Connected-Components Labeling

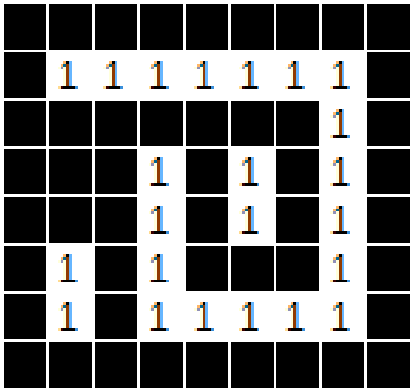
A Presentation by Craig Bester & Liam Pulles

Outline

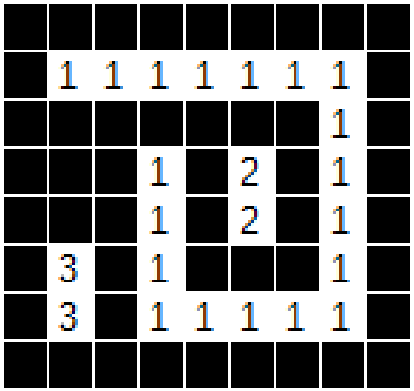
- ***Connected-Components Labeling*** is the task of finding and labeling the disjoint parts in a graph.
- In computer graphics terms, this becomes the problem of finding distinct regions in a binary image.
- This is often used to detect objects; regions of interest in an image.

Process Example

Thresholded image:



Labelled image:

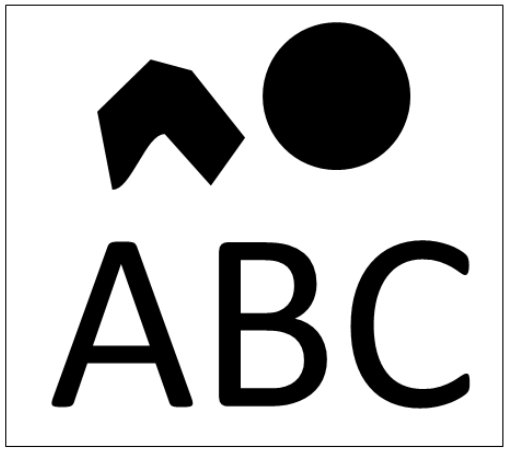


We do not label the black pixels – part of the background.

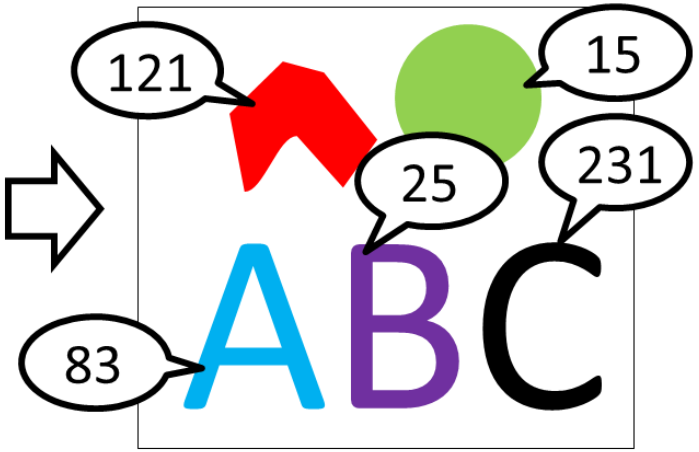
- After the connected-components labeling has completed, each pixel has a label corresponding to their region.

Image from: https://www.codeproject.com/Articles/825200/An-Implementation-Of-The-Connected-Component-Label

Example continued



Input : Binary image



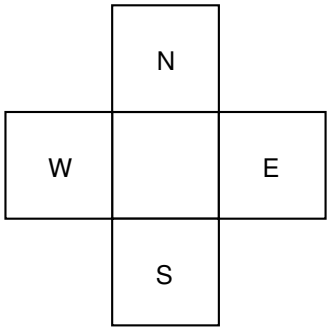
Output : Pixels in each connected component are assigned an ID unique to the connected component

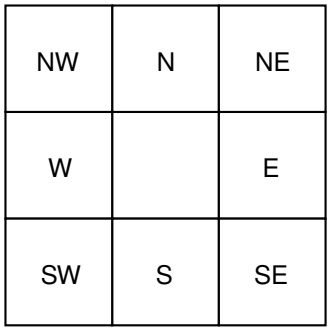
- To make the distinct regions easier to see, the output is colourised for demonstration purposes.

Image from: http://ito-lab.naist.jp/~n-sibata/cclarticle/index.xhtml

4-connect vs. 8-connect

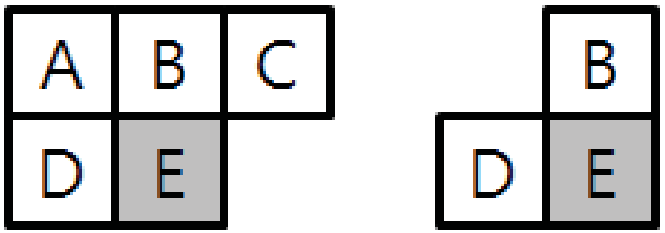
- There are different ways of regarding neighbors.
- **4-connect**: Pixels are connected according to their N,E,S,W neighbors.
- **8-connect**: Pixels are connected according to their N,NE,E,SE,S,SW,W,NW neighbors.
- We use 8-connect for our implementations.





Forward Pass

- Because it can be inefficient to check all 8 neighbors at each pixel, if we are going left-to-right and row-to-row, we can use a special mask.



- **Left**: a *forward scan mask* for 8-connect.
- **Right**: a *forward scan mask* for 4-connect.

Union-Find Overview

- The **Union-Find algorithm** is an efficient way to keep track of which pixels belong to which regions or “sets”. We use some variation of Union-Find for all our implementations.
- Initially, assign all pixels their own distinct sets.
- To put two pixels into the same set, we “union” the sets together.

Union-Find Overview

- Once we are finished unioning pixels into sets, we can retrieve their region label by doing a set lookup – a **find**.

- The find is not an immediate operation, and requires traversing a set of members to reach a “root” member, which holds the group label representing the connected region.

Union-Find Overview

- We use a 1-D array the same size as our image as a union-find data structure.
- The labels are also treated as the indices of the pixels.

Initial labeling

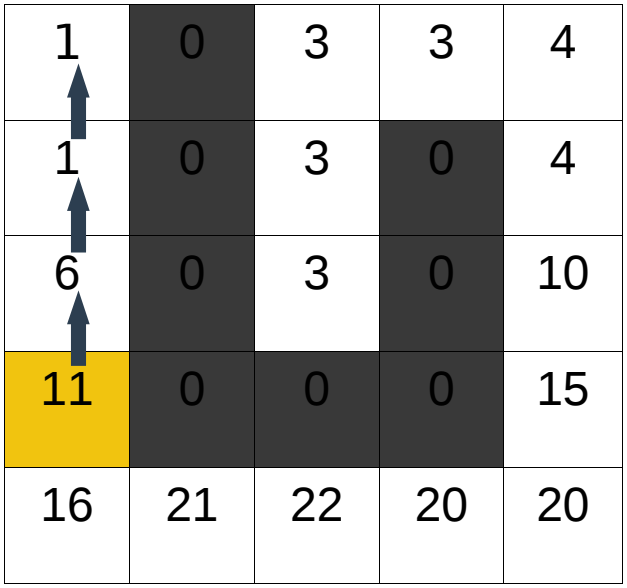
1	0	3	4	5
6	0	8	0	10
11	0	13	0	15
16	0	0	0	20
21	22	23	24	25

Find Operation

- During a find operation, root propagation works by using the label of a pixel as an index to traverse to another pixel’s label in the data structure.
- We do this recursively until we find that the pixel’s label matches its index. Then we are at a root label.

Root Propagation example:

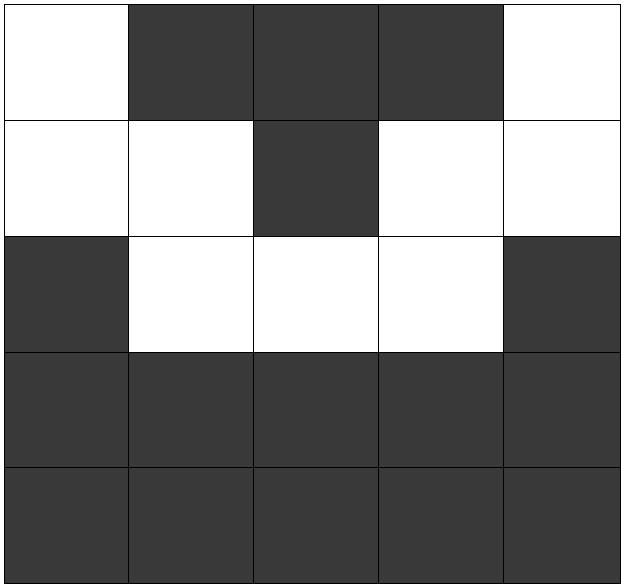
Pixel 16 with label 11 wants to find its root label.



- Here we can see that for pixel 16 to find its root node it needs to traverse: labels[11] = (6), labels[6] = (1), labels[1] = (1). We can see this is a root node as the label equals its index, and we stop.
- The label of pixel 16 can then become 1, this is known as path compression so we do not have to traverse the entire path again.
- Our MPI and Serial algorithms implement path compression by setting all the labels of the nodes along this path to the root node. This speeds up future finds.

Union-Find example

Input Binary Image



Initial labels

1	0	0	0	5
6	7	0	9	10
0	12	13	14	0
0	0	0	0	0
0	0	0	0	0

Union-Find example

Union(6,1)

1	0	0	0	5
6	7	0	9	10
0	12	13	14	0
0	0	0	0	0
0	0	0	0	0

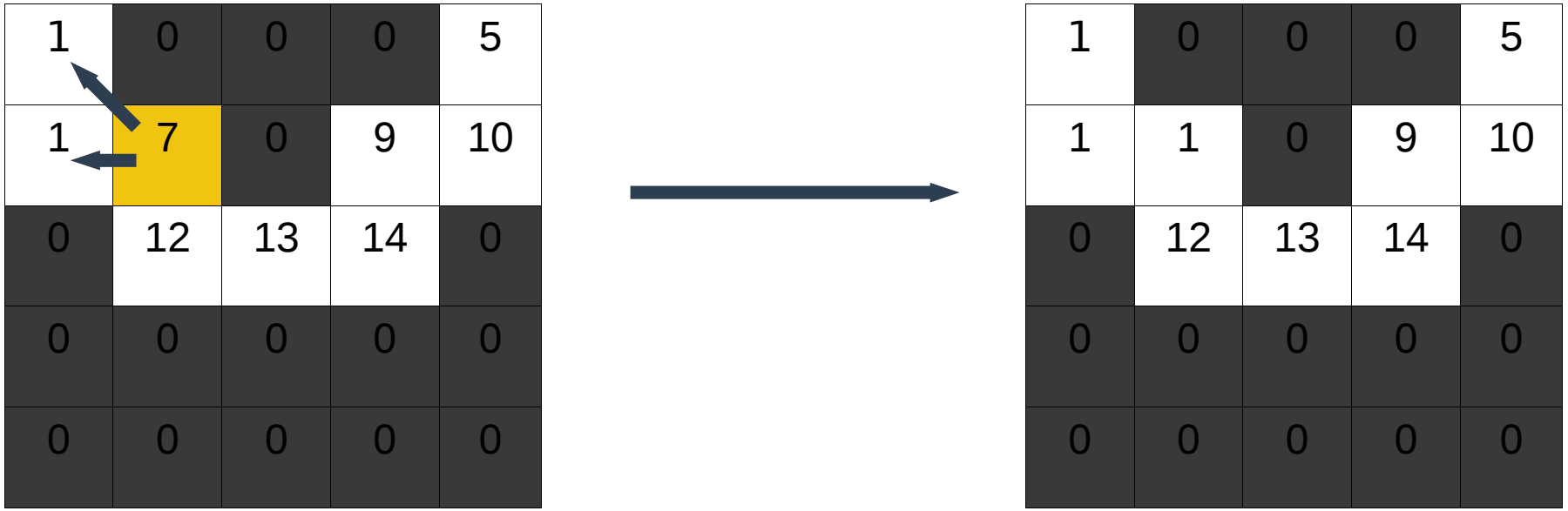
1	0	0	0	5
1	7	0	9	10
0	12	13	14	0
0	0	0	0	0
0	0	0	0	0

The forward pass mask results in us doing no operations on the top row. To do a union, we first need to call a find in both (6) and (1).

We see labels[6] = 6 and labels[1] = 1, so they are root nodes at this stage. labels[Max(1,6)] is then set to labels[min(1,6)], in this case labels[6] is set to 1.

Union-Find example

Union(7,1); Union (1,1)

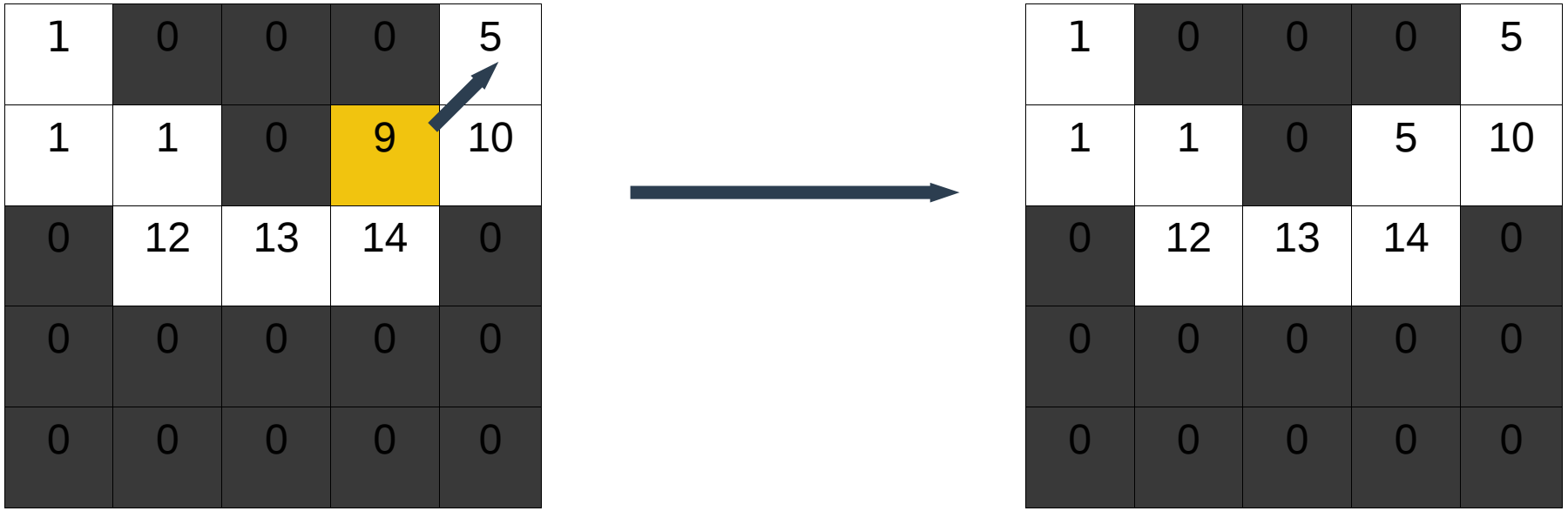


We see labels[7] = 7 and labels[1] = 1, so no traversal is needed.

Then labels[7] is set to 1. We also redundantly compare the top-left corner node with labels[7], though since labels[7] is now 1 there is nothing to do.

Union-Find example

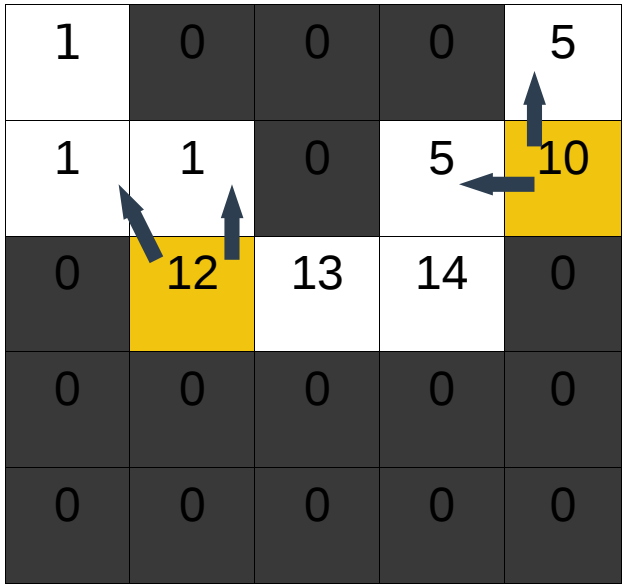
Union(9,5)

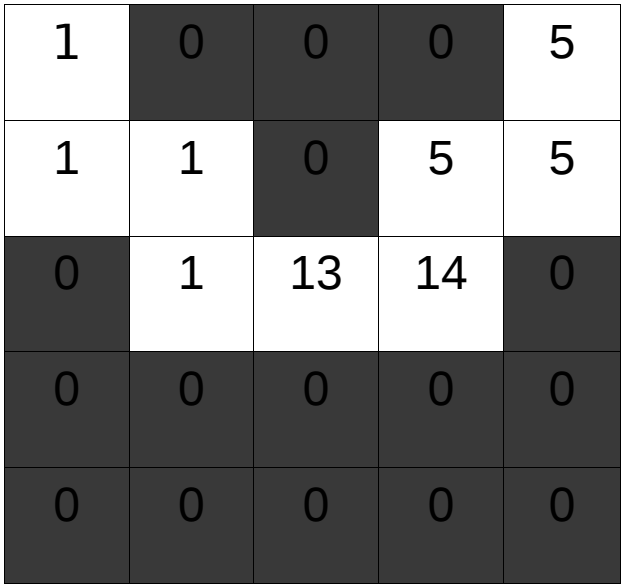


Similarly...

Union-Find example

Union(...,...)

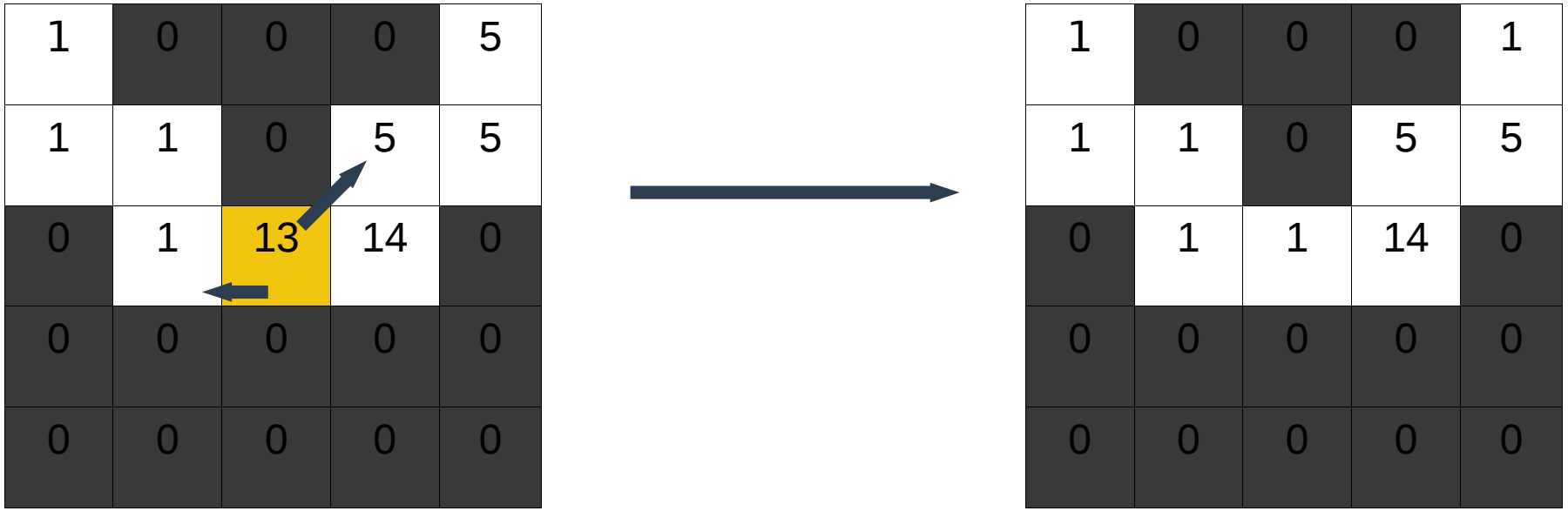




We repeat this process for nodes 10 and 12.

Union-Find example

Union(13,1); Union(1,5)

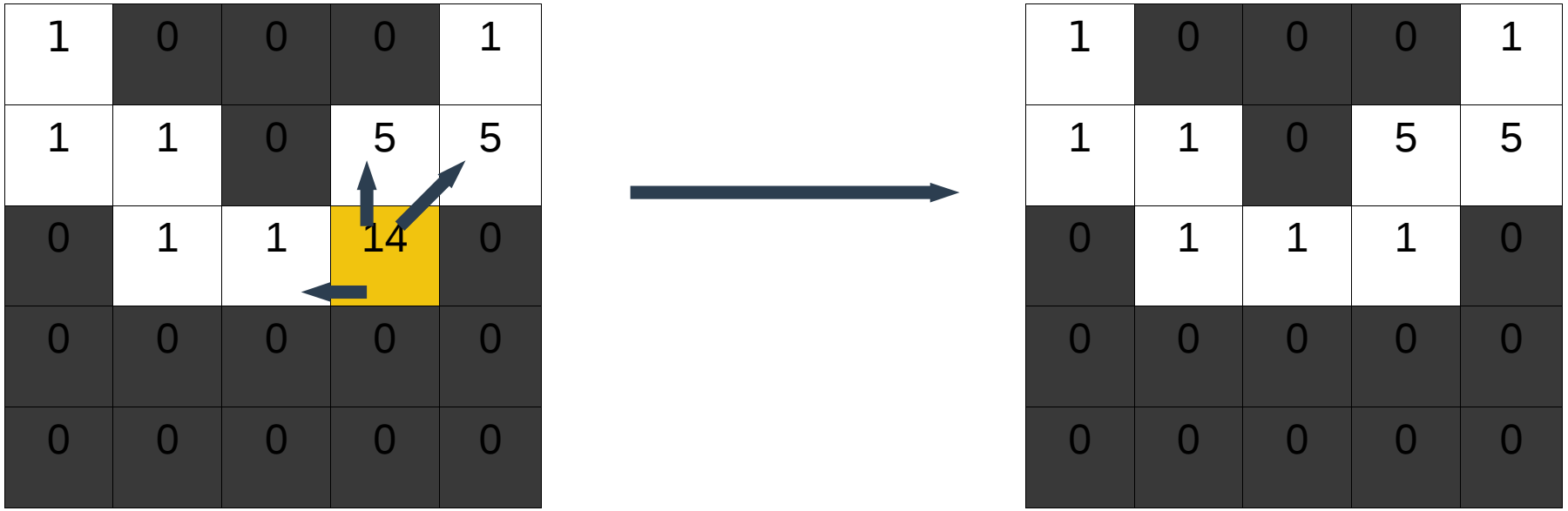


Here, after we have done union(1,13) (which sets labels[13] to 1), we will do union(1,5). Note that “labels[Max(1,5)] is set to labels[min(1,5)]” this means labels[5] (in the top right corner) will be set 1, not the 5 (labels[9]) next to labels[13].

This merges the regions (sets) 5 and 1.

Union-Find example

Union(14,1); Union(1,5); Union(1,5)



Finally, we set 14.

Union-Find example

1	0	0	0	1
1	1	0	5	5
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

1	0	0	0	1
1	1	0	1	1
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

After scanning, we do one more pass to retrieve all the final labels, by calling find() on each node.

The resulting image has one connected region.

Serial Implementation

- For all our programs, we load an image and binarize it (by thresholding) then feed it into the algorithm.
- In our serial algorithm, we do the following steps:

    1) We create a Union-Find object with as many elements as we have pixels.

    2) **First pass** – scan each pixel across the rows, top-to-bottom and merge each with their neighbours' labels from the forward scan mask.

    3) **Second pass** – for each pixel, set its label to that of the root node to get the final labeling.

    We colourise each pixel according to its label (thus giving independent regions distinct colors).

CUDA Implementation

- In our CUDA program:

  1) Copy the binarized image into a CUDA array and bind it to a surface –-- essentially a texture with write abilities. We use the surface because of its spatial locality speedup property when we do neighbor checking.

  2) We initialize one thread for each pixel in 8x8 blocks.

  3) **[Initial Labeling]** Each thread labels its pixel according to its unique thread id. Pixels with value 0 are not labeled or processed in any step.

CUDA Implementation contd.

4) **[Scan]** Each thread looks at its forward pass neighbors. The thread's label is then set to the minimum of it and its neighbors labels.
The label result of this may vary depending on how quickly individual threads can find the minimum, and set their own label (before other threads check it).

5) **[Analysis]** Each thread then uses its label as an index to recursively iterate through to its root label (region label) using the Union-Find propagation method.

CUDA Implementation contd.

6) Loop the following steps until no changes occur in the labels:

I) **[Link]** Each thread analyses its forward pass neighbors . The thread sets the pixel pointed to by its label (not itself, i.e set labels[labels[this_thread]]) to the minimum label of it and its neighbors' labels.

II) **[Relabel]** Each thread then takes the label pointed to by its own current label (labels[labels[this_thread]]), and sets it as its new label (other threads might have contributed a better minimum to the pointed label after the link phase)

CUDA Implementation contd.

III) **[Rescan]** Each thread compares the minimum of its forward pass neighbors. If the minimum of these neighbors' labels is not equal to the thread's own label, mark that we need to do another loop.

We stop looping when all the pixels have neighbors with the same label.

7) Copy the image from the GPU back to the host and attain our result.

CUDA Implementation notes

- Although thread contention is likely to occur when checking and setting neighbors, thread interleaving hides this latency.
- We also implemented a version which makes use of global memory (which is slightly faster).
- We did not make use of shared memory, as memory accesses are not block-local (threads can directly set their “root” nodes, potentially far away.)
- We also did not make use of constant memory, as all of the data changes quite rapidly, the data is potentially too big, and the amount of data we process varies in size greatly.

MPI Implementation

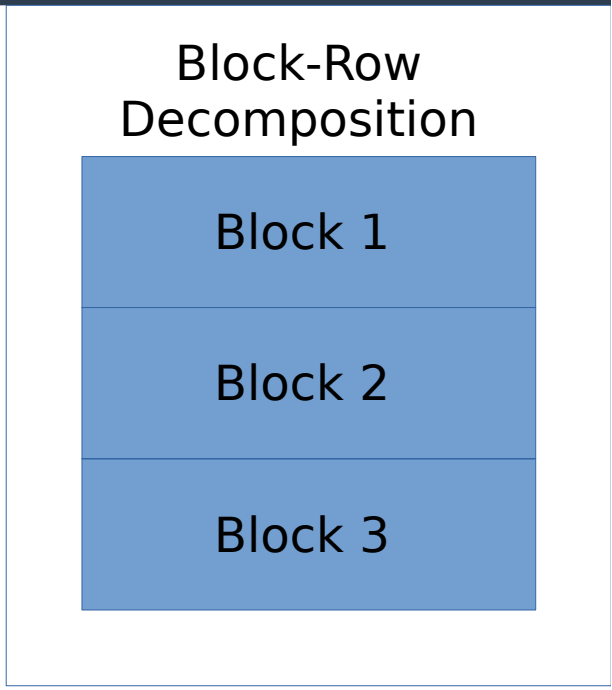
- In our MPI program for p processes:

  0) Load the image on the master process (rank 0).

  1) Use **block-row distribution**:

  – Split the binarized image into n/p sized blocks (n is #pixels).

  – Distribute the rows with MPI_Scatterv().



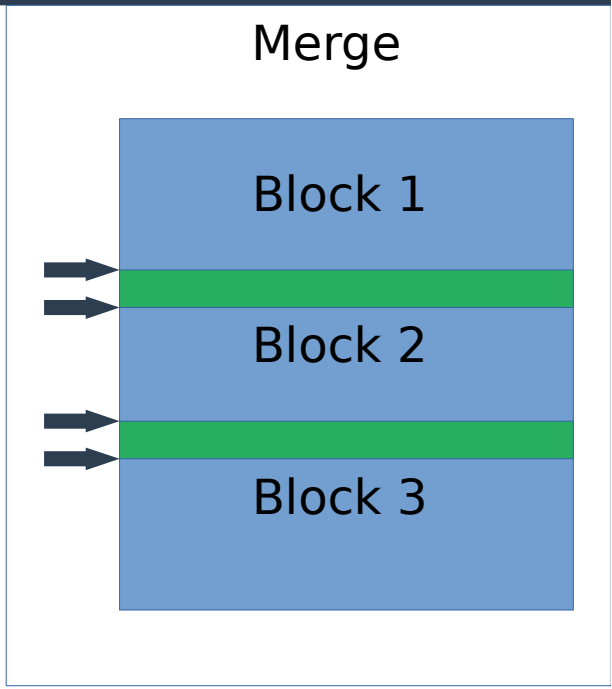
2) Each process then performs serial labeling algorithm on its block, finding and labeling the local regions.

3) The blocks are sent back to the master process with MPI_Gatherv().

4) The master process merges the regions along the boundaries between the blocks (first pass) and then relabels the whole image (second pass).

MPI Implementation notes

- Although it is possible to perform the merging of the regions in parallel using a binary tree method, it would entail additional communication costs and redundant work. The relatively small sizes of the images means the startup cost of communicating would outweigh any advantage of doing the merge in parallel. Also, the master process always has to do the final merge and global relabeling, making parallel relabeling redundant



- We do more than two passes on the image in MPI, however:

  I) The first two passes are done in parallel (for the local blocks).

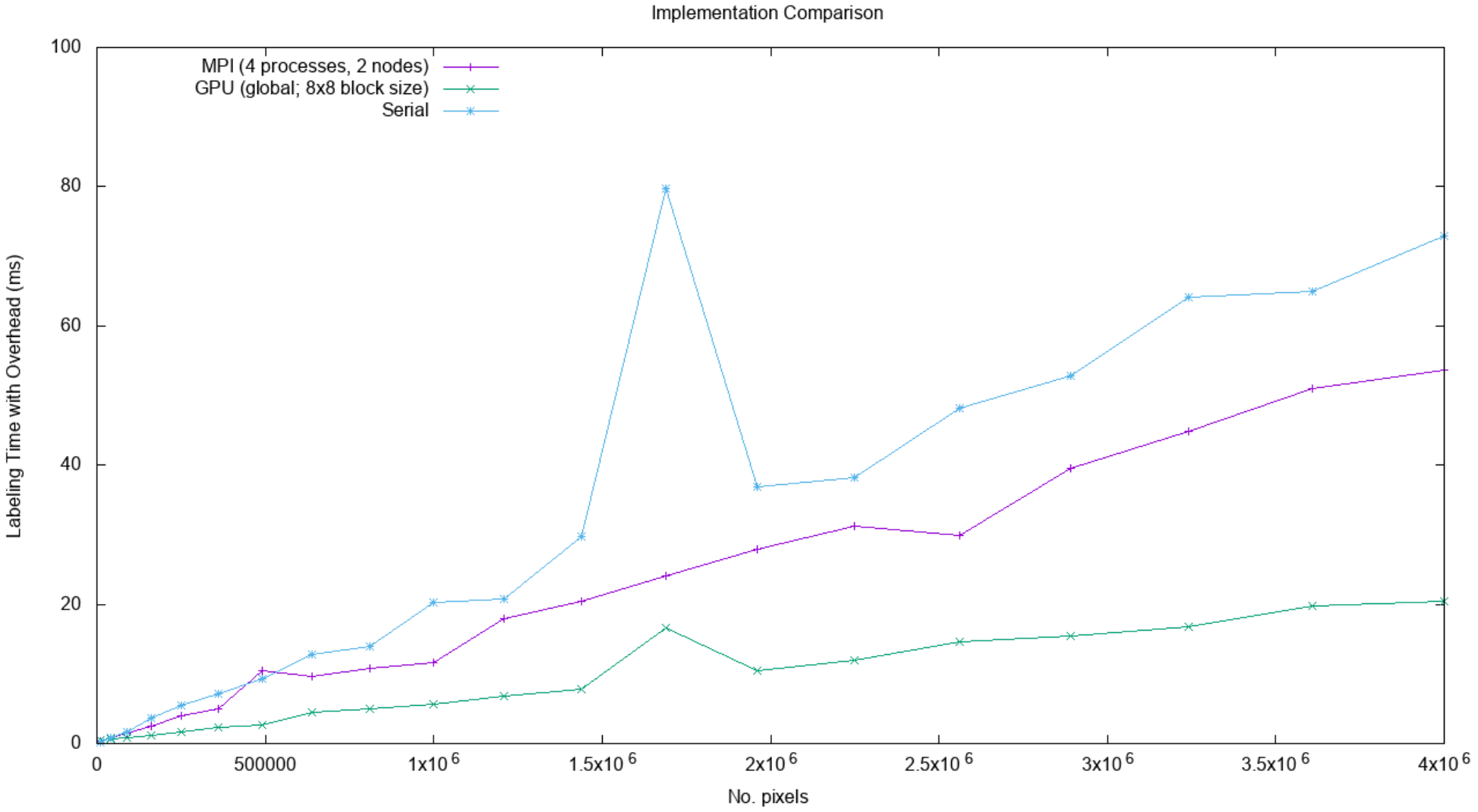
  II) The scan step of the final merge only requires the boundary rows of the chunks. There are p – 1 boundaries so this step is fast.

Demo...

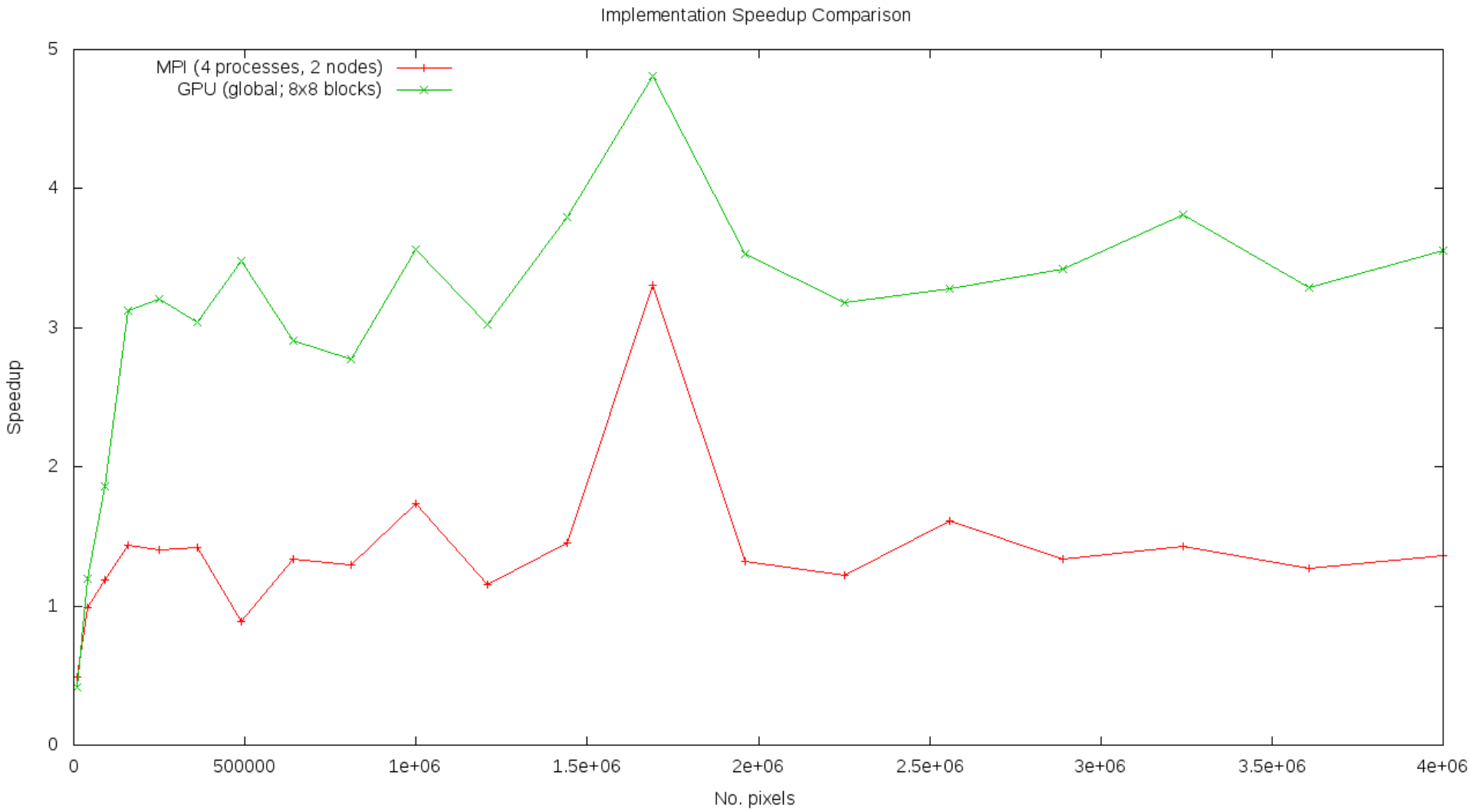
Example program execution

- ./ccl_serial -f f0001.bmp

Time Comparison



Speedup Comparison



Speedup example

- Speedup for the various algorithms on the hard-maze challenge image:

Implementation	Time (ms)	Speedup
Serial	5.27	1
CUDA (surface)	2.42	2.1
CUDA (global)	2.04	2.5
MPI	6.09	0.8