TDT4200-Problem Set 2 - Programming MPI - Image Convolution and Morphing (Graded)

Tor Andre Haugdahl, Zawadi Svela and TDT4200 Staff

September 24, 2021

Deadlines:

- **Theory:** TBD (Pass/Fail)
- **Programming - Part 1 & 2: Sept 29, 2021** by 22:00 in BB

This assignment must be done individually and w.o. help from others other than TDT4200 staff. We encourage that clarification questions be posted on the Forum in BB so all can benefit. All sources used found on the internet or otherwise must be referenced

Grading: This assignment is graded and counts towards your final grade in this course. Programming Part 1 and 2 together counts about 15% of your final grade and must be passed to take the next PS and the final in the course. Code documentation and code clarity will count 10% of the grade on this assignment, i.e 1.5% of your final grade. E.g document each function and what is does, variable names, etc

Deliverables

- Part 1: Deliver the file in the src-folder called "main.c"
- Part 2: Deliver the file in the src-folder called "morph.c"

Important: When you deliver your code, for this problem set and future ones, make sure that you only deliver the files explicitly asked for. This will only include code you have edited yourself, and not binary files, example input, etc. A consequence of this is that *you are not allowed to utilize external libraries not already included in the hand-out code.*

MPI programming - Imaging

In this exercise you are going to parallelize two programs using MPI. The first one (Part 1) is a image convolution program while the second (Part 2) is a implementation of the Beier-Neely image morphing algorithm.

MPI-Imaging Part 1: Parallel image convolution using MPI

In this program, a convolutional kernel is shifted across an image in order to produce a new image. A kernel is an equation that determines the value of a pixel in the new image based on pixels in the old one. A typical kernel will look like this:

$$\begin{array}{rlll} N[i,j] = & -1 * O[i-1,j-1] & -4 * O[i,j-1] & -1 * O[i+1,j-1] \\ & -4 * O[i-1,j] & +20 * O[i,j] & -4 * O[i+1,j] \\ & -1 * O[i-1,j+1] & -4 * O[i,j+1] & -1 * O[i+1,j+1] \end{array} \quad (1)$$

In words, for each color channel, the value in a new pixel $N[i,j]$ is a sum of the corresponding old pixel $O[i,j]$ and its neighbors. If we are applying the above kernel, and for the sake of example only look at a single color channel (say, red), then we'd multiply the "here"-pixel's red-value by 20, and then subtract 4 times the red-value of the orthogonal neighboring pixels and subtract 1 times the red-value of the diagonally neighboring pixels. The sum would be the new pixel's red-value. This process is then repeated across all pixels and channels, and often across multiple iterations.

These kernels are often provided in matrix form, the one corresponding to the kernel above would look like this:

$$\begin{array}{ccc} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{array} \quad (2)$$

The hand-out code in this assignment is a partially implemented parallel MPI program:

- Rank 0 reads the image and distributes it using MPI_Scatter.
- Each rank performs the convolution on its own partition for some number of iterations
- The result is gathered at rank 0 again and written to file.

The challenge is that because of the shape of the kernels, each process needs the value of pixels outside it's own partition. This is the problem you will have to solve by implementing what is called *border exchange*.

Tasks

Note: All necessary changes should be doable only by adding code to the main() function. You can assume the number of processes are a power of 2.

1. Compile and run the code using at least 2 processes. Look at the resulting image. You should see a solid line on the border between the partitions.

   You are provided a Makefile. This means the program can be compiled by simply running the command "make" in the root folder of the handout code. This will produce an executable called "main".

2. Fix the border exchange. Think about:
   - Which pixels need to be sent to which processes?
   - What order of communication is best to ensure quick and deadlock-free communication?
   - Do the local image partitions have room to store received pixels? And do you need to make changes to scatter/gather if you change this?

3. Using MPI_Wtime, measure how much time the program spends from start to finish using a single process (mpirun -np 1). Test with 2, 4 and 8 processes and document the *speedup* you get compared the single-process case.

MPI-Imaging Part 2: Parallel image morphing using MPI

For this exercise, you will take a pre-written serial implementation of the Beier-Neely algorithm and write a distributed version of this program.

The Beier-Neely algorithm is a process of morphing an image. Given an image and a set of feature lines, each meant to outline some feature in the image, like the angle of a mouth, the position of a jawline, ear, etc., the algorithm creates a new image for a new set of positions for these lines. The process of creating the morphed output image is to for each output-pixel determine a single position in the input to extract color values from. In short, for each pixel, we look at *all* the lines in the out-image to determine their relative closeness, and then use that information to find a corresponding position in the in-image to extract our color from. So the morphed image will look similar to the source-image close to each line, while looking more warped elsewhere.

A common use-case for this algorithm, and the one we will implement here, is to create a series of morphed images to transition from one image to another. Two input images, each with their corresponding feature lines, are morphed towards a set of equal line positions. If we calculate those common position as a weighted average of the two input sets, and fade the images in and out in a similar fashion, we can create a continuous transition, where one image warps into another.

Program description

The handout code comes with a Makefile to aid compilation. These are the commands you need:

- Compile program:

  ```
  make main
  ```

- Run program:

  ```
  mpirun -np N ./main src.jpg dst.jpg outpath/ STEPS lines.txt
  ```

- Example:

  ```
  mpirun -np 1 ./main images/man9.jpg images/man10.jpg
  
              output-png/ 10 lines/lines-man9-man10.txt
  ```

  STEPS is the number of "in-between"-images you want between the source and destination images. Runtime of the program does increase linearly with this number, so keep it low, e.g. 3, if you just want to test correctness. Keep in mind that the "-np" flag has no real effect until you implement the MPI-functionality.

  You can use any two images, but the line-sets provided corresponds to the images, so your output will look interesting if you use different images.

Tasks

Your task will be to parallelize the serial program using MPI, similarly to PS1 assignment. These are the requirements:

- Each process/rank should only produce one part of the output image. This includes not allocating more space than is necessary for this partition.
- The input is *broadcast* across all processes. This is because each output pixel is dependent on every pixel in both the input images and their line sets.
- Only rank 0 should read the input and write the final output image(s) to file. This means you need to *gather* the output partitions before writing.
- You can assume the image size is a power of 2.

A suggested approach:

1. Test the program. Compile and run using a single process.

2. Set up necessary MPI-variables and broadcast the input.
   The variables p, a and b are already broadcast. These are optional variables for the Beier-Neely algorithm.

3. Allocate partition space.

4. Implement the *gather* and test it. Perhaps on one of the input images?

5. Implement the necessary changes for each process to calculate its output partition. Be mindful of where you are working on coordinates in the local partition and where there are coordinates in the (global) input and output images.