KTH Royal Institute of Technology

SF2568
Parallel computations for Large-Scale problems

Year 2018

3D Region Growing

Group report

Picó Oristrell Sandra 940531 – 2308 sandrapo@kth.se
Corominas Larsson Miquel Sven 920614 – 5998 miquell@kth.se

Professors :
Hanke Michael

May 7, 2018

Contents

1 Problem description

In this project we will compare a sequential and parallel implementation of the 3D Region Growing algorithm. The main objective is to reaffirm the importance of parallel programming when applied to meaningful data, and study how to parallelize a sequential algorithm in a correct way.

1.1 Region growing description

Image segmentation is *the process of partitioning an image into multiple segments.*[1] and region growing is a *region-based image segmentation method.*[2]

The main goal of this algorithm is to partition an image into multiple regions. The first step is to determine the initial seed point, from which the region growing algorithm will examine neighboring pixels and determine if they should be added to the region based on the selected inclusion criteria.

The inclusion criteria is normally a predefined threshold determined by the type of analyzed data, but there are other methods also. In this project we will use the threshold inclusion method.

A basic formulation of the output of this algorithm could be the following one:

$$(\mathbf{a}) \bigcup_{i=1}^{n} R_i = R \tag{1}$$

(a) The segmentation must be completed, every pixel must be in a region.

$$(\mathbf{b}) R_i \text{ is a connected region, } i = 1, 2, ..., n \tag{2}$$

(b) The points in a region have to be connected based on some inclusion criteria.

$$(\mathbf{c}) R_i \bigcap R_j = \emptyset \text{ for all } i = 1, 2, ...n \tag{3}$$

(c) Each region of the image must be disjoint.

$$(\mathbf{d}) \text{ InclusionCriteria}(R_i) = \text{ True for } i = 1, 2, ...n \tag{4}$$

(d) Property that have to satisfy all the pixels in a region, i.e threshold inclusion criteria.

$$(\mathbf{e}) \text{ InclusionCriteria}(R_i \bigcup R_j) = \text{ False for any adjacent region } R_i \text{ and } R_j \tag{5}$$

(e) The multiple adjacent regions of an image are different in the sense of the inclusion criteria.

1.2 Project approach

As said before, the main approach of this project is to compare the performance, in terms of speed-up of the sequential and the parallel region growing algorithm.

For this, we are going to execute the same test cases in the sequential code and in the parallel code and compare the execution time needed in each case.

To be able to compare the results obtained in a proper way we will use a dedicated server with the following characteristics:

Number of cores	8
Core specification	Intel(R) Xeon(R) CPU E5-1630 v4 @3.70GHz
RAM	16GB
Memory	200GB HDD

1.3 Data

In order to achieve proper comparisons between the parallel region growing approach and the sequential one, meaningful 3D test data must be created. We are going to test the two algorithms with two different figures. Each figure will be created with help of a python script.

We decided to create our own data figures in order to achieve specific test cases.

2 Algorithm description

In this section we will explain our implementation in a detailed manner. First, we are going to define the sequential region growing algorithm and later, we will explain our parallel approach. Furthermore, the code and implementation can be found in the attached zip file.

2.1 Sequential 3D region growing

2.1.1 Description

The first step to implement the algorithm mentioned above is to receive the file that contains the data that we have previously created with the python script. The data is basically a 3D figure with different values assigned to each coordinate. The values will determine if they belong to one region or another.

Then, although we are treating 3D data, we will be storing everything into a one dimensional array, for simplicity. In order to achieve so, we implemented the following two functions:

- **coord2index**: Convert three-dimensional coordinates into a one-dimensional index.

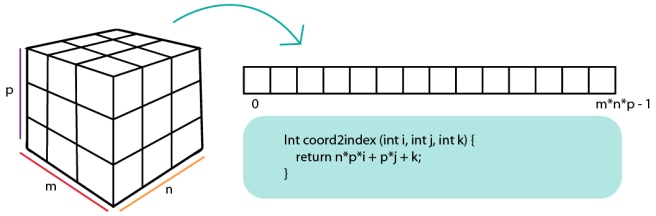

Figure 1: 3D coordinates to 1D index

- **index2coord**: Convert a one-dimensional index into three-dimensional coordinates.

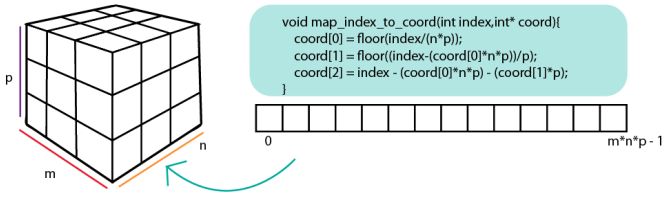

Figure 2: 1D index to 3D coordinates

Once we have all the image saved in the one-dimensional array, we create another array, where we will store the corresponding region number for each pixel in the image. So, at the beginning it will be an initialized to 0 and at the end we will have all the pixels assigned to a region number.

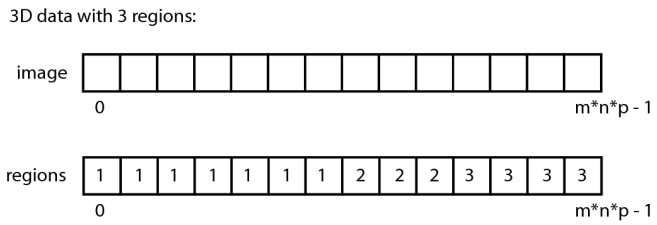

Figure 3: 1D index to 3D coordinates

This way, the growing region algorithm will be filling the regions array shown in the previous figure with the correspondent number. We will be calling the algorithm iteratively until the array is completely full; i.e without any 0 (unassigned coordinate). For each region, we will choose a new random seed. The seed point will be chosen randomly among all the pixels that have not yet been assigned to any region.

Therefore, the region growing algorithm will receive the regions array, the one dimensional image array and the seed point. In order to determine which points correspond to the same region as the seed, the algorithm must check the neighbours and only add those that verify the inclusion criteria. All neighbours will be stored in a stack.

2.1.2 Pseudo-code

In this section we will attach the pseudo-code of the sequential algorithm explained above.

Algorithm 1 Main function

```
1: procedure MAIN()
2:     image = ReadData(file);                          ▷ Read the image data from the file
3:     m,n,p = image_dimensions;                        ▷ Dimensions of the image
4:     int regions[m*n*p];                              ▷ Array to store the region value of each pixel of the image
5:     while all pixels not assigned do                 ▷ Region growing until all the pixels have a region assigned
6:         seed(x,y,z) = rand%((regions not assigned));  ▷ Select a seed point in the image
7:         region_grow(image,regions,m,n,p,seed);       ▷ Update regions array with the new pixels assigned
8:     file = WriteData(regions);                       ▷ Write the regions assigned in a file
```

Algorithm 2 Region growing

```
1: procedure REGION_GROW(IMAGE,REGIONS,M,N,P,SEED)
2:     region_num ++;                                   ▷ Increment the number of the region
3:     stack_create(needs_check);                       ▷ Create the stack structure to store the points to check
4:     addNeighbors(seed,needs_check,regions);          ▷ Add neighbors from the seed in the stack
5:     while elements in needs_check do                 ▷ Until the stack is empty
6:         point(x,y,z) = stack_pop(needs_check);       ▷ Pop one pixel
7:         if regions[point] != 0 then                  ▷ Check if the pixel is already assigned to a region
8:             continue;
9:         if check_inclusion(point,seed,image,m,n,p) == True then   ▷ If the pixel fulfill the inclusion criteria...
10:            regions[point] = region_num;             ▷ Assign the region to the point
11:            addNeighbors(point,needs_check,regions); ▷ Add neighbors from the point in the stack
12:                                                     ▷ Points from the same region assigned
```

2.2 Parallel 3D region growing

2.2.1 Description

In this section we are going to explain how our parallel region growing algorithm works. Nevertheless, the region growing itself will be exactly as it is implemented in the sequential algorithm.

The first step in making our algorithm parallel is to divide the data. We are going to divide the 3D image into a grid based on the number of processes selected by the user.

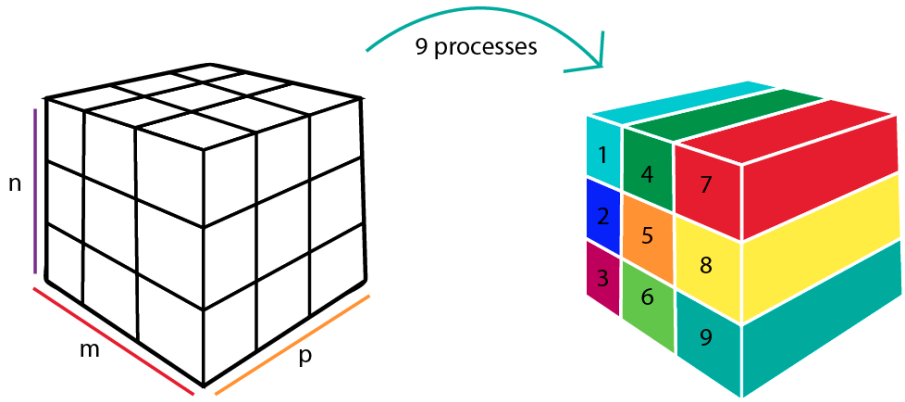

Figure 4: How the image is divided

This way, each process will have a small block of the original 3D image and then, each process will execute the sequential region growing with the assigned sub-image.

However, when all the processes have identified their own sub-regions, we have to check if the sub-regions form larger regions, i.e they are actually two contiguous sub-regions.

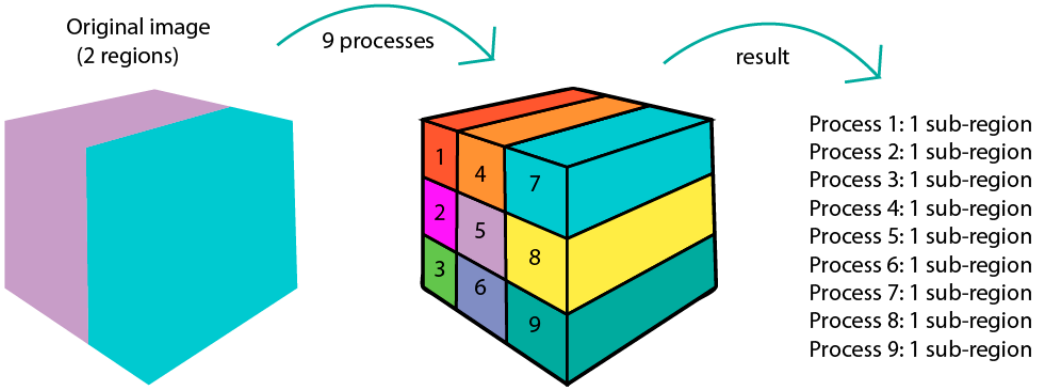

Figure 5: Sub-regions after the first step of the parallel approach

As can be seen in the previous image, as mentioned earlier, we need to look at whether the sub-regions are part of the same region. To solve this approach in a simple way, we decided to define the communication between processes as a two-dimensional grid. Specifically, using the MPI_Cart function. Then, in the case where we have 9 processes we will have the following MPI communication grid:

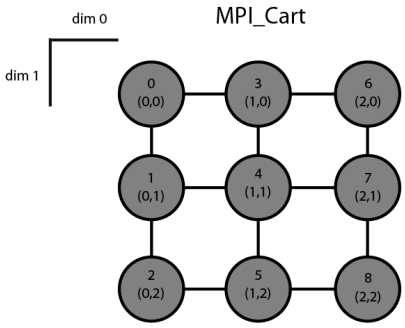

Figure 6: MPI_Cart communication

Finally, to know if the adjacent sub-regions form a complete region we are going to compare the pixels from the border of each sub-region. If two different adjacent sub-regions fulfill the inclusion criteria between their pixels, then we are going to consider them as a one region.

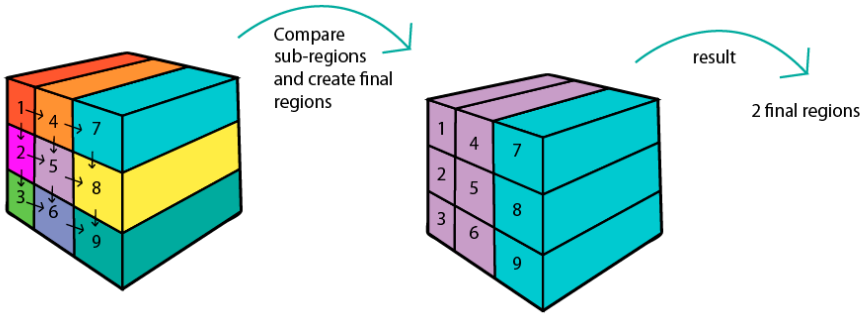

Figure 7: Final regions achieved through the multiple sub-regions

2.2.2 Pseudo-code

Algorithm 3 Main function parallel code

```
1:  procedure MAIN()
2:      image = ReadData(file);                                   ▷ Read the image data from the file
3:      grid_dims = InitMPI();          ▷ Initialize MPI communication and generate MPI grid dimensions
4:      n_local = (n/grid_dims[1]);                                          ▷ Update n local size
5:      m_local = (m/grid_dims[0]);                                          ▷ Update m local size
6:      p_local = p;                                                         ▷ Update p local size
7:      DistributeImage(image,sub_image);      ▷ Divide the image into multiple sub-images per each process
8:      int local_regions[m_local*n_local*p_local]                    ▷ Array to store the local regions
9:      while all pixels not assigned do          ▷ Region growing until all the pixels have a region assigned
10:         seed(x,y,z) = rand%((regions not assigned));                ▷ Select a seed point in the image
11:         region_grow(sub_image,local_regions,m_local,n_local,p_local,seed);  ▷ Update regions array with
    the new pixels assigned
12:     MPI_Barrier();
13:     CombineRegions();                          ▷ Combine the sub_regions into the original regions
14:     MPI_Barrier();
15:     Gather_Regions(local_regions,local_size);      ▷ Gather information to build the final regions array
16:     MPI_Finalize();
```

Algorithm 4 InitMPI

```
1: procedure InitMPI()
2:     int rc = MPI_Init(NULL,NULL);                          ▷ Init the MPI communication
3:     rc = MPI_Comm_size(MPI_COMM_WORLD,size);               ▷ Size = number of processes
4:     rc = MPI_Comm_rank(MPI_COMM_WORLD,rank);               ▷ Rank = number per each process
5:     MPI_Dims_create(size,2,dims); ▷ Create the dimensions of the grid based on the number of processes.
6:     MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,0,cart_comm);  ▷ Create the 2-dimensional
   MPI grid based on the dimensions.
7:     MPI_Cart_coords(cart_comm,rank,2,coords);              ▷ Assign coordinates per each process rank
8:     MPI_Cart_shift(cart_comm,1,1,north,south);             ▷ Link the processes in north-south direction
9:     MPI_Cart_shift(cart_comm,0,1,west,east);               ▷ Link the processes in west-east direction
```

Algorithm 5 CombineRegions

```
1: procedure CombineRegions()
2:     if west >= 0 then                                      ▷ If the local process has a west neighbour process..
3:         MPI_Recv(neigh_border_image from west process);    ▷ Receive its border image
4:         MPI_Recv(neigh_border_region from west process);   ▷ Receive its border region
5:         while all different regions in local_border_region do   ▷ For all different regions
6:             if InclusionCriteria(neigh_border_image,local_border_image) then
7:                 local_border_region = neigh_border_region;   ▷ Update the local region name
8:     if north >= 0 then                                     ▷ If the local process has a north neighbour process..
9:         MPI_Recv(neigh_border_image from north process);   ▷ Receive its border image
10:        MPI_Recv(neigh_border_region from north process);  ▷ Receive its border region
11:        while all different regions in local_border_region do   ▷ For all different regions
12:            if InclusionCriteria(neigh_border_image,local_border_image) then
13:                local_border_region = neigh_border_region;   ▷ Update the local region name
14:    if east >= 0 then                                      ▷ If the local process has an east neighbour process..
15:        MPI_Send(local_border_image to the east process);
16:        MPI_Send(local_border_region to the east process);
17:    if south >= 0 then                                     ▷ If the local process has a south neighbour process..
18:        MPI_Send(local_border_image to the south process);
19:        MPI_Send(local_border_region to the south process);
```

Algorithm 6 GatherRegions

```
1: procedure GatherRegions()
2:     if rank == 0 then                                      ▷ If its the master process...
3:         total_regions = MPI_Recv(local_regions from all processes)  ▷ Receive and combine sub_regions
   information
4:         CreateFile(total_regions);                         ▷ Create the final file results
5:     else                                                   ▷ If its not the master process..
6:         MPI_Send(local_regions to the master process);     ▷ Send local regions information to the master
```

3 Theoretical performance estimation

3.1 Time Complexity

To be able to calculate the theoretical performance of our parallel approach, we will define three different main times: t_{setup}, $t_{algorithm}$ and t_{comm}, being them the set up time, the region growing time and the communication time respectively. We also need to consider the dimensions of the whole image as (m,n,p).

We will define the setup time as the time of reading the information plus the time of creating the image array structure and the time of distributing the sub-image to each process.

$$\begin{aligned} t_{setup} &= t_{\text{readFile}} + t_{\text{createImg}} + t_{\text{distributeImage}} \\ &= m * n * p + m * n * p + (m * n * p) * (n_{process}) \end{aligned}$$

Then, the region growing algorithm will directly depend of the number of chosen processes.

$$\begin{aligned} t_{algorithm} &= t_{regionGrowing} * (n_{process}) \\ &= (n_{local} * m_{local} * p_{local}) * (n_{process}) \end{aligned}$$

Finally, the communication time could be the most complex one. The communication time takes into account the time of exchanging borders information and the time of gathering all the sub-regions into the final one. As it can be seen, the exchange time will directly depend of the MPI grid dimensions.

$$\begin{aligned} t_{comm} &= t_{exchange} + t_{gather} \\ &= (2 * (grid_{dims[0]} + grid_{dims[1]}) - 4) * 2operations \\ &+ ((grid_{dims[0]} - 2) * (grid_{dims[1]} - 2)) * 4operations + (m * n * p) * (n_{process}) \end{aligned}$$

Then, we can conclude that our parallel time complexity will be the sum of the three mentioned times.

$$t_{final} = t_{setup} + t_{algorithm} + t_{comm}$$

4 Implementation details

In this section we will explain more carefully the new functions of the MPI library that have allowed us to implement the communication between processes based on a 2 dimensional grid.

- **MPI_Dims_create***(int n_nodes, int n_dims, int dims[])*

  This function generates the dimensions of the MPI grid based on the selected number of processes. [3]

- **MPI_Cart_create***(MPI_Comm comm_old, int n_dims , int dims[], int periods[], int reorder, MPI_Comm *comm_cart)*

  This function creates the MPI grid.[4]

- **MPI_Cart_coords***(MPI_Comm comm, int rank, int maxdims, int* coords)*

  This function allows us to assign the coordinates of the MPI grid to each process (rank).[5]

- **MPI_Cart_shift**(*MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest*)

  This function returns the rank number of the process situated in the direction and position specified by the attributes.[6]

5 Results

5.1 Data cases

The data cases that we decided to test to be able to study correctly the parallel and sequential region growing approach are the following geometries:

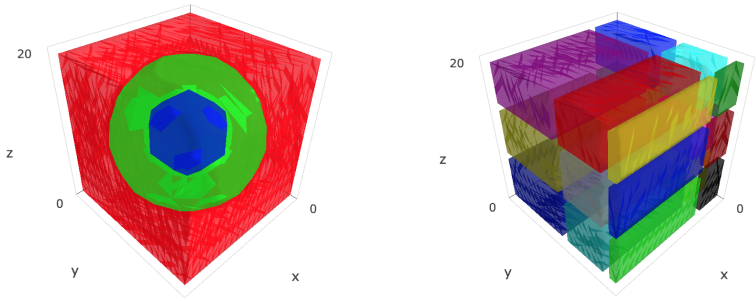

Figure 8: 2 used data types

Then, based on the previous data cases we will test different image sizes to be able to see the relationship between the performance parallel algorithm and the data size. These defined sizes are defined in the next section.

5.2 Graphical results

The region growing C code generates a text file where there is the relationship between each pixel and the region number. Based on that structure, we created a python file that reads the value of each pixel and visualizes the data as following:

- **Test case 1:**

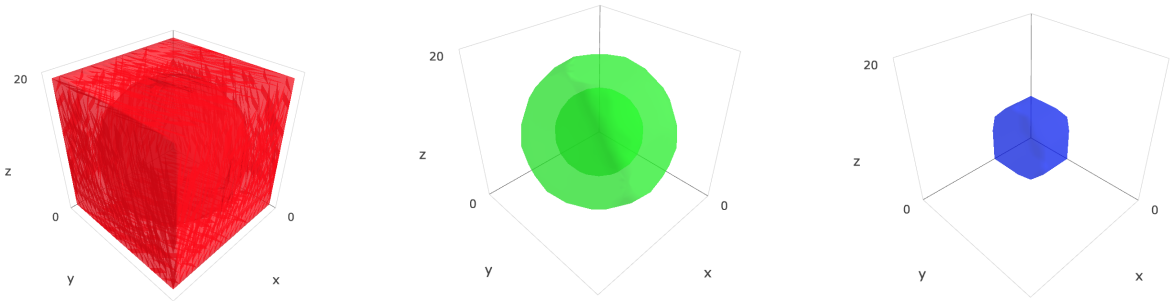

Figure 9: Regions of data type 1

- **Test case 2:**

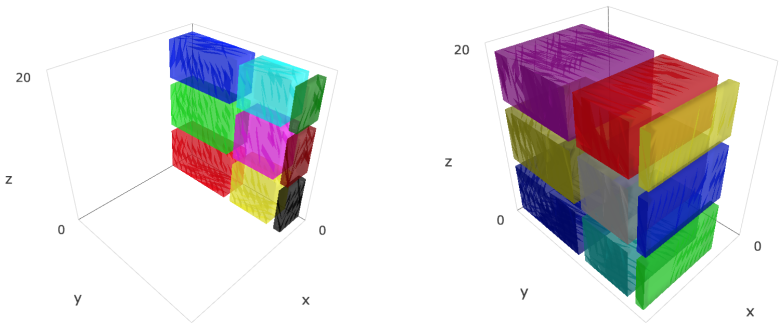

Figure 10: Some regions of data type 2

6 Experimental speedup investigation

Finally, in this section we are going to study the experimental speed-up comparing the sequential and the parallel approach using different number of processes and different data sizes.

6.1 Execution times

The experimental execution times can be observed in the tables below, for data type 1 and 2, respectively:

Data size \ N processors	1	2	4	8
20	0.049	0.050	0.054	0.064
40	0.075	0.068	0.067	0.077
80	0.277	0.204	0.181	0.182
160	1.852	1.260	1.079	1.313
240	6.149	4.114	3.525	4.500

Figure 11: Obtained execution times (seconds) for different combinations of inputted data size (side of the cube) and number of processors for data type 1.

Data size \ N processors	1	2	4	8
20	0.051	0.051	0.054	0.065
40	0.082	0.073	0.072	0.083
80	0.320	0.239	0.207	0.204
160	2.254	1.588	1.310	1.897
240	7.319	5.078	4.221	6.403

Figure 12: Obtained execution times (seconds) for different combinations of inputted data size (side of the cube) and number of processors for data type 2.

6.2 Speed-up

The obtained experimental speed-up can be observed in the following tables and plots:

Data size \ N processors	1	2	4	8
20	1.00	0.98	0.91	0.77
40	1.00	1.10	1.11	0.98
80	1.00	1.36	1.53	1.53
160	1.00	1.47	1.72	1.41
240	1.00	1.49	1.74	1.37

Figure 13: Obtained speedup for different combinations of inputted data size and number of processors for data type 1.

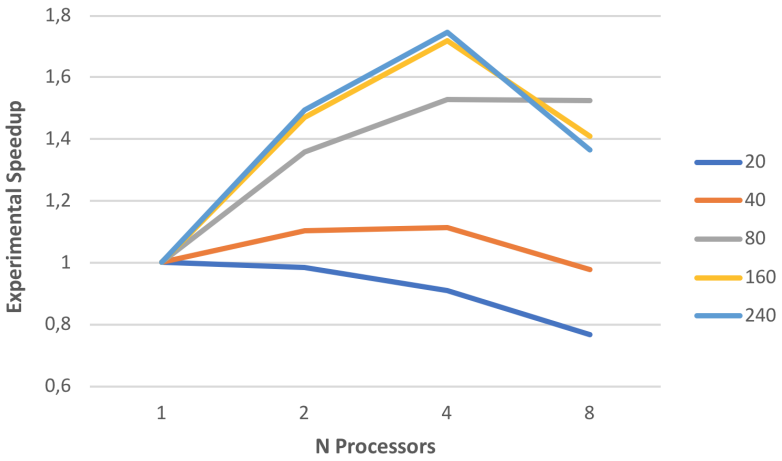

Figure 14: Plotted obtained speedup for different combinations of inputted data size and number of processors for data type 1.

Data size \ N processors	1	2	4	8
20	1.00	1.00	0.95	0.79
40	1.00	1.12	1.14	0.99
80	1.00	1.34	1.55	1.57
160	1.00	1.42	1.72	1.19
240	1.00	1.44	1.73	1.14

Figure 15: Obtained speedup for different combinations of inputted data size and number of processors for data type 2.

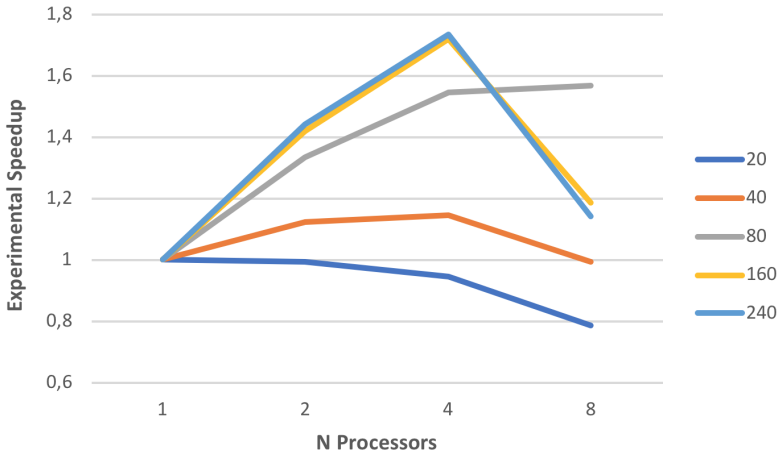

Figure 16: Plotted obtained speedup for different combinations of inputted data size and number of processors for data type 2.

7 Final conclusions

In the present project we have studied the popular segmentation algorithm of region growing, and have successfully implemented the algorithm in a sequential and a parallel way.

We would like to conclude the project mentioning the experimental results, as they confirm the idea that parallelizing a program is beneficial, but not in every situation. As we can observe in the previous tables and plots, for a large amount of data, execution time decreases with the number of processors, as expected, but there are some exceptions that have to be mentioned. First, parallelizing a program for a small amount of data will normally be disadvantageous, as a sequential implementation will outperform a parallel one due to communication overhead. Furthermore, there is a trade-off between speed and the number of processors, as adding more processors will end reducing the data each processor has to compute, but also increasing the total communication needed to synchronize the whole process, which can lead to higher execution times.

References

[1] "Wikipedia: Image segmentation," https://en.wikipedia.org/wiki/Image_segmentation, last accessed: 2018-05-5.

[2] "Wikipedia: Region growing," https://en.wikipedia.org/wiki/Region_growing, last accessed: 2018-05-5.

[3] "Mpi_dims_create," https://www.mpich.org/static/docs/v3.2/www3/MPI_Dims_create.html, last accessed: 2018-05-5.

[4] "Mpi_cart_create," https://www.mpich.org/static/docs/v3.2/www3/MPI_Cart_create.html, last accessed: 2018-05-5.

[5] "Mpi_cart_coords," https://www.mpich.org/static/docs/v3.1/www3/MPI_Cart_coords.html, last accessed: 2018-05-5.

[6] "Mpi_cart_shift," https://www.mpich.org/static/docs/v3.2/www3/MPI_Cart_shift.html, last accessed: 2018-05-5.