Ankit Tandon (at24473)
Suvamsh Shivaprasad (ss56236)

Parallel Edge Detection

Suvamsh Shivaprasad & Ankit Tandon
University of Texas at Austin

***ABSTRACT* - Our project is to parallelize the sobel edge detection algorithm. Sobel is one of the most widely used edge detection algorithms in fields like Computer Vision, Image processing, etc. Our experiment compares the execution time of the serial and parallel implementation of this algorithm. We utilized OpenMP to take full advantage of the hardware present at TACC. To benchmark our results we ran our sobel filter on hundreds of images at once. The final result was a massive (~12x) speedup on average after parallelizing our code, though there were some very random discrepancies discussed in detail below. Sobel is an extremely ideal algorithm to parallelize because not only is it a very tedious but also it is also very widespread. Quick edge detection can be very applicable to a field like robotics where traversing an environment is heavily dependent on understanding objects in the surrounding area.**

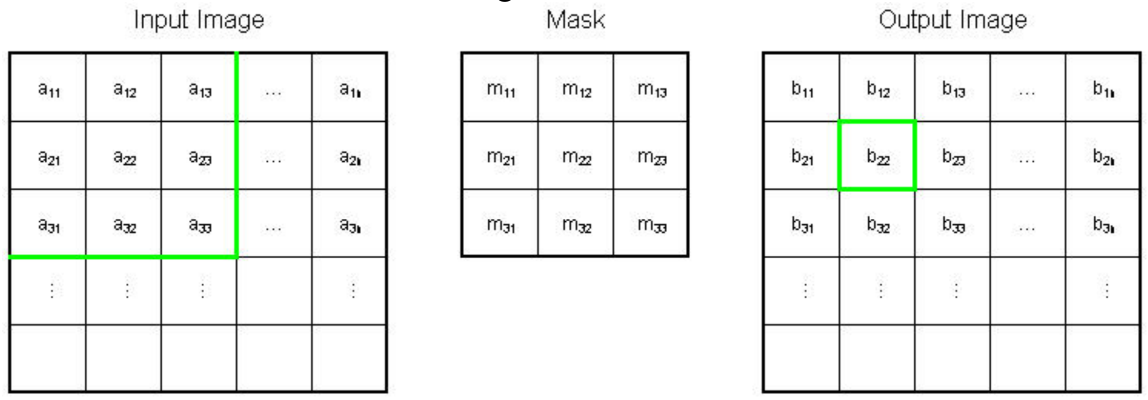
Image 1

Image 3

Image 2

Ankit Tandon (at24473)
Suvamsh Shivaprasad (ss56236)

The sobel operator is a discrete differentiation operator used to detect changes in pixels that might indicate the presence of an edge. There are 2 standard sobel kernels: one is a vertical and the other is horizontal. The first step of sobel is to use both kernels as a mask and go pixel by pixel across the input image to calculate the corresponding output pixel (the formula to calculate the new pixel can be found at the bottom of figure 1). Since we have an RGB value at each pixel we iterate through the input image using each filter 3 times and average the 3 results. So we now have 2 new output images: one created using the horizontal kernel and the other created using the vertical kernel. Next we take the square root of the sum of horizontal gradient squared and vertical gradient squared to calculate the new pixel intensity. Finally we compare all the new pixels to see if they are past a threshold, which will indicate an edge.


Input Image
Mask
Output Image
a_{11} a_{12} a_{13} … a_{1n}
a_{21} a_{22} a_{23} … a_{2n}
a_{31} a_{32} a_{33} … a_{3n}
m_{11} m_{12} m_{13}
m_{21} m_{22} m_{23}
m_{31} m_{32} m_{33}
b_{11} b_{12} b_{13} … b_{1n}
b_{21} b_{22} b_{23} … b_{2n}
b_{31} b_{32} b_{33} … b_{3n}

$$b_{22} = (a_{11}*m_{11}) + (a_{12}*m_{12}) + (a_{13}*m_{13}) + (a_{21}*m_{21}) + (a_{21}*m_{21}) + (a22*m_{22}) + (a_{23}*m_{23}) + (a_{31}*m_{31}) + (a_{32}*m_{32}) + (a_{33}*m_{33})$$

Figure 1 – Sliding across pixels

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 2 - Two sobel kernels (masks)

The application consists of pre-processing followed by three sequential parts:

Pre-processing ⇒ Read image ⟶ Apply sobel ⟶ Write image

There are two major portions where the application can be potentially parallelized:
- Parallelize the sobel filter itself
- Parallelize in such a way that each stage is run in parallel with respect to each image

Each of these parts must occur sequentially so we parallelized them both individually. We realized that parallelizing the entire sobel filter algorithm was slower than parallelizing the each stage of the algorithm separately, probably due to thrashing.

0. Pre-processing: The images are set with their source and output file names.

1. Read data(read_bmp): As bmp is a standard format we assume all bmp meta data is the same. We read all the required information and store it in its respective structure. Error checking is also done for safety of the application. Also the required space for each buffer is allocated in this function.

2. Apply Sobel(sobel): The main function in the application that actually applies the sobel filter to images. Though this function will walk through each and every pixel applying the sobel mask, it is not parallelized. The application could be made more efficient by correctly parallelizing this function.

3. Write processed data(write_bmp): All of the new image's data should be computed by this point. Using the bmp standard we will write all the image data to file.

As we were running in a 12 core machine(See Hardware Overview) we spawn 12 threads to execute each of the three stages individually. The structure of parallelism is as follows:
- 12 threads
- chunksize of 12

RESULT DATA:

FOR 1440 IMAGES:

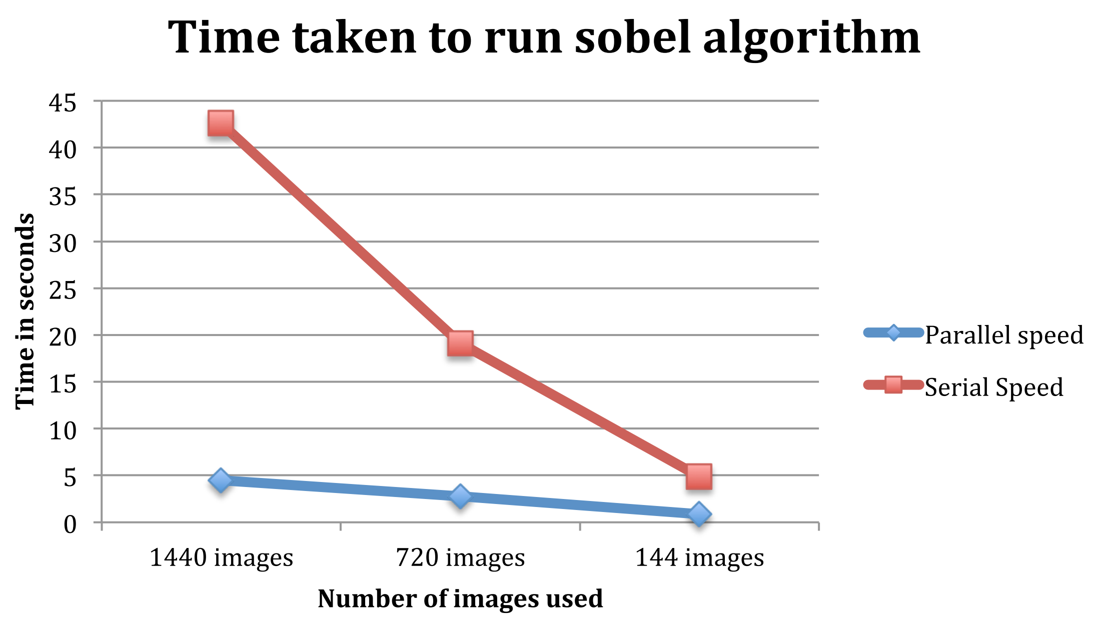
	Parallel time taken	Serial Time Taken
Read	0.421289 seconds	0.763061 seconds
Sobel	3.391279 seconds	35.562281 seconds
Write	0.705027 seconds	6.319146 seconds
Total	4.517595 seconds	42.64448 seconds

FOR 720 IMAGES:

	Parallel time taken	Serial Time Taken
Read	0.296568 seconds	0.376958 seconds
Sobel	1.768975 seconds	18.006772 seconds
Write	0.685290 seconds	0.703113 seconds
Total	2.750833 seconds	19.086843 seconds

FOR 144 IMAGES:

	Parallel time taken	Serial Time Taken
Read	0.102926 seconds	0.080688 seconds
Sobel	0.637327 seconds	3.956713 seconds
Write	0.134644 seconds	0.113800 seconds
Total	0.874897 seconds	4.877393 seconds



SPEED METRICS:

Speedup = $\frac{Ts}{Tp}$
Ts - Serial execution time
Tp - Parallel Execution time

Karp-Flatt Metric = $\frac{\frac{1}{y}-\frac{1}{p}}{1-\frac{1}{p}}$
y - speedup
p - number of processors

The smaller the value of the Karp-Flatt metric, better the parallelization. The expected speedup is ~12x because we are running on a 12-core machine. From our calculations one can see that the speedup is closer to ~11x. Also we can conclude that our parallelism is working correctly because Karp-Flatt metric is scaling according to the data set.

	Speedup	Karp-Flatt Metric
144 images	6.208	0.0846607
720 images	10.179	0.01614662
1440 images	10.468	0.013048221

The read times are consistent to <1 second. The write time is very random and erratic over a number of experiments. Write times vary anywhere from 0.5 seconds to almost 20 seconds. We believe that this is due to thrashing because our virtual memory system is in a constant state of jumping around exchanging data. Our data shows a ~12x time speedup from serial to parallel for the sobel filter.We can also see that this approximately scales according to the number of images processed. Therefore we are very glad to see that we are able to linearly scale. Our main goal when beginning this project was to speedup the sobel algorithm itself and we successfully achieved that. As an added bonus we were able to speed up the read/write process too!

HARDWARE OVERVIEW:

This application was developed on Lonestar Supercomputer at the Texas Advanced Computing Center (TACC). The Lonestar Linux Cluster consists of 1,888 compute nodes, with two 6-Core processors per node, for a total of 22,656 cores. It is configured with 44 TB of total memory and 276TB of local disk space. The theoretical peak compute performance is 302 TFLOPS. The system supports a 1PB global, parallel file storage, managed by the Lustre file system. Nodes are interconnected with InfiniBand technology in a fat-tree topology with a 40Gbit/sec point-to-point bandwidth. The Clos fat tree topology is formed by 648 port Mellanox IS5600 switches and 16-port endpoint switches in each chassis.
The application was run on a typical compute node. The compute specifications are shown below:

Component	Technology
Sockets per Node/Cores per Socket	2/6
Memory Per Node	24GB 6x4G 3 channels DDR3-1333MHz
Processor Interconnect	2x QPI 6.4 GT/s
PCI Express	36 lanes, Gen 2
146GB Disk	10K RPM SAS-SATA

SOURCE:

https://github.com/suvamsh/sobel_filter