University of the Pacific

Lab Report

MPI: Canny Edge Detection

Gary R. Roberts, Jr.

g_roberts@u.pacific.edu

Edited: April 2, 2017

1 Abstract

The purpose of this lab is to compare the performance difference between MPI processes and serial execution. In order to explore this, an algorithm for suppression, hysteresis and edge detection will be designed using serial logic and then parallelized using MPI processes.

2 Introduction

MPI provides a robust library of tools for parallelizing algorithms for large scale computing. MPI uses message passing to parallelize the execution of an algorithm across many machines. The distributed memory architecture of MPI allows for great performance but relies on efficient optimization of memory management through message passing. The goal of this project is to deduce the usability and identify the potential performance caveats when using MPI. Both serial and MPI methodologies will be used to generate suppression, hysteresis and edges images which can be seen in figure one below.

(a) Source

(b) Suppression Image

(c) Hysteresis Image

(d) Edges Image

Figure 1: Examples of Source and Processed Images

3 Methodology

3.1 Serial

The serial implementation utilizes the sequential memory addresses to allow the images to be accessed in place using offsets.

1. Suppression
   (a) The suppression algorithm accepts the previously generated phase image and searches local pixels.
   (b) If the pixel is connected to another higher value pixel it is culled.
2. Hysteresis
   (a) The hysteresis image accepts the previously generated suppression image.
   (b) Upper and lower thresholds are computed as filters. The upper threshold is generated by computing the 95th percentile of pixel values in the image. This is done using qsort. The lower threshold is computed by dividing the upper value by 5.
3. Edges
   (a) The edges image accepts the hysteresis image.
   (b) The local pixels are checked and if a pixel has a value of 255, the pixel is illuminated, otherwise it is culled.

3.2 MPI

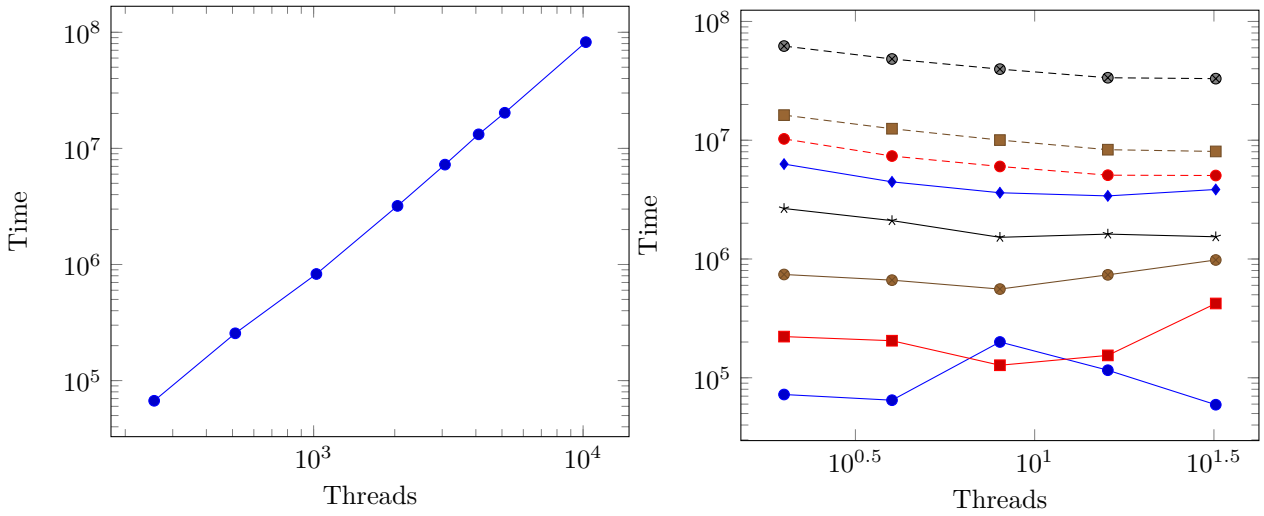
The MPI implementation differs from the serial implementation because it the image into p parts, where p represents the number of MPI processes. This chunks are distributed to each process using MPI_Scatter. The height and width values are broadcast from node 0. Before each convolution, the processes pass their ghost rows using MPI_SendRecv in a function called `sr_ghost()`. After each intermediary process is completed, MPI_Gather is used to collect the completed image by node 0.

4 Results

4.1 Performance

The serial and MPI implementations see a significant difference in their execution times. The MPI implementation experiences a longer execution time with less nodes, particularly with larger image sizes. The

serial algorithm exhibits a linear relationship with image size as seen in figure 1a. Figure two exhibits a break in this relationship with MPI for smaller images. This can be attributed to the message passing overhead involved with a greater number of processes. It may also be a product of a small sample size.



(a) Serial Execution time vs. Image Size

(b) Execution Time vs. MPI Processes

Figure 2: Execution Time vs. Num Threads

Speedup for MPI appears to be consistently linear until it plateaus at 32 processes, with more processes exhibiting a greater speedup. This identifies that the algorithm is strongly scalable but is dependent on the problem size being larger than parallel overhead. The algorithm also weakly scales, as the same number pixels per thread with an increasing number of threads results in speedup.

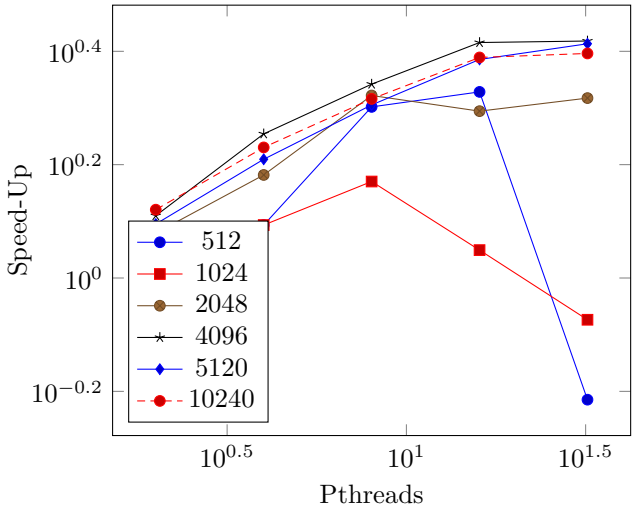


Figure 3: Speed-Up vs. Number of MPI Processes

5 Conclusion

MPI appears to be a very strong solution for large problems sets. Using images with a size greater than 1024, the speedup is consistently significant. Images of smaller sizes are too small of a problem size to be feasible for processing on a distributed architecture, as the message passing overhead begins to cost more than the computation itself.

Size	**p**	Ψ	ϵ
256	8	0.334759	0.041845
256	16	0.578757	0.036172
256	2	0.926895	0.463448
256	4	1.034820	0.258705
256	32	1.129885	0.035309
512	8	2.004284	0.250535
512	16	1.655363	0.103460
512	2	1.150006	0.575003
512	4	1.245717	0.311429
512	32	0.605037	0.018907
3072	8	2.006661	0.250833
3072	16	2.133237	0.133327
3072	2	1.150778	0.575389
3072	4	1.624407	0.406102
3072	32	1.882949	0.058842
4096	8	2.198145	0.274768
4096	16	2.603025	0.162689
4096	2	1.288091	0.644045
4096	4	1.796207	0.449052
4096	32	2.619307	0.081853

Size	**p**	Ψ	ϵ
5120	8	2.020678	0.252585
5120	16	2.435462	0.152216
5120	2	1.245812	0.622906
5120	4	1.621581	0.405395
5120	32	2.518844	0.078714
1024	8	1.484237	0.185530
1024	16	1.126006	0.070375
1024	2	1.118244	0.559122
1024	4	1.248191	0.312048
1024	32	0.844301	0.026384
10240	8	2.073482	0.259185
10240	16	2.450351	0.153147
10240	2	1.325959	0.662980
10240	4	1.703219	0.425805
10240	32	2.493489	0.077922
2048	8	2.099544	0.262443
2048	16	1.972752	0.123297
2048	2	1.201091	0.600545
2048	4	1.518516	0.379629
2048	32	2.077820	0.064932

Table 1: Observed Efficiency and Speedup