Parallel Canny Edge Detector using OpenMPI

Lai Wei Hong, 1226091

***Abstract*—Edge detection is an image processing technique used to identify edges within a given image and is a fundamental tool in computer vision, specifically in feature extraction. In this project, we explored the possibilities of parallelising an edge detection algorithm, Canny edge detector, using a combination of OpenMP and OpenMPI. We will discuss the design decisions made, along with presenting the final architecture of the solution proposed. At the end of the report, we present the performance results of our proposed system against a sequential reference.**

I. Introduction

EDGE detection is a popular technique widely used in computer vision as a means to remove unwanted noise and extract useful features from an image for machine learning tasks. With the advancement of technology, we observe an increase in image resolution used for image classification in various domains, such as cancer imaging. Consequently, this would increase the computational time required for processing these images. However, with minimal literature around parallelizing edge detection algorithms, it begs to question, would parallel edge detectors be worth the effort of implementing? This question serves as the basis for the decisions made in this project.

II. Sequential Code Overview

The edge detection algorithm that we are investigating in this project is the Canny Edge Detector[1], first introduced by John F.Canny in 1986. This is a particularly interesting problem for parallelisation as it is synchronous in nature. The operations in one stage is dependent on the results obtained in the previous stage.

The Canny Edge Detector is split into multiple stages.

A. Stage 1: Image Blurring

As with all computer-vision based feature extraction algorithm, Canny edge detectors are sensitive to noise in an image. In this context, we consider noise to be edges that do not contribute to the main structure of the image. A popular technique used for noise reduction in images is blurring, specifically Gaussian Blur, which is the technique used in Canny edge detectors. We first compute a Gaussian kernel of size (2k+1, 2k+1), where k = 1, 2 .. N using the algorithm shown below. Using this kernel, we apply a convolution operation to every pixel in the image. By the end of the convolution process, we should have a blurry version of the original image.

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

Fig. 1. Formula to generate Gaussian Kernel

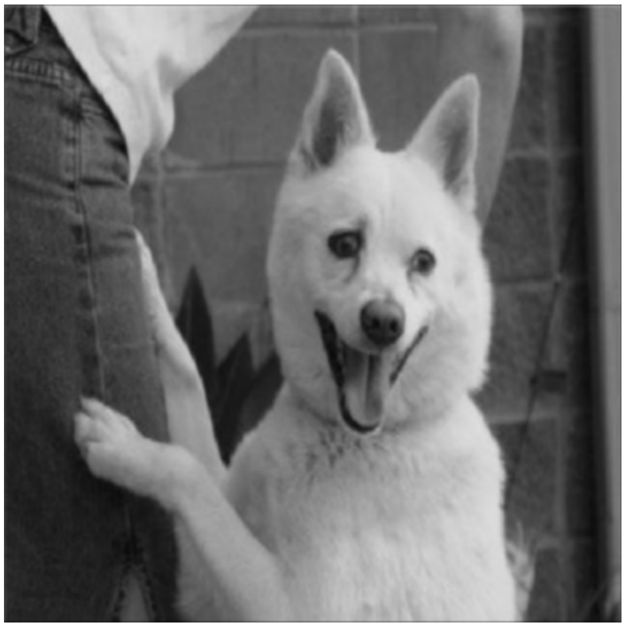

Fig. 2. Blurred sample image

B. Stage 2: Finding the intensity gradients of the image

Once we blurred the input image, we convolve a Sobel Filter throughout the image with the target pixel placed in the centre of the filter to find the gradient intensity and direction for the pixel. The gradient intensity helps to tell us the likelihood of a pixel to be an edge, while the gradient direction informs us of the direction that an edge could take.

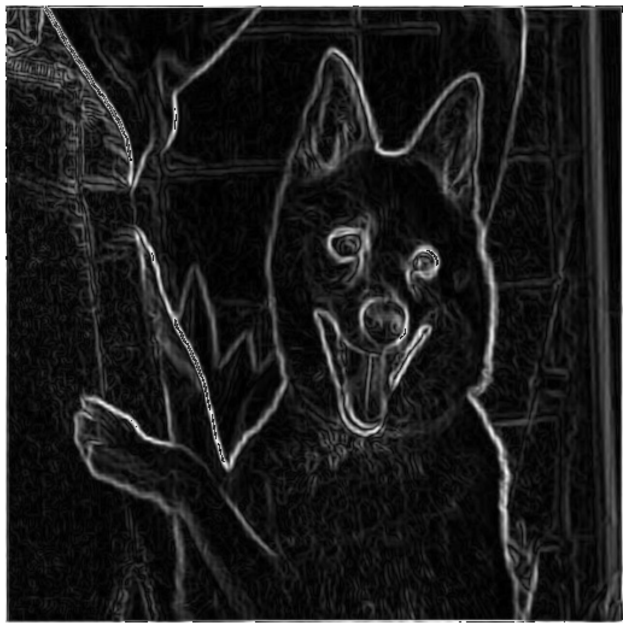
Fig. 3. Gradient intensity of sample image

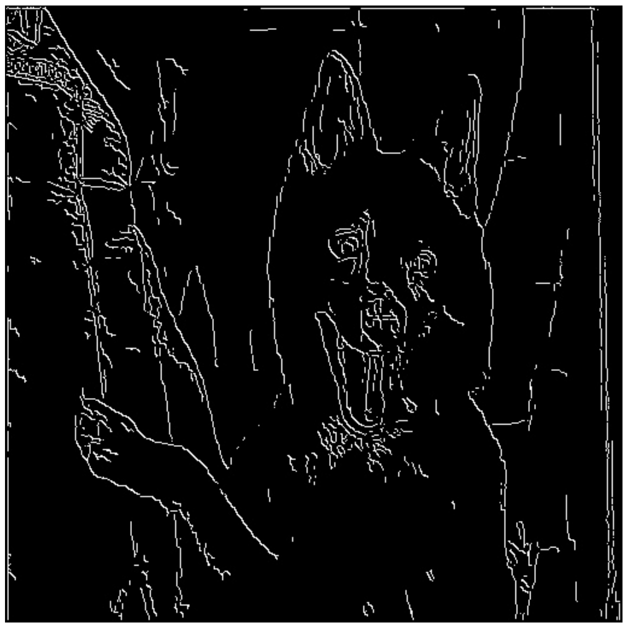
Fig. 4. Edge detected image

C. Stage 3: Apply gradient magnitude thresholding

Using the gradient intensity image, we can start filtering out pixels that are unlikely to be edges. In gradient magnitude thresholding, we compare the gradient intensity of a pixel against pixels at its positive and negative gradient direction. If the gradient intensity of the compared pixel is lower, it is suppressed and given a value of 0. Otherwise, the pixel retains its value. This operation is repeated to all pixels in the image.

D. Stage 4: Apply double thresholding and edge tracking with hysteresis

Now, we should have an image with a good representation of the actual edges. However, noise and different levels of pixel values are still prevalent in the image. With double thresholding and edge tracking, we aim to eliminate as much noise as possible.

With double thresholding, we categorise pixel values in the image into three categories, non-edges, weak edges and strong edges. This is performed by specifying two threshold values, one for weak edges and another for strong edges applied to all the pixels. Following this, we perform edge tracking by hysteresis. In an hysteresis operation, we compare a pixel categorised as a weak edge to its nearest neighbours. If the pixel is neighbours with a strong edge, it transforms into a strong edge. Otherwise, it is suppressed.

III. Parallel Implementation Design Decisions

Edge detection is an interesting problem to parallelize due to the high running time required by the convolution operation, particularly for problems with a large kernel size.

Prior to the development of the parallel solution, a number of design decisions have to be made to ensure that the solution is realistic and meets the performance requirements.

A. Experimentation Setup

When performing initial experiments for the project, the long wait times for jobs with high core count and memory requirement posed difficulty in experimenting and evaluating different implementations, given the the time constraint imposed. As such, we decide to pivot by placing more emphasis in memory management. Hence, in our experiments, we assume that each nodes have specifications similar to a Raspberry PI computer. The benefit of such a setup is that the solution can easily be scaled to incorporate more resources should that be made available.

B. Overlap vs Message Passing

Given the memory constraint, large images have to be partitioned to be processed by different nodes. This presents an interesting problem as convolution operations may rely on pixel values that are in another partition. Here, we consider two possible solutions.

Algorithm 1 Sequential Canny Edge Detector

```
 1: for All pixels p in the image do
 2:     for All surrounding pixels of p do
 3:         Apply image blurring using the Gaussian Kernel
 4:         Apply Sobel filter to get gradient intensity image g_i and gradient direction matrix g_d
 5:     end for
 6: end for
 7: for All pixels p_1 in g_i do
 8:     Retrieve pixels in the positive and negative gradient direction, g_i_p and g_i_n
 9:     if $p\_1 < g\_i\_p$ and $p\_1 < g\_i\_n$ then
10:         $p\_1 \leftarrow 0$
11:     else
12:         p_1 retains its original value
13:     end if
14:     Define a low threshold, t0 and a high threshold, t1
15:     if $p\_1 < t0$ then
16:         $p\_1 \leftarrow 0$
17:     else
18:         if $p\_1 > t0$ and $p\_1 < t1$ then
19:             $p\_1 \leftarrow 25$
20:         else
21:             $p\_1 \leftarrow 255$
22:         end if
23:     end if
24: end for
25: for All pixel p_2 in g_i with pixel value 25 do
26:     Get the highest pixel value, P from its nearest neighbours
27:     if $P == 255$ then
28:         $p\_2 \leftarrow 255$
29:     else
30:         $p\_2 \leftarrow 0$
31:     end if
32: end for
```

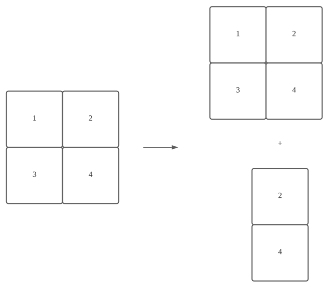

Fig. 5. Example of a overlap partition where the matrix is split columnwise

First, we consider overlapping the partitions by a length of 2k, where k is the k used in generating the Gaussian kernel. This approach loads all the data it needs for the convolution operation in the beginning, eliminates the need to request additional pixel data during the computation operation, which would help reduce the communication overhead required by the nodes in the network. However, a caveat of this approach is the higher memory requirement for every partitions, particularly for case when the size of the gaussian kernel is big.

An alternative is to broadcast the coordinates of the pixels required and gather it using MPI_Broadcast and MPI_Send. This approach is more memory efficient as each node will hold lesser image information at the start and request information that it needs. However, there are serious performance implications to this choice as the process requesting the pixel information needs to wait for all the processors to reach the point MPI_Broadcast is called.

The following diagram is the time taken in microseconds to distribute pixels in the hysteresis operation using row-based image partitioning against the number of pixels exchanged. In this experiment, only pixels classified as strong edges are sent to the node requesting it. From the graph, we can see that the running time of the algorithm increases as the number of pixels transmitted increases. Therefore, it is very likely that the performance will deteriorate even more in convolution operations where more pixels are required to be transmitted.

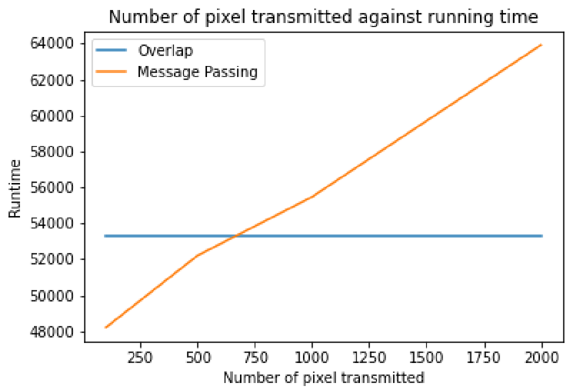

Fig. 6. Graph of Pixel Transmitted against Runtime

C. Master Slave Architecture vs P2P Network

Since it is unrealistic to have all the image data that needs to be processed in memory during program initialisation, there needs to be a strategy towards work allocation among OpenMPI processors.

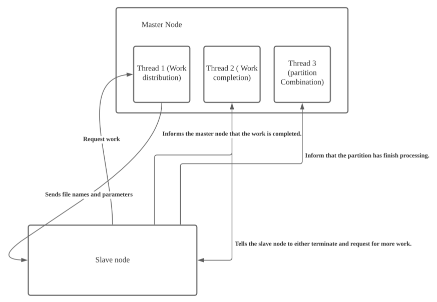
Master Node
Thread 1 (Work distribution)
Thread 2 (Work completion)
Thread 3 (partition Combination)
Request work
Informs the master node that the work is completed.
Inform that the partition has finish processing.
Sends file names and parameters
Tells the slave node to either terminate and request for more work.
Slave node

Fig. 7. Image of the Master Slave Architecture used in this project

One such strategy is the Master-Slave architecture. In this architecture, we define two types of nodes, a master node and slave node. The master node will be responsible for work distribution among all slave nodes. The slave node, on the other hand, simply executes the work once it has receive instructions from the master nodes and terminates when no more work is needed.

Another strategy is to implement a peer-to-peer network. One implementation experimented is a network where each node has a specific functionality. A node working on image blurring can broadcast a request to all the available gradient intensity nodes to pass on the work.

Both architectures have their pros and cons. The advantage of a Master Slave architecture is its simplicity in node management. With work distribution centralised onto a node, very minimal message passing is required to complete the task. However, it is not as scalable as a peer-to-peer architecture. Once the master node becomes the bottleneck, it is not as trivial as adding a new master node. Slave nodes have to be reassigned and communications between master nodes need to be established.

In this project, the Master Slave architecture is chosen due to the low number of nodes used in the experimentation setup and its simplicity. We have also observed significant performance difference between the two architectures.

D. Image partitioning for work division

Finally, we considered the method of partitioning the data chunks. Theoretically, both methods should be able to offer similar performance and return correct results.

However, we went with the row-based partitioning because it had a lower communication overhead when compared to the region-based partitioning when experimenting with the message passing operations outlined in subsection B above. While the message passing operations were eventually dropped, the row-based partitioning remained the de-facto choice in the final implementation.

IV. Overall architecture

A. Image Pre-processing

C++ does not have native image support. Therefore, this step has to be completed in Python. The script converts the jpg image into a Numpy array with each element represent the greyscaled pixel value for the image. This array is then converted to a text file. All the images are either resized to a dimension of 512x512 or 5000x5000 depending on which dimensions are closer to their native dimensions. At the same time, the script generates two lists, bigWorkList.txt and smallWorkList.txt. Each of this list stores the filenames of the image based on the their image dimensions.

B. Edge Detection Procedure

Algorithm 2 Work distribution thread

```
Require: Work queue containing filenames of images that need to be processed, Q, Processor Queue, P
1: Pop an element from the work queue, q
2: Pop p from the Processor Queue, P
3: if q is a large image then
4:     for $i < world_size$ do
5:         $yOffset \leftarrow 5000/world_size * i$
6:         $width \leftarrow 5000$
7:         $height \leftarrow 5000/world_size$
8:         $fileType \leftarrow 1$
9:         $messageTag \leftarrow 3$
10:        $i++$
11:        Param = [yOffset, width, height, fileType, messageTag]
12:        Send Param and Filename to the Slave Node
13:    end for
14: else
15:    $yOffset \leftarrow 50$
16:    $width \leftarrow 512$
17:    $height \leftarrow 512$
18:    $fileType \leftarrow 0$
19:    $messageTag \leftarrow 3$
20:    Param = [yOffset, width, height, fileType, messageTag]
21:    Send Param and Filename to the Slave Node
22: end if
```

Algorithm 3 Slave node reporting thread

```
Require: work queue, Q and processor queue, P
1: if q is empty then
2:     Sends a termination message to all other incoming messages
3: else
4:     pop the slave node back to P
5:     Send a message to the node that there is more work to be done
6: end if
```

We first retrieve the filenames from the two lists generated earlier and place them in a work queue. The master node will

Algorithm 4 Partition combination thread

```
Create a map with filename as key and number of encounters as values
Wait for any message with the message tag 3.
if Message with message tag 3 received then
    update the map with the filename seen in the message
end if
if All the big images have finished execution then
    for All keys, k in the map do
        Read all the temp files for with k as the filename and append into a 2D matrix
        Write the 2D Matrix into file once all temp files are read
        Delete all temp files for k
    end for
end if
```

Algorithm 5 Slave Node

```
Slave Nodes apply the same edge detection algorithm as the one outlined earlier in the report.
```

first process the images with larger dimensions first. For these images, we partition them into chunks based on the number of slave nodes available.

Upon the completion of partitioning, the master node sends the filename, along with a few parameters(width, height, yOffset) to the slave node. The slave node then performs the computation based on the Canny edge detector algorithm earlier. Once the slave image finish processing the partitioned images, it reports back to the master node. The master node, then spawn a thread to keep the results in a temporary file. Once all the partitions were computed, the thread combines all the partitions into a single image. The temporary files are subsequently removed. After reporting to the master node, the slave node is freed to either receive new work or terminate if no work is available.

When all the large images have been processed, the small images will then be placed in the work queue. The computation process is similar to that outlined above with the key exception being the workflow after the slave node reports back to the master node. In this case, the slave node work directly write the results to the file using the original filename to the results folder.

By the end of the program cycle, we will have all the transformed images in the results/ folder.

C. *Image Post-processing*

Here, we have a second Python script that converts text data into a Numpy array. Using the Pillow library, we transform the Numpy array into JPEG images that can be viewed.

V. Experiment

Here, we will be comparing the performance of the proposed solution against the sequential implementation in two different categories. First, we will compare the execution time of both algorithms when executing a single 5000x5000 image. The second experiment is used to compare the execution times used when processing a batch of 200 images with 10% of the batch being large images. An important note is that the time taken for image pre and post processing is not included in the experiment. For our experiment, we will be running it on nodes with 3 computing cores and 1 GB of memory.

A. *Single Image Performance*

Node	Average run time across 3 runs (10^-1s)	
	Sequential	Master-Slave
2	15.8	16.3
4	15.8	9.4
6	15.8	8.5
8	15.8	6.9

TABLE I

Average run time when master-slave is given different number of nodes

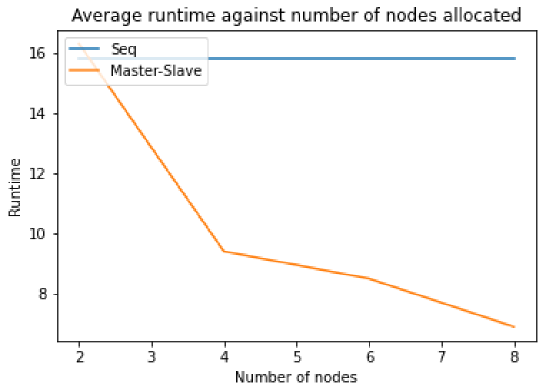

Fig. 8. Runtime against number of nodes

B. Batch Processing Performance

Node	Average run time across 3 runs (s)	
	Sequential	Master-Slave
2	63	75
4	63	39
8	63	25
16	63	21

TABLE II
AVERAGE RUN TIME WHEN MASTER-SLAVE IS GIVEN DIFFERENT NUMBER OF NODES

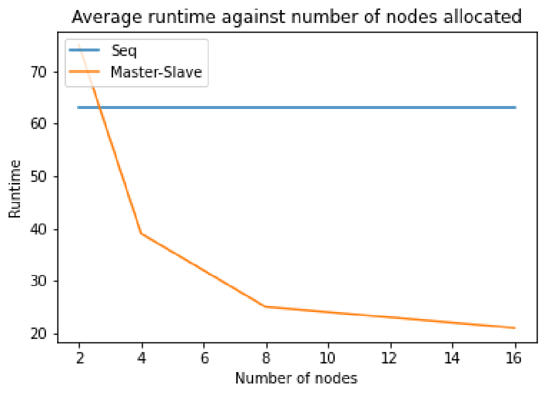

Fig. 9. Average run time when master-slave is given different number of nodes

C. Analysis

Overall, the results fell within the initial expectations. In both experiments, we were able to achieve a fairly respectable speedup compared to the sequential implementation. However, the results from the experiments have also exposed the weakness of the master-slave architecture. As the number of nodes increase, we observe a decrease in the additional performance gained. After further investigation, we observe congestion in thread 1 where work is being allocated. We believe that this happened because of the low processing time in a majority of our image batch, leading to this bottleneck.

While it is important to keep in mind of the resource restriction that we placed on the nodes, it is equally important that we realise that this bottleneck will hold true whenever a Master-Slave architecture is used.

VI. Conclusion

Overall, some level of speedup was achieved when compared to the sequential implementation. We have also uncovered the potential shortcomings of a Master-Slave architecture in cases where there is a high number of processing nodes. On top of that, the earlier observations that demonstrated the performance difference between overlapping the partitions against message passing required information shows us one of the key differences of OpenMPI compared to OpenMP. Threads will always outperform message passing processors. However, message passing is beneficial in giving a program more access to memory required for computation.

As the conclusion, we return back to our original question in the Introduction. Is parallelising edge detectors worth the effort? Based on the results of this experiment, we might suggest that an OpenMP based solution may be more efficient.

References

[1] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986. DOI: 10.1109/tpami.1986.4767851.