Tracking the Pulse of a Storm: Using Multithreading & Multiprocessing on the GraphCast Weather Model

Improving Efficiency on a Weather Model

Brooke E. Sherman*

Data Science Student at the Pennsylvania State University, bes5659@psu.edu

GraphCast is a recent graph neural network model made for producing more accurate weather predictions. The model also can predict extreme weather events way more accurately as well. With upcoming technology and how important weather prediction has become for the everyday person (whether it be deciding what clothing to wear by checking the weather app or leaving home to survive a hurricane), it is important to also ensure that GraphCast can run efficiently. GraphCast already does a much better job than some previous weather prediction models in efficiency (as it does not need multiple machines to run the code), but it still can be improved upon more in different areas of the model. Recently, multithreading and multiprocessing have been used to improve the time it takes the code to run while still trying to output accurate results. This paper covers the use of those methods and how they can be implemented into the code to make it produce more predictions and other values more efficiently. It shows that multithreading and multiprocessing can improve the code in areas where it overall took the code longer to run without completely affecting the accuracy of the results and predictions from the graph neural network model.

Keywords and Phrases: Multiprocessing, Multithreading, Weather Model, Graph Neural Network

1 INTRODUCTION

Weather prediction has always been a difficult problem in data science. Being able to accurately figure out what the weather will be like in two days can be hard, but it is even more of a struggle to predict it ten days ahead of time. There have been plenty of weather models to come out and be used for making weather forecasts from many different types of machine learning models. The goal with these weather models is to ensure that they can produce extremely accurate results while also trying to be as efficient as possible at creating these predictions. Weather models that can achieve this are helpful to people daily and in general, better for the future.

Weather models are an important part of people's lives. From using the iPhone weather app to watching the news for the weather in the morning, it helps people decide their choice in clothing for the day and notifies them of any severe weather. Having these models can help track hurricanes and other natural disasters that could warn and notify people that they may need to leave their homes to stay safe. These weather models need to be constantly updated to keep track of these significant events and to also keep weather apps and the news informed with the latest weather changes. Therefore, ensuring these models are efficient as possible while staying accurate is important.

---

* The research and work for this project was done in a Data Science class called DS 340W: Applied Data Sci at the Pennsylvania State University during the fall semester in 2024.

1.1 The GraphCast Model

With this new model network and all its different parts, GraphCast has done a very well-done job at improving accuracy and efficiency on previous weather models. It has been able to bridge the gap between needing to use multiple computers to run the code to using only one. It has been able to achieve accuracy on weather forecasts that previous models like HRES and Pangu-Weather have not been able to conquer. Figure 1 below shows an example of how GraphCast can outperform Pangu-Weather on the RMSE metric (lower RMSE is better).

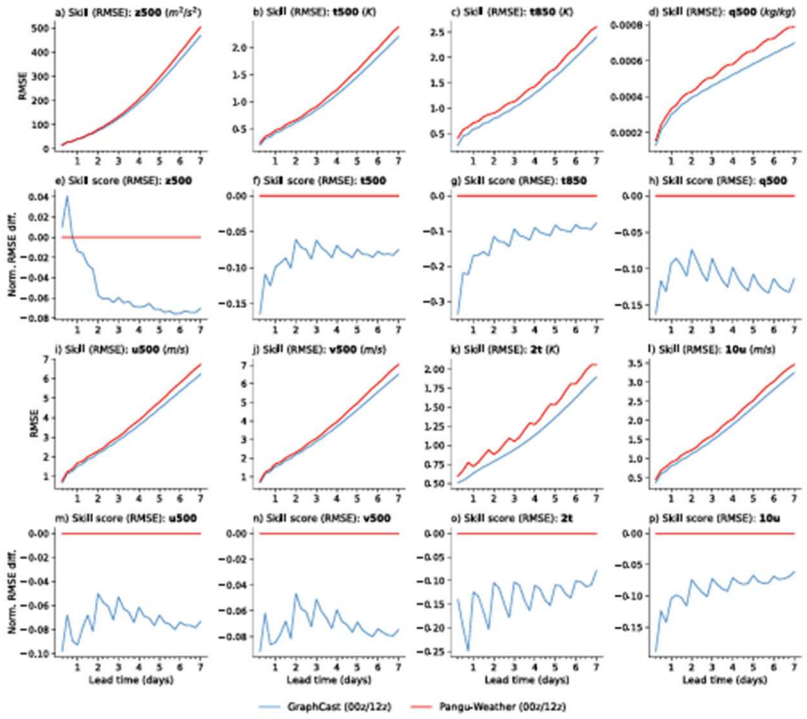

Figure 1: RMSE Score Comparison Between GraphCast and Pangu-Weather. Graphs Created by Remi Lam et al. via "GraphCast: Learning skillful medium-range global weather forecasting" (https://arxiv.org/pdf/2212.12794)

To understand how GraphCast works a little bit better, Figure 2 below shows the GNN model and how it predicts and makes forecasts. Following an encoding, processing, and decoding method as a GNN allows GraphCast to accurately predict values and do it more efficiently than previous models. It also reveals that it uses a mesh with nodes and edges to run the model properly.

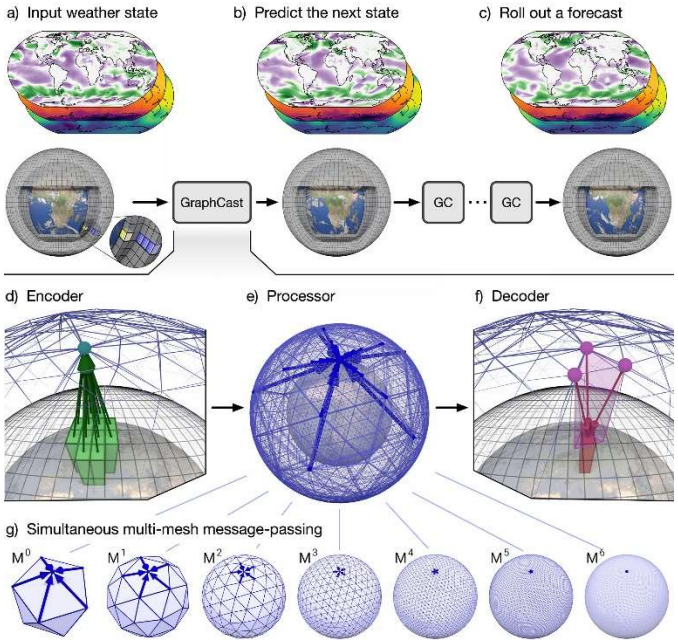

Figure 2: GraphCast Model. Model Created by Remi Lam et al via "GraphCast: Learning skillful medium-range global weather forecasting" (https://arxiv.org/pdf/2212.12794)

GraphCast has been able to make weather models more accurate and easier to use. However, it is important to note that while this model has created huge advancements in the weather prediction problem, there still could always be more room for making the code even more efficient.

1.2 The Problem

While GraphCast has been shown to be a very strong model in reporting weather, there still could be even more improvement made to this GNN framework. Even though GraphCast has been able to improve on efficiency a ton compared to previous models, there still could be room for more advancements in this area.

GraphCast has some bottleneck areas where the code runs slow due to a couple different reasons. The first area is loading in a selection of the data. Each selection of data has many features and is huge. This can make it difficult for the code to run fast because it is taking a large amount of time to compile all the parts of the data and bring it in for use on the model. The next part of the code where GraphCast could currently use some changes is in the running of the loss and gradient code. Both pieces of code take an extremely long time to run due to the data that must be passed through it and the fact that they both have a couple of functions to run through to achieve the loss and gradient values as a result. Since there are multiple functions to run through and the model predictor is being built in one of these functions with the large data, it takes a while for the loss and gradient to calculate properly.

Both problems lead to the code taking longer to run. GraphCast could be even more efficient if these issues were fixed without affecting accuracy.

1.3 Solution

Existing works have shown that there are multiple ways to increase efficiency. There is one paper that talks about multiprocessing and another that uses multithreading. Utilizing multiprocessing and multithreading are effective ways to decrease the time it takes to run the code with the weather data. These methods may cause some changes to the accuracy, so it is important to keep track of that. Both have been used in all types of models from supervised to unsupervised including GNN models like GraphCast. It is good for running multiple tasks at once and just speeding up tasks like loading in code, so it seems to be a perfect fit for the GraphCast issues. Specifically, multithreading and multiprocessing are used to speed up the different tasks to improve the problems in the code here. Therefore, by implementing these python packages in the GraphCast code on the two different problem areas (loading the data in and the loss and gradient code), the code should run more efficiently while hopefully still maintaining a strong accuracy with little loss and error. Because multithreading does not necessarily print out the code and save it as needed, the queue function was also imported and used to save the data and later output it. Multiprocessing has a queue function built in.

Because multithreading and multiprocessing can affect accuracy, it is important to check the evaluation metrics for both efficiency and accuracy for the changes applied to the code. To see how much the accuracy is affected, the loss, mean gradient, and the difference between the correct map and the predicted map will be used as evaluation metrics. For efficiency, the time of the two changes will be calculated using the time function. The code will be tested three different times with all the rest of the variables given in GraphCast remaining constant to find an average for each of the evaluation metrics. The overall conclusion is that with the changes implemented, the accuracy of the model can stay the same while highly improving for importing the data and slightly improving for the loss and gradient code.

2 RELATED WORKS

GraphCast itself has already improved upon the previous weather models, HRES and Pangu-Weather, but it still falls short in efficiency. The areas where the code runs longer could be solved more quickly by applying certain methods made for speeding up different parts of code. The challenge is being able to keep accuracy high while applying these techniques and testing them out.

On GraphCast, there were two main methods being considered: multiprocessing and multithreading. These processes both supposedly speed up the running of the code very well by making tasks use the processing units as efficiently as possible to produce outputs.

2.1 Multiprocessing

In a paper called “A Multiprocessing-based sensitivity analysis of Machine Learning algorithms for Load Forecasting of Electric Power Distribution System” by Zainab et al., it describes the usage of multiprocessing in different types of machine learning models including a GNN. This work showcases using the multiprocessing pool function and what its effects are on the efficiency of the model but also the accuracy.

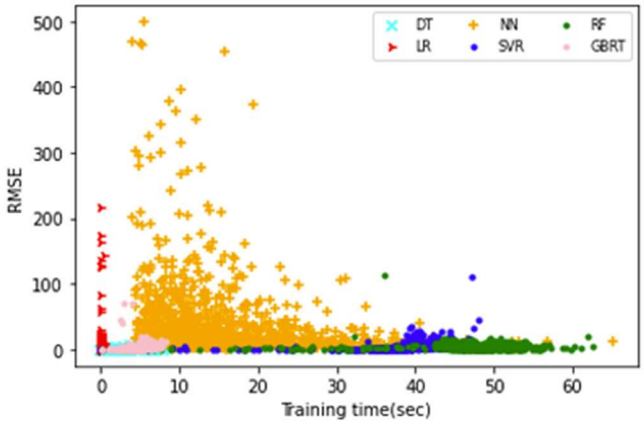

Figure 3: RMSE Score vs. Training Time of Different ML Models. Graph Created by Zainab et al. via "A Multiprocessing-based sensitivity analysis of Machine Learning algorithms for Load Forecasting of Electric Power Distribution System" (https://www.researchgate.net/publication/349418437_A_Multiprocessing-Based_Sensitivity_Analysis_of_Machine_Learning_Algorithms_for_Load_Forecasting_of_Electric_Power_Distribution_System)

Figure 3 above shows how the graph neural network model responds to multiprocessing. The training time of the models is super quick with the multiprocessing applied. However, for the GNN, it also shows that the RMSE is fluctuating on different attempts. The RMSE is high on some attempts, but low on others. Because of this, multiprocessing seems to affect the accuracy of the model a lot. Therefore, accuracy measures need to be calculated when applying multiprocessing to the code. Multithreading could also cause issues like this, so the accuracy is also checked when testing this as well.

There are some differences between how it was applied to GraphCast versus the models above. Instead of using the pool function, the process and queue functions were used because they were being performed on single tasks instead. The functions were used for the loading of the data while multithreading is used for the loss/gradient functions. In the paper, it is used to split up the data and run the model prediction code and then picked the one with the minimum error as the best performance. This GraphCast change is a combination of multiprocessing and multithreading due to the testing of multiprocessing on the loss/gradient code making the time be over eight minutes (without the changes it is a little over 3.5 minutes).

2.2 Multithreading

Another paper found was on using multithreading instead of multiprocessing. This paper is "STEP: A Distributed Multi-threading Framework Towards Efficient Data Analytics" by Yijie Mei et al. and discovers the use of multithreading in logistic regression, K-means, NMF (Nonnegative Matrix Factorization), and PageRank (a random walk model). The STEP framework can be used on more than just the models chosen and it is supposed to run tasks in parallel throughout the code with multithreading.

Figure 4 below shows the improvement upon the K-Means model in terms of efficiency using STEP versus using Spark or Petuum. This is just one of the models that STEP can improve upon using multithreading. It was also able to increase the efficiency of the logistic regression and NMF models as well. By seeing how it was able to increase the efficiency of these models, multithreading was chosen to apply to GraphCast.

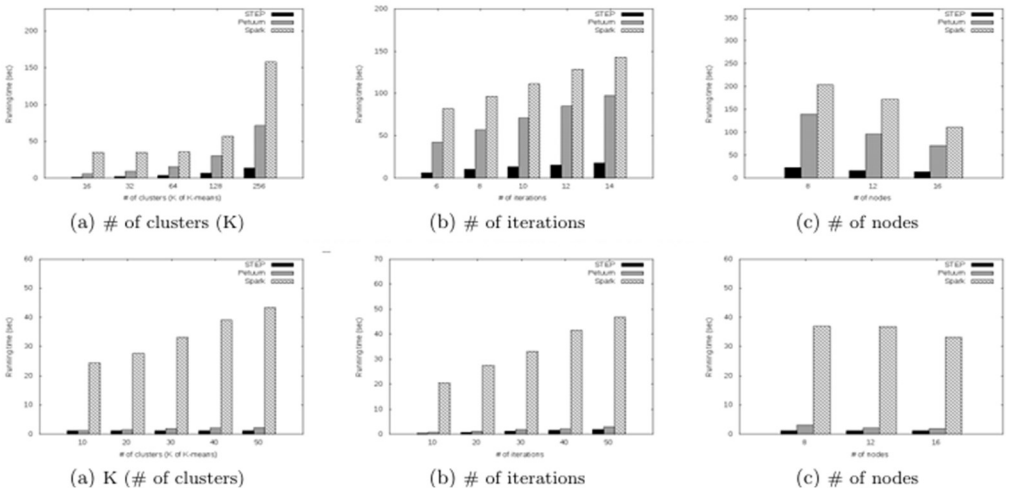

Figure 4: K-Means Results on Two Different Datasets. Table Created by Yijie Mei et al. via "STEP: A Distributed Multi-threading Framework Towards Efficient Data Analytics" (https://arxiv.org/pdf/1812.04971)

The difference between the STEP model and the changes in the code in GraphCast is that the application of multithreading in GraphCast was done for tasks that were dependent on the input-output system. Therefore, multithreading was used to improve upon tasks by doing them one at a time in GraphCast versus doing them at the same time in the STEP model. The multithreading still was able to improve upon the speed in GraphCast because the tasks were I/O bound (relying more on the input and output). The other difference is that this paper does not calculate the accuracy of the models with efficiency. This will be done in GraphCast to see if multithreading does influence how close the predictions were to the target. This GraphCast change was using multithreading only.

3 METHODOLOGY

To improve upon the efficiency of the GraphCast model, multithreading and multiprocessing are proposed to be utilized in the code. By implementing these packages, they can be effectively employed into the bottleneck areas of the code and make it run faster while producing accurate results. This would make the code run faster in these areas as the tasks rely more on the input and output, which is what multiprocessing and multithreading can be used for. Also, because of the areas these packages are being applied to, the accuracy should be unaffected compared to the results of the original code. The parts of my approach include adding the time function for evaluation, making the code that needs to run faster into functions, and then applying the multiprocessing/multithreading to the code. By going through this process, the two bottleneck areas can improve greatly.

3.1 Preliminary

Multiprocessing and multithreading are techniques introduced separately in different papers. Multiprocessing is the use of two or more processing units in a coding system. On the other hand, multithreading is the use of two or more users in a coding system. These papers showcase multiprocessing and multithreading and applying it to the entirety of the code by splitting up the data and running it in parallel process through a model at the same time with several workers. However, in the case of the GraphCast code, that is not how these packages are used. Instead, these packages are applied to the other

portion of where multiprocessing and multithreading might be useful which is speeding up input/output processes like bringing in data. Basically, it allows multiple workers/processors to focus on the one function to speed it up.

3.2 Efficiency Contributions

There were different contributions and changes made to the GraphCast code. These changes were made to make the code more efficient and to be able to evaluate results from the changes and the original code. There was also one cell at the end of the code that was deleted, as it was not used for the project.

First, the multiprocessing portion is added to the data code. The code that takes the data is made into a function to use for the multiprocessing package. The multiprocessing queue and process functions are used to improve the speed at which the data is brought in and collect the data into a variable called example_batch. Figure 5 shows the code below.

```
FUNCTION load_data:
        WITH a bucket google blob open dataset_file:
                LOAD dataset into example_batch using xarray and compute
        PUT example_batch into the queue
CREATE the multiprocessing queue
CREATE the multiprocessing process with the load_data function
START the multiprocessing process
GET the value from the queue into example_batch
KILL the process
```

Figure 5: Multiprocessing Code for Bringing in Weather Data.

For the multithreading part of the data code, the queue package and the multithreading packages were both used since queue is not built into multithreading. Just like multiprocessing, multithreading has a similar function to the process function called thread. It also was used in a pretty similar way, but instead it used t.join() before the q.get() line and did not kill the thread. Figure 6 displays those changes and the multithreading code below.

```
FUNCTION load_data:
        WITH a bucket google blob open dataset_file:
                LOAD dataset into example_batch using xarray and compute
        PUT example_batch into the queue
CREATE the queue
CREATE the multithreading thread with the load_data function
START the multithreading thread
JOIN the multithreading thread
GET the value from the queue into example_batch
```

Figure 6: Multithreading Code for Bringing in Weather Data.

The other part of the code that was changed was the running of the loss and gradient code. Once again, the loss and gradient code areas were changed into functions and then ran through the same multithreading process as seen in

Figure 5. They were run separately and not at the same time, as that was found to have time improvement. When they were run at the same time, the code took longer to produce results than it previously had with no changes. The code is pretty like Figure 5 with the use of two threads that each needed to be started, joined, and then retrieved by the queue. The code for these changes is below in Figure 7.

```
FUNCTION loss:
        GET loss, diagnostics from loss_fn_jitted function with train_inputs, train_targets,
        and train_forcings
        SAVE loss string into a
        ADD a to queue
FUNCTION grad:
        GET loss, diagnostics, next_state, grads from grads_fn_jitted function with
        train_inputs, train_targets, and train_forcings
        CALCULATE mean gradient and store in mean_grad with mean, lambda, and tree functions
        SAVE loss and mean_grad string into a
        ADD a to queue
CREATE queue
CREATE multithreading thread for loss function
CREATE multithreading thread for gradient function
START loss multithreading thread
JOIN loss multithreading thread
START grad multithreading thread
JOIN grad multithreading thread
PRINT item from queue (twice)
```

Figure 7: Multithreading Code for Calculating the Loss and Mean Gradient.

Finally, the only part to add was the time function which was imported into the original code, the multiprocessing and multithreading code, and the multithreading code as one of the measures of evaluation. Its job was to see how fast each of the functions ran with the changes that were applied to them. The time must be calculated at the start of the process and then calculated at the end. Finally, a print statement calculated the difference between those two times will leave the time it took to run the code with or without changes. Here is an example of importing and using time below in Figure 8:

```
IMPORT time
CREATE start time
PRINT hello
CREATE end time
PRINT end - start
```

Figure 8: Time Function Example. Code Created by NPE and Edited by Mateen Ulhaq via "How do I measure elapsed time in Python?" (https://stackoverflow.com/questions/7370801/how-do-i-measure-elapsed-time-in-python)

By adding all these changes, the new code was able to be implemented and give results that are comparable to show whether the code improved upon efficiency and how these changes affected the accuracy.

4 EVALUATION

The evaluation for this model is tested using the same dataset with some other constants. The model is tested based on the difference between the prediction and the target, the loss calculation, the gradient calculation, and how long it takes each of the two cells to run through the changes. The original code was run through adding the time function to set a baseline. Each set of the code (the original code with the time, the multiprocessing/multithreading change, and the multithreading change) were run three times to see how the metrics change. The changes applied show improvement in the efficiency of the model while keeping the accuracy unaffected.

4.1 Setup

To test this thoroughly, there are a bunch of values that will remain constant. The model was always a mesh size of 4 and latent size of 16 with 4 GNN message steps and 13 levels. The source of the data is the ERA5 data dating 2022-01-01 with a resolution of 0.25 and 1 step. Finally, the variable tested was always the 2m_temperature (air temperature two meters above the surface) and was shown on level 50. Also, these tests were done by running the code for the first time (resetting everything – closing the code out and restarting it) to have fairness.

The ERA5 dataset contains a bunch of values used to create and evaluate the model. It uses latitude, longitude, level, time, and datetime to help determine the coordinates for the graph. The variables used to predict the temperature value are geopotential at surface level, parts of land coverage that have both freshwater and seawater, mean sea level pressure, component vector winds (u and v) ten meters above the surface, total precipitation over a six-hour period, top of atmosphere solar radiation, temperature, geopotential, component vector winds (u and v), vertical velocity of air, and humidity. All the variables and coordinates are float values except level, which is an integer, time, which is a timedelta variable, and datetime, which is a datetime variable.

The baseline was created with these values as constants and the ERA5 data shown above. The only addition to the code in the baseline was the adding of the time function to calculate the evaluation metrics. This is shown in Table 1. The difference scale value is based on the scale created from subtracting the prediction to the target.

Table 1: Results on the Original Code

Evaluation Metrics	1st Results	2nd Results	3rd Results	Avg. Results
Data Load Run Time:	109.2974379	112.7363818	111.3274913	111.120437
Loss/Gradient Run Time:	284.2351198	223.4055829	263.3240824	256.9882617

Evaluation Metrics	1st Results	2nd Results	3rd Results	Avg. Results
Difference Scale:	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5
Loss:	11.0425	11.0425	11.0425	11.0425
Gradient:	0.125471	0.125431	0.125457	0.125453

For the GraphCast model, there was first the installation and initialization of the packages and all the parts of the GraphCast model. Then, the plotting functions were brought in for future usage. Next, model parameters were loaded and chosen, so the data could be selected properly. The data is then chosen and loaded. The cell where the data is loaded is one of the chosen areas to improve efficiency by applying multiprocessing and multithreading. Then, the data is explored showing some data plotted. Training, evaluation, and normalization data are created. Next, more functions are created with the GraphCast model and used like the predictor. The loss and gradient functions are also created here. Next, the predictions are created and compared to their targets creating the difference scale. Figure 9 shows the difference scale and image below. Finally, the loss and gradient values are calculated. This is the second area where efficiency will be improved.

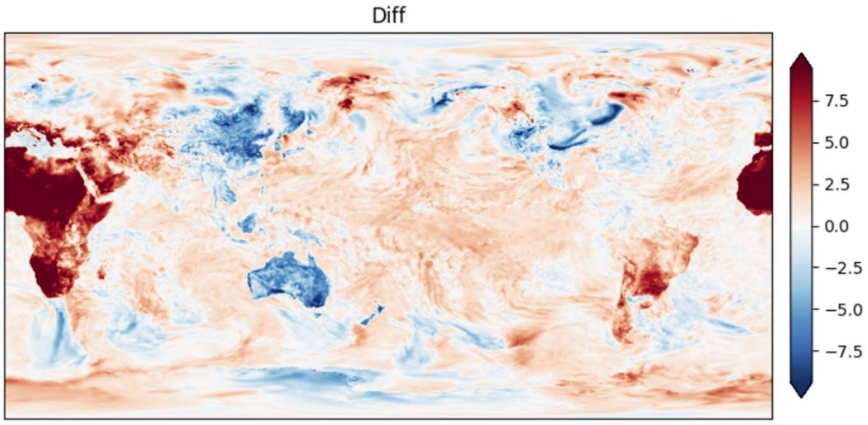

Figure 9: Difference Picture and Scale Between Prediction and Target Original Code.

The multiprocessing/multithreading experiment and the just multithreading experiment reveal interesting results compared to the baseline values presented in Table 1 and shown in Figure 9.

4.2 Multiprocessing/Multithreading Experiment

The first experiment was conducted with a combination of multiprocessing and multithreading. The goal of this experiment was to see if utilizing both methods could improve the timing of the cells and keep the accuracy of the model high. The data load run time is where the multiprocessing is used while the loss/gradient run time is where the multithreading is used here. If the results print out better with less time and about the same amount of loss, difference, and gradient, then the changes will have proved to be a strong change. This was also run three times to get an average of the results like in the original code.

Below in Table 2 shows the results of the multiprocessing/multithreading experiment.

Table 2: Results on the Multiprocessing/Multithreading Code

Evaluation Metrics	1st Results	2nd Results	3rd Results	Avg. Results
Data Load Run Time:	24.31912661	79.2135431	65.12379123	56.21882031

Evaluation Metrics	1st Results	2nd Results	3rd Results	Avg. Results
Loss/Gradient Run Time:	240.4171677	211.045925	209.3124219	220.2585051
Difference Scale:	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5
Loss:	11.0425	11.0425	11.0425	11.0425
Gradient:	0.125004	0.125451	0.125463	0.125306

Compared to the results of the original experiment, the data load run time improves significantly and the loss/gradient run time improves a little. The data load run time improves by about 50% while the loss/gradient run time improves by about 14%. In terms of accuracy, the loss and difference stay the same while the gradient changes very little when achieving the original goal. Below in Figure 10 shows the image received for one of the attempts in the multiprocessing/multithreading experiment.

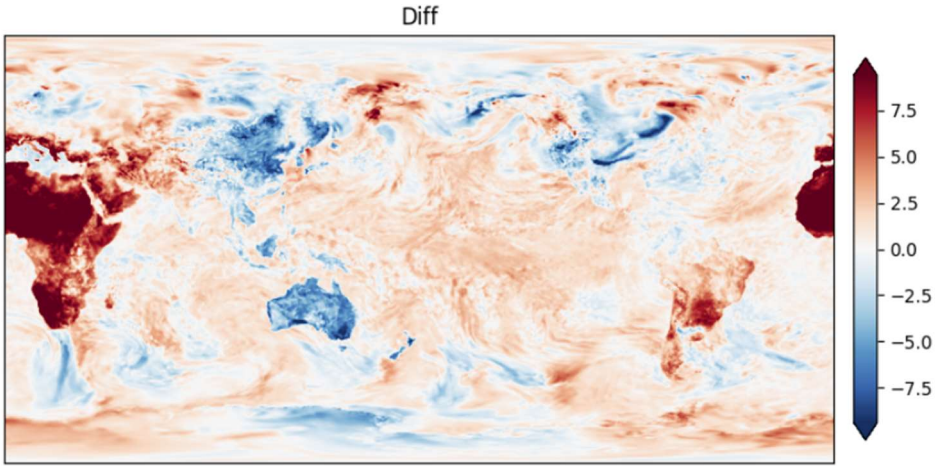

Figure 10: Difference Picture and Scale Between Prediction and Target Multiprocessing/Multithreading Code.

This experiment proved to be successful compared to the original code. The accuracy measures did not change or changed very little, and the efficiency improved for both bottleneck areas in the code.

4.3 Multithreading Experiment

The second experiment was to see how multithreading could improve both problem areas of the code alone. The goal was still to ensure that efficiency improved with accuracy remaining about the same. However, instead of having one cell be coded using multiprocessing, both cells had improvements with multithreading. This experiment was run three times like the previous experiment and the baseline. The same accuracy and efficiency values are used above as well.

Below in Table 3 are the results of the multithreading code.

Table 3: Results on the Multithreading Code

Evaluation Metrics	1st Results	2nd Results	3rd Results	Avg. Results
Data Load Run Time:	13.09670663	17.13241299	15.46217983	15.23043315
Loss/Gradient Run Time:	198.2784274	210.7360196	193.1230813	200.7125094
Difference Scale:	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5
Loss:	11.0425	11.0425	11.0425	11.0425
Gradient:	0.125456	0.125460	0.125444	0.125453

The multithreading experiment has also shown significant improvement in the timing results. The data load run time improved by about 86% while the loss/gradient run time improved by about 22% compared to the original code. The accuracy metrics also remained the same or about the same for this experiment as well. Below in Figure 11 is the difference between the prediction and target for this experiment.

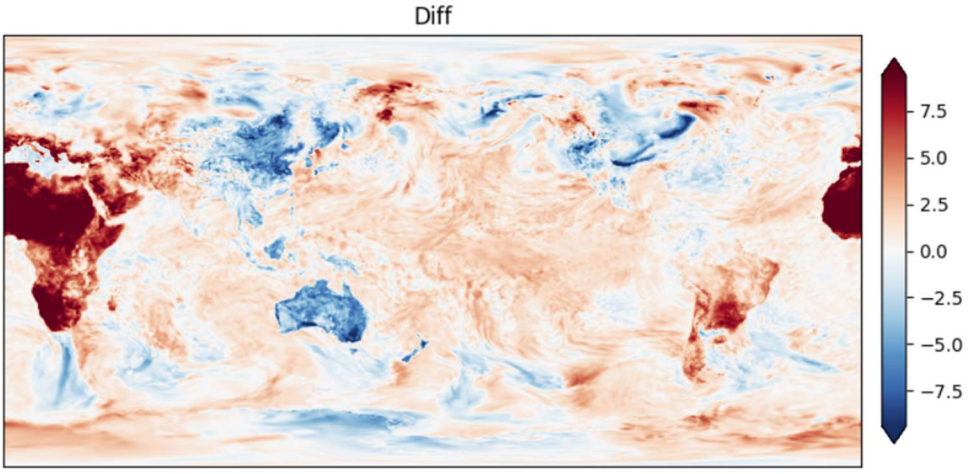

Figure 11: Difference Picture and Scale Between Prediction and Target Multithreading Code.

Because this experiment has more time improvement than the first experiment, multithreading has been revealed to be the best option for overall improving the GraphCast code. Here is a comparison of all the average results below in Table 4.

Table 4: Comparison of Average Results

Evaluation Metrics	Original Results	Multiprocessing/ Multithreading Results	Multithreading Results
Data Load Run Time:	111.120437	56.21882031	15.23043315
Loss/Gradient Run Time:	256.9882617	220.2585051	200.7125094
Difference Scale:	-7.5 to 7.5	-7.5 to 7.5	-7.5 to 7.5
Loss:	11.0425	11.0425	11.0425
Gradient:	0.125453	0.125306	0.125453

5 CONCLUSION AND FUTURE WORK

With the GraphCast model, it greatly improved in efficiency and accuracy than other previous weather models. However, it still could be improved upon the run time in certain areas of the code – the loading in of the data and the running of the loss and gradient functions. By applying multiprocessing and multithreading, the code could make these bottleneck areas better while ensuring that the overall accuracy of the model is unaffected.

For future work, there could be multiple opportunities to look at for improvement and testing on the GraphCast model. One of these would be testing out changing different variables that were constants to see how multiprocessing and/or multithreading would affect the change. For example, changing the number of steps or the resolution could

influence whether the multithreading/multiprocessing improves the code. Another idea is to test other ways that could improve the code more efficiently and without affecting the accuracy that are not multiprocessing nor multithreading. These ways proved to be effective, but there are other methods that could be tested. Overall, GraphCast seemed to improve with the multithreading and multiprocessing/multithreading experiments.

REFERENCES

[1] gilbert8. (2011, September 10). *How do I measure elapsed time in python?* Stack Overflow. https://stackoverflow.com/questions/7370801/how-do-i-measure-elapsed-time-in-python

[2] Lam, R., Sanchez-Gonzalez, A., Wilson, M., Wirnsberger, P., Fortunato, M., Alet, F., Ravuri, S., Ewalds, T., Eaton-Rosen, Z., Hu, W., Merose, A., Hoyer, S., Holland, G., Vinyals, O., Scott, J., Pritzel, A., Mohamed, S., & Battaglia, P. (2023, August 4). GraphCast: Learning skillful medium-range global weather forecasting. http://arxiv.org/pdf/2212.12794

[3] Mei, Y., Shen, Y., Zhu, Y., & Huang, L. (2018, December 12). STEP: A Distributed Multi-threading Framework Towards Efficient Data Analytics. https://arxiv.org/pdf/1812.04971.pdf

[4] Zainab, A., Syed, D., Ghrayeb, A., Abu-Rub, H., Refaat, S. S., Houchati, M., Bouhali, O., & Lopez, S. B. (2017). A Multiprocessing-Based Sensitivity Analysis of Machine Learning Algorithms for Load Forecasting of Electric Power Distribution System. https://www.researchgate.net/publication/349418437_A_Multiprocessing-Based_Sensitivity_Analysis_of_Machine_Learning_Algorithms_for_Load_Forecasting_of_Electric_Power_Distribution_System

APPENDIX

A.1 GitHub Link for Code

https://github.com/BrookeSherman03/DS340W-Project

- Implementation – Running Result folder is the original code with time function added
- Modified Implementation folder is the completed code changes

A.2 ERA5 Data

https://confluence.ecmwf.int/display/CKB/ERA5

A.3 Original GraphCast Code

https://github.com/google-deepmind/graphcast