Parallelization of an Edge Detection Method to Mark Shorelines from Remote Imaging Data

By: Peter Stein, Evan Burger, and Brandon Wilson

Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County

May 11th, 2021

Abstract

Thesis Statement

The globe is constantly changing, and with rising sea levels, increasing development, and sediment deposition it is a difficult task to keep accurate records of shorelines, especially in remote locations. This information is important for various fields, from navigation to ecology to land use (Al Mansoori and Al-Marzouqi [1]). Thanks to satellite imagery and aerial photography, there is a large quantity of fairly up-to-date data on what shorelines look like, with much available from the United States Geological Survey [8], which can be harnessed through machine learning to autonomously navigate boats, track erosion. The issue is that the shorelines in this data are not labeled. To apply machine learning to such data, the first major phase is feature detection and data labelling, which is often time-consuming. Our work focuses on developing a parallel algorithm to detect shorelines, which could be used as input to machine learning algorithms suited to specific applications. While such a shoreline-labeling program could be serial and have many copies run on different data simultaneously, making each instance itself run in parallel may more effectively use the hardware available. The goal of this project is to learn if parallelizing a shoreline labelling program will enable it to run faster than the serial version, given that both types of programs can have multiple instances running at once on the program level.

More resources would be required for studying shorelines globally. Available time on powerful computers designed for running many calculations on large data sets is limited, and on smaller computers working with so much data can lead to incredibly long execution times, so a study of parallelizing this application is important.

Results Summary

Of the three implementations, the parallel loading and saving versions were able to show good scalability, one on smaller images, one on the larger satellite images. Decomposing on the file level results in a speedup factor of two or greater over the serial version. However, the data decomposition version did not show a significant performance increase over the serial version. There were also two strategies for data preparation, blurring then a static threshold, or blurring then a dynamic threshold. The dynamic threshold was better at distinguishing the shoreline on the smaller colored images, but showed too much noise on the land in the grey satellite images.

The static threshold needed to be tailored to the dataset, so it was a good match for the satellite data, and worked well even with the data decomposition. It could not be used for the images not in the satellite data set, as it would not detect any edges.

Introduction

Background

Edge detection is an older problem in computer science, stemming from computer vision. In 1965, Roberts [6] was already working on this problem, but it wasn't until 1986 that Canny published the algorithm for which our work will be based on [2].This algorithm uses differences in color in an image to determine where edges should be. It has previously been used to study shoreline change in 2016 by Al Mansoori and Al-Marzouqi [1] and in 2020 by Vaccaro [9], and to study cloud types in 2013 by Dim and Takamura [3]. The shoreline studied by Al Mansoori and Al-Marzouqi consisted of 16 images [1], while the study by Vaccaro had only 2 [9]. These were studies focussed on specific populated areas. If a comprehensive study were to be done on larger areas of coastline requiring more images, performance would be more of a concern. Vaccaro mentioned his algorithm's performance (in speed) could be improved using parallelism [9], while Al Mansoori and Al-Marzouqi did not mention performance. Dim and Takamura used data that covered the globe, but the resolution of their data was smaller (and listed as one of the issues they encountered) and they didn't explain how they handled performance [3]. Thus a study of the performance of parallel versus serial methods of shoreline edge detection is of importance.

Project Question

How much will parallelization of edge detection in an image dataset improve performance?

Program Process and Parallelization

Program Environment Setup

Our implementation is written in python and utilizes the following libraries: OpenCV for image processing, Numpy for linear algebra and matrix operations, Threading and Multiprocessing for parallelization, Time for measurement, and OS for file I/O.

Data Preparation

The program takes in the images in grayscale before a 13x13 Gaussian blur filter and binary threshold operation is applied to each image in the dataset. The threshold operation

searches for pixels that exceed an arbitrary grayscale value limit to discriminate between land and water. The threshold operation uses binary thresholding, and otsu thresholding depending on the implementation, to discriminate between land and water, then sets that pixel to 1 or 0. Otsu thresholding finds a threshold value between two peaks in the pixel value distribution so that their within-class variances are minimal. The output of Otsu thresholding has shown it is impactful in finding edges for aerial images when used in combination with binary thresholding. Afterwards, the Canny algorithm is applied onto each image to output edge detection results.

Two Implementations

In the early stages of program development, we found that any runtime improvement from parallelization wouldn't be noticed due to the size of some images. In response, we created two distinct parallel methods. The first method uses one processor to apply this algorithm to an image. The second method applies the algorithm with only binary thresholding and uses several processors on segments in large satellite imagery. The second method was made specifically to address the image size issue that the first method couldn't overcome. The difference in thresholding operations also reveal interesting insight; satellite imagery does well without dynamic thresholding while aerial imagery needs dynamic thresholding, like Otsu thresholding, for better results.

Results and Analysis

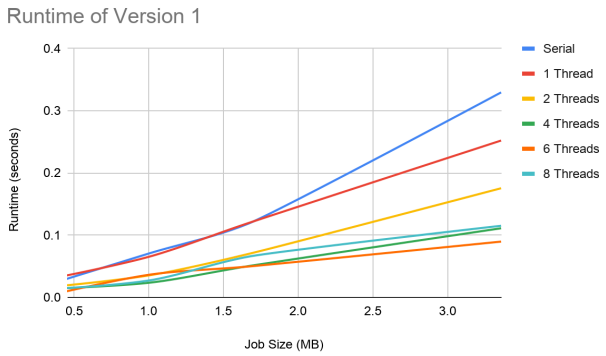
Results

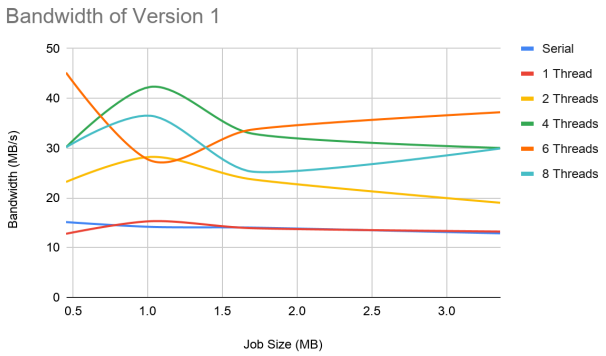
We were able to gain insight by measuring the bandwidth and runtime for each of our trials. Measurements were made by increasing the number of threads up to eight and increasing job size for datasets of both smaller aerial images and large satellite images. Then we analyzed the runtime and bandwidth to learn about the algorithm's performance. Then, the images were visually analyzed to determine the accuracy of our approaches. The importation of the image dataset, the algorithm, and the exportation of the dataset to a separate directory were timed for each test.

Performance Analysis (Parallel v. Serial)

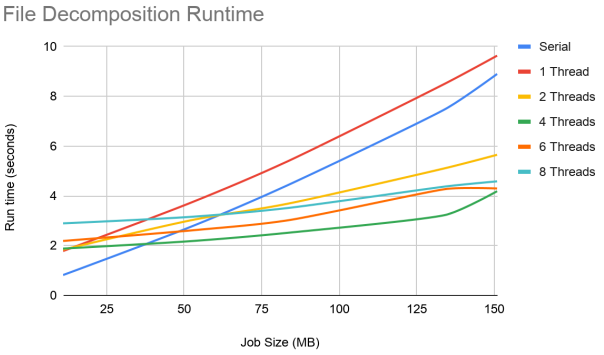
This setup utilized the threading library in Python to implement Pthreading. The first dataset used many small images; each processor took in an image and applied the preprocessing

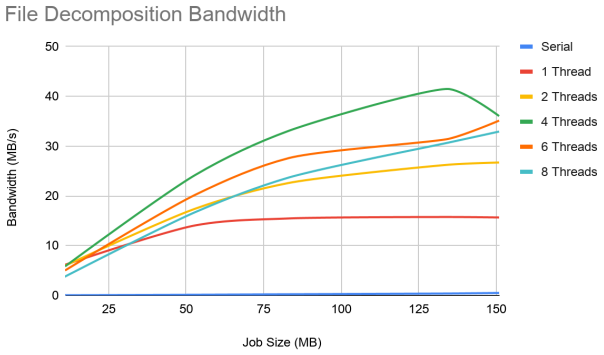
and edge detection algorithms then the runtime (left) and bandwidth (right) were recorded and plotted for varying job size at a set number of processors. Our results expectedly show that increasing job size increases runtime and increasing the number of processors lower runtime.





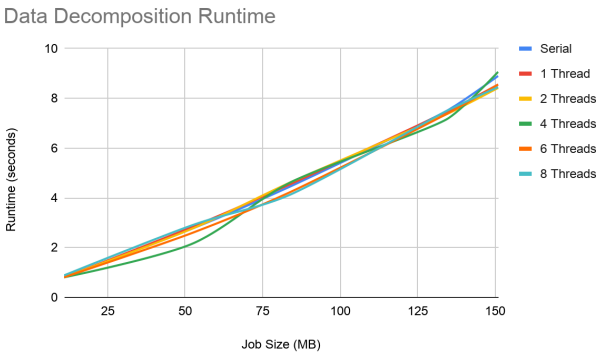
We also used Python's multiprocessing module on a dataset of higher resolution satellite images and got the same effect as before. So, both our parallel model scales well for small or large image datasets of varying sizes. Also, both threading modules are effective in decomposing datasets between processors.

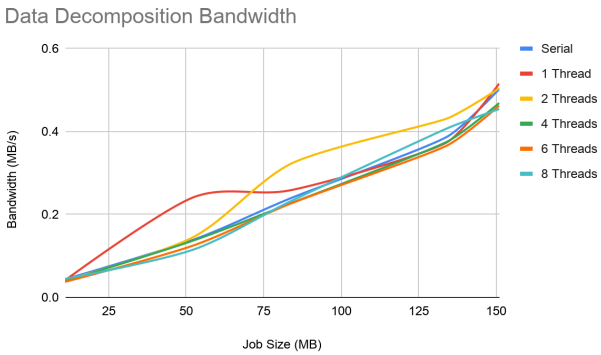




We used Python's threading library to implement our other method, decomposing a single image among many processors. Each image was decomposed by segmenting them by rows of pixels. For example, a processor could be responsible for the top 10% of the image. We expected a performance increase in this case too, but our results ran contrary to that expectation. Runtime still increases with the job size, but parallelization has no impact. This dataset was gathered using satellite imagery, so we initially thought this was due to hardware limitations from processing large images, however, further experimentation showed that this impact is seen when used on lower resolution images as well. Alternatively, it could be due to the effects of

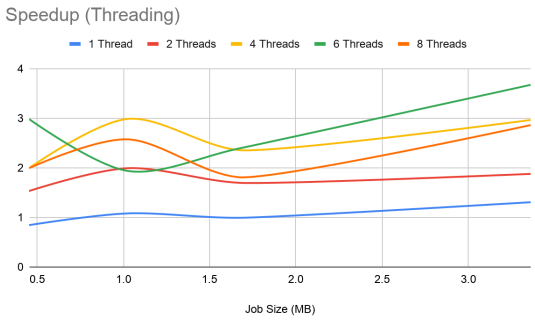
optimization in the imported Python libraries; so the difference in size between the whole image and each segment is not significant enough to reduce runtime.

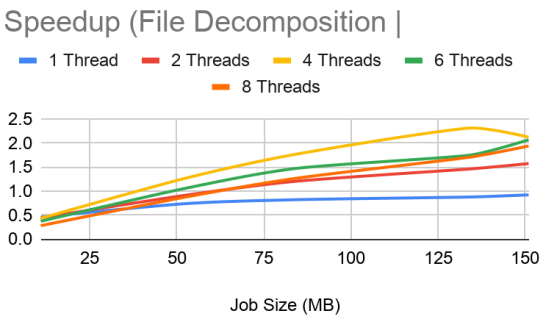




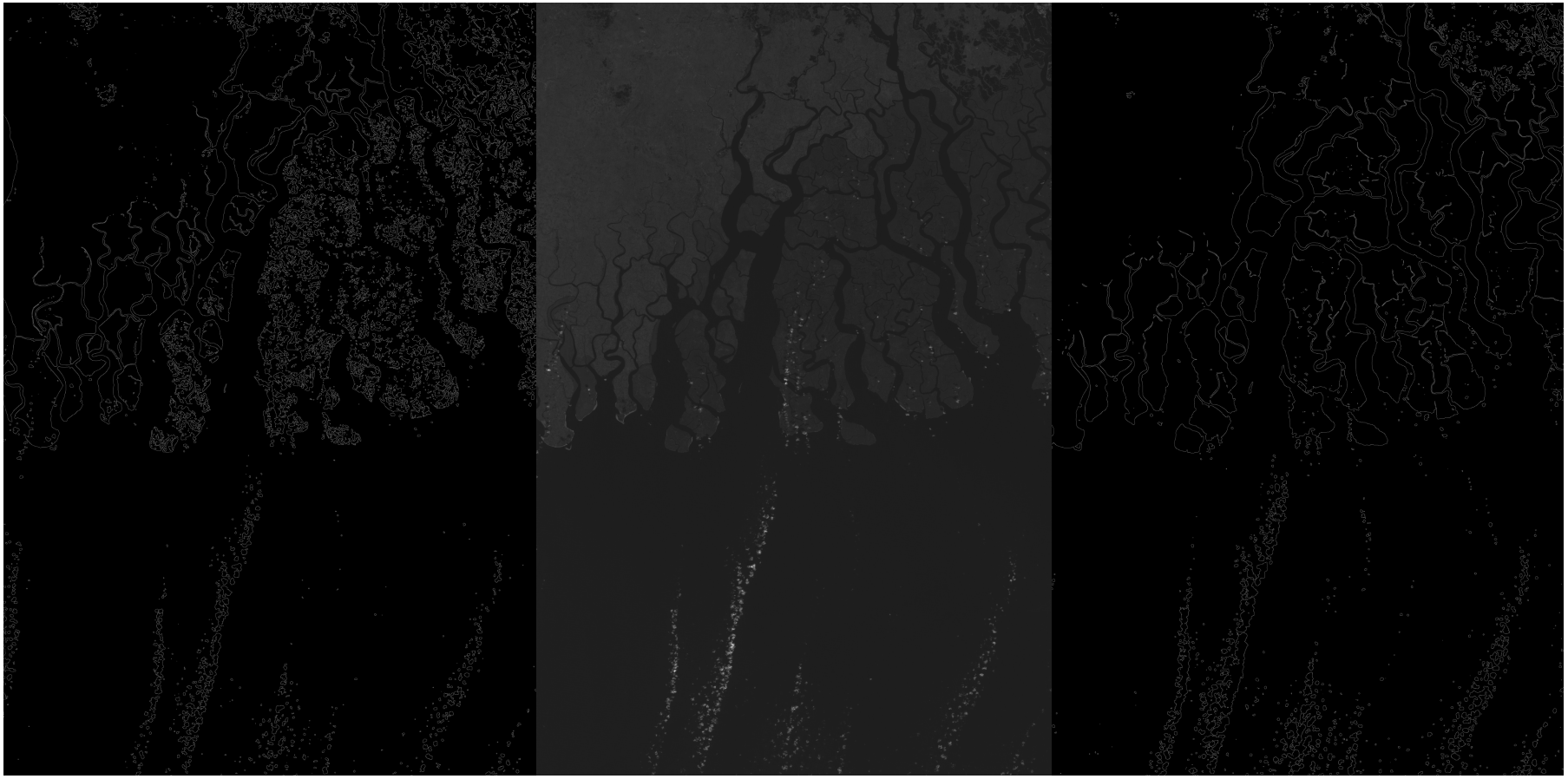
Speedup Analysis

We see speed increase by a factor of three or higher using the threading module (left) and increase by a factor of two or higher using the multiprocessing module (right). The threading module was used on small images while the multiprocessing module was used on large satellite imagery, but both used file level decomposition. The difference between both methods could be explained by the average image size difference in both datasets, so both modules hardly have any differences between them performance wise.

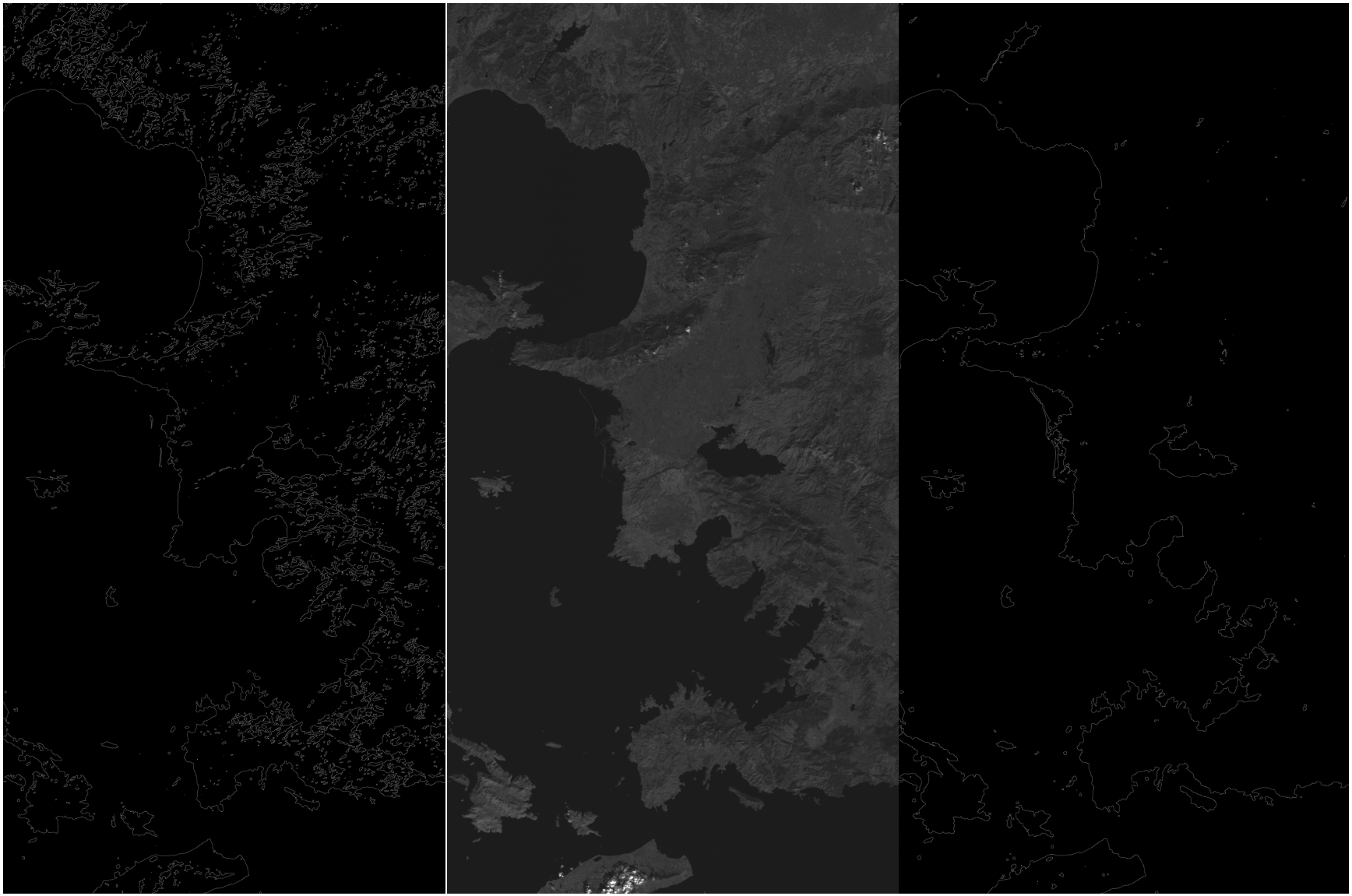




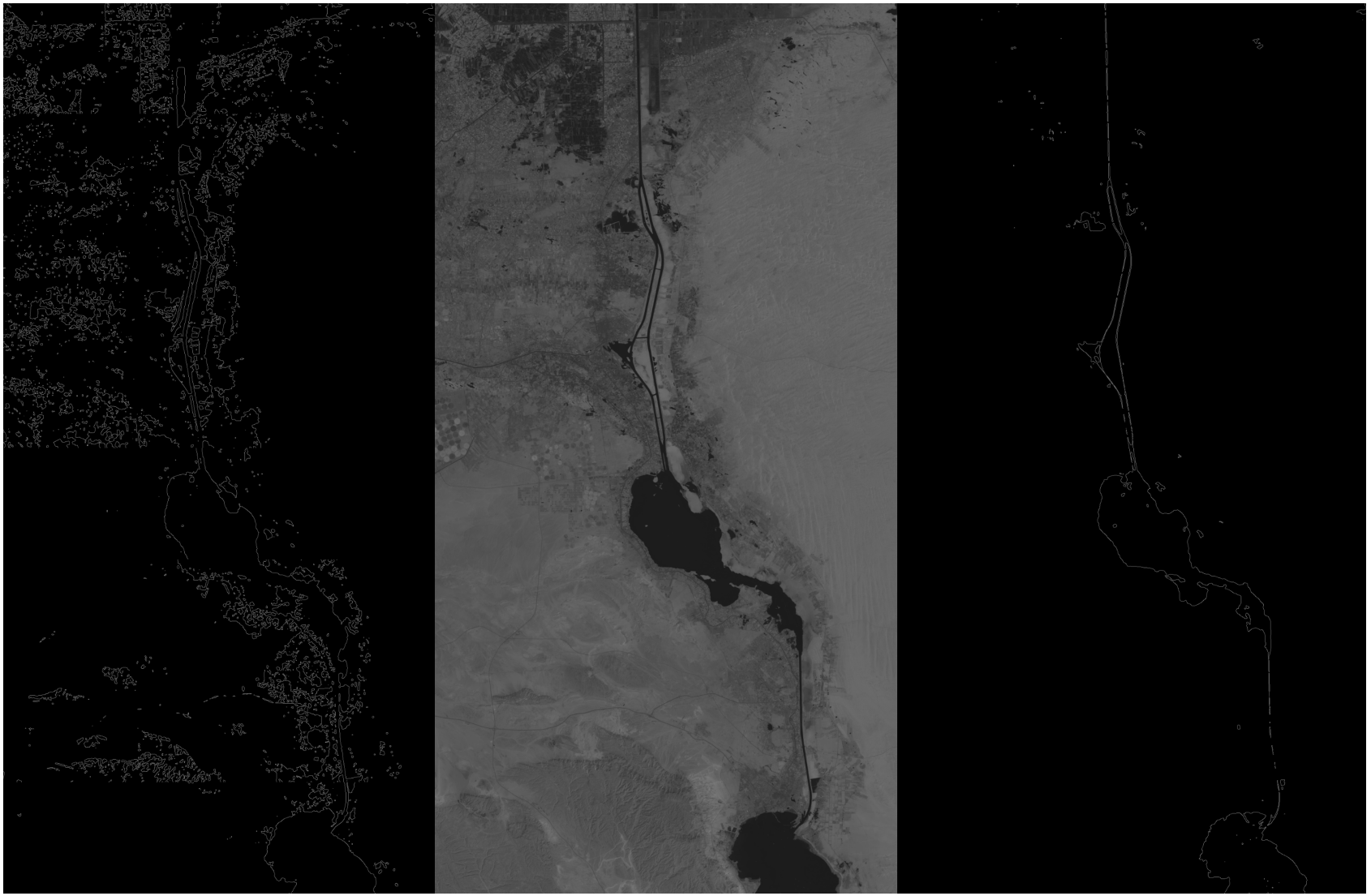
Accuracy Analysis



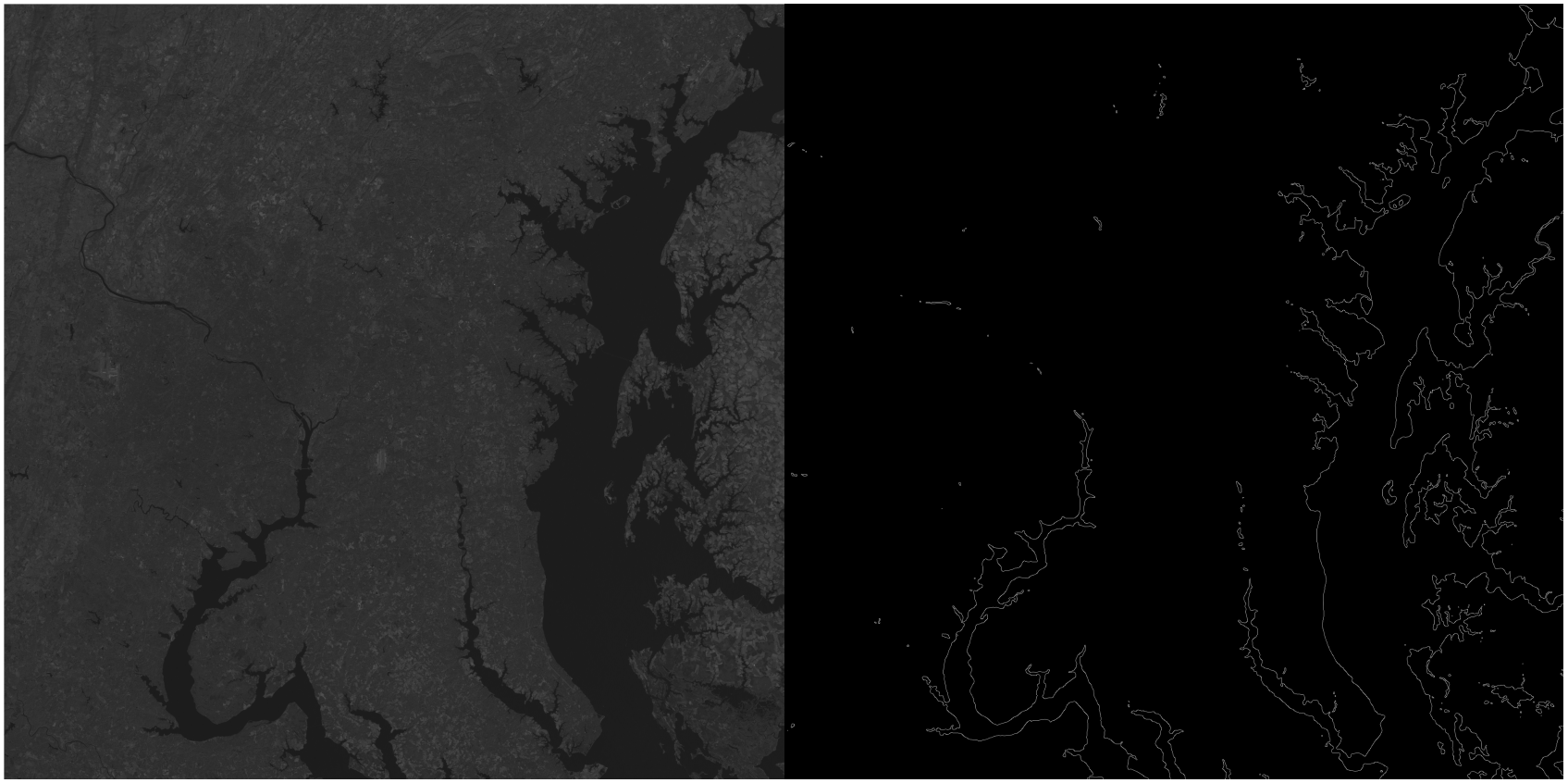
With adaptive thresholding (left), clouds still recognized as land and dark land is also considered water. With the static threshold (right), clouds are recognized as land, but dark land is successfully categorized.



The static threshold (right) also tends to have less noise on the land.



The data decomposition with adaptive threshold results in artifacts as different processes choose different thresholds. Avoiding either of these strategies removes the artifacts.



The static threshold algorithm run on the Chesapeake Bay showing good accuracy and low land noise.

The dynamic threshold applied to a cityscape. There is little noise within the city and the boats cause little blips, but the accuracy is on par with the static threshold on the satellite images. But this is significantly better than the static threshold which returns no edges, since all the pixels once blurred are a higher value than the threshold designed for the satellite data.

Conclusion

Project Summary

We aimed to create an edge detection algorithm then parallelize it on the file level and data level. Then, we collected data on it's performance and visually analyzed the results. Our parallel implementations were able to increase speed by a factor of two or more. However, parallelizing on the data level showed no improvement due to the size of the image or the Python optimization in the python library making runtime differences negligible in our serial and parallel implementations. Furthermore, we were able to ascertain that static thresholding is useful in images with low variance while dynamic thresholding, like Otsu thresholding, is useful for images with higher variance.

Impact

The results of this experiment gives insight into the performance of edge detection algorithms, an aspect that is commonly overlooked by other academic articles. Our results demonstrate that parallel edge detection is capable of improving speed by at least two times. From here more work can focus on improving runtime and accuracy in edge detection algorithms for high resolution satellite imagery and large image datasets. Also, the software and performance data can support the development and innovation of big data architectures for image data.

Future Improvements

We came across issues that give interesting insight into how to improve upon our work. Firstly, decomposing images between processors, using either the threading or multiprocessing modules in Python, will not show a performance improvement regardless of the number of active processors. We speculate this is due to the overhead that comes from initializing processors or the size differences between the whole image and image segments are not large enough to affect performance on our rigs. Therefore, an implementation that can reduce that overhead may reduce runtime. Also, with loading and saving being ninety percent of the total process time, working with faster storage would reduce this, making any data decomposition improvements a larger percentage of the runtime for each image.

Acknowledgements

We'd like to thank the professor, Tyler Simon, for his patience and teaching. This project was very interesting and the content we learned in class made the task much easier to understand.

References

[1] A. Al Mansoori, F. Al-Marzouqi. “Coastline Extraction using Satellite Imagery and Image Processing Techniques,” *International Journal of Current Engineering and Technology,* vol. 6, pp. 1245-1251, Aug 2016, E-ISSN 2277-4106.

- An experiment aiming to detect the coastline through image processing techniques. Provides rationale for creating binary masks over images to discriminate between land and water.

[2] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

- Canny’s original paper outlining his edge detection algorithm. Includes explanations for the use of the Gaussian blur and for one of the edge thinning techniques.

[3] J. R. Dim, T. Takamura, “Alternative approach for satellite cloud classification: edge gradient application,” *Advances in Meteorology,* vol. 2013, pp. 1-8, 2013, doi: 10.1155/2013/584816.

- An in-depth article discussing the whole process from choosing which bands of satellite data to use, to methods to improve contrast, to edge formation. It was useful for understanding which methods of edge detection (Sobel, Pewitt, Roberts) apply to what scenarios, and for another look at the edge detection process. This also served as inspiration for the two level paper outline.

[4] T. B. Moeslund. *Canny edge detection,* (2009). Accessed: Mar. 5, 2021. [Online]. Available: https://web.archive.org/web/20150421090938/http://www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf

- A step by step overview of Canny’s algorithm with an example image. It explains why and how each step is done, and follows an example image through the process.

[5] J. M. S. Prewitt, “Object enhancement and extraction,” in *Picture Processing and Psychopictorics*, Academic Press, 1970.

- The original derivation of the Prewitt operator and its use in improving object recognition. It also explains some of the issues of linear gradient operators on more complex scenes.

[6] L. Roberts, *Machine Perception of 3-D Solids, Optical and Electro-Optical Information Processing*, MIT Press, 1965.

- Roberts' thesis deriving the Roberts operator. It also explains a method different from Canny's for forming lines from the edge pixels.

[7] I. Sobel, "An isotropic 3×3 gradient operator," in *Machine Vision for Three-Dimensional Scenes*, H. Freeman, Ed., pp. 376–379, Academic Press, New York, NY, USA, 1990.

- The original paper that derives the Sobel gradient operator. It covers its benefits and drawbacks and its major use cases.

[8] U.S. Department of the Interior. "EarthExplorer." Website. URL https://earthexplorer.usgs.gov/ (accessed Mar. 7, 2021).

- A United States Geological Survey site that allows for the searching of satellite images from many available data sets. Specifically the landsat 2 collection is available for download, which includes reflectance and temperature data for much of the globe. This is likely the dataset that will be focussed on for the project.

[9] A. Vaccaro. "4 steps to detect coastline changes from satellite." Website. URL https://towardsdatascience.com/satellite-coasts-detection-model-with-python-and-opencv-28d1b4b8474e (accessed Mar. 6, 2021).

- An implementation of a serial method to detect coastlines using edge detection, then using edges from between two sets of images of the same location at different times to visualize coastline change. This gives a good idea of the process involved in this project, except it is serial and uses clustering to determine what is water.