COMP 429/529:
Parallel Programming
Assignment 1

Project (Image Segmentation)
Report

Group members:

Mustafa Saraç

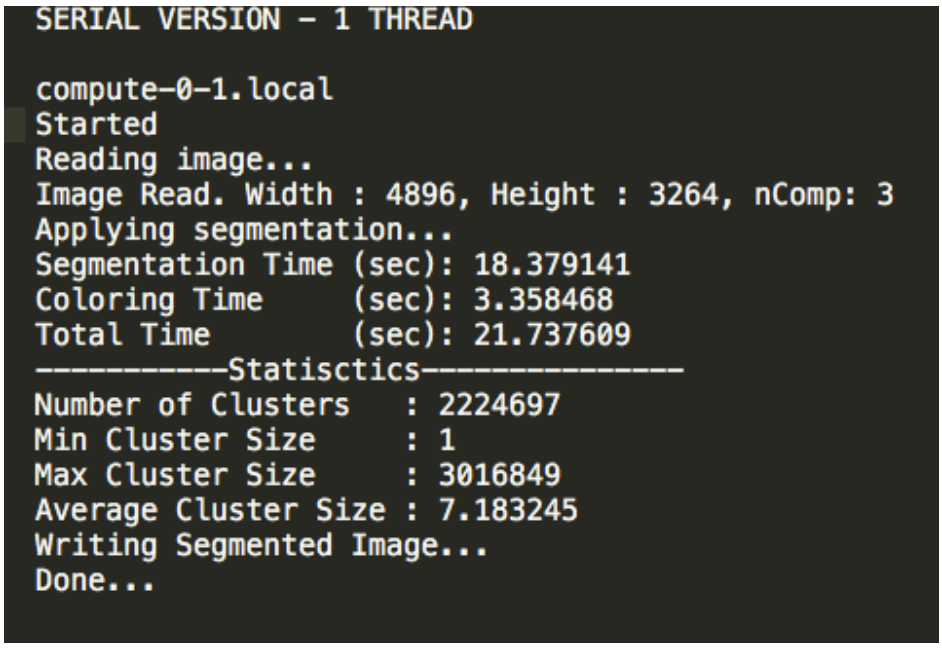
Mustafa Mert Ögetürk

Introduction:

In the image segmentation project, the algorithm first read the image and collect the data of each pixel. After that, all the data are labeled and then some pixels share the same label to determine the segments. After completing the segmentation, all labels are colored regarding to their segments then the simplified version of the picture is obtained. Image segmentation algorithm is a good choice to make it parallelized because of the process that I mentioned about it. It has different big data and their structures, and different works that can be done at the same time. So, we encountered different kind of difficulties in this project. Solutions will be detailed in related titles for each part. We completed all parts of the project and achieved considerable speed-ups in this project.

Part-I Parallel Segmentation

In this part, loop nest-1 first passes over the whole image and gathers all the pixel labels in a global linear matrix array namely “labels”. Then, it operates by finding neighbor pixels with better labels. When we execute the code serially in Lufer cluster, performance test results shown in the following figure-1:



In the case that both of the “for” loops are working as parallel, the loop with j index results a data waste, because all of the threads try to reach many different same memory addresses and this causes so many page faults. For example, when we get an address of an index in the array, it comes with some adjacent memories of another array elements by the cache working logic, these additional memories become unused in so many loop steps while working as parallel. Therefore, this affects the performance in a bad way.

On the other hand, making the other “for” loop with i index parallel allows us to divide the image into stripes as it is described in the assignment instructions. In this way, we partition the whole data into sub-regions. This sub-regions work as parallel but they work serial in their region. We think that, in the junction stripes of these sub-regions, some race condition situations can be occurred technically but these are not harmful because we work with millions of pixels so these rare race conditions on a single junction stripe do not affect the result. In addition, when we compare the serial and parallel outputs of the algorithm, this technical assumption was not observable.

To achieve a considerable speed-up with OpenMP, putting a pragma just for the outmost “for” loop was essential. It is shown below for the loop nest-1:

```
//LOOP NEST 1
// first pass over the image: Find neighbors with better labels.
#pragma omp parallel for num_threads(numThreads)
        for (int i = height - 1; i >= 0; i--) {
                for (int j = width - 1; j >= 0; j--) {  ...
```

In the loop nest-2, the algorithm passes over the whole data and searches for a label that is updated and then propagates the updated label to determine the segments. By this way, the image segmentation process is completed. This process goes on the stripes again. Therefore, the solution and parallelism idea was the same as we explained above in the loop nest-1 part. We did not make the inside loop as parallel because of the same reason. We inserted a pragma command just before the outmost loop in order to get a high performance segmentation.

//LOOP NEST 2

// second pass on the labels, propagates the updated label of the parent to the children.

#pragma omp parallel for num_threads(numThreads)

```
for (int i = height - 1; i >= 0; i--) {
    for (int j = width - 1; j >= 0; j--) {  ...
```

When parallelism is completed, we examined the segmentation time of the algorithm for several times with different thread numbers such as 1, 2,4,8,16,32. Segmentation time decreased obviously, by increasing the thread numbers.

```
compute-1-9.local
Started
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 6.902352
```

(With 1 thread)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 6.413860
```

(With 2 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.725939
```

(With 4 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.549750
```

(With 8 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.009473
```

(With 16 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 3.943964
```
(With 32 threads)

To sum up, we achieved an almost 4.7x speed up with executing between 1 and 32 threads. Even though, technically segmentation time must decrease by increasing the thread numbers, sometimes it might be possible to record some exceptions.

Part-II Parallel Coloring

After the segmentation process, the aim of this part was to assign colors to labels and then assigning these label colors to pixels. In loop nest-3, the serial implementation, takes all the unique label numbers of each pixel and assign to them a random color in the hash map. First, we came up with an idea which is dividing work between threads based on label numbers. Then, we realized that labels are getting unique numbers and they don't increase in order. So, dividing work between these numbers will not provide a correct result, because we cannot handle the distribution of the threads to the label numbers. Then, we decided to divide the work regarding to the index numbers of the array. To do that we tried the implementation below;

```
//LOOP NEST-3
#pragma omp parallel
{
int tid = omp_get_thread_num();
 int tnum = omp_get_num_threads();
 int totalWork = width * height;
 int labelWork = totalWork / tnum;
 #pragma omp critical for // we add this line after getting a segmentation fault,
                          but it decreases the efficiency, it is detailed below.
(int i = tid * labelWork; i < (tid * labelWork) + labelWork; i++) {
int label = labels[i];
```

```
 if (label == 0) //background {
red[0] = 0;
green[0] = 0;
blue[0] = 0;
 } else {
if (red.find(label) == red.end()) {
clusters++;
count[label] = 1;
 red[label] = (int) random() * 255;
 green[label] = (int) random() * 255;
blue[label] = (int) random() * 255;
 } else count[label]++;  } } }
```

Before we had inserted this “critical” command into the implementation, we were trying to make this “for” loop parallel but it resulted with a segmentation fault. According to our research, reason of the segmentation fault that we faced in this part is because more than one thread tried to make an operation on the same memory address simultaneously. In order to solve this issue, we thought that using “critical” command in this part will not allow more than one thread to have a chance to operate on the same memory address. Therefore, by using this “critical” command, we had solved the issue. However, even though we had solved this problem, we could not reach our goal to have a high performance on this part because “critical” command actually makes algorithm work as serial. When “critical” is used in a “for” loop, it means that all of the threads will work on this task one by one. Therefore, this change did not give us a speedup. Moreover, the operating order of a thread among all threads might differ at each run, so the output image file might have a different color dispersion. This discrepancy indicates that threads did not work well and using “critical” is not a good way to implement in this part. Therefore, we realized that trying to write into the same memory address of a hash map from more than one thread is a crucial error for a parallel algorithm. So, executing serially was the best choice to get the highest performance.

In loop nest-4, this time we read the data from the hash map, when we try to make it parallel we did not encounter a segmentation fault because reading data do not cause a data dependency. On the other hand, after reading the data the algorithm operates a writing but writing part has a data dependency. Therefore, we decided to make on loop nest-4 parallel and to execute loop nest-3 as serial. Implementation for loop nest-4 is shown below:

```
//LOOP NEST 4
  #pragma omp parallel for num_threads(numThreads)
      for (int i = 0; i < height; i++) {
       for (int j = 0; j < width; j++) {

         int label = labels[i*width+j];
         seg_data[(i*width+j)*3+0] = (char)red[label];
         seg_data[(i*width+j)*3+1] = (char)blue[label];
         seg_data[(i*width+j)*3+2] = (char)green[label];
       }
      }

      for ( auto it = count.begin(); it != count.end(); ++it )
       {
         min_cluster = std::min( min_cluster, it->second);
         max_cluster = std::max( max_cluster, it->second);
         avg_cluster += it->second;
```

When parallelism is completed, we examined the coloring time of the algorithm for several times with different thread numbers such as 1, 2,4,8,16,32. Coloring time decreased obviously, by increasing the thread numbers.

```
compute-1-9.local
Started
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 6.902352
Coloring Time     (sec): 2.726328
Total Time        (sec): 9.628680
```
(With 1 thread)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 6.413860
Coloring Time     (sec): 2.436157
Total Time        (sec): 8.850017
```
(With 2 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.725939
Coloring Time     (sec): 2.496771
Total Time        (sec): 7.222710
```
(With 4 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.549750
Coloring Time     (sec): 2.477648
Total Time        (sec): 7.027398
```
(With 8 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.009473
Coloring Time     (sec): 2.277492
Total Time        (sec): 6.286965
```
(With 16 threads)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 3.943964
Coloring Time     (sec): 2.237942
Total Time        (sec): 6.181906
```
(With 32 threads)

To sum up, we achieved an almost 1.5x speed up with executing between 1 and 32 threads. Even though, technically segmentation time must decrease by increasing the thread numbers, sometimes it might be possible to record some exceptions.

Part-III Optimization

In this part, we tried to get rid of unnecessary operations in the algorithm. The "for" loop that controls the phases has log2 (n) phases which is the worst case. So, every time the algorithm checks for whole data and do the labeling and updating operations, many of them was unnecessary because it does not work in the worst case scenario every time. When it completes the task, it continues to work unnecessarily. Therefore, we added a Boolean expression which is called correction into the phases for loop. This Boolean variable is true initially, whenever labeling statements apply an operation on the labels, this variable turns to false and this indicates that process is uncomplete, then phases loop continues to work. On the other hand if the labeling operations don't do any change, correction Boolean will stay as true, which shows the process is complete and for loop breaks. Briefly, this optimization has a considerable impact on performance improvement. This is implemented as shown below:

```
void imageSegmentation(int *labels, unsigned char *data, int width, int height, int pixelWidth, int Threshold) {
 int maxN = std::max(width,height);
int phases = (int) ceil(log(maxN)/log(2)) + 1;
for(int pp = 0; pp <= phases; pp++) {
bool correction = true;
……..
if (ll < labels[idx]) {
 labels[ll - 1] = std::max(labels[idx],labels[ll - 1]);
correction = false;
```

```
        }
  } }
   //LOOP NEST 2
 // Second pass on the labels. propagates the updated label of the parent to the children.
#pragma omp parallel for num_threads(numThreads)
          for (int i = 0; i < height; i++) {
           for (int j = 0; j < width; j++) {
                    int idx = i * width + j;
if (labels[idx] != 0) {
labels[idx] = std::max(labels[idx], labels[labels[idx] - 1]);
// subtract 1 from pixel's label to convert it to array index
                                          }
                                    }}
                    if(correction) break;
}
```

First, we executed the parallel code without optimization and then did the same execution with optimization. We achieved a positive performance improvements up to 4 threads but after that, by increasing the thread numbers it became hard to observe the speed-ups.

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 11.376405
Coloring Time     (sec): 3.333183
Total Time        (sec): 14.709588
```

(With 4threads)

Before the optimization

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.725939
Coloring Time     (sec): 2.496771
Total Time        (sec): 7.222710
```
(With 4 threads)

After the optimization

It corresponds to almost 2x speed-up.

Part-IV TBB

In this part, we converted the OpenMP implementation to the TBB implementation directly. The logic behind all the parts was exactly same. The test results are shown in below:

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 9.047058
Coloring Time     (sec): 2.793162
Total Time        (sec): 11.840220
----------Statisctics--------------
```
(With 1 thread in TBB)

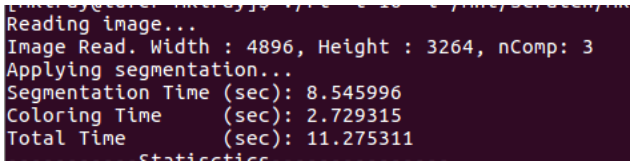
```
compute-1-9.local
Started
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 6.902352
Coloring Time     (sec): 2.726328
Total Time        (sec): 9.628680
```
(With 1 thread in OpenMP)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 8.662660
Coloring Time     (sec): 2.725380
Total Time        (sec): 11.388040
----------Statisctics--------------
```
(With 2 thread in TBB)

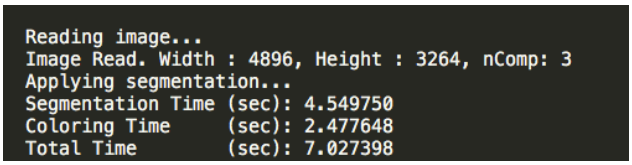
```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 6.413860
Coloring Time     (sec): 2.436157
Total Time        (sec): 8.850017
```
(With 2 thread in OpenMP)

```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 8.587156
Coloring Time     (sec): 2.727560
Total Time        (sec): 11.314716
----------Statisctics--------------
```
(With 4 thread in TBB)

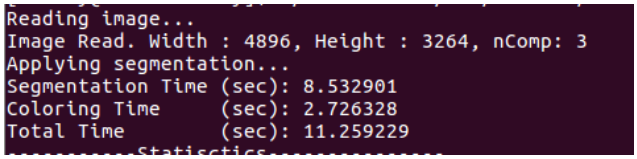
```
Reading image...
Image Read. Width : 4896, Height : 3264, nComp: 3
Applying segmentation...
Segmentation Time (sec): 4.725939
Coloring Time     (sec): 2.496771
Total Time        (sec): 7.222710
```
(With 4 thread in OpenMP)



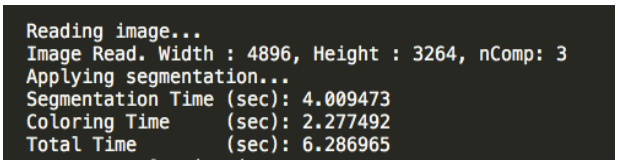
(With 8 thread in TBB)



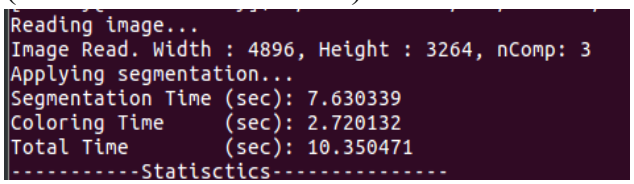
(With 8 thread in OpenMP)



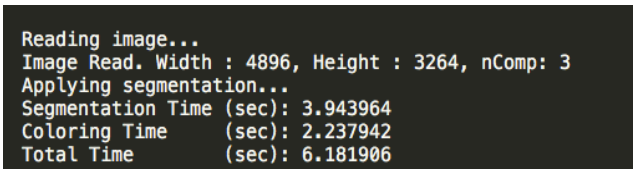
(With 16 thread in TBB)



(With 16 thread in OpenMP)



(With 32 thread in TBB)



(With 32 thread in OpenMP)

You can find the converted implementation for all parts below:

Part-I:

```
            //LOOP NEST 2
            // Second pass on the labels. propagates the updated label of the parent to the children.


tbb::parallel_for( tbb::blocked_range<int>(0, height),
 [&]( const tbb::blocked_range<int> &r ) {
  for(int i = r.begin(), i_end = r.end(); i < i_end; i++){
          for (int j = 0; j < width; j++) {
                  int idx = i*width + j;
                  if (labels[idx] != 0) {
                    labels[idx] = std::max(labels[idx], labels[labels[idx] - 1]);
                          }
                  }
        }
}
```

Part-2

```
tbb::parallel_for( tbb::blocked_range<int>(0,height),
[&]( const tbb::blocked_range<int> &r ) {
for(int i = r.begin(), i_end = r.end(); i < i_end; i++){
            for (int j = 0; j < width; j++) {
             int label = labels[i*width+j];
               seg_data[(i*width+j)*3+0] = (char)red[label];
               seg_data[(i*width+j)*3+1] = (char)blue[label];
               seg_data[(i*width+j)*3+2] = (char)green[label];
                        }
          }
}
```

Conclusion:

In this assignment, we had learned how to make a serial algorithm, which does image segmentation, parallel with most preferable frameworks OpenMP and TBB. In this learning process, we had experienced how to solve any issue related to parallel programming, such as data dependence and thread overhead. By using both of these frameworks inside the project, we had chance to examine the differences between them.