Connected-Components Labeling in Java

Antonio Tirone

December 20, 2018

Definition of the problem

The connected-components labeling problem consists in assigning a unique label to every connected (8-neighbours connection) group of foreground pixels. To validate the efficiency of the solution, tests have been run on random matrices, and on images translated to bitmap matrix thanks to a threshold application[1].

The Algorithm

The chosen algorithm, has been taken from The connected-component labeling problem: A review of state-of-the-art algorithms, section 3.3.1, and described in detail in Optimizing two-pass connected-component labeling algorithms. With this algorithm a solution is provided in time $O(N^2)$, as the results in the next section confirm. This algorithm uses a union-find structure to store the temporary labels, in order to quickly solve equivalences.
The algorithms is of the 2-scan type, meaning that 2 scans on the matrix are needed. In the first one temporary labels are assigned to every foreground object met: following the described order the neighbours in the mask are examinated, and a proper label is assigned to the current pixel.
Follows a flattening procedure to get a continuos sequence of labels and which according to the authors of the OCL algorithm reduces the computation time. In tests section is shown that the time taken from this procedure is not significant overall.
The second step simply assigns to every foreground pixel the label corresponding to its father's value (note that thanks to the flattening procedure the father is the root).

The main problem of this implementation is its memory usage: since an additional union-find structure is needed, in particular $O(NxN/4)$, the max number of different labels. It would have been possible to allocate memory just when it is needed, implementing a specific union-find datatype, but for efficiency reasons has been chosen to use an array of int instead. This leads to

[1]The threshold application has been taken from `https://introcs.cs.princeton.edu/java/31datatype/`

the allocation of the maximum possibly consumed memory since the beginning of the execution.

Parallel Implementation

To make the algorithm work in parallel, a simple master-slave approach was chosen, dividing the matrix in rectangular submatrices, and assigning each of them to one thread. This division is not an actual division of the matrix in such submatrices, but only a choice of arguments to pass to the threads, to make them work only on the desired section. Two types of threads are defined, to work in two different steps of resolution: CCLThread is used to solve labeling in the local matrix.
Then the master process unifies the results of threads. This section of code shall be executed in sequential order: if more threads were to access the same union-find structure synchronization would be required, slowing down the whole execution. Then a second step executed in parallel is the assignment of the root label to every point in the matrix, done by `{LabelSolverThread}`. A test run on my laptop with 4 cores provided the following results for times relatives to the various part of the algorithm.[2]

```
#nThreads   #workload    # time   #speedup   #efficiency
 Sequential time = 1737
 parallel: 1022, Submatrices joining: 0, flattening: 25, parallel: 470
1    10002x10002    1537    1.130        1.1301236174365648
 parallel: 592, Submatrices joining: 1, flattening: 27, parallel: 236
2    10002x10002    856     2.029        1.0146028037383177
 parallel: 515, Submatrices joining: 1, flattening: 25, parallel: 153
3    10002x10002    694     2.502        0.8342939481268011
 parallel: 396, Submatrices joining: 1, flattening: 25, parallel: 116
4    10002x10002    538     3.228        0.8071561338289963
```

From this test we can see that almost all the computation time is spent in parallel execution. We can also see that the algotithm for the sequential solution is not optimal, so from now on we will refer to the 1 thread exection of the parallel algorithm as sequential execution.

Test cases

Aside from performance tests, two other kind of tests have been performed. A correctness test, which checks that the output arrays are the same for every number of threads. And a visual test, which shows graphically the labeling.

To report significant results two different kind of tests were run: execution on pictures of various dimensions, and execution on randomly generated bitmaps.

[2] Now and later on the results are truncated at the third decimal digit for easiness of reading

For random generated bitmaps have been executed tests with a foreground objects density of 0.3 and 0.6, for square matrices of dimensions 1000, 10000, 20000. On my personal laptop only results up to 10000 are provided, due to heap memory limitations.

Personal laptop

The system has 4 cores, Intel i7 7th generation cores.

Alpha University supercomputer

Tests run on the university cluster, with 12 processors

The first test compares the computation on random square matrices of size 1000x1000, 10000x10000, 20000x20000, having density of 0.3. The results for the the first matrix don't seem accurate, showing a non linear behavior. The other two instead give similar results, increasing linearly until a speedup of almost 5. The efficiency otherwise goes as down as 0.4.

The second test is the same as the first, but with density 0.6. Again the first matrix doesn't produce valuable results. For the others, the speedup is better then for the test with low density, going as up as almost 9. Efficiency for these matrices doesn't go lower then 0.7.

Third test is run on images of different sizes, 502x377 7954x4813 10918x9987 10342x16907, with threshold 100. Similarly to the previous tests the result for the smallest input are not significant, but is interesting to note that pic10 (10918x9987) provides better speedup then the biggest image. This is probably due to the density of foreground pixels in the images.

The fourth test is run on the same pictures as test 3, but with a different value of threshold, 200. This leads to a minor number of foreground pixels, and accordingly to this the performance are worsened (confirming observation made on random matrices). It has to be noted that speedup increases until 8 threads, and then starts to remain stable, assesting at a factor between 5 and 6.

Figure 1: First test, speedup on random matrix with density = 0.3

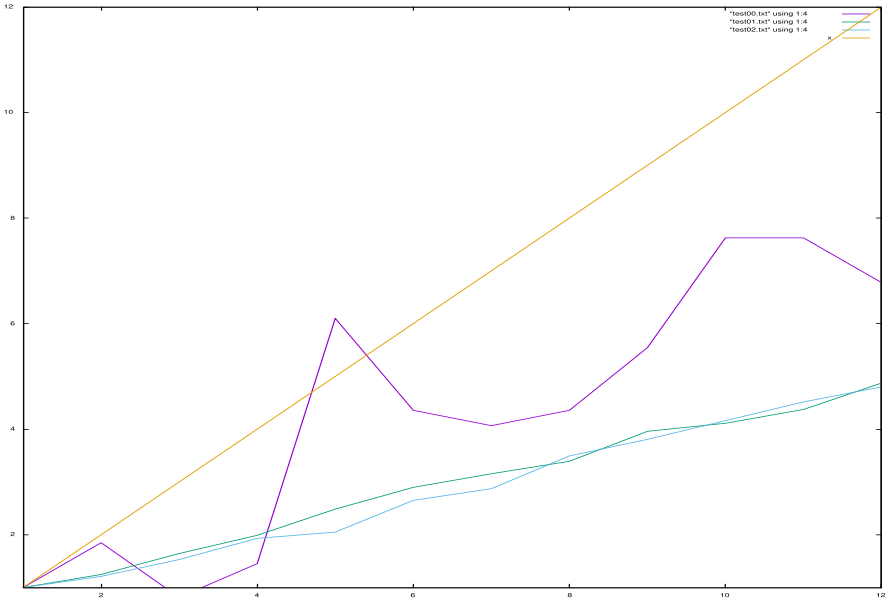


Figure 2: First test, efficiency on random matrix with density = 0.3

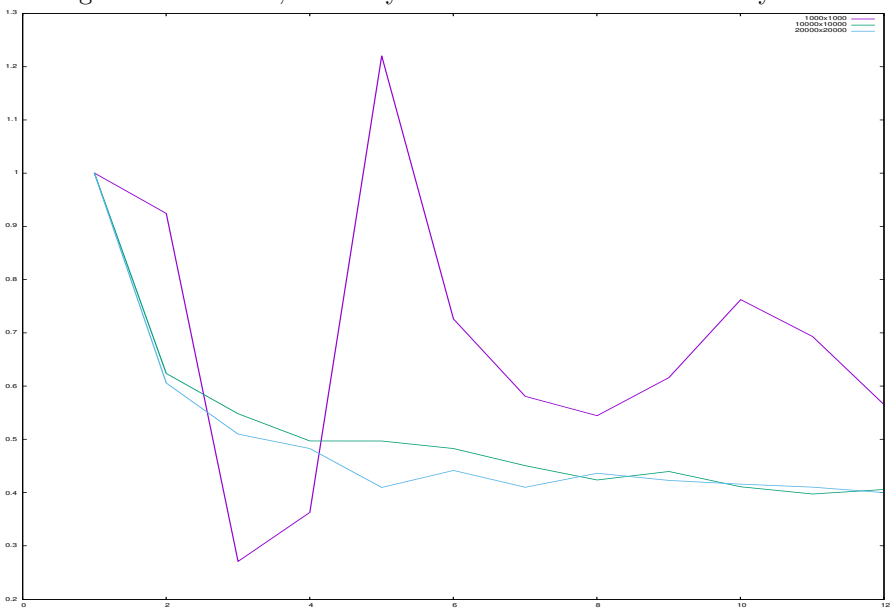


Figure 3: Second test, speedup on random matrix with density = 0.6

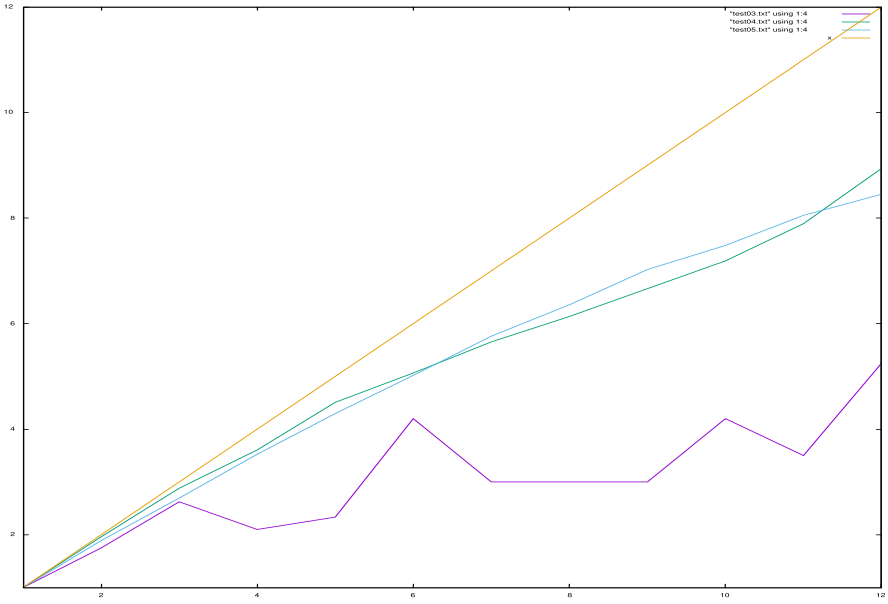


Figure 4: Second test, efficiency on random matrix with density = 0.6

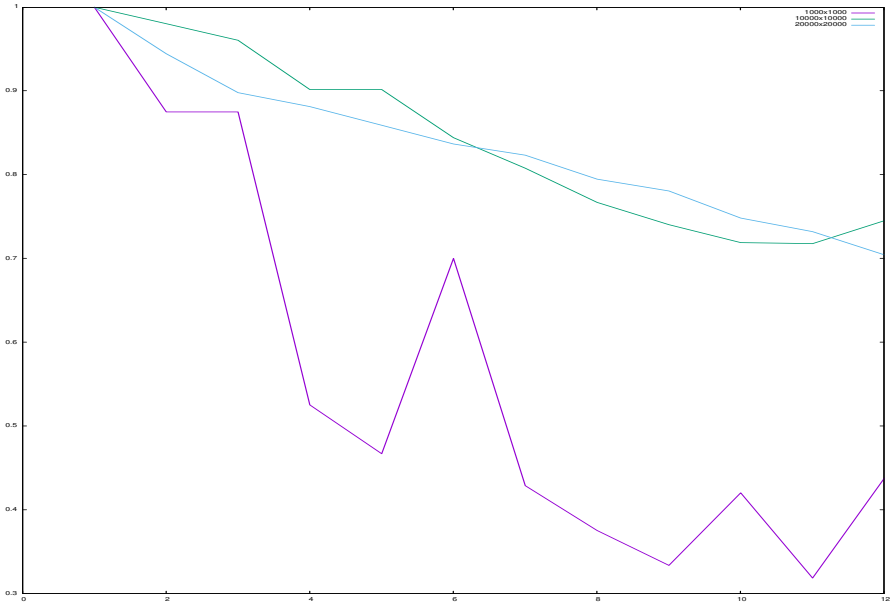


Figure 5: Third test, speedup on pictures processing with threshold 100

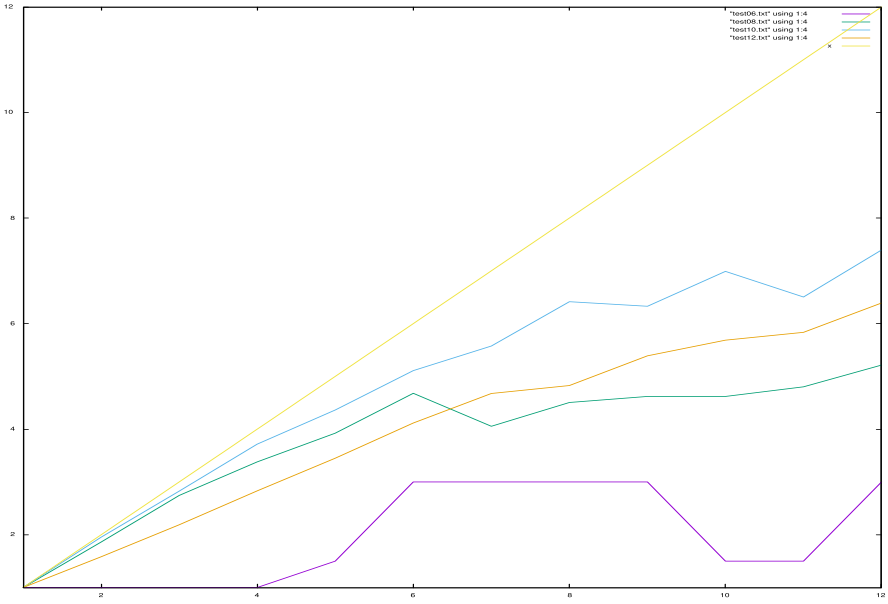


Figure 6: Third test, efficiency on pictures processing with threshold 100

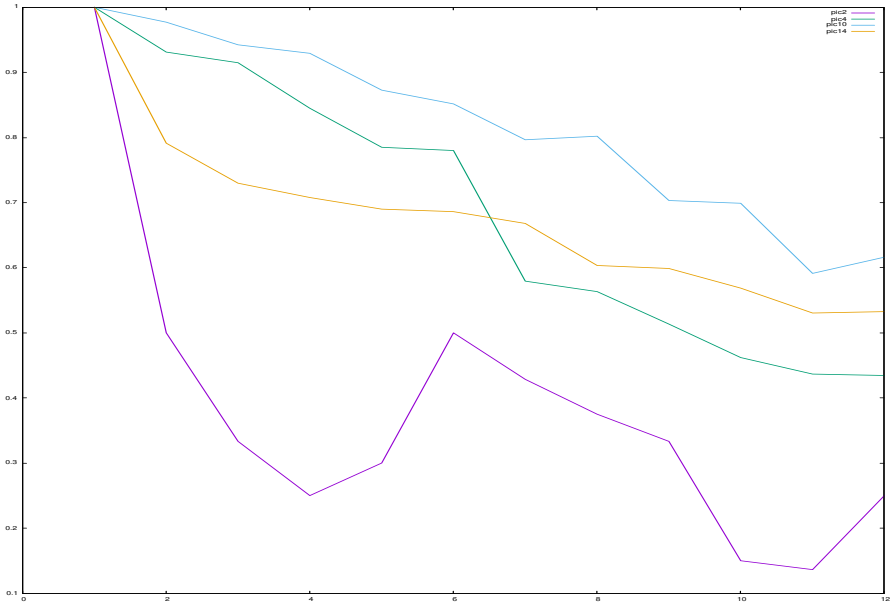


Figure 7: Fourth test, speedup on pictures processing with threshold 200

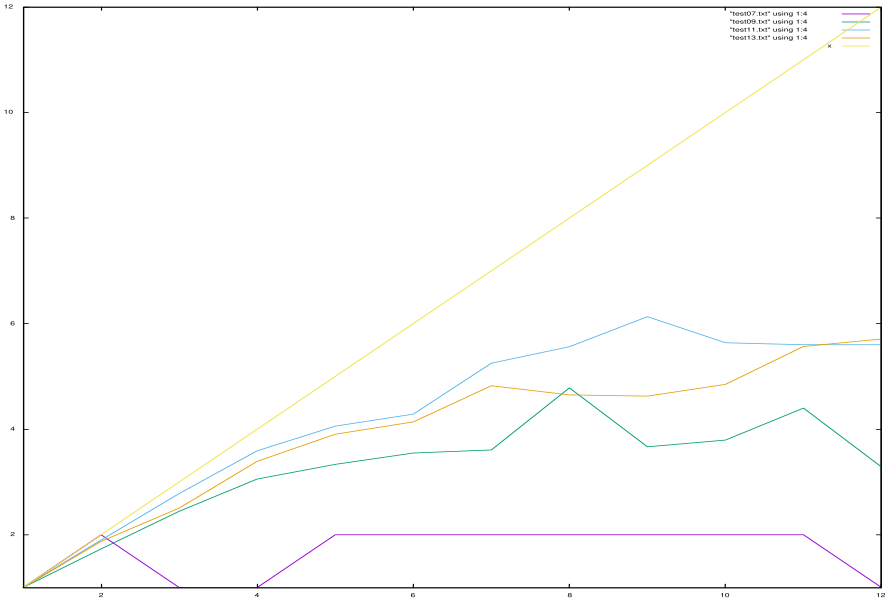
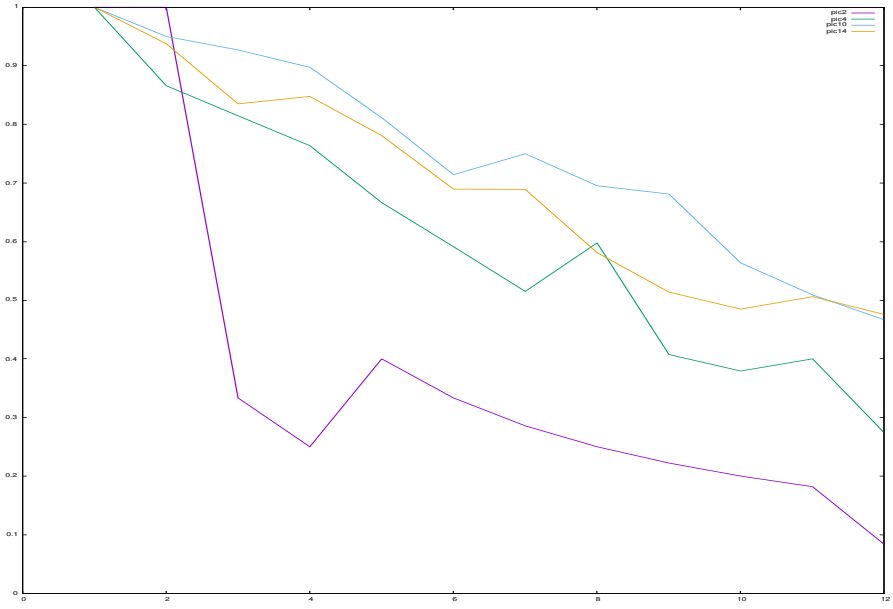


Figure 8: Fourth test, efficiency on pictures processing with threshold 200



Conclusions

As the results of the tests point out, a significant speedUp factor can be achieved thanks to parallelization. The efficiency factor is not high, decreasing inversally to the number of processors used for computation.

For little input, in the order of $N = 1000$ the parallelization doesn't prove itself to be useful. As the matrix size goes up, a greater speedUp factor can be achieved.

It is also interesting to note that for random generated matrix the speedup factor has a continuos increase, while for image processing a limit to speedup is reached. For higher density of foreground pixels (density of 0.6 and threashold of 100) better speedup factors are achieved.

Also, the $O(N^2)$ time complexity is confirmed. This can be seen in the results, and is very evident in the computational time for the random matrices: going from 10000 to 20000 the time increases of 4 times, as to be expected.