Kernel Image Processing

Hasib Kolaković
`89221033@student.upr.si`

Abstract

This paper studies Kernel image processing algorithm by implementing it in a variety of modes. In particular, we demonstrate what it is, how it works, how we can run this in sequential, parallel and distributive modes. For the distributed part of the implementation, we present a parallel method on distributed-memory. Finally we obtain computation performances and compare the results.

1 Introduction

Kernel image processing is a fundamental concept in a field of digital image processing, playing a key role in techniques such as image sharpening, bluring, edge detections, color processing, noise reduction etc. Processing is done by convoluting every pixel of an image with kernel matrix.Below we have examples of three kernel matrices, from left to right: identity kernel, sharpen kernel and blur kernel.

```
[0 0 0]      [0 -1 0]      [1/9 1/9 1/9]
[0 1 0]      [-1 5 -1]     [1/9 1/9 1/9]
[0 0 0]      [0 -1 0]      [1/9 1/9 1/9]
```

Figure 1: Three kernel matrices displayed next to each other.

For implementation of kernel image processing, we are using Java programming language. Thus, for storing and manipulating with our images, we will use BufferedImage class. It allows us to use various functions such as pixel manipulation, drawing and reading or writing files. Images have 24-bit pixel representation, that is pixels have RGB color model where each color's intensity is represented with 8-bits.

2 Sequential computitation

After loading the image, the techniques for image processing are done via convolution, where we have our input image and two dimensional kernel matrix. For each pixel in our input image, we overlap it with our central value in kernel matrix, and each overlapping value is multiplied and then added to final resulting pixel. This pixel is added to the new output image at the placement of the original pixel. For edge handling, the image is conceptually tiled and values of "missing" pixels are taken from the opposite edge or corner.

Algorithm 1 Pseudocode for convolution

```
1: set outputImage = inputImage.blank
2: for each row in inputImage do
3:     for each pixel in row do
4:         set accumulator = 0
5:         for each kernelRow in kernel do
6:             for each value in kernelRow do
7:                 Calculate pixel position(wrapped image handling)
8:                 Multiply pixel by value and add to accumulator.
9:             end for
10:        end for
11:        set corresponding outputImage pixel = accumulator
12:    end for
13: end for
```

2.1 Time complexity of kernel image processing

The time complexity of kernel-based algorithm depends on the size of a image and kernel. Image size is determined by number of rows(height) n and number of pixels in a row(width) m. Thus the time complexity of processing the entire image is often represented as O(n×m). Time complexity also depends on kernel matrix size. The kernel is a small matrix, typically of size k×k, that slides over the image to modify each pixel. When kernel is applied to an image, it means it needs to be applied to each pixel. Thus the overall time complexity of kernel image processing is

$$O(n \times m \times k^2)$$

In our examples we will test our modes with relatively small k values, and sizeable n and m values.

3 Parallelization Problem

To parallelize our sequential computation, we have to decide how to distribute the work among a number of workers. In both parallel and distributive modes, we will give each worker a specific region of the image, that it will have to process. In our case, we send each worker information about where to start and where to end. This division is based on the width of the image. In other words each worker gets parameters from and to, and the whole height. Both in parallel and distributive modes, I take the number of workers, and divide the image into that number of regions. Each worker is assigned with one region. Usually, all workers get the same number of work to do, except the last worker that gets the last region which sometimes has extra (image width modulo number of workers). This does not influence our time complexity excessively, since width and height of image are relatively large and similar in size, and size of kernel is small-scale.

4 Multi-threaded mode

Loading an image here is same as in loading an image in a sequential mode. In our multi-threaded mode we have Runnable class, where the run method defines how each segment of the image is processed.

Algorithm 2 class Task implements Runnable

```
1: Constructor Task(start, end)
2: Function run() {
3: Convolution(start,end)
4: }
```

Afterwards in our main function we define an array of the Runnable objects with parameters ***start*** and ***end***, where each region of the image is assigned.

An array of threads is initialized, followed by the immediate start of each thread to execute its assigned task concurrently. After all threads are started, the main program waits for them to complete using the function ***join()***. This step is crucial to ensure that the final output image is fully processed before any further operations(such as image saving or displaying).

Algorithm 3 Multi-threaded main function

```
1: int parts = number_of_threads
2: Task[] task = new Task[parts]
3: Assign each task[i] start and end parameters
4: Thread[] thread = new Thread[parts]
5: Assign each thread[i] a task[i]
6: Start each thread[i]
7: Join each thread[i]
8: Display output image
```

5 Distributed Mode

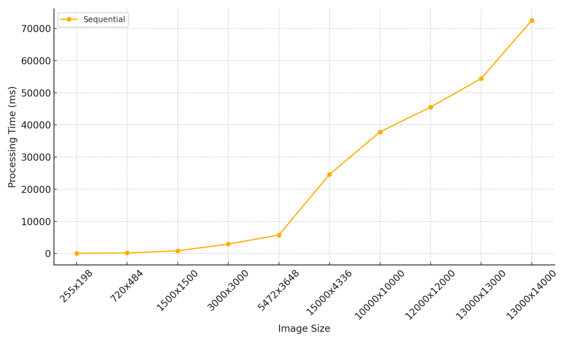
The implementation of distributed mode is accomplished with MPI(Message Passing Interface). We use MPJ Express, which is an open-source Java message-passing library. In this mode, we have root and slave processes. We assigned the process with rank 0 to be our root process. Unlike the multi-threading mode, processes do not share memory. The root process loads an input image, then divides the image into regions based on the number of processes. Since the MPI can not communicate with images in their JPEG format, first the root process translates each region into an array of bytes and sends it to each slave process. The slave process transforms this byte array back to region of the input image, and then do the processing on this subimage. In the meantime, the root process is assigned to process a subimage as well. Subsequently, the processed(by slaves) subimages are transformed to an array of bytes again, and sent back to root process to translate the array and assemble the fully processed image.

Algorithm 4 Distributed Algorithm

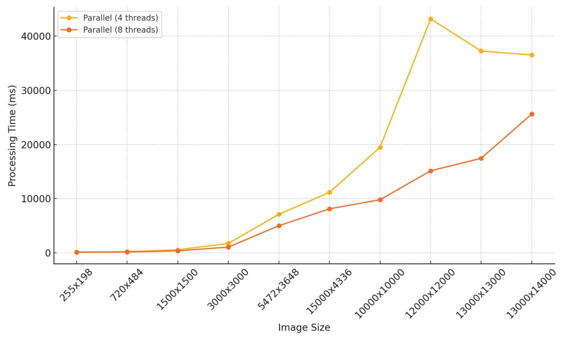
```
 1: if (rank == 0) {
 2: Load the input image.
 3: Send slaves their subimages in form of byte arrays.
 4: }
 5: if (rank != 0) {
 6: Receive the subimage.
 7: }
 8: Common Part:
 9: Process the subimage.
10: if (rank != 0) {
11: Send to the root: the processed subimage in form of a byte array.
12: }
13: if (rank == 0) {
14: Receive all byte arrays from slaves.
15: Transform byte arrays into subimages.
16: Assemble the subimages into final output image.
17: }
```

6 Results

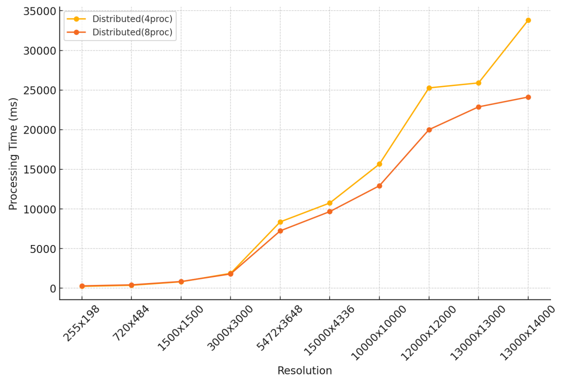
The program was evaluated on Intel® Core™ i5-1135G7 Processor which has 4 cores, and 2 threads for each of the cores. For testing we took 10 images of various dimension, ranging from 255x198 to 14000x13000. Each image was tested 5 times for each mode(sequential, multi-threading and distributive). For each image and mode, the average of those measurments was taken into final comparison.



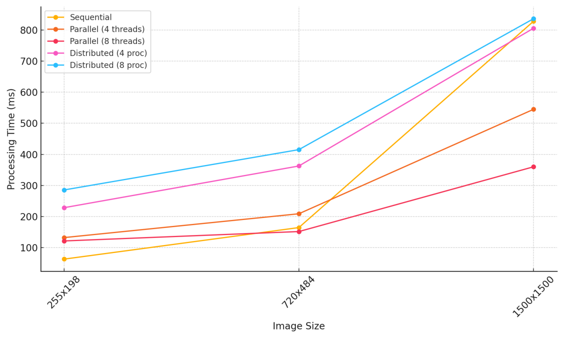
Sequential mode is the simplest mode, since it uses only one thread for execution. Each task or step must be finished before the next one begins.



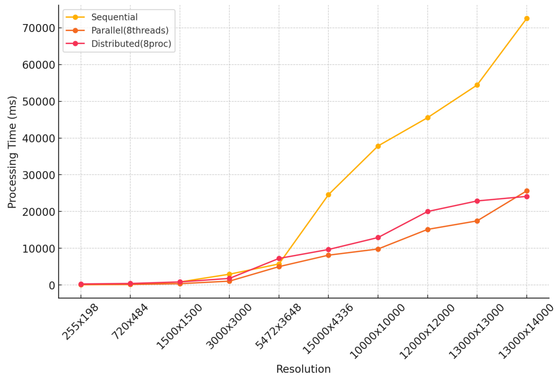
Multi-thread mode has to take time and prepare the workers before processing. As it can be seen from the graph, multi-thread mode with 8 threads outperforms multi-thread mode with 4 threads.



Communication in distributive program mode plays a huge role when it comes to time efficiency.



Comparing results of all modes together for relatively small images show that sequential outperforms for really small images.



Comparing sequential, parallel(8 threads) and distributive(8 processes).

7 Conclusion

Overall parallel(multi-threading) and distributive modes are faster than sequential mode. This makes sense, since using more workers to divide a task and perform it in parallel will speed our process.

Sequential mode performs better only when we test on smaller samples. This is because multi-threading and distributive modes take time to define regions and set workers to work, especially in distributive mode where we have to communicate over different processes.

Using 8 threads for parallel program will yield us better results than using only 4, that is using all threads that our CPU has to offer will yield performance improvement.

For distributive part, dividing the problem onto 8 processes yielded us better results than dividing the problem onto 4 processes.

Parallel and distributive programs are far more complex than sequential programs. Having to do more steps in execution takes more time, but when dividing the work onto more threads/processes, parallelization surpasses the sequential execution of kernel image processing.