Joshua Poole
M10196190
CS5068: Parallel Computing

Final Project

Parallel implementation of climate forecasting

1. Introduction

The global climate has been a major point of discussion and research in the scientific community for decades. As this is the only plant humans will probably ever inhabit, it is of utmost important to track the climate and humanity's impact on it.

One of the most important aspects related to climate studies is forecasting. Climate forecasting is used to predict certain climatic properties such as weather averages for a few weeks or even a few years into the future (Merryfield). These forecasts not only allow scientists to be able to model the predicted future of the climate on Earth, but also allow policy makers and citizens to see what the future holds if no changes are made. By creating an accurate model of our past and future climate we open the door for new understanding of the Earth and our effect on it.

Because of the importance of this topic, I decided to develop an application that was capable of predicting this year's climate. Given the restrictions presented with a one-person team size, modeling and predicting the climate on a large scale was deemed impractical. Therefore, the project scope was localized to modeling and predicting yearly average minimum and maximum observed temperatures.

This project gathers data from a query-able archive of historical weather and climate data called Meteostat. Meteostat provides fast access to this historical climate data through a JSON API ("Introduction"). This API allows you to query historical climate data gathered from "weather stations" throughout the world. For this project, the "Newark Airport" station in New York was used because of its location and because it had collected climate data for over a century. This quantity of data was deemed a reasonable amount to make predictions with. This project's parallel implementation is mainly a map-reduce operation that averages data in chunks in parallel. The parallel implementation also leverages the asynchronicity of HTTP

responses by avoiding the process to be blocked when waiting for a response from the Meteostat API.

In totality, the project queries daily weather data from 1920-2019, then averages the daily observed temperature minimums and maximums into yearly average temperature minimums and maximums. It then plots this data and determines the trendlines for the minimum and maximum temperature data. It then uses these trendlines to predict the average minimum and maximum temperature for the year 2020.

2. Design and Optimization Approach

This project is coded in python 3 and is split into a few different methods. The main work from this application takes place in the method "get_daily_data_and_avg". This method accepts an integer year as input. It uses the 'requests' package to query the Meteostat API for all the daily weather data collected for the span of that year. The design and creation of the request portion was aided by a tutorial on "DataCamp" by Olivia Smith on using the 'requests' package. After receiving and parsing the response, all of the minimum temperatures recorded for each day are averaged into a single yearly average minimum temperature. The same is done for the daily observed maximum temperature. The method then returns the yearly average minimum and maximum temperatures. This was the method that was chosen to be parallelized.

To implement parallel processing, the package 'multiprocessing' is used. This package allows for the creation of multiple processes that can run independently. The operating system need only assign threads to each process to achieve parallelization. It should first be mentioned that the structural design of this application's parallel functionality is heavily inspired from Frank Hofmann's article on "Stack Abuse" called "Parallel Processing in Python". A reference to this article can be found in the end of this report.

The 'get_data_parallel' method manages the high-level parallelization of the application. First it determines how many processes can be created by getting the number of cores available on the machine. This functionality should result in more speed improvement the more cores are available on the machine. After determining the amount of processes that can be created for maximum speed, it initializes a processor pool and starts creating processes. Each process is

given the ‘worker’ method as its target. The worker method takes three variables as arguments: a string representing the process’s name, a global task queue, and a global result queue. The task and result queues allow for the ‘get_data_parallel’ method to manage what the processes are doing and when they should stop. It also allows for the processes to return the result of their task(s) to a global thread-safe queue for later reference.

The worker method is quite simple in its functionality. It is essentially an infinite loop that gets a task from the task queue, completes the tasks, gives the result of that task to the result queue, and repeats. This loop, and subsequently the overall process, only stops it receives a task of value “-1”. Each task is either an integer year or a -1. If the task is a year, then it calls the aforementioned ‘get_daily_data_and_avg’ method, passing it the year it received as a task. That method will return two dictionaries which store the yearly average minimum and maximum temperature. After these dictionaries are put into the result queue, the loop repeats.

The task queue is first populated by all the years the need to be queried. Two global constants START_YEAR and END_YEAR determine the lower and upper bounds of data for the application to gather. Every year in between START_YEAR and END_YEAR, excluding END_YEAR, is put into the task queue. The end of the task queue is then filled with -1s. A -1 is inserted for each process created earlier. Since queues are First In, First Out (FIFO), all the actual tasks are given to the processes, and when there are no more left, each process will be given a -1 to end it. After all the processes have been stopped, the result queue is parsed. All the individual dictionaries containing the average minimum and maximum temperature for each year are combined into two large dictionaries containing all the yearly average minimum and maximum temperatures.

The parallelized portion of the application as described above was also implemented sequentially through a simple loop. This loop passes every year from START_YEAR to END_YEAR, again excluding END_YEAR, to the ‘get_daily_data_and_avg’ method and stores the resulting yearly averages. The results of the sequential run are not used, and its existence is purely for the purpose of comparison.

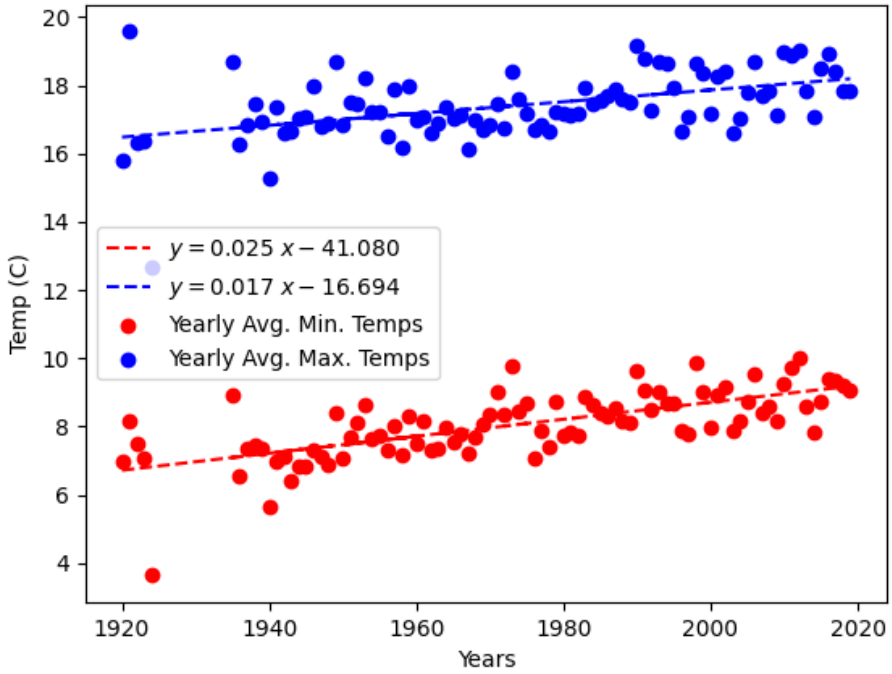

Figure 1: Matplotlib plot of gathered temperature data

Lastly, the dictionaries containing all the yearly average minimum and maximum temperature data are formatted and split into x and y coordinates. These lists of x and y pairs are plotted on a scatter plot using the 'matplotlib' package. The resulting plot can be seen in Figure 1 above. Additionally, best fit lines were calculated and plotted for each data set. The creation and labeling of these best fit lines are based on an answer by Ashot Matevosyan to a question posted "Stack Exchange" about drawing best fit lines using 'matplotlib'. The equations for these lines are then used to predict the average minimum and maximum temperature for the year 2020.

3. Application Performance Analysis and Project Results

The application was run on two different systems in both sequential and parallel fashions. For the timed runs, daily temperature data from 1920-2019 was gathered and averaged. It should be mentioned that data from 1925-1934 in the Meteostat API is missing, so a bit less than 100 years of data was gathered. I am not sure why this is the case, but is it clear that this particular weather station was not recording data during this time period.

First the application was run on my local machine which features an Intel i5-8400 running at 2.80 GHz with an available 6 cores for processing. It took 15.39s to gather and average the data sequentially and 4.83s to do the same in parallel. The parallel version on my local machine boasts over a 3x speed increase over the sequential version. This speed increase is even more exaggerated when run using the Ohio Supercomputer Center (OSC).

The application was run on a single Owens node which features a NVIDIA Tesla P100 GPU with an available 28 cores. The sequential version took 8.44s to complete, while the parallel version only took 483.02 milliseconds. On an environment with 28 cores at its disposal, this parallel application was able to accomplish over a 17x speed increase over the equivalent sequential version. It is clear from these results that even when only given a few cores to work with, the parallelized version of this application will provide a significant speed increase over a sequential operation.

One very important topic that should be mentioned is that the response time of an external API plays a huge role in the overall completion time of the application. Occasionally, the volume of the requests to the API by the application become overwhelming and a 429 response is received instead of data. When this happens, the process that made the request is slept for one second before attempting to query the API again. An excess of these 429 responses could have a significant negative impact on the speed of the application. Additionally, since the application is only querying historical data that will theoretically never change, a future improvement to the application would be to store the data gathered (in reduced form) locally for future runs. This would not only result in a significant improvement to the speed of the application, but also make it less reliant on the internet and the responsiveness of an external API.

In addition to the speed comparison between sequential and parallel data gathering and processing, the application also had a primary goal of predicting the average maximum and minimum temperature for 2020. Based on the data gathered from 1920-2019, the application predicted an average minimum temperature of 9.21°C and average maximum temperature of 18.21°C.

The fundamental concepts of this application could be extrapolated onto a larger scale by attempting to model and predict global climates. Instead of using a single weather station in

New York as a data source, stations across the nation, or perhaps even the entire world could be used. The data from these stations would most likely be split into quadrants across the globe that would allow scientists to make models and predictions about the global climate.

4. Division of Work and Self-Assessment

As I was the only individual who worked on this project, I completed all of work required to create and test the application. I first found a well-structured, functional, external API that could be queried for historical weather data. I then looked for a weather station with an appropriate amount of data and decided on what weather data would be collected and processed. Next, I needed to create a method that would be able to query this API and process the data received. This method would need to be modified multiple times to best work with the parallel process structure I had decided on. Lastly, the application was tested until it worked properly, and was timed for efficiency. Throughout development I relied heavily on internet resources such as “Stack Overflow” and various package documentation to determine how to accomplish different functionalities.

Works Cited

Hofmann, Frank. “Parallel Processing in Python.” *Stack Abuse*, Stack Abuse, stackabuse.com/parallel-processing-in-python/.

“Introduction.” *Meteostat Developers*, dev.meteostat.net/docs/.

Matevosyan, Ashot. “How Can I Draw Scatter Trend Line on Matplot? Python-Pandas.” Stack Overflow, 30 Nov. 2019, stackoverflow.com/questions/41635448/how-can-i-draw-scatter-trend-line-on-matplot-python-pandas/41635626.

Merryfield, William J. “Advancing Climate Forecasting.” *Eos*, 27 Nov. 2017, eos.org/science-updates/advancing-climate-forecasting.

Smith, Olivia. “Performing an HTTP Request in Python.” *DataCamp Community*, 19 Sept. 2019, www.datacamp.com/community/tutorials/making-http-requests-in-python.

Code Appendix

https://github.com/poolejosh/ParallelComputingFinalProject