Program 4

Canny Edge Detector – MPI Style

Glenn Contreras
g_contreras2@u.pacific.edu
University of the Pacific

Abstract—**The purpose of Program 4 is to finish the implementation of the Canny edge detector, and to use the Open MPI library to show how parallelization can greatly speed up very compute-intensive image processing. We ran the parallel code in the 4-node cluster achieving a performance increase over the serial version by a factor 29 while maintaining a parallel efficiency of over 100%.**

I. Introduction

Program 4 expands on the previous labs by adding the non-maximum suppression, double threshold and hysteresis functions to complete the Cany edge detector algorithm. The suppression algorithm thins the edges of the magnitude image using the phase image lookup. The resulting image shows only a faint outline of the picture and needs by amplified by the double threshold function to clearly show the edges. Finaly, the hysteresis function filters out the pixels that do not have a pixel color of 255, leaving only the white and black pixels that clearly show the edge of the picture.

II. Implementation

A. MPI Parallel Code

The image was first broken up into chunks via the MPI_Scatter method and sent to the corresponding processors. Each processor recieves an equal chunk of the image and proceeds to convolve the image to obtain the horizontal and vertical gradients. However, before convolution, image rows from adjacent processors need to be exchanged to avoid lines in the image. This was achieved by using the MPI_Sendrecv method to pass the neseccary rows into the proccessor's image chunk. Once each processor has the padded image with the ghost rows, it can compute the nececessary images to achieve the final edge image. Using MPI_Gather, the program puts back the image chunks to form the final product shown in figure 1.

III. Results and Analysis

A. Output

The image below shows the output of the Canny edge detector algorithm of a source image of size 4096px. The horizontal lines are due to a faulty implementation of the convolution function.

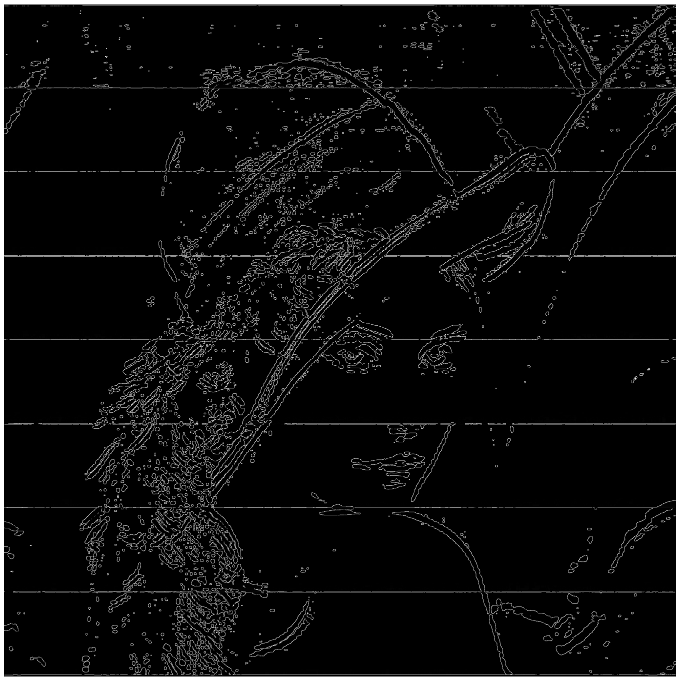

Figure 1: Sample output from a 4096px source image size.

B. Speedup

Table 1 shows the speedups of the program when run in a 4 node configuration on the SOECS cluster using MPI. MPI achieves impressive speedups by running on average 24x faster than the serial implementation when using 32 tasks. The overall speedup average was 14.199. However, a very noticeable drop in performance occurs in with the largest image sized. This could be due to inefficient resources on the nodes.

Image	Tasks=4	Tasks=8	Tasks=16	Tasks=32
1024	3.2100	17.6349	16.6105	31.5277
2048	2.7379	18.2403	29.1634	37.3104
4096	2.9899	18.8220	29.4920	38.5608

7680	3.0374	9.5283	15.8194	20.0642
10240	3.0241	8.3158	14.4642	17.1850
12800	2.9694	0.0137	0.0237	0.0284
AVG	2.9948	12.0925	17.5955	24.1128

Table 1: Speedups with a 4 node MPI configuration.

C. *Efficiency*

Table 2 shows the parallel efficiency of program 4 using a 4 node configuration. From this table, the efficiency seems to peak at 8 tasks giving an average of 1.5. Overall, the parallel efficiency was 1.0284.

Image	Tasks=4	Tasks=8	Tasks=16	Tasks=32
1024	0.8025	2.2044	1.0382	0.9852
2048	0.6845	2.2800	1.8227	1.1660
4096	0.7475	2.3527	1.8433	1.2050
7680	0.7594	1.1910	0.9887	0.6270
10240	0.7560	1.0395	0.9040	0.5370
12800	0.7423	0.0017	0.0015	0.0009
AVG	0.7487	1.5116	1.0997	0.7535

Table 2: Efficiency with a 4 node MPI configuration.

D. *Analysis*

The open MPI implementation grealty sped up the processing by, on average, a factor of 14 with an efficiency of greater than 100% in most cases. In terms of scalability, this program is weakly scalable, since the speedup up tends to decrease with an image size greatr than 4096.

IV. Conclusion

We implemented the Canny edge detection algorithm using the Open MPI libraries to demonstrate how an image processing algorithm can benefit from parallelization. On average, the parallel MPI code was 14 times faster than the serial version while being 100% efficient with its processors. There was a significant drop in performance when processing the 12800x12800px image size possibly due to limited resources on the cluster. Overall, these results justify implementing a serial program in parallel when using any kind of image processing algorithm.