Edge Detection for High-Resolution Images with Paralleled methods

Cuong Dao, Donggyun Park
(Group 3)
`{cuongdd,donggyun}@kth.se`

1 Problem description

Edge detection is an image processing technique for finding the boundaries of objects within images. Generally, it works by detecting the discontinuities in brightness. It is one of the fundamental image processing technique in modern computer vision. Among many edge detection algorithms, Canny [1] is a popular one. In this project, we strive to apply Canny algorithm for detecting edges in high-resolution images.

Since the dimensions of the image we will be working on is large ($\approx$ 20 million pixels), sequential implementation of the Canny algorithm would be inefficient. We are going to implement a paralleled version of the algorithm based on OpenMPI. We will conduct an analysis and comparison on the performance of the algorithm on sequential and paralleled versions with different kernel size of the algorithm. Specifically, we will show the theoretical estimation of the complexity of the paralleled algorithm and provide empirical data and analysis on the experiments we run with real images on 1 – 8 processors. We use C as the main language in our implementation on OpenMPI. Additionally, `python` will also be used for visualization purposes.

2 Method

In this section, we present the methodology for this project. We start with a description of main steps of the Canny edge algorithm. We then propose and analyse our paralleled version of the algorithm.

2.1 Canny edge detection algorithm

Canny edge [1] is an edge-detection algorithm developed by John F. Canny in 1986. It is a multi-stage algorithm to detect a variety of edges in images. The algorithm can be broken down into 5 major steps:

1. **Apply Gaussian filter to denoise the input image**
   The result of edge detection can be affected if the input is noisy; therefore, we first need to denoise the input image. To that end, a Gaussian filter convolved with the image. For a Gaussian kernel with size $(2k+1) \times (2k+1)$, the elements of the filter are calculated as follow

$$F_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right); \qquad 1 \leq i, j \leq (2k+1) \tag{1}$$

   We use $k = 2$, thus, our Gaussian filter is of the size 5×5.

2. **Find the intensity gradients along the x and y directions of the image**
   Gradient is the first-order derivative of the image along each direction. It is rough estimation of all the edges in the image. That gradients in each direction ($\mathbf{G_x}$ and $\mathbf{G_y}$) are computed by convolving the image with an edge-detection operator (e.g., Sobel, Prewitt) of each direction. The gradient $\mathbf{G}$ of the image and its direction $\mathbf{\Theta}$ can then be computed as

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2} \qquad \mathbf{\Theta} = \arctan \frac{\mathbf{G_y}}{\mathbf{G_x}} \tag{2}$$

In this project, we experiment with Sobel [2] operator of two different sizes, 3 and 5.

$$\mathbf{G_x} = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$\mathbf{G_x} = \begin{bmatrix} +2 & +1 & 0 & -1 & -2 \\ +2 & +1 & 0 & -1 & -2 \\ +4 & +2 & 0 & -2 & -4 \\ +2 & +1 & 0 & -1 & -2 \\ +2 & +1 & 0 & -1 & -2 \end{bmatrix} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +2 & +2 & +4 & +2 & +2 \\ +1 & +1 & +2 & +1 & +2 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -2 & -1 & -2 \\ -2 & -2 & -4 & -2 & -2 \end{bmatrix}$$

3. **Apply gradient threshold to get rid of the spurious response in edge detection**
   The edges obtained from the step above are thick. Therefore, we will apply the non-maximal suppression (NMS) algorithm to remove spurious edges and to find 1-pixel edge lines. The general principle of NMS is to go through all the pixels in the gradient matrix and find the those with maximum value in the gradient direction.
4. **Apply double threshold to determine potential edges**
   The edges obtained after NMS should be fairly accurate. However, there might still be spurious edges due to the variation in colors, or noise. To remove these edges, we will apply a high and a low thresholds to remove the edges whose gradients are lower than the low value and higher than the high values.
5. **Track the edge by hysteresis**
   After the above steps, strong edges are clearly identified. To improve the detection of weaker edges, we consider a window of 8 connected pixels for a weak edge. If there is any strong edge in those 8 pixels, we keep the weak edge. Otherwise, it'll be removed.

2.2 Parallelization

A straightforward approach to parallelize the Canny edge-detection algorithm on multiple processors is to divide the original image into sub-images which are then processed on each processor. To that end, we could leverage MPI collective communication to distribute the sub-images into slave processors. One could immediately think of `MPI_Scatter` for this distribution. However, in our case, the heights of the sub-image in each processor vary (for the reason described below); thus, we cannot achieve this with `MPI_Scatter`. Instead, we use `MPI_Send` and `MPI_Recv` to send each sub-image to its corresponding processor. Regarding data distribution, we use the linear data distribution as its suitable for our need. Each sub-image contains a continuous part of the original image, identified as the data between `start_index` and `end_index`. The algorithm to compute the indices is presented below.

We also pad the input image with lines filled with zeros. This serves two purposes 1) make the height of the image divisible by the number of processors P, and 2) preserve the edges in the top and bottom of the image. The details of padding are as follow

- First, the program reads the input image and pad it with zeros on the top and at the bottom according to the size k of the stencil/kernel being used in order to prevent ghost cells. The stencil in this case would be used for computing the gradients of the original image.
- Then, if needed, more row of zeros are padded to the bottom of the image so the height the padded image is divisible by P. This is expected to not harm the performance when the image partitions are distributed to each processor since the number of additional zeros is at most $P - 1$ rows.

Figure 1 depicts the of zero-padding pre-processing step and the distribution of data. Additionally, $m = \frac{k-1}{2}$ **rows are taken from the neighbouring partitions between each image partitions to prevent the ghost cell effect.**

We use `MPI_Bcast` to broadcast the height and width of the image (after padding) to all processors so they could compute their local indices. Here one important thing is the processors have to be synchronized before they can execute the next tasks. `MPI_Barrier` is used for the synchronization in this part. The algorithm to compute the local indices of the sub-image on each processor is as in Algorithm 1

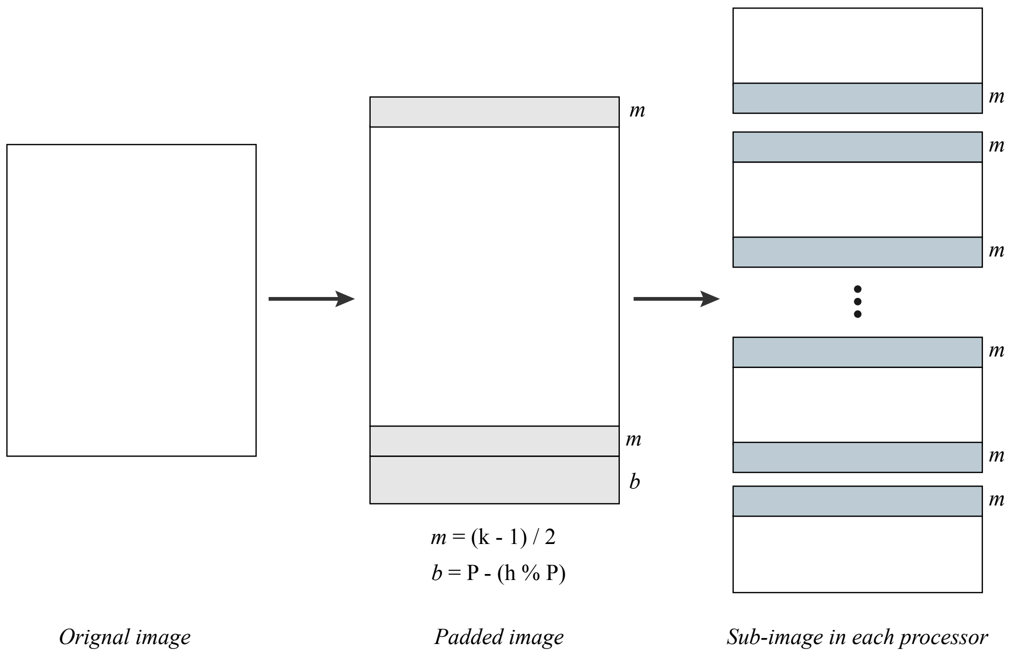


Figure 1: Illustration of image padding. If needed, more zeros (b lines are added after zero-padding on the top and the bottom in order to make the image's height divisible by P.

Algorithm 1 Algorithm to compute local indices

```
 1: if rank $= 0$ then
 2:    start_index = 0
 3:    end_index = height / P + b
 4: else
 5:    if rank $> 0$ or rank $< P - 1$ then
 6:       start_index = rank * height / P - b
 7:       end_index = (rank + 1) * height / P + b
 8:    else
 9:       start_index = rank * height / P - b
10:       end_index = (rank+1) * height / P
11:    end if
12: end if
```

3 Results

3.1 Comparison between theoretical performance estimation and our experiments

3.1.1 Theoretical time estimation

(i) Sequential time

- Time complexity of the convolution operation

$$t_{\text{conv}} = h \times w \times k^2 \times t_a$$

- Time complexity to compute the gradient of the image

$$t_{\text{grad}} = h \times w \times t_a$$

- Time complexity of the non-maximum suppression algorithm

$$t_{\text{nms}} = h \times w \times 2 \times t_a$$

Thus, the total time for the sequential version of the Canny edge algorithm is:

$$T_s = 2t_{\text{conv}} + t_{\text{grad}} + t_{\text{nms}} = h \times w \times (2k^2 + 3) \times t_a \tag{3}$$

where t_a is the time for a basic operation (e.g., multiple two numbers, computing square root of sum of two numbers, etc.) and the convolution operation is done 2 times, x-axis and y-axis respectively.

(ii) Parallel execution time
In this part, we do not consider the padded pixels since they are negligible compared to the original width and height, which are few thousands. The approximate time complexities of the whole program on each processor are shown below.

- $t_{\text{comm,i}} = t_{\text{startup}} + (h_{\text{parallel}} \times w)\, t_{\text{data}} + h_{\text{parallel}} \times w \times t_a + t_{\text{startup}} + t_{\text{data}}$
- t_{startup}: is the startup time, or latency
- t_{data}: time to send a sub-image partition.

Thus, the total time for the parallel implementation is:

$$T_P = t_{\text{comp}} + t_{\text{comm}} \tag{4}$$

$$\cong \frac{T_S}{P} + [2t_{\text{startup}} + (h_{\text{parallel}} \times w + 1)\, t_{\text{data}} + h_{\text{parallel}} \times w \times t_a] \tag{5}$$

3.1.2 Experimental results

All of the times measured in the experiments were averaged over 3 repetitions. Two different sizes of kernel for convolution were used to ensure that there is no implementation error in the parallel algorithm.

(i) Total run time w.r.t thd number of processors P.

The raw data for the total amount of time is shown in Table 1.

No. of Processors	1st Run	2nd Run	3rd Run	Average
8	1.4784	1.5352	1.5300	1.51449
7	1.5618	1.5499	1.7322	1.61464
6	1.6840	1.6353	1.6564	1.65857
5	1.7823	1.7578	1.7914	1.77721
4	1.9000	1.9522	1.9362	1.92947
3	2.1933	2.2629	2.3837	2.27997
2	3.1635	3.3087	2.9949	3.15570
1	3.8237	4.2668	4.1766	4.08906

Table 1: Total running time w.r.t. different number of processors. For each number of processors p, we did 3 runs with convolution kernel size $k = 3$

Figure 2 shows the total run time of the sequential (with a single processor) and the parallel (with 2 – 8 processors) algorithms. This measures the amount of run time for data distribution, pre-processing in each processor, Canny edge detection algorithm and writing the sub-image edge output results.

(ii) Local run time w.r.t the number of processors P

The raw data for the local amount of time is shown in Table 2.

This measures the amount of time for pre-processing and Canny edge detection algorithm in each processor. Thus **the communication time is not added here**.

We were able to observe the local computing time decreases almost proportionally as the number of processors increases.

(iii) Parallel speedup and parallel efficiency

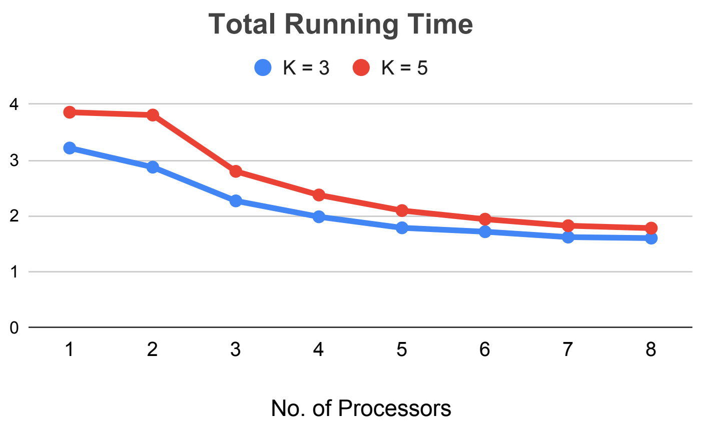


Figure 2: Total running time of the program w.r.t number of processors for two different kernel sizes $K = 3$ and $K = 5$. The $x-$axis is the number of processors and the $y-$axis is the running time, measured in seconds.

No. of Processors	1st Run	2nd Run	3rd Run	Average
8	0.3863	0.3822	0.3687	0.37907
7	0.4249	0.4266	0.4569	0.43613
6	0.5153	0.5158	0.5094	0.51350
5	0.6141	0.6119	0.6421	0.62271
4	0.7812	0.7882	0.7847	0.78472
3	1.0473	1.1732	1.2243	1.14827
2	1.8696	1.9951	1.7728	1.87918
1	3.5890	4.0292	3.9375	3.85189

Table 2: The local running time w.r.t. different number of processors. For each number of processors p, we did 3 runs with convolution kernel size $k = 3$

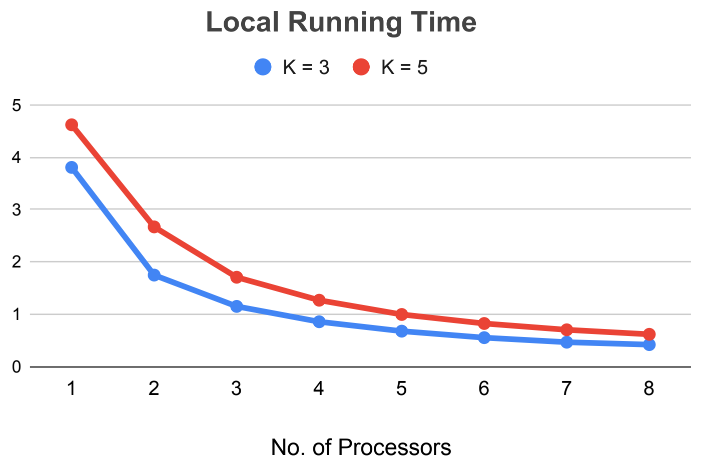


Figure 3: Local running time (i.e., time for the Canny algorithm) w.r.t. number of processors for two different kernel size $K = 3$ and $K = 5$. The $x-$axis is the number of processors and the $y-$axis is the running time, measured in seconds.

For measuring the parallel speedup and the efficiency, the times are measured with the kernel size $k = 3$.

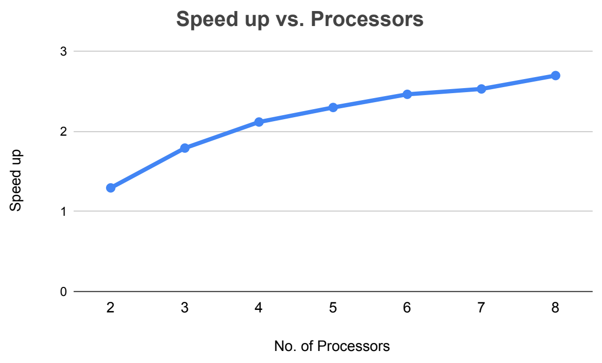
The parallel speedup is

$$S_P = \frac{T_s^*}{T_P} \tag{6}$$

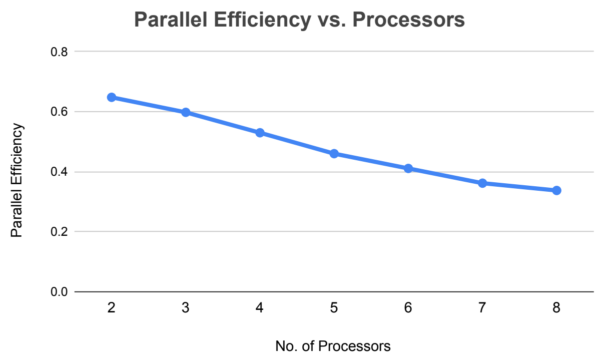
and the parallel efficiency is

$$\eta_P = \frac{S_P}{P} \tag{7}$$

This is the result for the parallel speedup and the efficiency obtained from our experiments.



(a) The parallel speedup along with the number of processors clearly shows that it does not linearly increase.



(b) The parallel efficiency decreases as the number of processors increases. Therefore it implies that many number of processor does not necessarily mean it reduces the run times efficiently.

Figure 4: Speedup and Parallel efficiency w.r.t number of processors

3.2 Examples of output images

Figure 5 shows the detected edges for two example images. As can be seen from the plot, the output edges are of high quality and most of the noise in the input image are removed.

Additionally, as shown on the time plots (Figure [2] [3]) above, it takes slightly more time to detect the edges for the images with bigger size of convolution window (or stencil) k as the number of the arithmetic operations is greater than with smaller size of windows.

4 Discussion

We have observed in the assignments and this project that as the communication time is relatively larger than the local arithmetic operations, if the image is bigger we would have better parallel speedup and parallel efficiency both in theory and practice. In other words, the critical point regarding parallel computing is how long the communication is compared to the total run time of the algorithm. Since the communicative operations between multiple processors are significantly longer than arithmetic operations but necessary at the same time, in some cases the parallelization might not be useful for reducing the run time.

5 Conclusion

While working on this project, we have realized that the parallel computing plays an important role when the workload is in large-scale. If the parallel efficiency is high even with many number of processors, then parallelization will save a lot of time. From choosing the topic to implementing the program, everything has been done on our own pace. This made the project more meaningful and we were able to enjoy it overall.

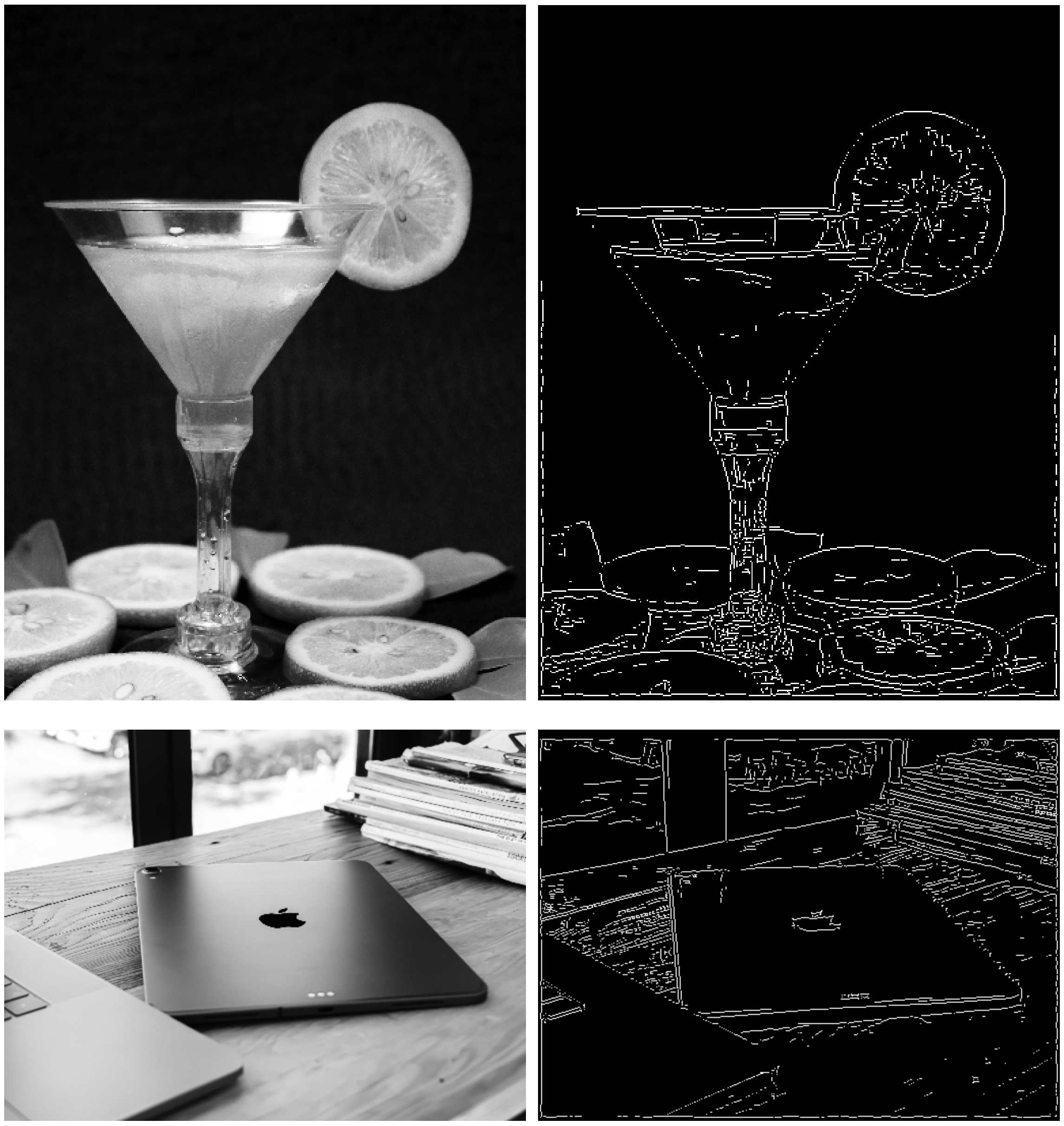


Figure 5: *Left*: Original images; *Right* Edge images obtained by our implementation. Note: for visualization purpose, we enhance the edge image by making the edges thicker so they're more visible in the report. In real edge image, since the edges are only 1-pixel and the dimension of the image is large, it is hard to show the details in this report.

References

[1] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, p. 679–698, Jun. 1986. [Online]. Available: https://doi.org/10.1109/TPAMI.1986.4767851

[2] N. Kanopoulos, N. Vasanthavada, and R. L. Baker, "Design of an image edge detection filter using the sobel operator," *IEEE Journal of solid-state circuits*, vol. 23, no. 2, pp. 358–367, 1988.