TDT4200 PS 3a - MPI Graded and Mandatory

A. C. Elster, B. Gotschall and Z. Svela

Due Friday, Sept 27, 2019

Introduction

This Problem Set counts 10% of your final grade. You can get max 110pts, were 100pts is considered full score, if you complete Bonus task + Documentation. Points are given as follows:

Table 1: GRADING of PS 3a

Programming Assignment	Documentation
Task 1: 50 pts	1e - 5 pts
Task 2: 80pts (do not turn in Prog. Task 1)	2b - 2pts, 2c - 2 pts, 2d - 6pts
Bonus Task: 90 pts (do not turn in T1 or 2)	3c - 4 pts, 3d - 6pts
+ 5 pts for good in-line doc.	you get pts for either 2 or 3

In this assignment, you will implement an edge detection algorithm. You will start with a black and white image, and produce a new image only containing the lines within the original. Our method for doing this is illustrated in Fredrik Berg Kjølstad's handout on ghost cells (pdf), also covered in class, but the necessary information will also be explained here.

The method is shifting a convolutional kernel over an image which computes the underlying data together with the kernel weights into a new value. In simpler words: a new pixel is generated by multiplying the pixel and its neighbor values with kernel coefficients and summing them up. Some kernels are multiplying the accumulated value at the end with a constant factor (e.g. the gaussian kernel). This mathematical operation allows relatively easy to blur or sharpen images. Special kernels can also amplify edges, which is called edge detection. The mentioned paper includes the following laplacian kernel as an example. Let O be the original image and N the new image. The convolutional operation using a 9-point laplacian kernel can be expressed as follows (see also Figure 7a in Kjølstad's write-up):

$$\begin{array}{rccc} N[i,j] = & -1*O[i-1,j-1] & -4*O[i,j-1] & -1*O[i+1,j-1] \\ & -4*O[i-1,j] & +20*O[i,j] & -4*O[i+1,j1] \\ & -1*O[i+1,j+1] & -4*O[i,j+1] & -1*O[i+1,j+1] \end{array} \quad (1)$$

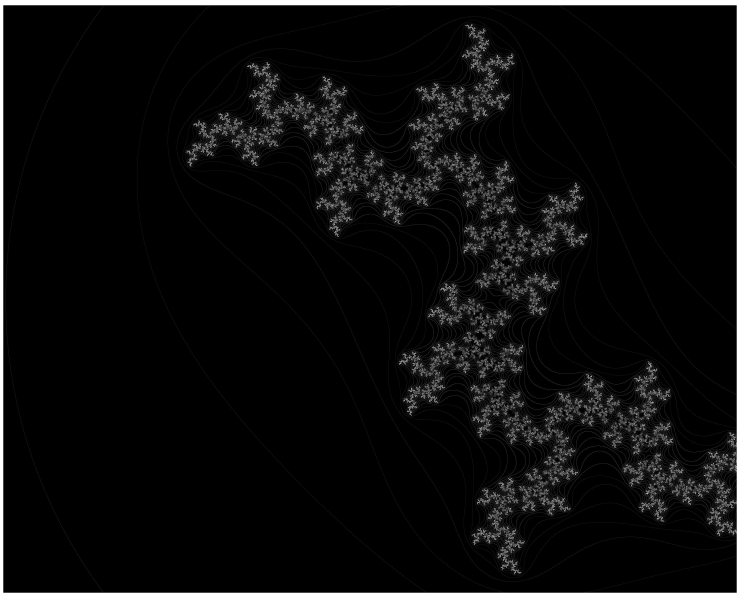

Figure 1: 9-point laplacian kernel applied on a Mandelbrot rendering

This kernel applied on the already familiar Mandelbrot rendering from the previous assignments can be seen in Figure 1.

By re-applying this kernel on the new image, and repeating the process a number of times, we will produce an image where only the most prominent lines are visible.

With this assignment you are handed a code which reads in a bmp image, extracts a black and white representation, applies a chosen kernel a number of iterations and saves the result in a new bmp image. The black and white representation is done to make the kernel operation simpler. However, the code which is provided is completely serial. **Your task is to parallelize the code using MPI.**

As described in the paper and above, the algorithm computes a pixel based on its own value as well as the values of all 8 of its nearest neighbors. This poses a challenge when calculating the next iteration on the local boarders since some of the data needed are stored on other processes.

Preparation

NB! You do not need to document this task

1. Get familiar with the code
2. Try out different provided kernels
   (a) laplacian1Kernel, laplacian2Kernel, laplacian3Kernel
   (b) gaussianKernel (! adjust the kernel dimension this is a 5x5 kernel !)
3. Take a baseline - laplacian1Kernel
   (a) measure the wall-time with one and 1024 iterations

Note: Read though all the following tasks first, you might save work by already thinking about future tasks.

Task 1: Parallelize with MPI - Single iteration

(Submit your code for this Task or Task 2, but do this first)
Create an MPI application which applies the laplacian1Kernel just one time on the input image. Distribute the work across all ranks. You can comment out the iteration loop for now, or start the application with just one iteration: (`--iterations 1`)

a) Form the provided code into an MPI application

b) Scatter the input image across all ranks. A row wise division is sufficient for this task.
NB! You application must work for arbitrary image resolutions. Do not assume that the data can be equally divided across all ranks!

c) Let all ranks apply the kernel to their respective chunks.

d) Gather the final image and output it.

e) Measure the speedup for one iterations and compare it to the baseline from Task 0.3a

Note: If you have trouble distributing the data, try flattening the image data array (1-dimensional array). Note that this requires adjustments to the calculation.

Task 2: Parallelize with MPI - Multi iteration

(Submit your code for this Task or Task 2)
Now your application must be transformed to apply the laplacian1Kernel an arbitrary number of times on an input image. This will need a more sophisticated communication algorithm.

a) Extend your MPI application to communicate the chunk boundaries between all adjacent ranks after each iteration
NB! For bandwidth reasons you are not allowed to redistribute the whole image again. Only adjacent ranks are supposed to communicate their boundaries (boundary exchange, ghost cells).

b) Describe the communication pattern for 3 iterations. Count how many communications are necessary and calculate the amount of data that is exchanged in total.

c) Measure the speedup for one and 1024 iterations and compare it to the baseline from Task 0.3a and the single iteration implementation from Task 1.1e

d) **Briefly** describe your approach of solving this task with a focus on the MPI communication and changes done to the kernel computation (if changes were necessary).

Task 3 – Bonus task (+10 pts)

(If solved, submit your code for this one instead for Task 1 or Task 2)
To support bigger kernels like the provided gaussianKernel, just a single border exchange is not enough. This kernel computes a pixel with two neighbors in every dimension, requiring more ghost cells and and an advanced line exchange for every iteration. The naive solution would be to exchange $\lfloor kernelDim/2 \rfloor$ rows or columns each iteration, but that wastes bandwidth as not all data is actually required in each iteration. Instead the concept of a deep halo boundary exchanged, as described in the paper from the beginning, would be an efficient communication scheme.

a) If not already done, divide the input image in rectangles and scatter it across all ranks (not row wise). This results into more adjacent ranks (north, east, south, west), which need to communicate each iteration.

b) Use the gaussianKernel and implement a deep halo communication scheme to exchange **only the necessary** chunk boundaries each iteration.

c) Describe the communication pattern for 3 iterations. Count how many communications are necessary and calculate the amount of data that is exchanged in total. How much data do we save from a naive **boundary** exchange in which all two rows and columns from the neighbors are exchanged each iteration?

d) Measure the speedup for one and 1024 iterations and compare it to the baseline from Task 0.3a and the single iteration implementation from Task 1.1e

Deliverables

The documentation must be delivered as a PDF file together with the source code packed in a ZIP file and submitted in time on blackboard. The code must be documented and in a runnable state. Do not use any libraries apart from standard libraries or those provided. Do not change the calling conventions (parameters) of the application.

Provide only **one** version of your code! It should be the latest (most advanced) code either from Task 2 or the Bonus Task. The following tasks require documentation (if solved):

- 1e, 2b, 2c, 3, (Bonus 3, 4)

Do not include binaries or object files in your delivery and don't include any BMP image files.

PS 3b (Theory), will be Pass/Fall, but mandatory and coming out this coming week.