COMS4040A: High Performance Computing and Scientific Data Management

Connected-Components Labeling

Craig Bester - 783135
Liam Pulles - 855442

19th May 2017

1 Introduction

Connected-components labeling (CCL) is the task of detecting disjoint connected regions in binary images. It is one of the most fundamental processing steps required for computer vision algorithms to detect objects or patterns in images. The problem involves assigning a label to each pixel in an image such that the label uniquely identifies the entire connected segment of the image that the pixel resides in. Due to the importance of detecting connected image regions for computer vision applications, real-time video processing for example, there has been extensive research into increasing the performance of CCL algorithms. We present three such algorithms and analyse their performance.

2 Background

There are two different neighbourhoods that can be used in CCL algorithms when checking whether a pixel p is contained in the same segment as its neighbour pixels - the 4-connect neighbourhood, $N_4(p)$, and the 8-connect neighbourhood, $N_8(p)$. In the 4-connect neighbourhood the top, left, bottom and right neighbors of a pixel are considered for connectedness. In the 8-connect neighbourhood, the diagonal neighbors are also considered. We will be using the 8-connect scheme, which is the most commonly used since it generally matches human perception of distinct objects better.

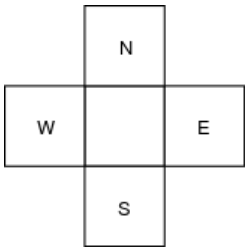

Figure 1: 4-connect Neighbourhood

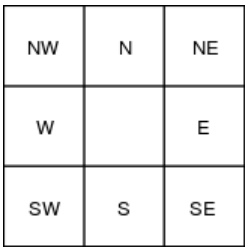

Figure 2: 8-connect Neighbourhood

A common approach to the connected-components labeling problem is to scan sequentially across the image and give each pixel either a unique label or a label based on the labels of the surrounding pixels. This initial labeling step will generally produce several subregions of labels per connected region. The merging of subregions requires equivalence classes of neighbouring labels to be resolved into a single subset. While numerous historical techniques exist that target specific architectures, the basis of modern CCL algorithms is the *union-find* algorithm. [1]

The union-find algorithm involves performing two operations on sets: finding which subset a point belongs to, and union—joining two subsets into a single subset. The union-find data structure for N subsets can be represented as an array with N elements. Each possible subset is associated with the numerical value of its position in the array. Finding the subset of a point with label x can be performed by a recursive lookup of the value of the array at position x; this operation is recursive because we want to find the 'root' label, which defines the entire equivalent subset such that all labels in the subset are transitively equivalent to the root. Merging two subsets is performed by assigning the label of one of the subsets to the other in the array. For example, if we merge the subsets 2 and 5, we write 2 to the array at position 5.

3 Algorithms

We present three connected-components labeling algorithms for binary images based on union-find: a serial implementation, a parallel GPU implementation (using NVIDIA's CUDA programming model),

Figure 3: $FS_8(p)$ — Forward-Scan Neighbourhood for 8-connected

and a distributed parallel implementation (using MPI).

3.1 Serial Implementation

The input to a CCL algorithm is a binarised image, where pixels values are either 1, pixel is part of a region, or 0, pixel is part of the background or not part of a region. The basic serial connected-components labeling algorithm using union-find involves three steps:

1. **Initial labeling**: Each pixel is assigned a unique, sequential label based on its position in the image. We use a 1D array of labels for our union-find data structure.

2. **Forward-scan**: We sequentially scan across the rows of the image from top to bottom. At each pixel, we merge its label with the labels of the pixels in the forward-scan neighbourhood, merging the adjacent subsets. This step creates local equivalent subsets and creates chains of labels (since label 3 may be marked as equivalent to label 2 and label 2 could be marked as equivalent to label 1) that need to be resolved to merge entire regions.

3. **Equivalence Resolution**: We now examine each pixel and find its 'root' label. This involves recursively looking-up the label, following the path of equivalent subsets created in the forward-scan step until we reach the root, which points to itself in the label array. We then assign the root label to the current pixel. In this way, all pixels are assigned their final label corresponding to its disjoint, connected region in the image.

We note that the path length in the recursive lookup varies based on the input image. Some regions may be completely described by a single label after the forward-scan, while others could have many subregions due to irregular shapes. Some images may have many regions while others could consist of one large region. Additionally, we do not process background pixels in the algorithm. Because of this, it is impossible to provide an accurate theoretical complexity analysis of the algorithm as the work done depends on the structure of the input image, not only its size. We can, however, provide a rough upper bound.

We perform $\mathcal{O}(5n)$ label lookups and comparisons in the forward-scan step, and we perform at most n union-find (recursive find and merge) operations in the equivalence resolution step. The asymptotic complexity of n union-finds on n pixels is $\mathcal{O}(nlog^*n)$, where log^*n is the *iterated logarithm* [3]. Hence we can estimate the total asymptotic complexity of the serial algorithm to be

$$\mathcal{O}(5n+nlog^*(n)).$$

We do not count the initial labeling because it consists of assignments, not union or find operations; although it would only add another $\mathcal{O}(n)$ term. Note that the iterated logarithm function can be considered as almost constant, so the overall algorithm scales almost linearly with respect to the number of pixels in the image. This is supported by our empirical results in section 5.

3.2 MPI Implementation

Our parallel algorithm for distributed memory systems uses MPI and is based on the union-find serial algorithm. The master process reads in the image and scatters it. We use a row-block distribution scheme

Algorithm 1: Serial CCL

```
Input  : I — image with n pixels
// Step 1: initial labels (0 reserved for background pixels)
for i ← 0 to n − 1 do
    if I[i] = 1 then
        I[i] ← i + 1
    else
        I[i] ← 0
    end
end
// Step 2: forward-scan
for i ← 0 to n − 1 do
    if I[i] ≠ 0 then
        for p ∈ FS_8(I[i]) do
            if label(p) ≠ 0 then
                Union(I[i], label(p))
            end
        end
    end
end
// Step 3: resolve equivalences
for i ← 0 to n − 1 do
    if I[i] ≠ 0 then
        I[i] = Find(I[i])
    end
end
```

where each process gets a single contiguous block of rows to process. The use of row-blocks is primarily due to the indexing and labeling convention required in CCL; using a row-block distribution, we can easily convert the process-local labels to global indices that correspond to the union-find array over the entire image when merging.

Each process resolves the connected components in its allocated block using the serial CCL algorithm. Once this is done, we gather all blocks and merge the local results. This is done by scanning the row boundaries between each of the blocks and resolving equivalences between labels in these bordering regions. Finally, the labels are updated to their final label using the recursive find. We outline the algorithm in pseudo-code (algorithm 2). As with the serial implementation, we use a 1D array to represent the union-find data structure. We only communicate portions of this label array to the processes, as it corresponds to the image anyway.

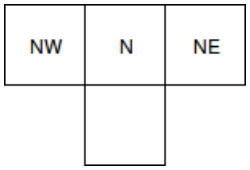

Figure 4: Merge Scan Mask

We use `Scatterv` and `Gatherv` to account for the number of processes not evenly dividing the number

Algorithm 2: MPI CCL

Input : I — image with n pixels
$I_p \leftarrow$ `Scatterv`(I)
`CCL`(I_p)

// Merge on master process
$I \leftarrow$ `Gatherv`(I)
if *rank* $= 0$ **then**
  `Merge`(I)
end

of rows in the image, leading to the last process possibly receiving fewer rows. While it would be possible to merge the local blocks over several processes, it would be redundant work because the final process always has to resolve the global labels over the entire image (since introducing even one extra equivalence may require changing all the labels in the image). Having one gather step also reduces the amount of communication overall. Considering the algorithm takes only a few milliseconds, the communication overhead relative to the work required can be large.

The overall work required increases with the MPI algorithm relative to serial. With p processes and assuming p divides n, the work done for each process is

$$\mathcal{O}(5\frac{n}{p} + \frac{n}{p}log^*(\frac{n}{p}))$$

for the CCL step on the process-local blocks. Let t_s be the startup time of a communication and t_l be the time taken to transfer a label in the image, the total communication overhead for a scatter and gather is then

$$2(t_s log_2(p) + t_l(p-1)n).$$

Additionally, the merge step requires scanning the boundary rows between each block. We use the scan mask given in Figure 4 since we do not need to check every neighbour in $FS_8(p)$, only the pixels in the neighbouring region. This means the master process (rank 0) that merges the local results performs

$$\mathcal{O}(4 * width + nlog^*(n))$$

operations to produce the final image, where *width* is the width of the image. The total work we do is then:

$$\mathcal{O}(5n + nlog^*(\frac{n}{p}) + 4 * width + nlog^*(n) + 2(t_s log_2(p) + t_l(p-1)n)).$$

Unfortunately, we note that this implementation is prone to load balancing problems because of the nature of the CCL algorithm being dependent on the structure within the input image. For example, one process may have only background pixels in its block, and so cannot do any processing. Additionally, the merge is only done on the master, but we noted that this method reduces communication overhead, which is a bigger factor than load balancing in this case.

3.3 CUDA Implementation

For our CUDA implementation, we used an algorithm described by [2]. In basic terms, the algorithm works as follows:

1. Assign one thread to each pixel.

2. Each thread sets its pixel's label to its unique thread id unless its pixel is part of the background, i.e. value 0.

3. Each thread looks at its forward neighbors (Figure 3) and updates its label to the minimum label in the neighbourhood.

4. Each thread uses its label as an index in the image (which we treat as the union-find data structure) and sets its label to the root node, found by a recursive lookup.

5. The labels now directly correspond to a few select indices. All threads again analyse their forward neighbors, and update their root label to the minimum of the two labels if they find a neighbor with a different label. At the end of this, some of the representative labels will have changed, representing merged regions.

6. Update the labels to be the root label, as in step 3. There will be a thread race here, but CUDA's atomicMin() can alleviate this to a large degree.

7. Repeat steps 4 through 6 until there are no more representative label updates. We can keep track of this by means of a boolean flag.

In our implementation we made use of surfaces (textures with write ability) to make use of the spatial locality speedup of the surface/texture data structure. However, as we see in section 5, our global memory implementation is generally faster. This may be due to being able to use atomicMin() with global memory, but not with surfaces, during step 5.

Note that this is a multi-pass algorithm, with the number of passes performed during the loop depending on the structure of the input image. We used separate kernel calls for each step in order to synchronise the operations. This does not add significant overhead because the time taken is only a few microseconds per call (wheras the overall algorithm takes milliseconds to run).

4 Experimental Methodology

We ran two sets of experiments over images of varying sizes and segmentation complexities.

- **Random images**: Generated images with a random binary value at each pixel. The sizes vary from (100×100) to (2000×2000) in increments of 100 per side. That is, images of size (100×100), (200×200), (300×300), ... , (2000×2000). We draw columns of black pixels at evenly spaced intervals to make more regions in the resulting random image, otherwise the image would have several tiny clumps of regions and one region covering almost the entire image.

- **Challenge images**: These images are of various sizes and contain particularly interesting segmentations. The images include:

  - A page of binarised text from Hamlet.[1]
  - A maze image.[2]
  - A fingerprint. [4]
  - A tobacco company letter with binarised text.[3]

The time taken in milliseconds (ms) for each algorithm to label each image is recorded, including any overhead of communicating the image across nodes or copying it to the GPU. We do not measure the time taken to load in the initial image or colour the final labeled image in the experiments, because each implementation performs these steps in the same manner as serial. We average timings over 10 runs of each algorithm for each image.

[1] Project Gutenberg (http://www.gutenberg.org)
[2] https://www.allkidsnetwork.com/mazes/
[3] http://legacy. library.ucsf.edu/

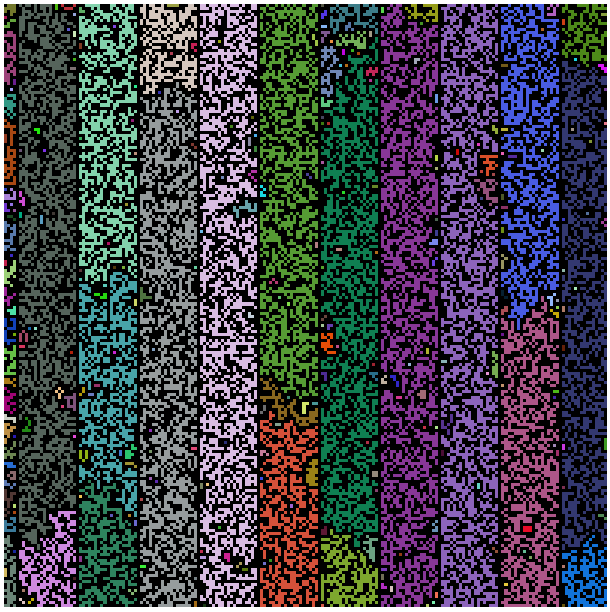

Figure 5: Randomly generated (200×200) image after being labeled and colourised.

5 Results and Analysis

1. We first compared using surface memory versus global memory for our GPU implementation (Figure 6) and found that the global memory was slightly faster. Possibly due to being able to use atomicMin() during the update step.

2. We then did a search for the ideal block size for CUDA over different sizes of random images (Figure 7). We see that (8×8) blocks are generally the fastest for our algorithm.

3. Similarly, we searched for the ideal number of processes to run our MPI algorithm, noting that the cluster only allows 3 processes per node. The results of this search are presented in Figure 8. We chose 4 processes over 2 nodes because it is generally the fastest and most consistent.

4. We then tested the average execution time of the serial, GPU (using global memory and 8x8 thread blocks), and MPI (using 4 processes and 2 nodes) algorithms on randomly generated binary images, Figure 9. We see that the GPU version is the fastest, followed by MPI, and then the serial version. The spikes in the execution time can be attributed to running all the experiments on the cluster at once. It is possible that the serial and CUDA versions may have been running on the same node at once and conflicted with the processing time of each other by chance. The speedup of the parallel algorithms is shown in Figure 10. The GPU has an average speedup of around 3.3, while the MPI version has an average speedup of around 1.4.

5. For MPI, we measured the time taken for per-process labeling and the overall time of the algorithm. With this we can roughly derive the overhead by taking the difference of these two times. The results are shown in Figure 11. We also calculated the ratio of the overhead and total execution time, shown in Figure 12. We see that the overhead ratio reduces slightly as the image size increases, a positive indicator for the scalability of the algorithm.

6. We similarly measured the overhead versus labeling time for CUDA. The results are shown in Figure 13. Note that the overhead time for CUDA is less than that of MPI. The ratio of the overhead and total execution time is shown in Figure 14. We see that the overhead ratio reduces more prominently than with MPI as the image size increases, however the overall fraction of overhead time is much greater than with MPI. Note that we see the spike in processing time from previous graphs explained by the spike in overhead in Figure 13.

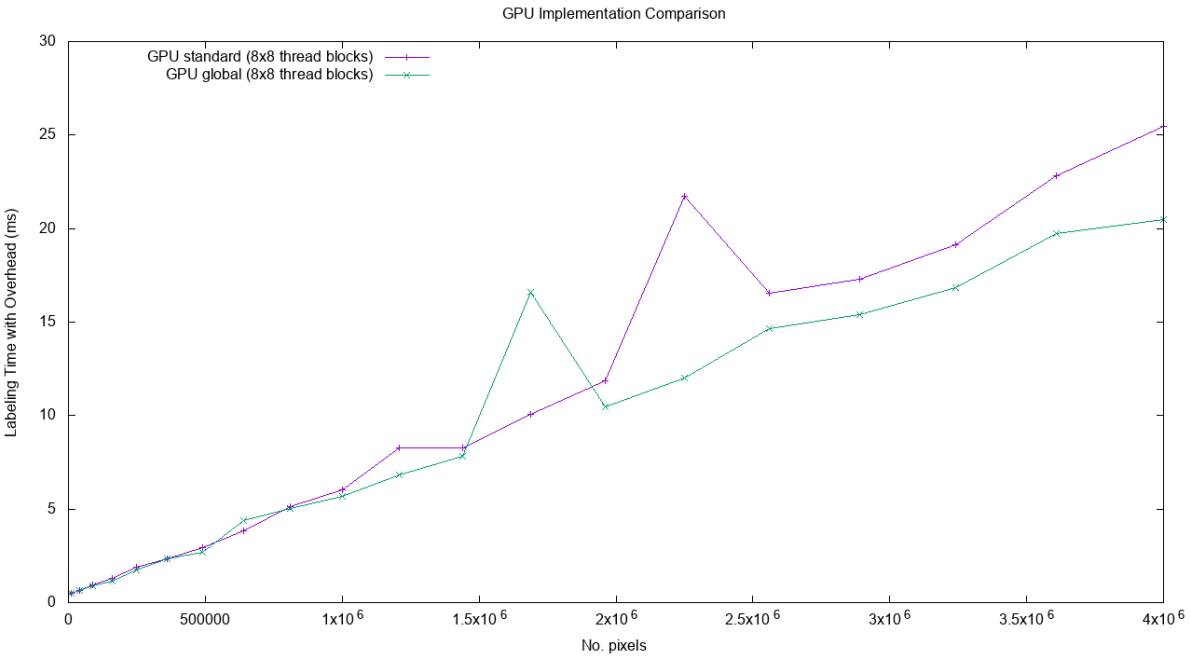

Figure 6: GPU execution times with surface memory vs global memory on randomly generated images (lower is better).

7. For MPI we empirically calculated the serial fraction from the Karp-Flatt metric using different numbers of processes and a fixed image size of (1700×1700), Table 1. The serial fraction increases (overall efficiency falls) as we increase the number of processes, as expected since we are increasing the overall overhead. We also calculated the serial fraction for a fixed number of processes (4) with increasing image sizes, shown in Table 2. The serial fraction remains relatively constant.

Processes	2	4	8	16	32
Speedup	0.978	1.334	0.762	1.028	0.835
Serial Fraction e	1.045	0.666	1.357	0.971	1.204

Table 1: Karp-Flatt serial fraction metric for MPI on a randomly generated (1700×1700) image with different numbers of processes.

Image Width	1400	1500	1600	1700	1800	1900	2000
Speedup	1.322	1.223	1.613	1.335	1.429	1.273	1.360
Serial Fraction e	0.675	0.757	0.493	0.666	0.600	0.714	0.647

Table 2: Karp-Flatt serial fraction metric for MPI on randomly generated images using 4 processes and 2 nodes.

8. Finally, we tested our implementations on real-world images, which we called 'challenge' images. We present the comparative results in Figure 15. The varying nature and structures of these images highlight the differences between our implementations when there are different numbers of disjoint regions and large black sections in the images. Notably, our MPI implementation performs worse than serial on most of these challenge images.

We note that the time complexity of the CCL algorithms follow a roughly linear trend as seen in Figure 9, supporting our theoretical analysis. Despite being a multi-pass algorithm, the labeling part of the CUDA implementation is shown to be much faster than serial or MPI, with overhead making up

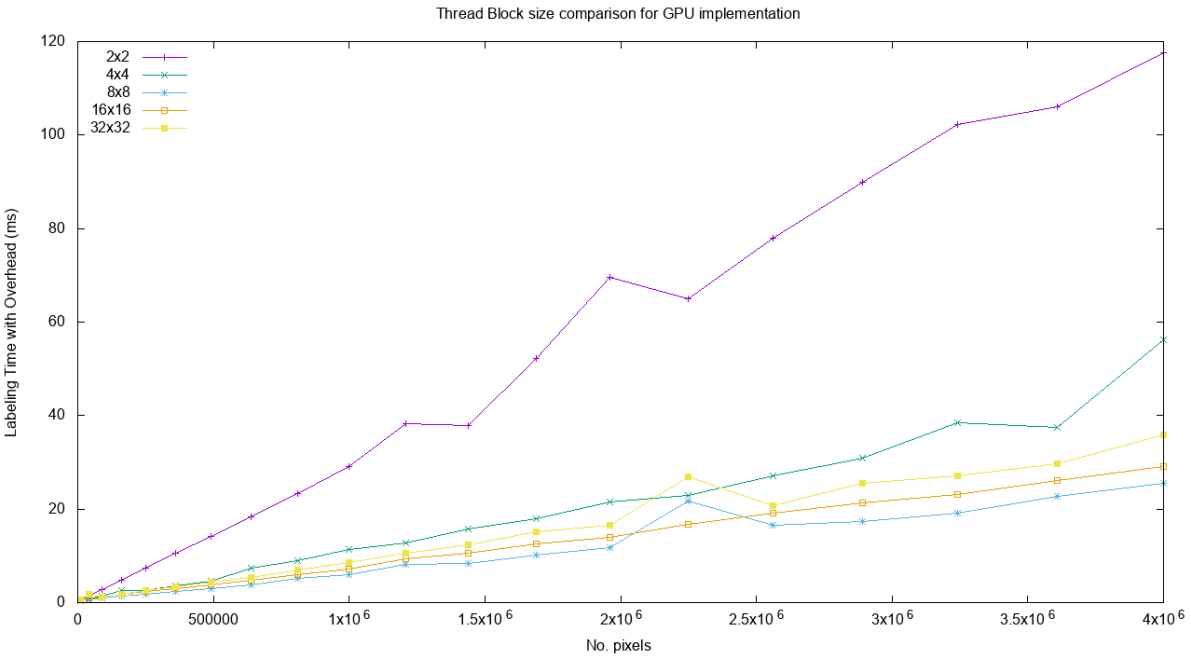

Figure 7: GPU execution times with different block sizes on randomly generated images (lower is better).

$25\% - 50\%$ of the execution time. The overhead of the MPI implementation, on the other hand, makes up about 10% of the total time, indicating that we could better exploit MPI if we could improve the serial labeling component, possibly by using OpenMP.

The fact that using global memory was faster than the surface in CUDA was an unexpected result, and likely due to the fact that we were able to use atomic operations with global memory.

6 Conclusion

Connected-components labeling has a variety of applications, ranging from optical character recognition, through to object detection. In the case of video (where the images need to be labeled in real-time) it is vital that the connected-components labeling software works rapidly to keep up with each frame.

We presented three CCL implementations, including two parallel algorithms using MPI and CUDA respectively. Our results showed that parallelising CCL resulted in a significant performance increase with CUDA and indicated that our parallel algorithms are potentially scalable. However, we saw that this problem is not well suited to distributed memory parallelisation using MPI due to the cost of network communication being relatively high.

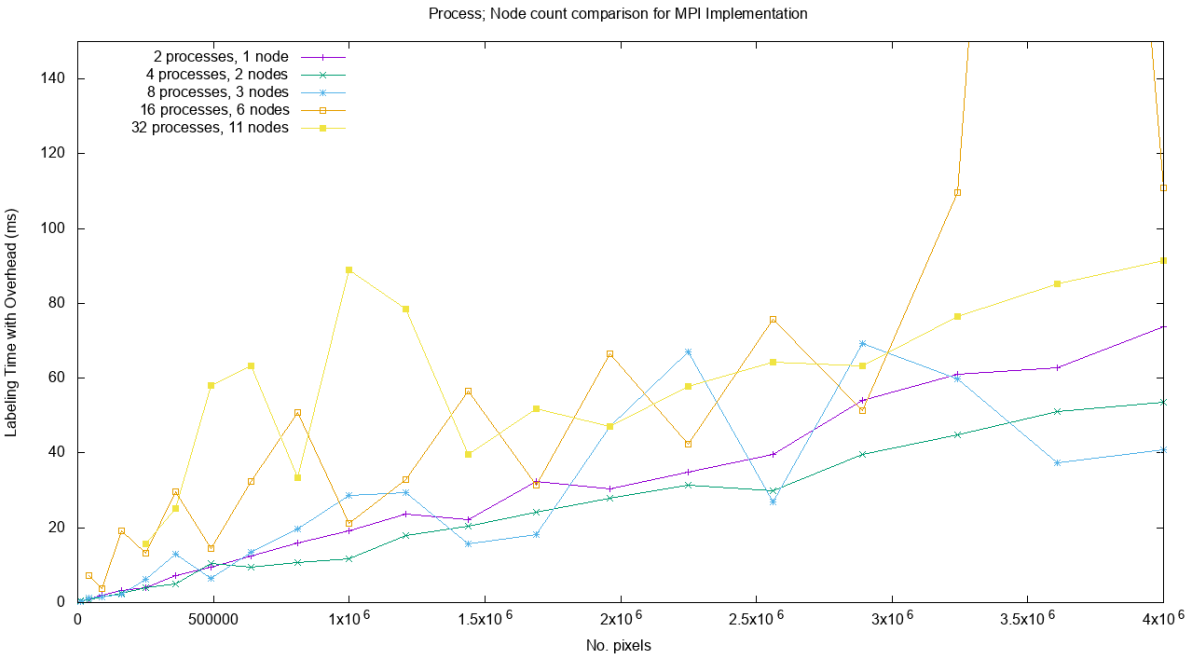

Figure 8: MPI execution times with different numbers of processes on randomly generated images (lower is better).

References

[1] Grana, Costantino, Daniele Borghesani, and Rita Cucchiara. "Connected component labeling techniques on modern architectures." *Image Analysis and Processing.* ICIAP, 2009, 816-824.

[2] Jung, In-Yong, and Chang-Sung Jeong. "Parallel connected-component labeling algorithm for GPGPU applications." *Communications and Information Technologies (ISCIT), 2010 International Symposium on.* IEEE, 2010.

[3] Raimund Seidel and Micha Sharir. "Top-down analysis of path compression", SIAM J. Computing 34(3):515–525, 2005.

[4] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, Handbook of fingerprint recognition. Springer Science & Business Media, 2009.

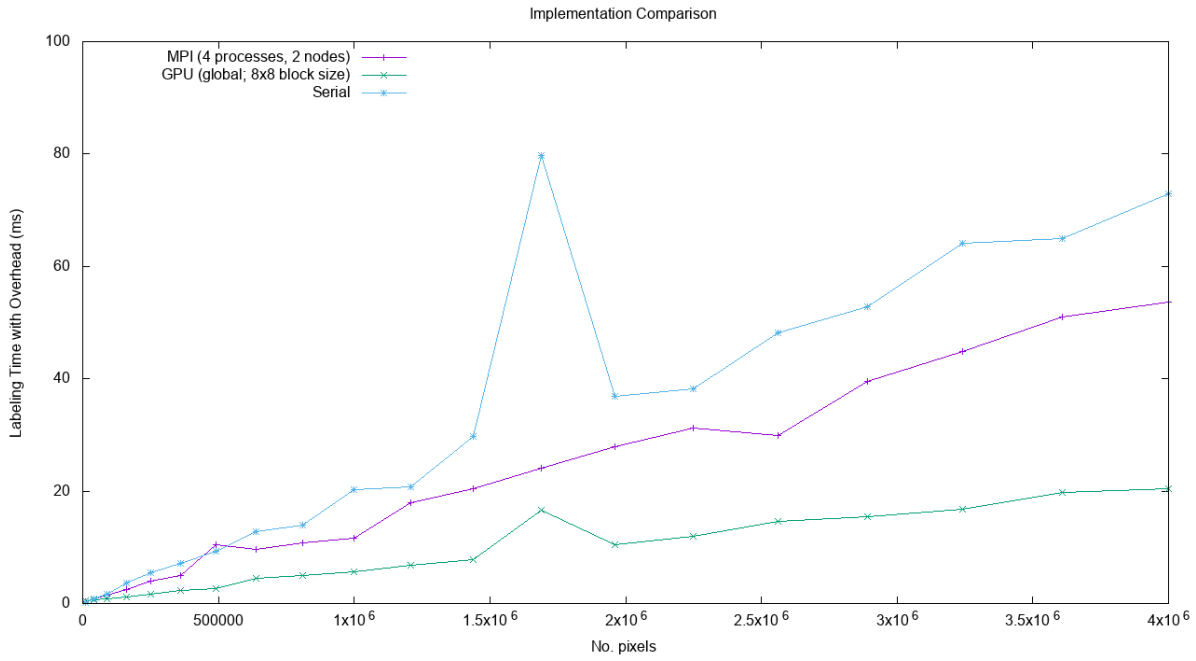

Figure 9: MPI execution times with different numbers of processes on randomly generated images (lower is better).

Figure 10: Speedup of the parallel algorithms on randomly generated images (higher is better).

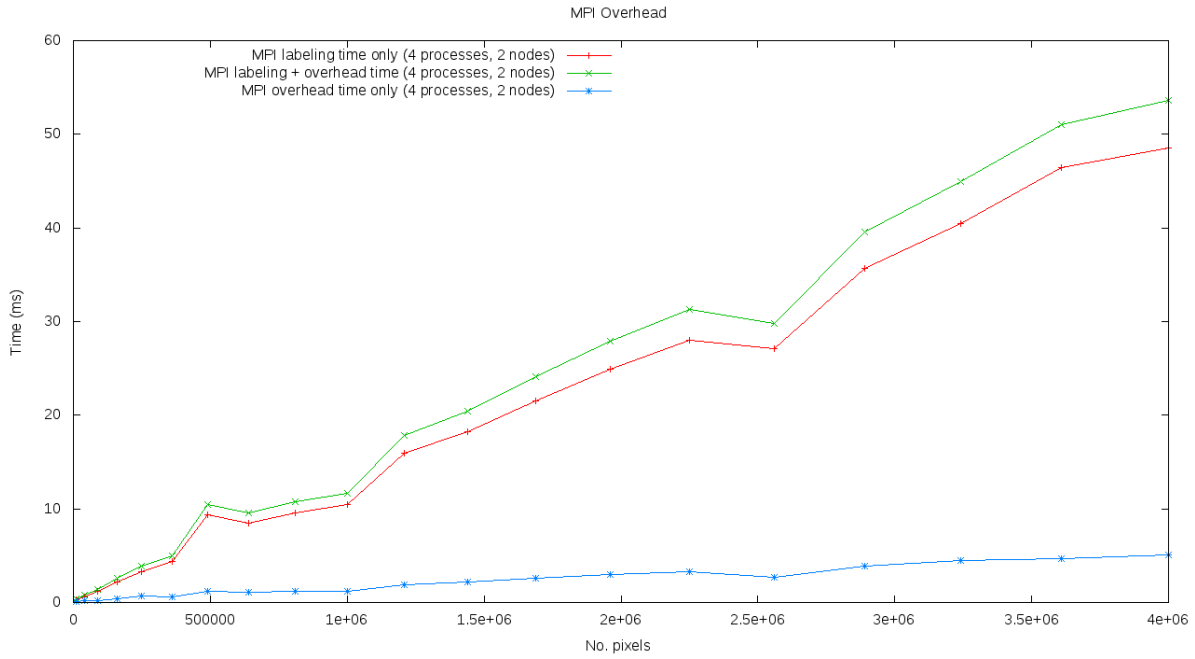

Figure 11: MPI average total execution time, per-process labeling time, and overhead time for 4 processes over 2 nodes.

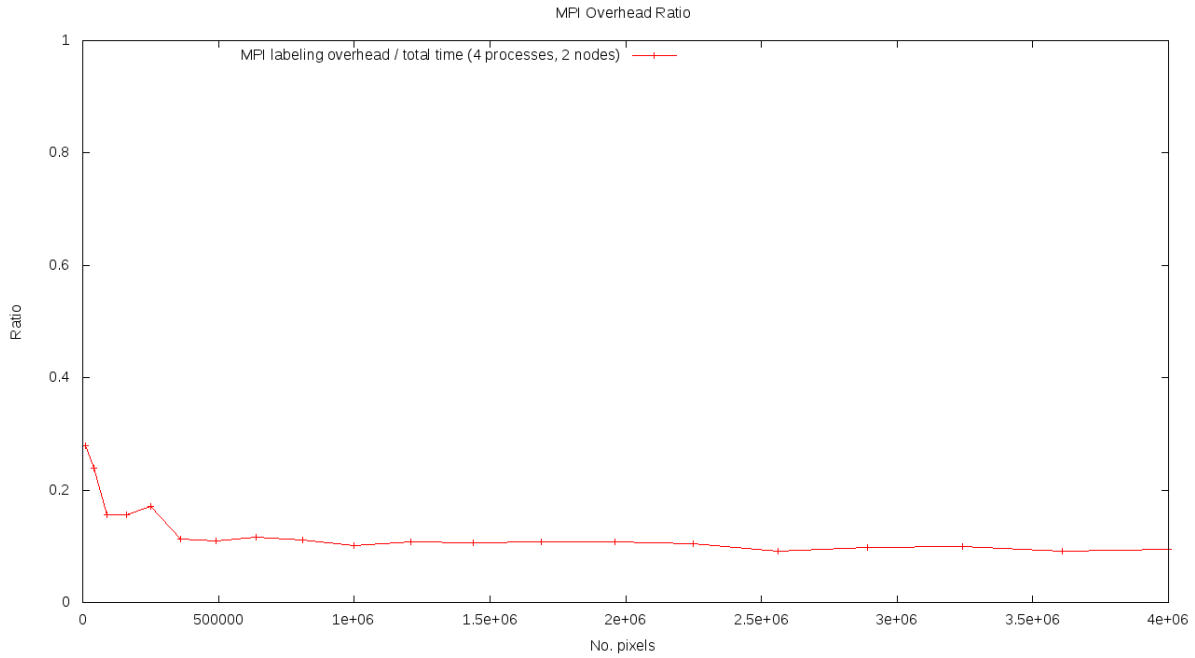

Figure 12: Average ratio of MPI overhead time versus total execution time for 4 processes over 2 nodes.

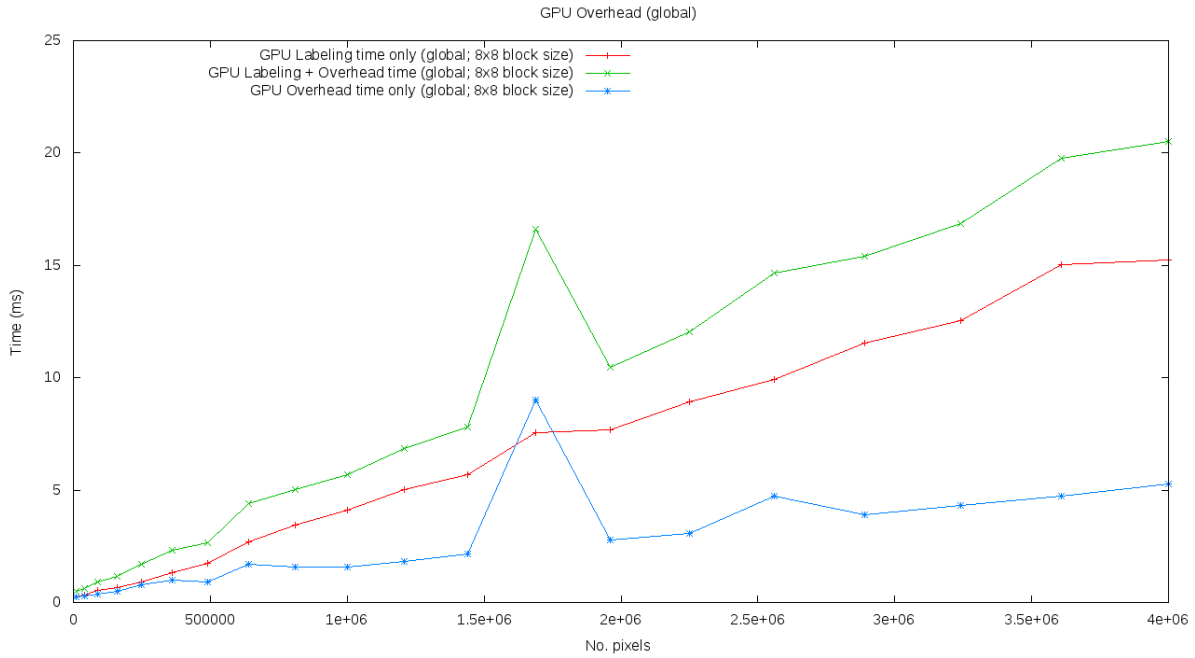

Figure 13: GPU average total execution time, per-process labeling time, and overhead time using global memory and (8×8) thread blocks.

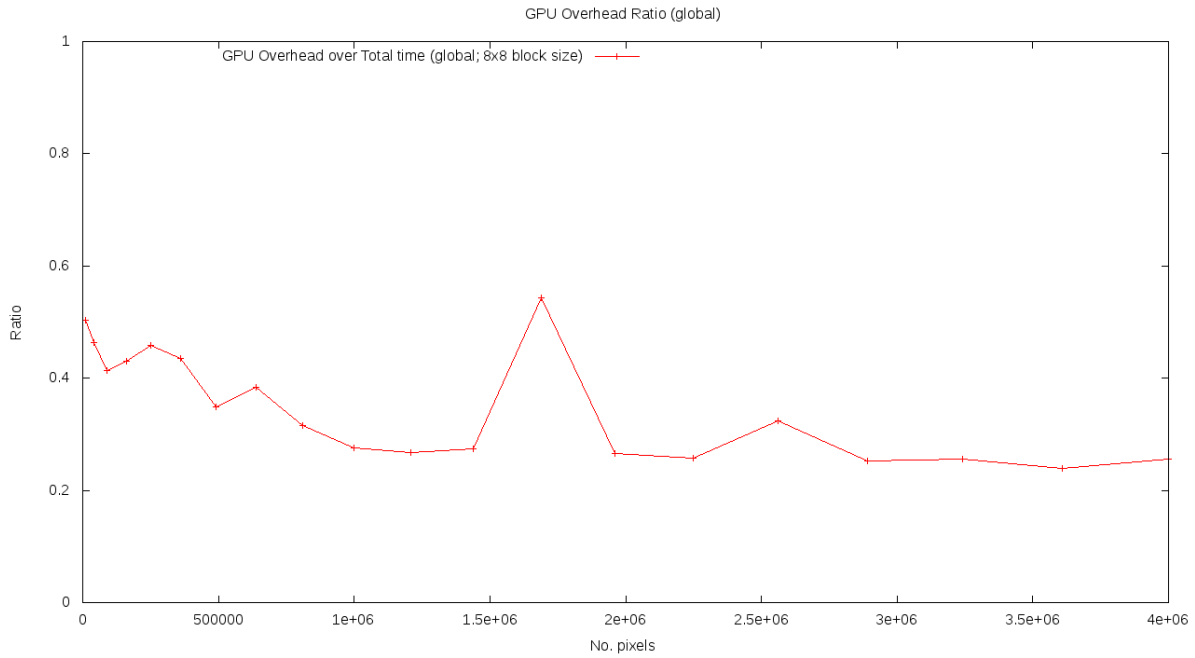

Figure 14: Average ratio of GPU overhead time versus total execution time using global memory and (8×8) thread blocks.

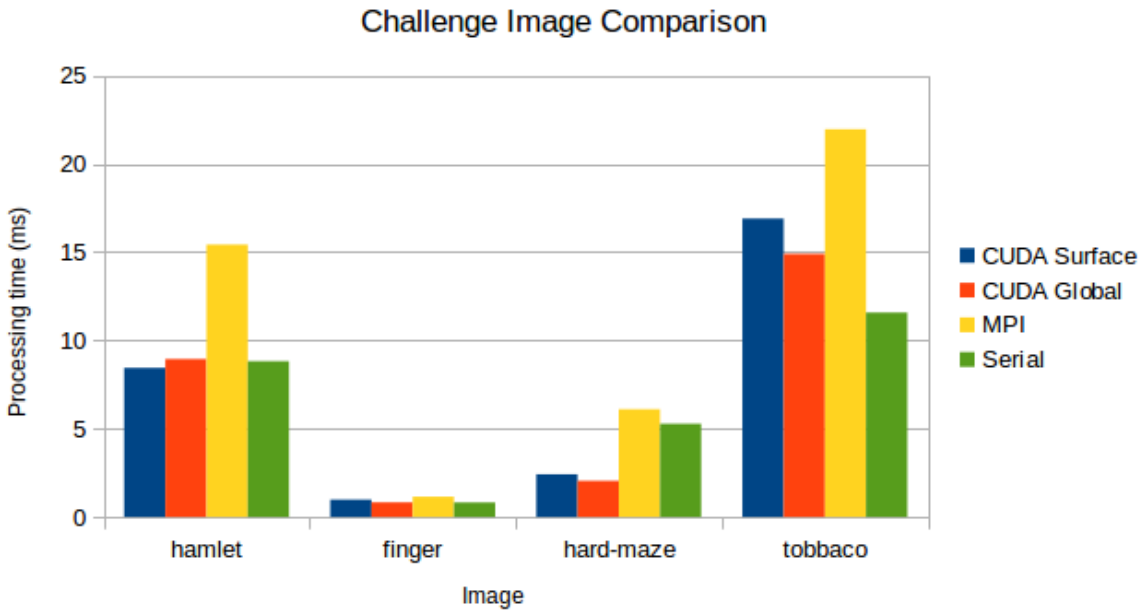

Figure 15: Execution times of CCL implementations on challenge images.