate root pixels, each threads check the core pixel's above and left neighbor again and trace back to get both pixel's root pixel. If you got two different root, then remove one by make the root having bigger value point to the one with smaller value with atomic operation. For example, label 67 leads you to root '7' but label 85 leads you to root '81'. You can remove root '81' by setting A[81] = 7;

Code Verification and Manual

All three versions are tested by checking each pixel's neighbor to see if two connect white pixel have the same label.

To run the program, you should first prepare a gray-scale image and use image2input.py to convert the image to plain text binary input which can be read by my program without C++ image processing library.

./image2input.py 1.jpg source

Then, the 'source' file can be used as input.

Use command 'make' to build all three versions, all versions are built with compiler flag '-O3'. The command used to run each program is:

Serial: ./ccl_le_cpu source > result
MPI: mpirun -np 32 ./ccl_le_mpi source > result
CUDA: ./ccl_le_gpu source > result

Performance Evaluation

The input data consists of 3 images of different resolution. The smallest one is 1024x768, the medium one is 5120x3840, the biggest one is 10240*7680.

The result of MPI is shown below:

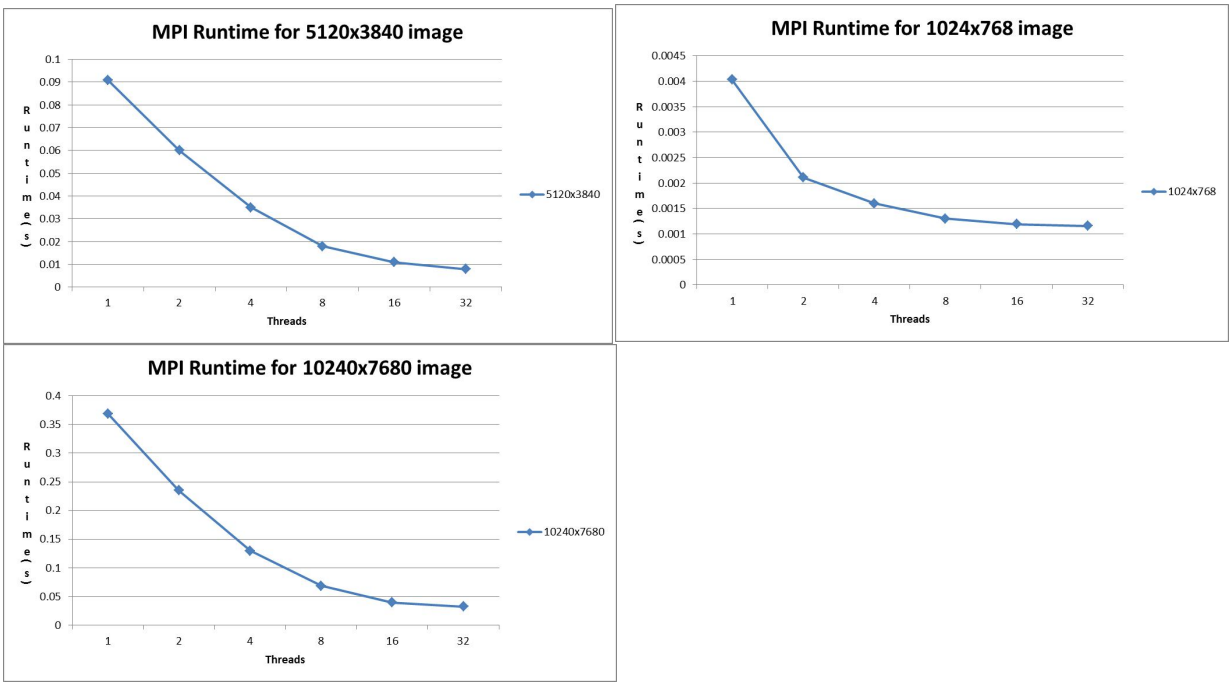

Fig7. the Run Time of MPI Parallel CCL

The speedup for medium and big image are around 10x using 32 cores. But for the smallest image, the speedup is only 5x.

	Serial (Baseline)	CUDA (speed up)	OpenMPI 32 threads (speedup)
1024x768	0.006(1.0x)	0.009(0.67x)	0.00116(5.172x)
5120x3840	0.077(1.0x)	0.027(2.85x)	0.008(9.625x)
10240x7680	0.335(1.0x)	0.067(5x)	0.033(10.15x)

Tabular1. Run Time Comparison

The efficiency of CUDA SIMD version is even worse than the baseline for 1024x768 because even serial CCL can finish in a short time for small images and I included the time of copying data between GPU and the host which is time consuming. Further, the SIMD algorithm's serial version is not as efficient as two-pass algorithm which makes it even slower for small images. But, the bigger a input image is, the more speedup my parallel algorithms have.

Conclusion

In this project, I implemented two-pass serial CCL algorithm and developed parallel CCL using OpenMPI and CUDA. Thanks to Foota and his project at github.com/foota/ccl. I mimicked his way of processing the image with python and use some of his function declarations. But unfortunately, I think his CCL algorithm is like the serial version of my CUDA algorithm but is not correct without considering duplicate roots problem I mentioned in explanation of my CUDA implementation. And his algorithm is even not built for binary images.

Both of my parallel algorithms can generate correct result and gives considerable speedup so far. But I think I still need to try more input images and images with different amount of connected components to further evaluate the performance of my code.

OpenMPI version perform better than CUDA version because two-pass CCL algorithm itself is not easy to be parallelized using SIMD. And the time used to copy data between the host and GPU can not be neglected when the total processing time is short.

Reference:

1. Rosenfeld, Azriel; Pfaltz, John L. (October 1966). "Sequential Operations in Digital Picture Processing". J. ACM. 13 (4): 471–494. doi:10.1145/321356.321357. ISSN 0004-5411

2. Yongtao Zhao, Qingkui Chen. Algorithm for Connected Component Labeling of Binary Image Using CUDA[J]. Journal of Computer-Aided Design & Computer Graphics, 2017, 29(1):72-78.

3. Yihang Ma, Lijun Zhan. Parallelization of Connected Component Labeling Algorithm[J]. Journal of Geography and Geo-Information Science, 2013, 29(4).