Edge Detection Using Parallel MPI

Identify the Edges in a Given Image As Fast As Possible

Robert Hughes

An Edge detector is the program that is being parlellized. The data will be examined to see if the parallelization techniques proved to work. The parallelization is done using MPI and OpenMP in conjunction with each other.

Image Processing, canny edge detection, MPI, OpenMP

I. Introduction

In our case an edge will be defined as the boundary between two regions. The image we will use use is the Lenna and the regions to draw the edges with can be seen. The detection will isolate the pixels of interest and mark them as edges. This is an important image processing technique because it allows to simplify the image while preserving a lot of the structural elements. What is interesting is that image processing can be found in lots of different fields like pattern recognition. In this particular example of MPI the edge detector that is being used is the Canny Edge Detector.

While the processes of Canny Edge detection can be computationally expensive so in this paper we will be interested in the speedup and scalability of the edge detector using MPI. The paper will first have a brief overview of the canny-edge process, move into MPI and then discuss the results and speedup that was found.

II. Canny Edge Detection

A. Prepping the Image

To start we read in a given image and since we are running this using the MPI we will need to split the image into equal sized chunks and distribute them to each of the processors that we have active. Once this is done we need to distribute the ghost rows. The ghost rows are the rows that are in other chunks but we want access to in order to preform our calculations. In my version of the program before I scatter the image to each o the processors I allocate enough memory for the image chunk and the ghost rows. The chunk is then sent to the middle of this memory and the ghost rows can be added to the top and bottom of the images. Once the chunks and the ghost rows are distributed to each of the processors then the chunks are ready to be processed.

B. Edge Detection

The first step that needs to be preformed is applying a filter to image. This is done by utilizing a Gaussian kernel and the Gaussian Derivative Kernel and convolving them around the image. Once this is finished the Magnitude and Gradient images need to be found. This is done by looking at the intensity change between pixels and by looking at the distance between pixels. The gradient and the magnitude are going to be used in all the steps moving forward so it is important to know that they are correct. The next step in our canny edge detection is the non-maximal suppression this is done by turning the pixel off if they are not the max value in the direction that they are pointing the magnitude and gradient images are referred to in these situations to figure this out. Once this is done we will need to preform the hysteresis on the image, this is probably the most important step in the entire process. It sets the pixels that are not off to be either a strong edge or a weak edge. It determines this by taking the sorted list of pixel intensity and the value at 90% of the number of pixels will be what makes a strong edge. So if the intensity is above that value it will be turned on to 255. If the intensity is below this value and above 20% of this then the edge is determined to be weak and it will be set to 125. In our implementation we used the qsort the original goal was to do a parallel merge sort but I was unable to fix the segmentation faults. I was trying to utilize a merge sort that was written in the Comp 137 class and but do to running out of time and the segmentation faults I was unable to adapt it to the program in time. This is definitely something that I want to work on in the future. Finally the edges nee to be linked through the edge lining process this is done by retaining all the strong edges and then the weak edges that are neighbors to the strong edges. After this is complete the edges image should be ready to be out put to the user.

III. MPI

MPI stands for Message Passing Interface. This is due to the communication that needs to occur between the different processors. We will be using MPI in C and using the ECS cluster that UOP has to offer. This section will give a brief overview of the functions that we will be using.

A. Functions Used In MPI

The first thing that we need to do is break up the image and send the chunks to each to the processors. So the first thing we need to do is us the MPI_Scatter this will evenly split up the image and send it to the various different processors that we will be using. Another message passing technique that we are going to be using is MPI_Bcast this is useful to send data to all of the processors so in the case of our program we wan to broadcast the size of the image to all the processors. Then within the functions we need to use MPI_Send, MPI_Recv, and MPI_SendRecv these will be used to send the ghost rows to each other the ghost rows are a very important part because without them the function will not work correctly. Finally the last message passing that we will need is the MPI_Gather this is useful to get all the pieces back together so that the final images can be printed out. MPI has many other functions that can be used for other purposes but for us all we need to worry about is the message passing at the beginning.

IV Results

Now it is time to get to important part, what was the improvement from the serial run. All the Data can be found in the tables proceeding this section on the last pages of the report. All the times are reported in Seconds. And the sigma value that was used to obtain the Gaussian information was 1.25. In table 1 we can see the execution

times of the MPI program this is promising as they seem to decrease as we increase the number of processors but let us take a closer look at this. By looking at the speed up which is the serial time divided by the parallel time we can learn a lot so lets compare the serial time to the 32 processes and see what kind of speed up was achieved. If we look at table 4 which can be found at the end of the document we see that there is speed up across the processor. This is even easier to see in the Figure 2 where the tabulated data is graphed. Due to the linearity of the system we can also say that this is a pretty efficient system. Efficiency is calculate by looking at the speedup over the number of processor which is what the graph tells us and the more linear the the more efficient. However what was interesting is when we tried to run the system using both MPI and OpenMP. I found that it did not help to increase the time and when the thread count reached a certain level it actually started to make the performance worse. I think this has to do with a mishap in using the OpenMP as I believe it should make it faster.

The only other explanation is that there were more threads created than the processor can handle creating some kind of issue. The other thing that needs to be noted is the couple of drops that exist in the efficiency which I believe can be chalked up to having the communication out weighing the data size so when the image was too small the cost and overhead to send the data was higher than needed and created more time than we could make up for.

V Conclusion

This is a good candidate for MPI as sen by the speedup graph, I think future work can be done to further increase the efficiency like the inclusion of a parallel merge sort. I also believe that if the OpenMP is worked with more we can find a solution to make that speed up the calculations as well.

	Comm Size	1	2	4	8	16	32
Image Size							
1024		1.015	.372	.1876	.1013	.097	.117
2048		9.66	1.374	.7225	.388	.4125	.2657
4096		25.876	5.52	2.903	1.501	1.22	.929
7680		45.2895	19.22	9.92	5.279	3.5006	3.165
10240		121.4895	34.39	18.06	9.43	6.39	5.404
12800		190.7055	53.89	27.707	14.598	10.37	8.39

Table 1: MPI Run Times

10240						
	Num_threads	2	4	8	16	32
Comm_size						
2		35.877	35.84	35.74	36.505	35.5
4		17.989	18.114	18.002	17.927	17.968
8		8.8349	8.66	8.893	8.801	8.703
16		9.174	8.544	8.746	8.70	8.59
32		9.194	10.40	8.938	9.98	9.27

Table 2 OpenMP and MPI 10240

12800						
	Num_threads	2	4	8	16	32
Comm_size						
2		55.92	56.204	56.886	56.443	55.904
4		29.002	28.89	28.285	28.88	28.65
8		14.614	14.1527	14.12	13.98	14.93
16		14.32	14.72	14.24	13.747	13.5
32		14.79	16.128	18.025	18.828	15.32

Table 3 OpenMP and MPI 12800

Speed Up							
	Image Size	1024	2048	4096	7680	10240	12800
Processes							
2		2.96	7.05	4.69	2.36	3.53	3.6
4		5.41	13.37	8.91	4.57	6.75	6.89
8		10.02	24.9	17.25	8.58	12.93	13.07
16		10.46	23.42	21.21	13.09	19.01	18.39
32		8.68	36.36	27.85	14.33	22.5	22.73

Table 4: Speed Up

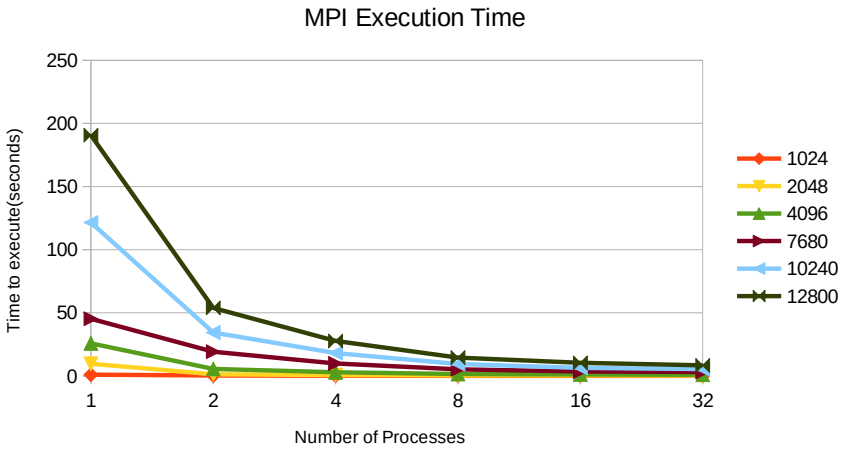

Figure 1: MPI Execution Time

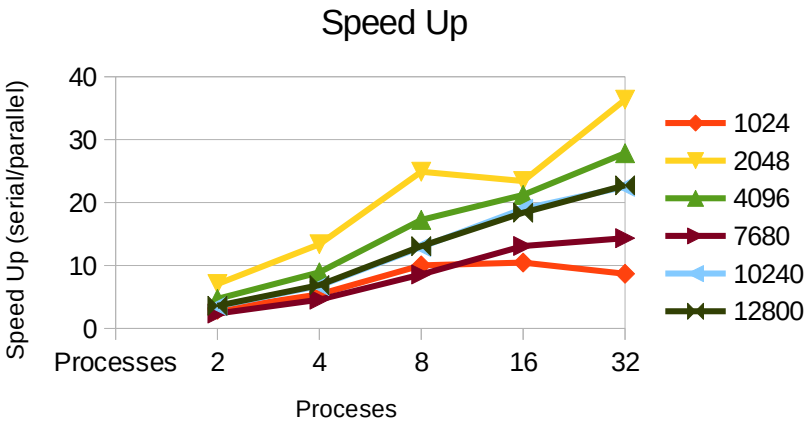

Figure 2: Speed UP