

NM08: Parallel Computing For Weather Forecasting Using MPI or OpenMP

Final Report

by

James Kotic, Shaan Hossain, Vahe Poghosyan, and Arjun Dureja

Computer/Electrical Engineering Capstone Design Project
Ryerson University, Year 4

Acknowledgements

First, we have to thank our research supervisor, Dr. Nagi Mekhiel. Without their assistance and dedicated involvement in every step throughout the process, this project would have never been accomplished.

Getting through our dissertation required more than academic support, and we have many people to thank for helping us in various ways. We cannot begin to express our gratitude and appreciation for their friendship. James Kotic, Shaan Hossain, Vahe Poghosyan, and Arjun Dureja have been unwavering in their personal and professional support during the time we spent at the University.

Most importantly, none of this could have happened without our families. To our parents and grandparents – it would be an understatement to say that, as a family, we have experienced some ups and downs in the past year. This report stands as a testament to your unconditional love and encouragement.


Certification of Authorship

We hereby certify that we are the authors of this document and that any assistance we received in its preparation is fully acknowledged and disclosed in the document. We have also cited all sources from which we obtained data, ideas, or words that are copied directly or paraphrased in the document.

Student A

Name: James Kosic

Student#: 500885336

Signature: 

Student B

Name: Shaan Hossain

Student#: 500882569

Signature: 

Student C

Name: Vahe Poghosyan

Student#: 500919654

Signature: 

Student D

Name: Arjun Dureja

Student#: 500884644

Signature: 

Table of Contents

Acknowledgements	2
Certification of Authorship	3
Table of Contents	4
Abstract	5
Introduction & Background	6
Objective	7
Theory and Design	8
Parallel Computing Theory	8
Data Sources	10
Sliding Window Methodology	11
Design of Sequential Prototype	12
Data Dependencies	16
Message Passing Interface	17
Design of Parallel Prototype	19
Final Design of Sequential Program	20
Final Design of Parallel Program	23
Alternative Designs	25
Decision-Tree Algorithm	25
Multi-Day Forecast Implementation	25
MPI/OpenMP Hybrid Implementation	27
Material/Component List	28
Performance Accuracy Results/Testing Procedures	29
Accuracy Methodology, Parameters, and Dataset Metrics	29
Dataset Parameter Plots	30
Sample Computations for Prediction Parameter Accuracy	31
Tabulation of Accuracy Results	32
Analysis of Algorithm Accuracy	33
Analysis of Performance	34
Performance Gain	34
Conclusions	37
References	38
Appendices	39
Sequential Python Prototype	39
Parallel Python Prototype	44
Sequential C Program	47
Parallel C Program	56

Abstract

In this project, the goal is to examine the real-world applications of parallel processing. Throughout the semester, it was discovered that parallel processing is a powerful tool that is used in many aspects of day-to-day life. To fully grasp the importance of this computing technique, the team underwent a multi-stage research and experimentation process. The first step in fulfilling this objective was to understand the theory and logic behind parallel processing. The prominent parallelization theory that this project was designed around was Amdahl's law, which gave a further understanding of the capabilities and limitations of parallel processing, aiding in implementation. The second step once the theory was discovered, was to decide on a problem to which this theory could be applied. The topic of weather forecasting was chosen, in which a program parses massive amounts of data to output a prediction of future weather conditions. This was selected as the topic as it has great potential to display the improvements when parallel processing is applied to a problem. After theory discussion and topic selection, the matter of implementation and application had to be addressed. In the way of weather forecasting, an algorithm had to be picked which would be suitable for parallelization. The sliding window algorithm was the final decision, as it was the most practical for the purpose of this project. Furthermore, the sliding window was an obvious choice as it had the capability to be parallelized without massive alterations to the functionality of the algorithm. Once a prototype of the algorithm applied to the problem of weather forecasting was implemented, the parallelization of the program began. In order to parallelize the prototype, Message Passing Interface (MPI) was used. Finally, with help from the supervisor, along with research and many hours spent programming, a finished product was created. This program successfully demonstrated the ability that parallel computing has to take a task that would otherwise be too long and costly to be viable and break it down into smaller, easier to compute operations. Thus, the objective of this project was achieved, and the step-by-step process will be further explored in this report.

Introduction & Background

This project is relevant in the field of Computer Engineering, specifically hardware engineering, as it defines the importance and pitfalls of hardware optimization. Something taught in engineering courses is that for a computer system to operate at maximum efficiency, both the hardware and the software must be optimized to a tee. To complete this project, the hardware aspect must be examined carefully, and the bottlenecks that come with optimization via parallelization must be examined. The end goal of this project is to create a predictive weather forecasting algorithm using given data sets and chosen algorithms on a parallel processing unit, and compare the gain as the amount of processing power is increased. First, a commonly used algorithm in the field of weather prediction will be selected. This algorithm must require significant computing power to be run sequentially. Then, this algorithm will be analyzed and developed in a selected programming language. After running the algorithm on a specific dataset, its accuracy will be analyzed and the parameters will be adjusted accordingly. Once finalizing the sequential code, an effort will be made to parallelize it. This will be done with the use of low-level programming libraries that allow interfacing with individual processes and threads. Using concepts researched earlier, they will be applied to the sequential program to create a highly efficient parallel program that delivers a large performance improvement. To wrap up the project, the parallel program will be thoroughly tested and examined to understand the improvements and bottlenecks. Finally, a conclusion will be made that will determine if parallel computing is suitable for the chosen application.

Objective

In this project, the goal is to deepen the understanding of parallel computing and to reinforce the importance of hardware and software optimization. This was accomplished through the application of parallel processing on a weather forecasting program, by examining and analyzing parallel computing theory, various programming models, algorithms for parallel computing, and performing a design and synthesis for implementing the parallel program. Key concepts were explored such as Amdahl's Law, with a brief touch on alternative theories. Additionally, an approach and various methods have been analyzed such as runtime optimization comparisons between MPI, OpenMP, and MPI+OpenMP along with parallelization of code through data dependency.

Theory and Design

In this section, the theory which was researched in the first semester will be explained further, and the design process will be illustrated.

Parallel Computing Theory

In order to optimize a program via parallelization, the first area of concern is the amount of the program which is able to undergo parallelization versus the amount of the program which must remain serialized. The performance gain is directly related to the number of resources dedicated to the parallelized program, however, there is a plateau of improvement as the gain begins to decrease. The theory used in this project for calculating gain is called Amdahl's Law [2], which states that the latency of a task can only be improved by parallelization as much as the parts of the program that can be parallelized. A conclusion can be drawn that no matter how many resources are diverted to the program, serial code will not see any performance gain. Amdahl's Law is explained mathematically in Equation (1).

$$v = \frac{1}{(1-P) + \frac{P}{S}} \quad (1)$$

Where v is the gain, P is the portion of the task accelerated, and S is the speedup of the portion of the task which is being optimized [3]. This equation relates to the gain of the project by adding more processors. Another theory that was considered for this project was Gustafson's law [1], which is a modernized version of Amdahl's Law. This law differs from Amdahl's in the degree of speedup increase, where Amdahl's gain is considered exponential, however, Gustafson's is considered linear [4]. This theory was not pursued further as it is more well suited for a large number of processors (the law was designed around a hypercube with 1024 processors) which is not appropriate for the hardware in use. Please see below for a graphical representation of both laws proposed in this section. (Figures 1.1 and 1.2).

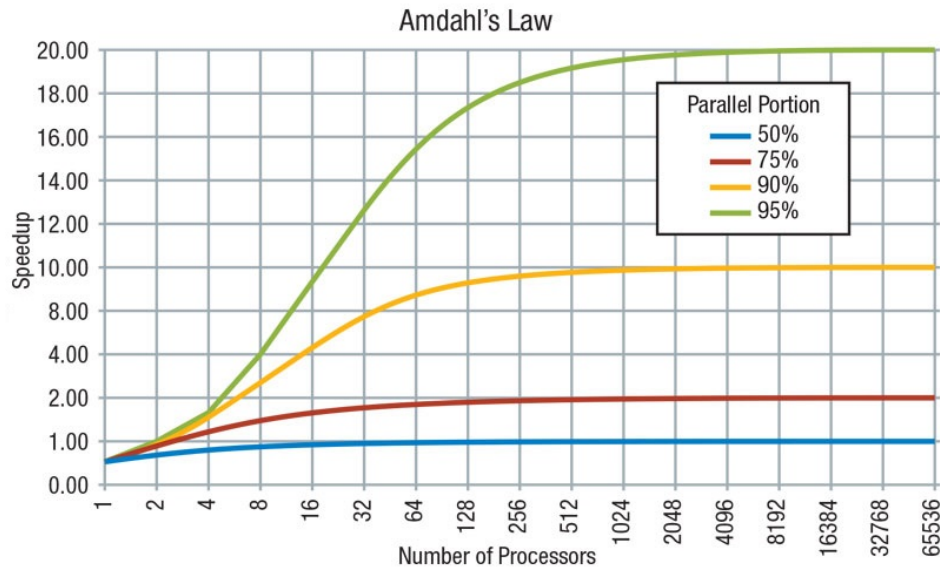


Figure 1.1: Graphical representation of Amdahl's Law.

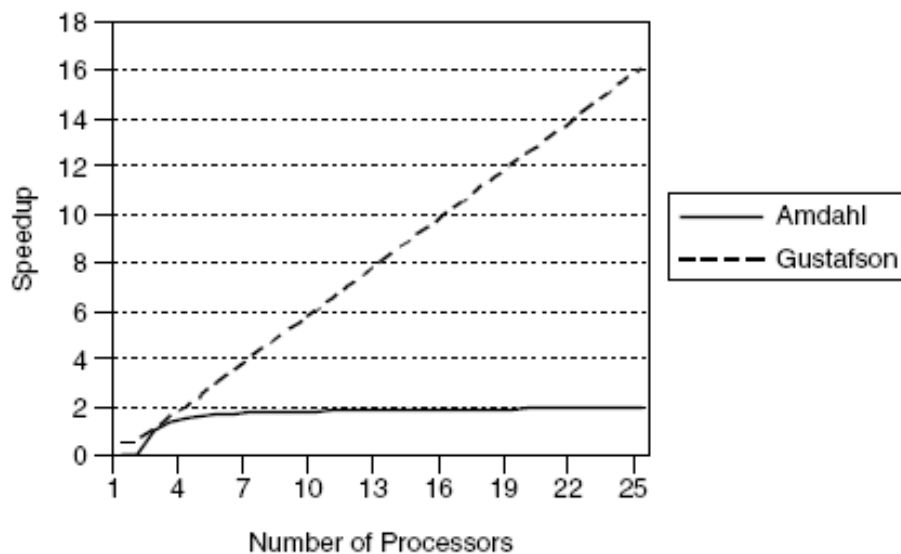


Figure 1.2: Graphical representation of Amdahl's Law vs Gustafson's Law.

When Amdahl's Law is applied to the sequential prototype using theoretical gain numbers, the following graph is obtained. This graph illustrates the decrease in execution time of the program as the amount of processing power is increased. This behavior is to be expected as it matches that of Amdahl's theory. The increase of performance of the prototype plateaus at around 70 processors, where any more power given to the project does not result in a noticeable further gain.

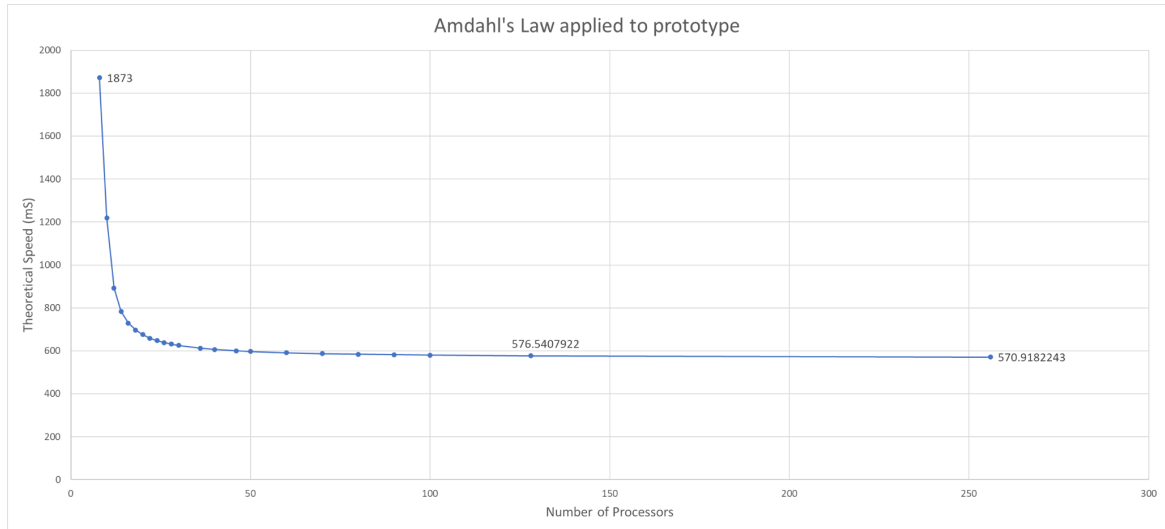


Figure 1.3 - Graph of Theoretical Decrease in Time in Sequential Prototype.

Data Sources

The data used is gathered from the official Canadian Government Climate website. The location used for forecasting in the first prototype is the Toronto Pearson International Airport located in Mississauga, Ontario. These datasets range from 2013 to the present day, thus providing us with enough information to test the functionality of the implemented algorithm. The dataset contains a maximum of 11 variables, some of which are unavailable on a per-season basis. For the initial prototype in python, a predefined set of data was downloaded and used in the program; The Canadian Climate website provides a list of bash commands that can be used to directly access a specified set of data. Below is an example provided of using the bash command to get a specified set of data.

```
Monthly data interval (station complete history):
for year in `seq 1998 2008`;do for month in `seq 1 1`;do wget
--content-disposition
"https://climate.weather.gc.ca/climate_data/bulk_data_e.html?format=csv&st
ationID=1706&Year=${year}&Month=${month}&Day=14&timeframe=3&submit=
Download+Data" ;done;done
```

Having the ability to parse the data through bash commands is extremely useful when working with C. Although some of the variables are found in the datasets such as “Total Rain” and “Snow on ground” are to be calculated based on the forecasting season, the common variables such as “Minimum Temperature”, “Maximum Temperature”, “Mean Temperature” and “Wind Speed” will be the main priority of forecasting with this program.

For the final sequential prototype, the program is responsible for forecasting weather conditions of twenty major cities in Canada. The datasets for the twenty cities were downloaded and stored in two-dimensional arrays. The set of weather conditions being forecasted is as follows:

1. Maximum Temperature (°C)
2. Minimum Temperature (°C)
3. Mean Temperature (°C)
4. Heating Degree Days
5. Cooling Degree Days
6. Total rain (mm)
7. Total Snow (cm)
8. Total Precipitation (mm)
9. Snow on Ground (cm)

Predicting nine weather conditions over 365 days of twenty datasets requires a large amount of computation power, hence why an efficient algorithm is required. The next section will go into detail about the selected algorithm and its functionality.

Sliding Window Methodology

The sliding window algorithm is a common algorithm used for weather forecasting. The algorithm has an average accuracy of 92.2% [5]. As the algorithm consists of numerous matrix multiplications, it required a lot of computation time for large datasets. A step-by-step description of the algorithm is listed in [5, Alg. 1]. The sliding window algorithm can forecast an **n** number of weather conditions for a future day **D** (Forecast Day) using the last seven days of data preceding day **D** and a set of fourteen days from the previous year around the same timeframe as day **D**. As described in the Data Sources section above, there are a total of nine weather conditions being predicted for each day of the forecast. The following is an in-depth explanation of each step of the sliding window algorithm as shown by P. Kapoor in [5, Alg. 1].

Since nine weather conditions are being predicted, the first step is to take the previous seven days of data from day **D** and put it in a 7 X 9 matrix labeled as “CD”. The second step is to get a fourteen-day dataset from the previous year's dataset as a 14 X 9 matrix. The timeframe of the data is selected in a way to allow day **D** to approximately me in the center of the timeframe. This 14 X 9 matrix is labeled as “PD”. The “PD” matrix allows us to create eight 7 X 9 matrices as shown in [5, Fig. 1]. These matrices are labeled as $W_1, W_2 \dots W_8$ and are utilized in the next step. The 7 X 9 matrices are the sliding windows and are used in step four to find the sliding window with the nearest match to the “CD” matrix in terms of dataset values. To do so, the Euclidean distance between each sliding window and the “CD” matrix is computed using the Euclidean distance formula (2) below:

$$d(p, q) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2} \quad (2)$$

The Euclidean distance formula is used to calculate the distance between two points in Euclidean space, which results in the distance between two points. The Euclidean distance formula can be utilized to find the closes matching sliding window to the matrix “CD” by using the formula. The mean of each weather condition is calculated for the matrix “CD” and each sliding window. The mean of each variable in the “CD” matrix is q_i and the mean of each variable in each sliding window is p_i . Each difference is squared and the sum of the difference is square rooted to find the Euclidean distance between “CD” and each sliding window. Step five is to select the sliding window with the lowest Euclidean distance as the sliding window to be used for predicting the weather conditions of day **D**.

In step six, the variation vectors are computed for each weather condition. To compute a variation vector, the difference of each parameter **n** compared to other days is stored in a 6 X 1 matrix. Variation vectors are computed for each variable of both the “CD” and the “PD” matrices and stored in matrices “VC” and VP” respectively. Afterward, the mean of each matrix is calculated and the average of the means is stored as a prediction for each prediction parameter.

Design of Sequential Prototype

The sequential prototype of the program is implemented using Python. The reason behind implementing the initial prototype in Python is to use the “matplotlib” module in Python to create visual representations of the forecasted data. As previously mentioned, the prototype uses the sliding window algorithm and is capable of forecasting nine weather conditions per day for 365 days in twenty locations. The datasets are imported into DataFrames and stored in an array using the “pandas” module. The datasets are illustrated in Figure 2.1 below.

```

data_files_2020 = [
    pd.read_csv("data/2020_tor_data.csv"),
    pd.read_csv("data/2020_van_data.csv"),
    pd.read_csv("data/2020_mtl_data.csv"),
    pd.read_csv("data/2020_cal_data.csv"),
    pd.read_csv("data/2020_edm_data.csv"),
    pd.read_csv("data/2020_ham_data.csv"),
    pd.read_csv("data/2020_hfx_data.csv"),
    pd.read_csv("data/2020_kel_data.csv"),
    pd.read_csv("data/2020_kit_data.csv"),
    pd.read_csv("data/2020_lon_data.csv"),
    pd.read_csv("data/2020_osh_data.csv"),
    pd.read_csv("data/2020_ott_data.csv"),
    pd.read_csv("data/2020_peg_data.csv"),
    pd.read_csv("data/2020_qbc_data.csv"),
    pd.read_csv("data/2020_reg_data.csv"),
    pd.read_csv("data/2020_sas_data.csv"),
    pd.read_csv("data/2020_stcath_data.csv"),
    pd.read_csv("data/2020_stjn_data.csv"),
    pd.read_csv("data/2020_vic_data.csv"),
    pd.read_csv("data/2020_win_data.csv"),
]

data_files_2021 = [
    pd.read_csv("data/2021_tor_data.csv"),
    pd.read_csv("data/2021_van_data.csv"),
    pd.read_csv("data/2021_mtl_data.csv"),
    pd.read_csv("data/2021_cal_data.csv"),
    pd.read_csv("data/2021_edm_data.csv"),
    pd.read_csv("data/2021_ham_data.csv"),
    pd.read_csv("data/2021_hfx_data.csv"),
    pd.read_csv("data/2021_kel_data.csv"),
    pd.read_csv("data/2021_kit_data.csv"),
    pd.read_csv("data/2021_lon_data.csv"),
    pd.read_csv("data/2021_osh_data.csv"),
    pd.read_csv("data/2021_ott_data.csv"),
    pd.read_csv("data/2021_peg_data.csv"),
    pd.read_csv("data/2021_qbc_data.csv"),
    pd.read_csv("data/2021_reg_data.csv"),
    pd.read_csv("data/2021_sas_data.csv"),
    pd.read_csv("data/2021_stcath_data.csv"),
    pd.read_csv("data/2021_stjn_data.csv"),
    pd.read_csv("data/2021_vic_data.csv"),
    pd.read_csv("data/2021_win_data.csv")
]

```

Figure 2.1: Dataset Arrays in Python Prototype.

The “data_files_2021” array is used for illustrating a comparison between the predicted weather parameters and the real-world dataset. The resulting accuracy and weather condition graphs will be illustrated in the Performance Measurement Results section. The python prototype has a function named “forecast_day” which takes parameters of the day for prediction, the previous year's data, and the current year's data. The function then gathers the “CD” and “PD” matrices from these datasets based on the day parameter of the function. Afterward, the means for all parameters are calculated for the “CD” matrix. An iterator passes through each sliding window and calculates the means for each variable in the current sliding window. Next, the program computes the Euclidean distance for each sliding window and stores it in an array with the respective index of the sliding window. This step is illustrated in Figure 2.2 below.

```

# calculating means for present year 7 days
s4 = time.time()
present_sums = [0] * NUM_OF_PARAMS
for line in CD:
    for i in range(len(line)):
        if not math.isnan(line[i]): present_sums[i] += line[i]
present_means = get_mean(present_sums)

# mean of parameters for each window
ed_list = []
for window in windows:
    prev_sums = [0] * NUM_OF_PARAMS
    for vals in window:
        for i in range(len(prev_sums)):
            prev_sums[i] += vals[i]
        prev_means = get_mean(prev_sums)

    euclid_distance = sum([(present_means[i] - prev_means[i]) ** 2 for i in range(len(present_means))])
    ed_list.append(euclid_distance)
s4e = time.time()

```

Figure 2.2: Mean Calculation and Euclidean Distance Computation For Each Sliding Window.

As explained in the sliding window methodology section above, after locating the sliding window with the smallest Euclidean distance, the variation vectors of the matrices “CD” and “PD” are calculated. Finally, the average of the means of each weather condition is computed and stored as the prediction for that variable. This process is illustrated in Figure 2.3 below.

```

present_variation_vector = []
for i in range(len(CD[0])):
    vector = []
    for j in range(len(CD)-1):
        vector.append(CD[j+1][i] - CD[j][i])
    present_variation_vector.append(vector)

prev_variation_vector = []
for i in range(len(selected_prev_window[0])):
    vector = []
    for j in range(len(selected_prev_window)-1):
        vector.append(selected_prev_window[j+1][i] - selected_prev_window[j][i])
    prev_variation_vector.append(vector)

mean_present_var = []
for i in range(len(present_variation_vector)):
    mean_present_var.append(sum(present_variation_vector[i]) / len(present_variation_vector[i]))

mean_prev_var = []
for i in range(len(prev_variation_vector)):
    mean_prev_var.append(sum(prev_variation_vector[i]) / len(prev_variation_vector[i]))

mean_variation_vector = []
for i in range(len(mean_present_var)):
    mean_variation_vector.append((mean_present_var[i] + mean_prev_var[i]) / 2)

prev_day = CD[-1]
for i in range(len(prev_day)):
    prev_day[i] += mean_variation_vector[i]

```

Figure 2.3: Step 6 of Sliding Window Algorithm.

A sample prediction for one day is shown in Figure 2.4 below. This illustration does not include all the predicted parameters as depending on the season of the year, some of those parameters are simply zero.

Max Temp (°C)	Min Temp (°C)	Mean Temp (°C)	Heat Deg Days (°C)	Cool Deg Days (°C)
Actual	4.1	-3.8	0.2	17.8
Prediction	5.81	-0.77	2.57	15.42
Accuracy	99.384%	98.877%	99.131%	99.184%

Figure 2.4: Sample Forecast For One Day.

Calculating accuracy of temperature is slightly inaccurate on the Celsius scale; to calculate accuracy, the Kelvin scale is used. The temperatures are converted from Celsius to Kelvin using equation 3 and the accuracy is calculated using equation 4 below.

$$T_K = T_C + 273.15 \quad (3)$$

$$Accuracy = 1 - \left(\frac{|Actual - Prediction|}{Actual} \right) * 100 \quad (4)$$

Finally, the runtime of the program for one city and twenty cities are listed below. There is a significant increase in computation time, thus the program will greatly improve with the implementation of parallel programming.

Number of Cities	Runtime (in seconds)
1	0.425
20	8.386

Figure 2.5: Table of Number of Cities Compared to Runtime.

Note: the runtime of the program may vary depending on the single-threaded performance of the machine it's running on.

Data Dependencies

During the parallelization of the algorithm, the identification of data dependencies within an algorithm is vital to improving its performance via parallelization. In particular, analyzing loop statements within the sliding window algorithm and verifying the execution of statements, such that the data storage is accessed by successive statements within each iteration [8] (loop independent dependence). An example of this can be shown below in the Python prototype, where data storages of `present_max`, `present_min`, `present_rain`, `max_sum`, `rain_sum`, and `min_sum` are accessed and then incremented by different parsed values of the dataset, aforementioned in the previous section, for the calculations of the mean temperatures at each window. Since both loops are loop independent, which can be executed independently and asynchronously, this section of this code can be parallelized with the use of MPI and Processes/Threads without affecting the functionality of the algorithm.

Each MPI process spawns multiple OpenMP threads

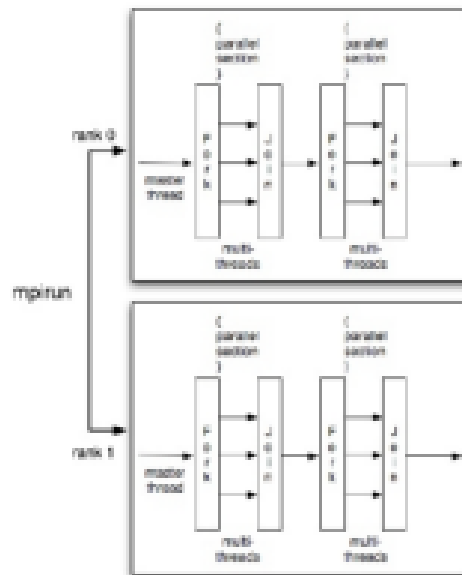


Figure 2.5: Use of Processes to Spawn Threads for Parallel Execution.

```
for line in CD:
    present_max += line[0]
    present_min += line[1]
    present_rain += line[2]
```

```
# mean of parameters for each window
for line in w_dct:
    max_sum = min_sum = rain_sum = 0
    for values in w_dct[line]:
        max_sum += values[0]
        min_sum += values[1]
        rain_sum += values[2]
```

Figure 2.6: Examples of Loop-Independent Dependence Within the Prototype.

Message Passing Interface

Message Passing Interface (MPI) is a standardized library widely used in parallelism and high-performance computing. It allows for the exchange of messages across several computers/processes running the same program. This means computation can be evenly split between processes to significantly reduce execution time. It provides various commands which can be used to obtain details about the parallel program as well as to facilitate communication between the processes. An example of commands is “MPI_Comm_size” and “MPI_Comm_rank” which return the number of processes and the current rank respectively. It is important to know this information because it is useful when splitting computation and assigning tasks to certain processes. A simple example of an MPI program can be seen in the figure below.

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {

    int rank, size;

    MPI_Init (&argc, &argv); //initialize MPI library

    MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my process id

    //do something
    printf ("Hello World from rank %d\n", rank);
    if (rank == 0) printf("MPI World size = %d processes\n", size);

    MPI_Finalize(); //MPI cleanup

    return 0;
}
```

Figure 2.7: Example MPI Program [6].

There are two types of communication methods in MPI: point-to-point and collective. Point-to-point communication involves sending and receiving data to and from one process to another. Figure 2.7 showcases how this works. The two commands used are “MPI_Send” and “MPI_Recv” which require information such as the data, data type, number of elements, destination ID, etc. Both of these commands are considered “blocking”. This is because when a process calls the receive command, it enters a blocked state while it waits for another process to call the send command. This is important when analyzing performance because a process that is blocked for too long can cause a bottleneck in a parallel program.

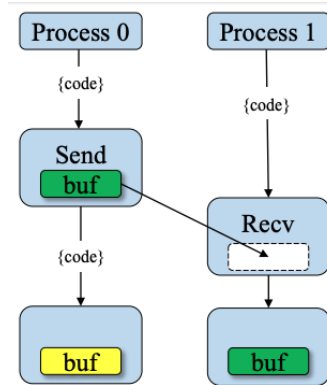


Figure 2.8: Point-to-Point Communication Between Processes.

Collective communications are far more advanced than point-to-point. It allows for 1-to-many, many-to-1, and many-to-many communication between processes. Examples of collective communications are “MPI_Scatter” and “MPI_Gather”. Figure 2.8 illustrates these two commands. Scatter is used to distribute pieces of data from one process to several others. The data can be represented as an array and MPI will automatically split it up so that each process gets a unique piece. Every process will need to call this command when it is being used, the root process is inputted so that MPI knows where the data originates from. When the child processes are done working on the data, they can send it back using the gather command. MPI will take each piece of data and re-construct the array in the root process. There are also other forms of collective communications such as “MPI_Bcast” and “MPI_Allgather” which are used for different purposes.

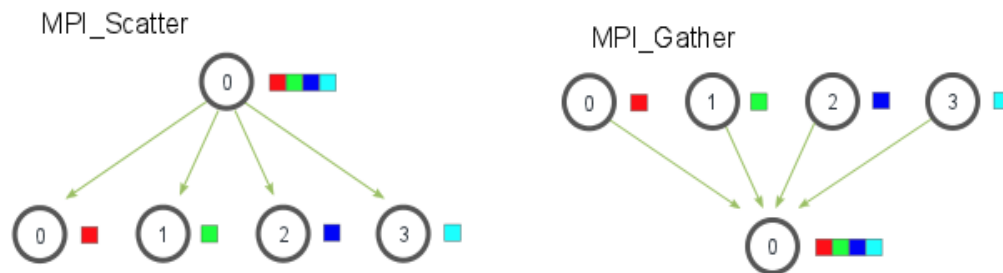


Figure 2.9: Illustration of MPI_Scatter and MPI_Gather [8].

Another useful command is “MPI_Barrier”. This is used for process synchronization. Given that all the processes are executing independently, it is unlikely that they will finish their tasks at the same time. There are instances when these processes need to be forced to wait for the other processes to finish before moving on. This is where the barrier command can be used. It will block the process until every process has called the command, then it will continue its execution. The figure below showcases this.

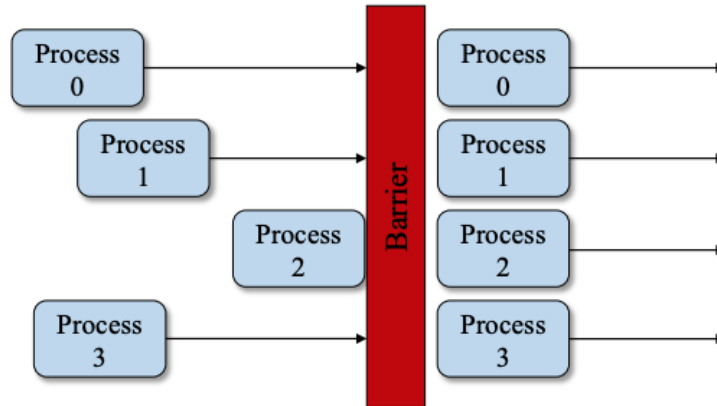


Figure 2.10: MPI_Barrier Being Used on 4 Processes.

In order to run a program in MPI, it needs to be compiled using a compiler that has MPI implemented. For this project, MPICH will be used. The compile command is “mpicc” and is used like other C compilers such as GCC or Clang. To run the program, the command “mpirun” is used. This command allows the user to specify arguments to pass into the program such as the number of processes. Users may also specify a list of hosts within this command. The hosts are separate computers running the same program. This is useful when one computer does not have the required number of CPU cores. When specifying multiple hosts, MPI will open a TCP connection between them to pass messages. An example of an MPI program being run can be seen in the figure below.

```

~/COE70B-Engineering-Design-Project main*
> mpirun hello_world -np 4
Hello world from processor rank 2 out of 4 processors
Hello world from processor rank 1 out of 4 processors
Hello world from processor rank 0 out of 4 processors
Hello world from processor rank 3 out of 4 processors

```

Figure 2.11: Running an MPI Program.

Design of Parallel Prototype

The prototype for the parallel program was developed as a simple proof of concept for parallel computing. The purpose of the prototype was to determine how this specific program might be parallelized. It gave insight as to how the final design should work and what can be expected from it. The program would predict a specific number of days based on the number of processes for a single location. For example, if there were four processes, it would predict three days of weather since one process is dedicated to sending and receiving data while the others perform the prediction. This was done using point-to-point communications. First, the root process would read the data from the CSV files and then filter it to retrieve the days needed.

After this, using a loop, it uses the send command to distribute the data to the other processes. The child processes then use the receive command to retrieve the data and begin predicting their designated day. The parallel portion of this program along with its output can be seen in the figures below.

```
if rank == 0:
    data_20 = pd.read_csv("data/2020_tor_data.csv")
    data_21 = pd.read_csv("data/2021_tor_data.csv")
    data_21.head()
    present_days = data_21[data_21["Month"] == 6]
    prev_days = data_20[data_20["Month"] == 6]

    for i in range(size-1):
        comm.send(data_20, dest=i+1)
        comm.send(data_21, dest=i+1)
        comm.send(present_days, dest=i+1)
        comm.send(prev_days, dest=i+1)
else:
    start = MPI.Wtime()

    data_20 = comm.recv(source=0)
    data_21 = comm.recv(source=0)
    present_days = comm.recv(source=0)
    prev_days = comm.recv(source=0)
```

Figure 2.12: Parallel Prototype Code.

```
~/COE70B-Engineering-Design-Project main*
> mpirun --oversubscribe -np 4 python3 weather_mpi.py

Prediction for June 16, 2021
-----
Max Temp | Min Temp | Rainfall
Actual:   22.3 | 10.5 | 0.0
Predicted: 22.28 | 12.53 | -0.45

Time taken to predict weather by process 1: 0.024512

Prediction for June 17, 2021
-----
Max Temp | Min Temp | Rainfall
Actual:   26.3 | 9.2 | 0.0
Predicted: 22.4 | 10.09 | -0.45

Time taken to predict weather by process 2: 0.026903

Prediction for June 18, 2021
-----
Max Temp | Min Temp | Rainfall
Actual:   24.9 | 18.0 | 5.4
Predicted: 25.7 | 8.13 | 0.0

Time taken to predict weather by process 3: 0.03125
```

Figure 2.13: Parallel Prototype Output.

The program is run using the “mpirun” command and four processes are specified. As expected, it predicts three days worth of weather data. The program also outputs how long each process took to execute. This is done using the “MPI_Wtime” command to keep track of watch time. Each process takes approximately the same amount of time to compute their respective days. This means that they are truly running in parallel, fully independent from each other. Also, since the predicted values are accurate, the point-to-point communications were executed successfully.

Final Design of Sequential Program

The final sequential program, which will be parallelized afterward, was developed in C. This program is functionally identical to the Python prototype. It is essentially a translation of the Python code to C code. This brings forth a considerable challenge due to the program needing to be written from scratch. For time-saving purposes, the Python program made use of libraries such as Numpy and Pandas. These libraries provided functionality for reading from files and manipulating matrices. These needed to be written from scratch in C, which required a strong understanding of areas in low-level programming such as pointers and memory management. It is important that these functions are written as efficiently as possible to ensure that no bottlenecks are present.

The sequential program was split into three files: `csv.c` which contained a CSV reader, `utils.c` which contained utility functions such as a matrix filterer, and `weather.c` which contained the code for the algorithm itself. The CSV reader takes a three-dimensional array of characters representing the result and a file name string. It makes use of the “fopen” and “fgets” APIs to read each cell in the CSV file and copies its contents to the result array. The matrix filtering function takes two three-dimensional arrays of characters, one representing the input matrix, and one the output. It also takes a start row, end row, an array of integers representing which columns need to be filtered, and the size of the columns array. The algorithm loops through the matrix, only appending the values contained within the inputted bounds to the output matrix. The `utils` file also contains functions for calculating the euclidean distance between two lists, and a function for locating the minimum value in an array. The definitions for the four functions can be seen in the figure below.

```

void filterMatrix(
    char matrix[][50][50],
    char result[][50][50],
    int startRow,
    int endRow,
    int *colsToFilter,
    int numColsToFilter
);

double calcEuclidianDistance(
    double *presentMeans,
    double *prevMeans,
    int numPresentMeans
);

double min(double *arr, int len);

void readCSV(
    char result[][50][50],
    char * filename
);

```

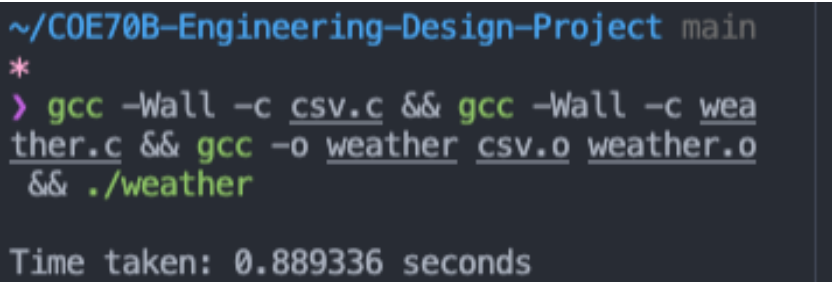
Figure 2.14: Definitions of Functions Used in the Sequential Program.

The sliding window algorithm itself computes the same six steps implemented in the Python prototype. It is abstracted into a function that is meant to predict the weather for a given day. The function takes two arrays of type `double` representing the final prediction and actual values respectively. It also takes two three-dimensional arrays of characters retrieved from the CSV function representing the present data and previous data. Lastly, it takes the day of the year desired to be predicted, a list of parameters that should be included in the prediction, and the size of the parameter list. The definition for this function can be seen in the following figure.

```
// Predicts the weather of a particular day of a year
void predictWeatherForGivenDay(
    double *prediction,
    double *actual,
    char presentData[][50][50],
    char prevData[][50][50],
    int day,
    int *cols,
    int numParams
) {
```

Figure 2.15: Definition of the Prediction Function.

The function itself first calls the matrix filterer to retrieve the present and previous days for the specified day of the year. It then performs the sliding window algorithm and assigns the output to the inputted prediction array. The main function makes calls to the prediction function to retrieve predictions for the twenty cities previously mentioned. It does this by looping through the list of locations, calling the CSV reader to retrieve the data for that location, and finally loops 351 times calling the prediction function for each day of the year. The output of the sequential program can be seen in Figure 2.16.



```
~/COE70B-Engineering-Design-Project main
*
> gcc -Wall -c csv.c && gcc -Wall -c weather.c && gcc -o weather csv.o weather.o
&& ./weather
Time taken: 0.889336 seconds
```

Figure 2.16: Output of Final Sequential Program.

The program simply prints out the execution time to predict the weather for all twenty cities. The prediction values can also be printed but that affects the execution time and provides an inaccurate result. As expected, the C program is significantly faster than the Python prototype given that C is compiled and Python is interpreted. The C program is approximately 17 times faster. This, however, can still be improved with the introduction of parallel computing which will be discussed in the next section.

Final Design of Parallel Program

The finalized design consists of a program developed in the C language with the use of the MPI. The program was developed on top of the final sequential C program. This program will be run across several processes so it is important to evenly distribute the workload. The final technique decided to parallelize the program is to split the number of days being predicted by the number of processes. Given that there are twenty locations and 351 days per location, there are 7020 days being predicted in total. This number represents the number of times the prediction function is called in the sequential program. Considering that each of these function calls do not depend on each other, they can be called in parallel. For example, if there are four processes, each process would compute 1755 days of weather. The first step to achieving this implementation is to write an algorithm that evenly splits the number of days between the processes. This can be seen in the figure below.

```
// Divide days evenly between processes
int count = 344 / numProcs;
int remainder = 344 % numProcs;
int startDay, endDay;
if (rank < remainder) {
    startDay = rank * (count + 1);
    endDay = startDay + count;
} else {
    startDay = rank * count + remainder;
    endDay = startDay + (count - 1);
}
```

Figure 2.17: Splitting the Number of Days Between Processes.

This simple algorithm determines the loop bounds for each process on each iteration. The rank and number of processes are retrieved using the “MPI_Comm_rank” and “MPI_Comm_size” commands. The algorithm ensures that even if the division results in decimals, it adds a remainder so that the value remains an integer. For example, the start day and end day for the first process could be 0 and 50, the start day and end day for the second process could be 51 and 100, and so on. No overlaps can occur here, otherwise, two processes would be predicting the same day, resulting in a waste of resources.

After implementing this feature, the program does not seem to have a large improvement in performance. This is due to a bottleneck caused by an abundance of I/O operations. Since the program is being run across several processes, the CSV reader is being called an unnecessary amount of times. The sequential program called it 40 times, twice for each location loop iteration. The number of times the parallel program is calling it is 40 multiplied by the number of processes. This is a very large number of I/O operations that are known to be quite slow. To solve this problem, a collective communication within MPI can be used. The chosen command is “MPI_Bcast”. This communication allows a single process to broadcast data to every other process in a highly efficient manner. An illustration of how this works can be seen in the figure below.

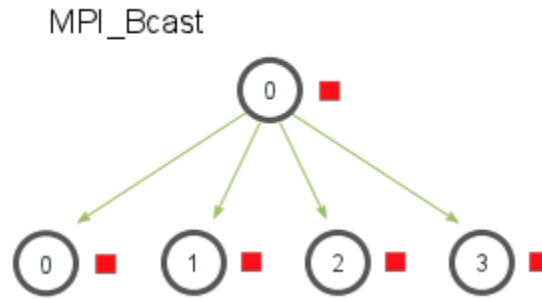


Figure 2.18: Illustration of MPI_Bcast.

To implement this optimization in the program, a conditional statement can be added to only allow the root process to call the CSV reader. After reading from the CSV files, every process can call the broadcast command to retrieve that data. Therefore, the number of I/O operations is reduced to the same amount as the sequential program, thus increasing the performance of the program. The implementation of this optimization can be seen in the figure below.

```

if (rank == 0) {
    // Get data from CSV files
    readCSV(presentData, presentCSV[location]);
    readCSV(prevData, prevCSV[location]);
}

// Retrieve data from rank 0
MPI_Bcast(&presentData, 400*50*50, MPI_CHAR, 0, MPI_COMM_WORLD);
MPI_Bcast(&prevData, 400*50*50, MPI_CHAR, 0, MPI_COMM_WORLD);

```

Figure 2.19: Implementation of MPI_Bcast Optimization.

After implementing this, the performance improved drastically. The benefits of parallel computing were clearly shown. The output of the program after running it on four processes using the “mpirun” command can be seen in Figure 2.20. The overall performance of the program will be thoroughly analyzed in the Performance Measurement Results section.

```

~/COE70B-Engineering-Design-Project main*
> mpicc -Wall -c csv.c && mpicc -Wall -c utils.c && mpicc -Wall -c weather_mpi.c &
& mpicc -o weather_mpi csv.o utils.o weather_mpi.o && mpirun --oversubscribe -np 4
weather_mpi

Time taken: 0.152085 seconds

```

Figure 2.20: Output of Final Parallel Program.

Alternative Designs

This section goes into more detail regarding alternative designs that were discussed during the timeline of this project. Each alternative design is briefly explained, and the reasoning behind choosing a different design is stated.

Decision-Tree Algorithm

The decision tree algorithm is a data mining algorithm that is commonly used in the machine learning field. When researching weather forecasting algorithms in the fall semester, the decision tree algorithm seemed to be the most suitable for this project. As stated by Y. Song, “decision trees are one of the most effective methods for data mining; they have been widely used in several disciplines because they are easy to be used, free of ambiguity, and robust even in the presence of missing values” [9, p. 130]. When researching, the properties of decision trees were suitable for this project due to real-world datasets having numerous missing values due to human error.

The benefits of using a decision tree algorithm for this project were that the algorithm itself can be easily parallelized, thus taking advantage of parallel programming. The decision tree algorithm has many properties such as branches, splitting, stopping, and pruning, which was explained in detail in the final report of COE70A.

As the decision tree algorithm required various machine learning libraries, it became difficult to implement the algorithm without heavily relying on external libraries. Relying on external libraries is an issue since the final parallel program uses MPI, which requires the program to be written in the C programming language. Implementing some of the required machine learning libraries in C requires previous experience in machine learning and other libraries are simply not supported in C. Implementing the not supported libraries would have to be written from scratch, thus heavily impacting the timeline of this project.

Multi-Day Forecast Implementation

As described in the Sliding Window Methodology section, the sliding window algorithm is capable of forecasting weather conditions for one day at a time. The implementation of the multi-day forecast program was a modification to the main sequential program. The multi-day forecast program was implemented in Python to visualize the forecasted data. As the main sequential program stored the forecasted data after iterating through each day, the multi-day forecast program uses the forecasted day as part of the set for the next prediction.

This program was responsible for using predicted data to predict the weather for the first two months of the year 2022. The only data required for executing the multi-day forecast program was the first seven days of data from 2022 and a dataset containing all data from the year 2021. Figure 3.1 below illustrates the maximum temperature ($^{\circ}\text{C}$), the minimum temperature ($^{\circ}\text{C}$), the mean temperature ($^{\circ}\text{C}$), and the heating degree days ($^{\circ}\text{C}$).

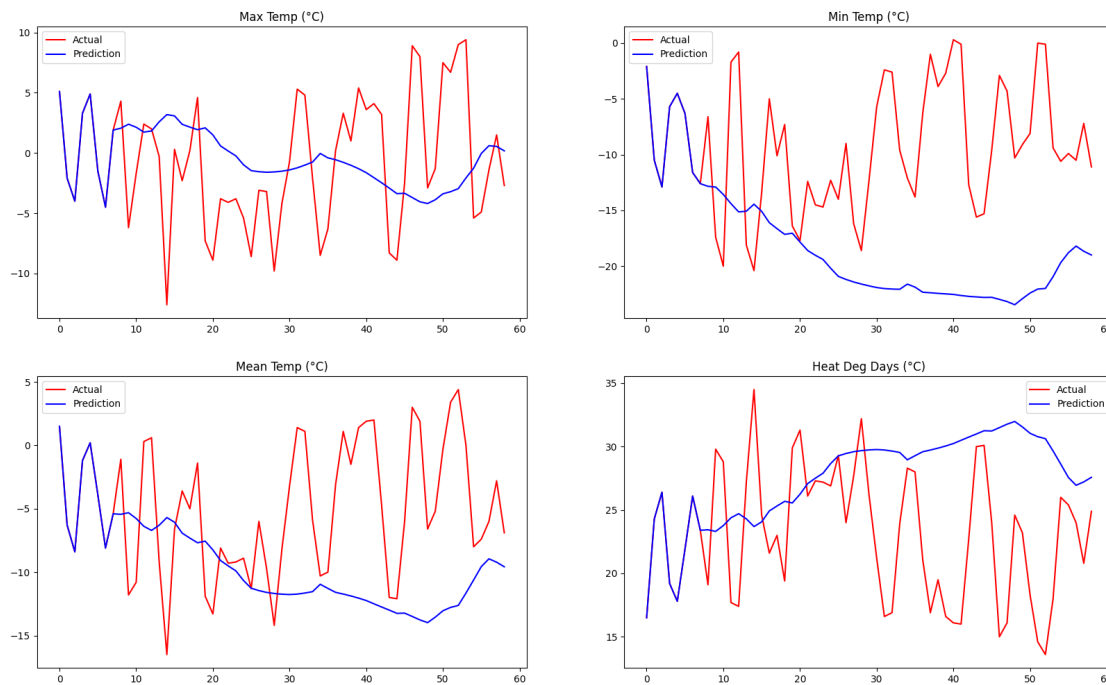


Figure 3.1: Multi-day forecast Program Output.

After analyzing the results, it became clear that forecasting multiple days becomes extremely inaccurate after two to three days. There are many factors that account for the inaccuracy of the multi-day forecast program, some of which are: seasonal changes, the variation size of the sliding window, and discrepancies relative to the previous year's data.

Finally, this program was not implemented in the final design of the project due to not having the functionality to be parallelized. As each iteration is dependent on the previous day's forecasted data, the processors will spend most of their time waiting for previous data to be predicted. Therefore, parallelizing this program will serve no benefit on performance.

MPI/OpenMP Hybrid Implementation

Another library commonly used for parallelizing sequential programs is OpenMP. While MPI is for distributing memory across several processes and devices, OpenMP allows for programming on shared memory devices. It does this by spawning multiple threads to work on a specific task within a program. Instead of multiple instances of the program running, the threads work within a single instance and can be spawned on command. This can be useful when dealing with loops because they can be configured to run in parallel and complete the overall task quicker. The figure below illustrates how an OpenMP program might be configured.

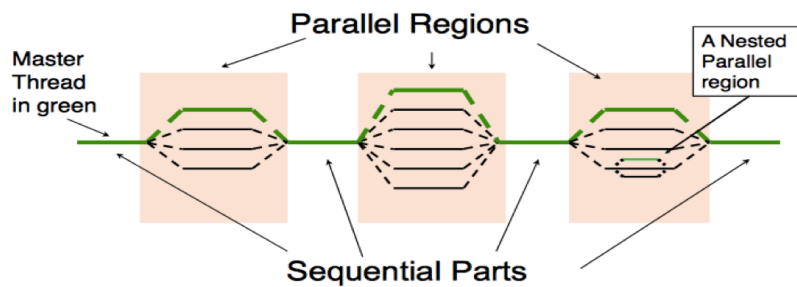


Figure 3.2: OpenMP Program With Sequential and Parallel Regions [11].

The idea for this project design was to make use of both MPI and OpenMP, creating a hybrid implementation. Since MPI runs across multiple processes, each of those processes has threads that are sitting idle. Logically, it makes sense to make use of those threads in an attempt to speed up the program. This was done by adding OpenMP directives to each of the loops in the final parallel program. The directives include specifications for the portion of code that needs to be parallelized. For example, if parallelizing a for loop where each iteration accesses a different location in memory, the directive “`#pragma omp parallel for`” is sufficient. However, if the threads will be accessing the same variable, such as when determining the sum of an array, a reduction clause needs to be added to ensure thread safety. Examples of OpenMP loop directives can be seen in the figures below.

```
double windows[8][7][numParams];
memset(windows, 0, 8*7*numParams*sizeof(double));
#pragma omp parallel for
for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 7; j++) {
        for (int k = 0; k < numParams; k++) {
            windows[i][j][k] = atof(PD[j+i][k]);
        }
    }
}
```

Figure 3.3: OpenMP Loop Directive for Creating Sliding Windows.

```
double euclidianDistance = 0;
#pragma omp parallel for reduction(+:euclidianDistance)
for (int i = 0; i < numPresentMeans; i++) {
    euclidianDistance += pow(presentMeans[i] - prevMeans[i], 2);
}
return sqrt(euclidianDistance);
```

Figure 3.4: OpenMP Loop Directive with Reduction Clause.

After adding the directives to every loop in the parallel program, the hybrid implementation was ready to be run. The results from running the program however were far from what was expected. The execution time increased drastically to the point where it was even slower than the sequential program. This was unexpected as it was hypothesized that OpenMP would reduce the execution time, given that multiple processes and threads would be working on the same algorithm. It was found that this was caused due to bottlenecks introduced by OpenMP. Since the loops present in the sliding window algorithm are not very large, the time it takes for OpenMP to create the threads is greater than the execution time in the loop. Also, the operating system needs to engage in context switching when working with threads, this adds even more time to the total execution time. In order to reap the benefits from OpenMP, the loops would need to be much larger, in the millions of iterations, so that the bottlenecks become negligible. In conclusion, this design was determined to be unsuitable for this project.

Material/Component List

To complete this project, a multitude of virtual resources were required. The biggest necessity was a computer with a multi-core processor, to allow for functional parallelization. This was initially a bottleneck since the team did not have any more than 8 processors available, however, the use of the Ryerson network overcame this issue. The second most important resource was a Message Passing Interface, a tool that enables parallelization. MPI allows for data to be moved from one process to another through cooperative operations, thus creating the capability of parallel processing. In this project, MPICH was the implementation of MPI used. MPICH is a portable version of MPI, which functions on Linux and MAC OS/X, with Linux being the operating system in use on the Ryerson network. Another important resource was the data on which the program would operate. The Canadian government offers free-to-use data sets of weather conditions collected in multiple different locations for widely varying dates. Finally, the programming had to be done in suitable development environments. The primary IDEs used were Visual Studio Code and Pycharm. When developing the Python prototype, multiple libraries were used such as pandas, NumPy, and matplotlib.

Performance Accuracy Results/Testing Procedures

This section goes into detail about the testing and benchmarking methodologies and the result and analysis of the accuracy of the sliding window algorithm

Accuracy Methodology, Parameters, and Dataset Metrics

Testing the accuracy of the sliding window algorithm required analysis of the entirety of both predicted and actual datasets. Standard Deviation and Mean were used as parameters to test the accuracy of the distributed predictions in relation to the actual data. The justification for these metrics is that they provide insight into a general weather pattern over a large set of data and as such, both datasets were compared in this statistical manner to test the accuracy of our algorithm. In particular, both Mean and Standard Deviation provides information on the variability/volatility of a dataset and as such, if these metrics were used to find the variability/volatility of both the datasets, one produced by the prediction algorithm and the other, the real/actual dataset, both would have to be following a similar pattern of volatility/variability in order to state the algorithm as accurate and in overall, feasible to the objective of this project. The formulas below were used to calculate these metrics and were computed with the use of Excel.

$$\overline{X} = \frac{\sum X}{N}$$

Figure 4.1: Mean Formula.

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

Figure 4.2: Standard Deviation Formula.

Dataset Parameter Plots

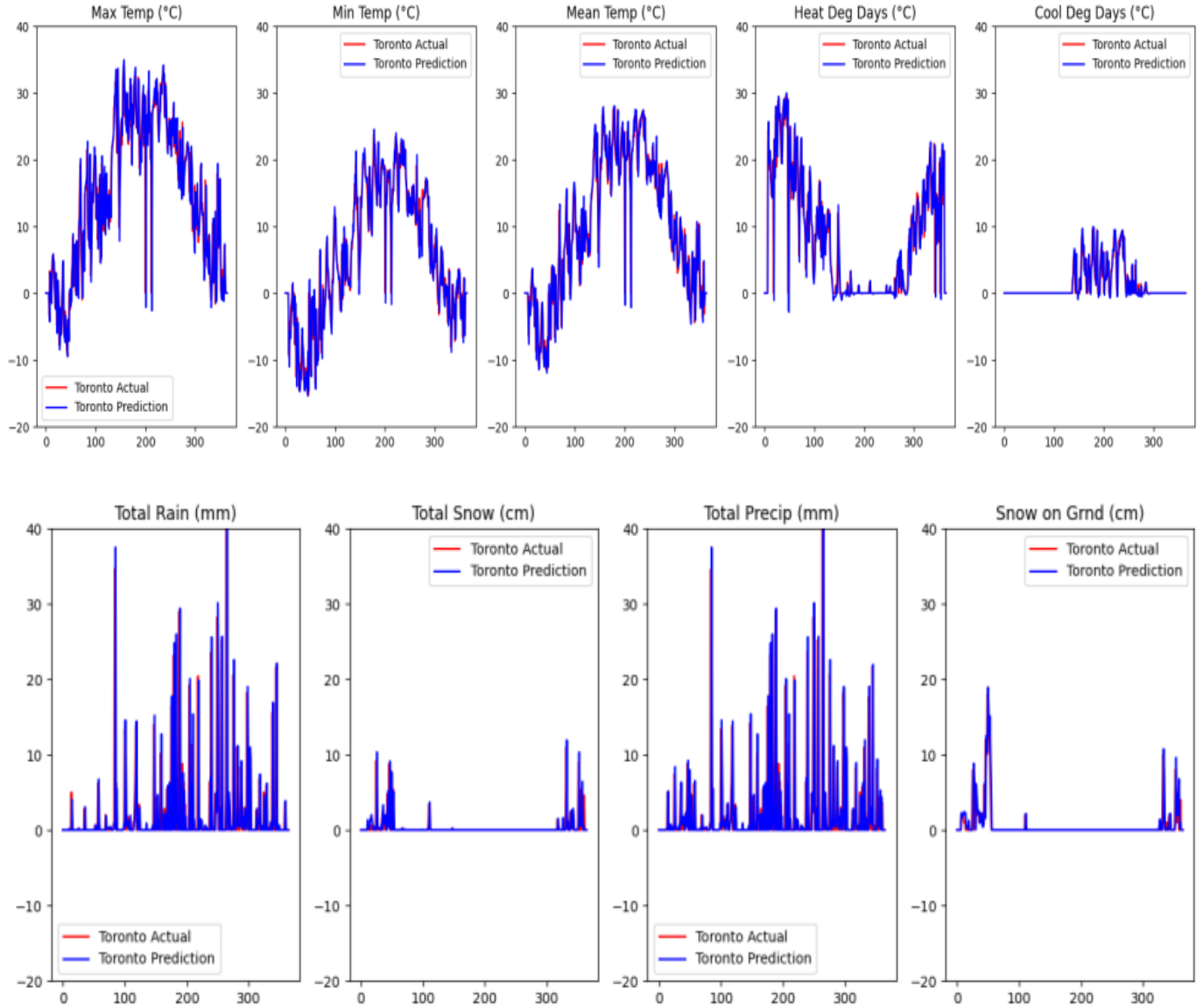


Figure 4.3: Plots of Predicted and Actual Datasets and their respective Parameters.

Sample Computations for Prediction Parameter Accuracy

$$\% \text{ Error} = \left| \frac{\text{Theoretical Value} - \text{Experimental Value}}{\text{Theoretical Value}} \right| \times 100$$

Figure 4.4: Percentage Error Formula used in calculation below.

Sample Calculation to compute Max Temp parameter :

Standard Deviation Calculated Using Real Dataset of 365 days of MAX Temp for Toronto from the year 2021:

$$\sigma = 10.538$$

Standard Deviation Calculated of Max Temp Using Predicted Data:

$$\sigma = 11.342$$

$$\% \text{ Accuracy} = 100 - \% \text{ Error}$$

$$= 100\% - (| (11.342 - 10.538) / 11.342 | * 100)\%$$

$$= 100\% - 7.1\%$$

$$= 92.9 \%$$

∴ Feasible

Sample Calculation to compute Precipitation Parameter :

Standard Deviation Calculated Using Real Dataset of 30 days of Precipitation for Toronto from the month of April:

$$\sigma = 5.199$$

Standard Deviation Calculated of Precipitation Using Predicted Data:

$$\sigma = 5.958$$

$$\% \text{ Accuracy} = 100 - \% \text{ Error}$$

$$= 100\% - (| (5.958 - 5.199) / 5.958 | * 100)\%$$

$$= 100\% - 12.7\%$$

$$= 87.3 \%$$

∴ Feasible

Tabulation of Accuracy Results

Month	Accuracy % of Prediction
January	97.02%
February	98.26%
March	91.35%
April	87.35%
May	89.14%
June	92.32%
July	98.05%
August	90.32%
September	84.37%
October	86.31%
November	88.34%
December	79.85%

Table 4.1: Table of Computed Percentage Accuracy for all Parameters.

The values calculated in Table 4.1 were the cumulative accuracy of different parameters of the algorithm. These parameters include Maximum Temperature, Minimum Temperature, Mean Temperature, Heating Degree Days, Cooling Degree Days, Total Rain, Total Snow, Total Precipitation, and Snow on the Ground. In the format of the sample calculations above, the accuracy of each parameter was computed with respect to the corresponding month (30 days), then the mean of these accuracies was used as the final accuracy calculation for a specific month, and was then recorded in the table above. This process was repeated until the table above was filled with necessary accuracy data to benchmark the correctness of the algorithm throughout different times of the year and thus, allowed for analysis of the discrepancies in the algorithm, visualized by the plot in Figure 4.5.

Analysis of Algorithm Accuracy

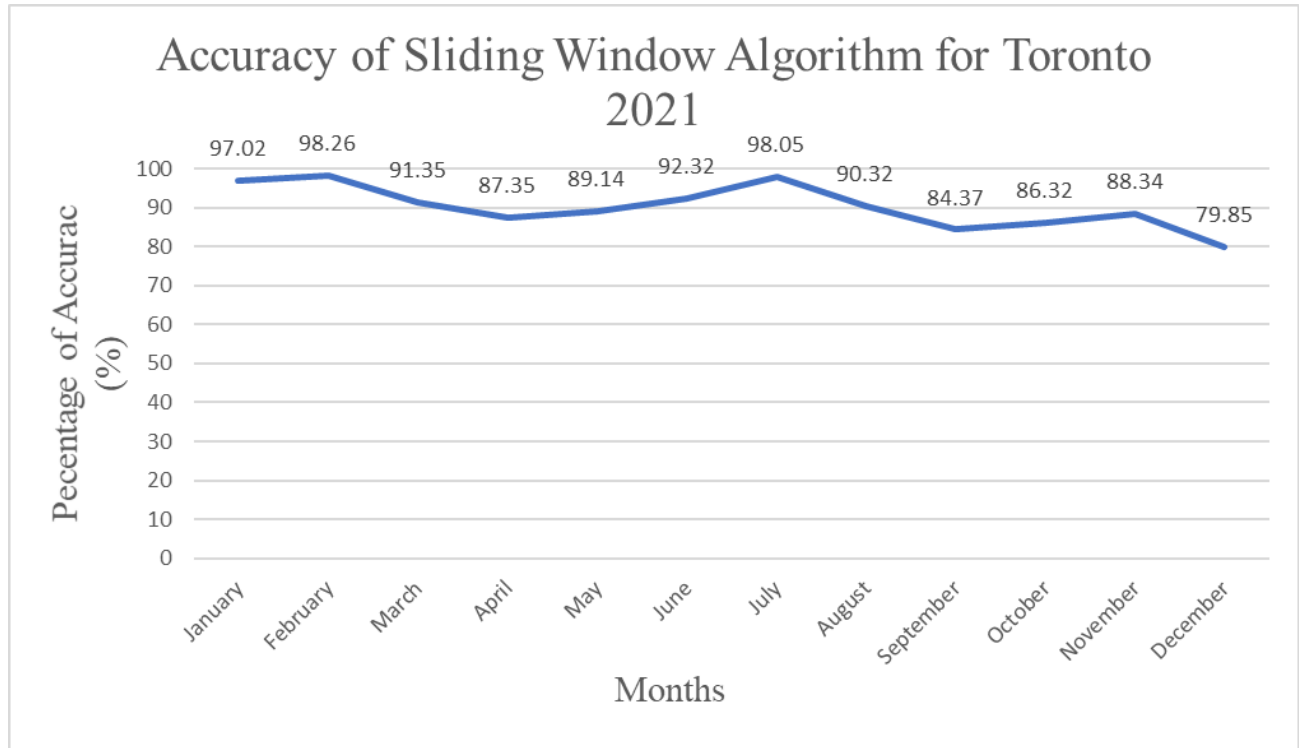


Figure 4.5: Accuracy between Predicted and Actual datasets

Regarding the accuracy of the sliding window algorithm, there are deviations or reductions in accuracy at certain times of the year for different places. Observing the data obtained from the accuracy testing of the algorithm (Figure 4.6), these discrepancies usually occur during the months of seasonal change, where the most volatile and anomalous data appear. The clearest example can be observed during the month of December, when the season in this particular example, Toronto, is transitioning from fall to winter. However, in Months like February and July, temperatures stay relatively fixed and stable, thus resulting in a high accuracy within the metrics of mean and standard deviation. With these factors in mind, the algorithm had an approximate accuracy of 90% and as result, was evidently feasible for weather prediction. Note: In this particular example, the algorithm observes the pattern for the city of Toronto, however, for different cities/locations that have different weather conditions, the accuracy graphs for the algorithm vary depending on the months a city/location experiences drastic seasonal changes in which results in anomalous data and less prediction accuracy for those particular months.

Analysis of Performance

This section will discuss the analysis of the final parallel program and will apply the theory mentioned in the Theory and Design section to the finished product.

Performance Gain

Once the program was converted to C and parallelized, the actual gain of performance could be calculated. Given that Amdahl's law, which was used to calculate the theoretical gain, is a proven theory, the behavior observed was expected. For the problem considered in this project, in which a sequential iteration took around ten seconds to run, the cut-off for performance improvement is in a lower range as opposed to the theoretical mark of 70 processors explained in the Theory section of the report. In practice, the plateau point was at six processes, where after this point the increase in performance was negligible when giving the program more power. This can be explained by the fact that certain parts of the program must remain serial. In a perfect world, the entirety of the program could be parallelized, in which the speed of the program would be defined by $1/N$, with N being the number of processors devoted to the parallelization of the program. However, since there will always be some portion of the process which cannot be parallelized, this cannot hold true. For example, the initialization of a program must be done sequentially, as oftentimes the next step of the process relies on the previous step. This leads to the conclusion that if the serial portion of the program is nearly the same size as the portion which can be parallelized, there will not be much improvement in performance. This understanding explains the lack of gain after six processors as seen in Figure 5.1 below.

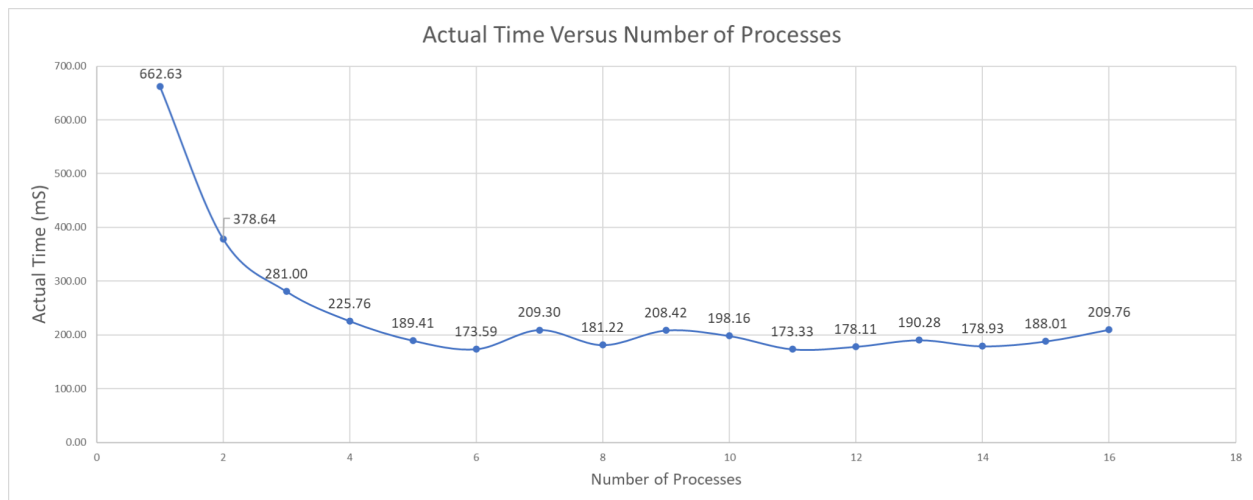


Figure 5.1: Decrease in Overall Execution Time Versus Number of Processes.

When observing Figure 5.1, there is a minor fluctuation in the execution time after six processors. Amdahl's law is a powerful and eloquent tool to explain the theoretical gain and bottlenecks of parallel processing, however, the flaw in this theory is that it does not account for the overhead of communication and initialization. The fluctuation after the performance plateau can be explained by the fact that once the program is parallelized, each individual process still takes time to initialize the portion that must be serial, and to communicate with other processes. In each individual case, the overhead varies due to different costs of communication.

In Figure 5.2, the overall gain of the program when parallelized is graphed against the amount of processes. The gain shown in this figure is different from Amdahl's gain, as the latter depicts the gain of the portion of the program which has been parallelized, rather than the gain of the program overall. In this graph, the cost of overhead is much more visible when compared to Figure 5.1. This can be attributed to the scale used in gain versus the scale used for speedup.

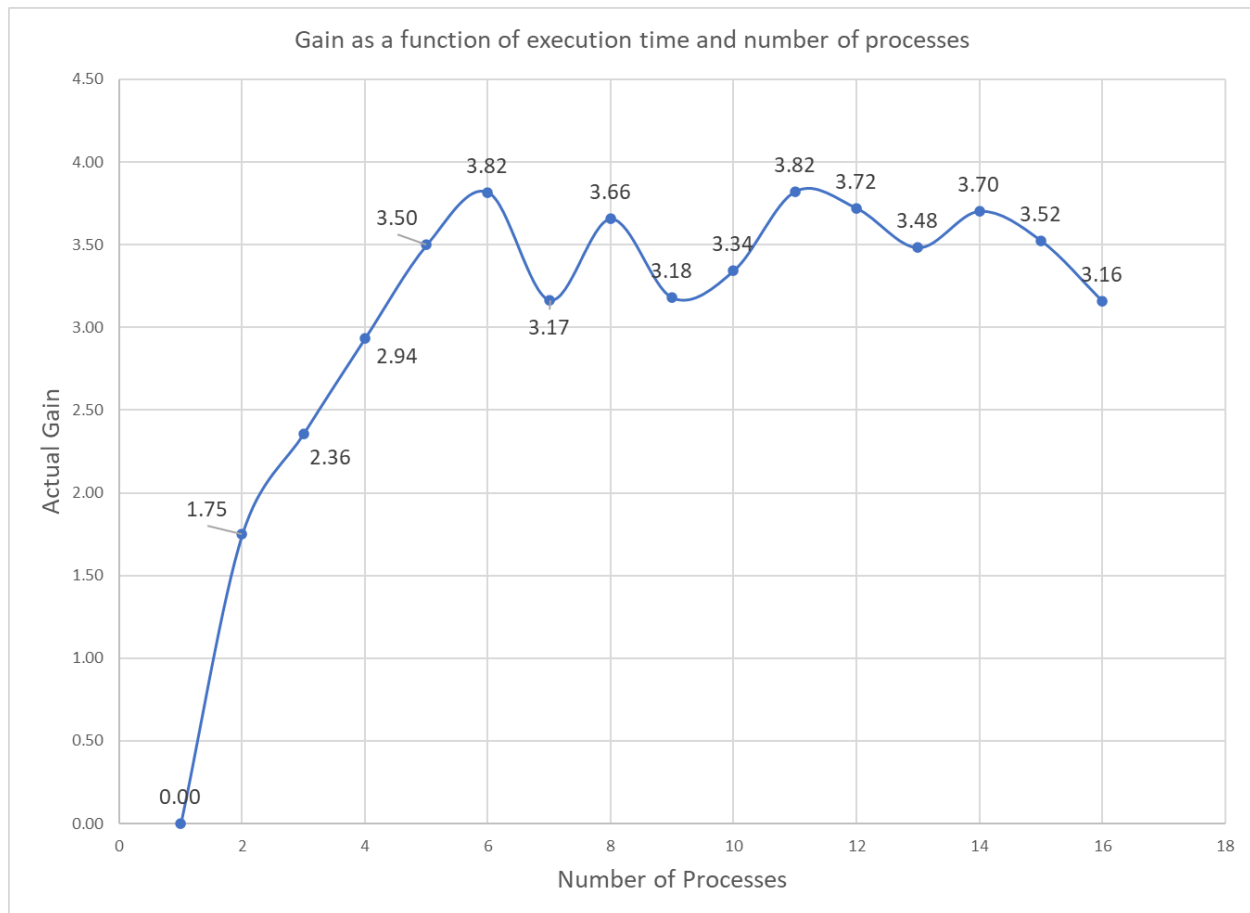


Figure 5.2: Overall Gain of Program.

The gain calculated using Amdahl's law is graphed in Figure 5.3, where each data point represents the gain of that portion of the code when parallelized. The performance increase calculated using Amdahl's is lower than that of the overall increase due to the fact that the problem being solved is of a small size, which leads to parallel operations being similar in size to serial operations. The increase between a pure sequential program and a parallelized program using only 2 processors is severe, however, the same plateau occurs after the six process point. The instability of Amdahl's gain after the plateau point can be explained by the cost of overhead and communication, similar to the overall gain.

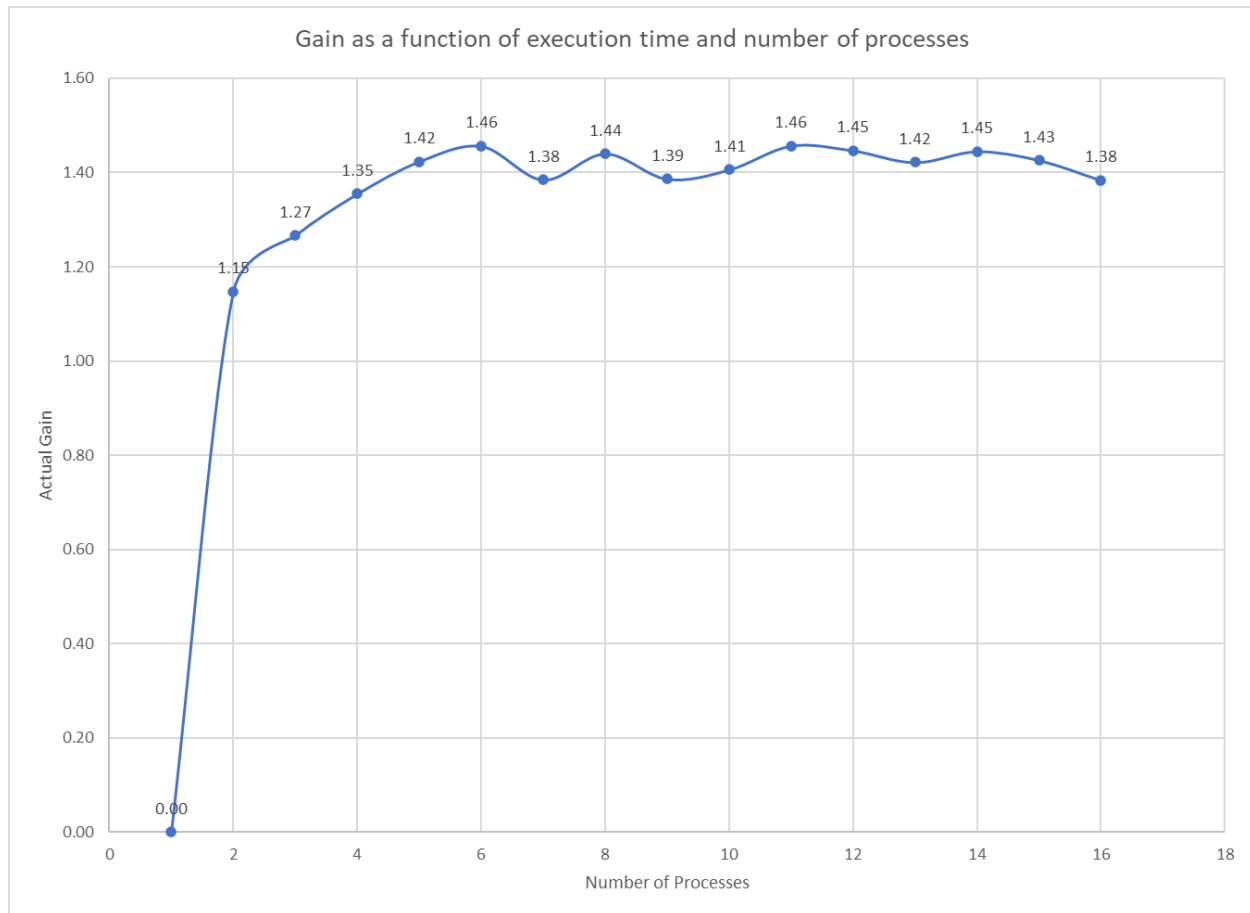


Figure 5.3: Amdahl's Gain of Performance.

Conclusions

In conclusion, the team was able to successfully analyze the benefits and drawbacks of parallel processing when applied to a real-world problem. Beginning with research into the basics of parallel computing, where the theory of how the increase of performance due to parallelization was discovered alongside the most efficient methods of implementation. In addition to deciding on a common weather prediction algorithm to use, practical data sets were chosen to operate on. The sliding window algorithm was parallelized using MPI, and upon analysis, the team found that Amdahl's Law stands true in the case of the problem being considered. Once the program was parallelized, statistical analysis was performed to determine the accuracy of the algorithm. This analysis proved that the sliding window algorithm creates very accurate future predictions. At the tail end of the project, an attempt was made to use a hybrid of OpenMP in tandem with MPI, however, it was found that the bottlenecks caused by this implementation outweighed the benefits, and so it was not pursued any further. Overall, this project was a thorough look into the world of parallel computing, illustrating how a large program can be optimized through the use of hardware manipulation.

References

- [1]. LaPlante, Phillip A. "Chapter 7.1.5 - Gustafson's Law." *Chapter 7.1.5 - Gustafson's Law | Engineering360*,
<https://www.globalspec.com/reference/14699/160210/chapter-7-1-5-gustafson-s-law>.
- [2]. "Ahmdal'S Law". *Codingforspeed, Code Optimisation Tips, Faster Than Faster, Don't Waste 1 CPU Cycle Or 1 Byte*, 2021, <https://codingforspeed.com/ahmdals-law/>.
- [3]. "Computer Organization | Amdahl's Law and Its Proof." *GeeksforGeeks*, 11 June 2018, www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/.
- [4]. Gustafson, John L. "Amdahl's Law" www.johngustafson.net,
www.johngustafson.net/pubs/pub13/amdahl.htm
- [5]. Piyush Kapoor, Sarabjeet Singh Bedi, "Weather Forecasting Using Sliding Window Algorithm", *International Scholarly Research Notices*, vol. 2013, Article ID 156540, 5 pages, 2013. <https://doi.org/10.1155/2013/156540>
- [6] *Introduction to parallel programming with MPI and OpenMP*. (n.d.). Retrieved February 12, 2022, from https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf
- [7]. *VW: Intro to parallel processing: 6. converting from serial*. [Online]. Available: http://www.math.udel.edu/~plechac/M578/Intro.Par.Processing/section_6.html [Accessed: 29-NOV-2021].
- [8] *MPI scatter, gather, and Allgather*. MPI Scatter, Gather, and Allgather · MPI Tutorial. (n.d.). Retrieved February 12, 2022, from <https://mpitutorial.com/tutorials/mapi-scatter-gather-and-allgather/>
- [9]. Y.-Y. Song and Y. Lu, "Decision tree methods: Applications for classification and prediction," *Shanghai archives of psychiatry*, 25-Apr-2015. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4466856/>
- [10]. R. Kumar, "Decision tree for the weather forecasting - ijcaonline.org," *International Journal of Computer Applications*, Aug-2013. [Online]. Available: <https://research.ijcaonline.org/volume76/number2/pxc3890620.pdf>
- [11] Nersc, "OpenMP¶," *OpenMP - NERSC Documentation*. [Online]. Available: <https://docs.nersc.gov/development/programming-models/openmp/> [Accessed: 17-Apr-2022].
- [12] "Coding games and programming challenges to code better," *CodinGame*. [Online]. Available: <https://www.codingame.com/playgrounds/349/introduction-to-mpi/introduction-to-distributed-computing>. [Accessed: 17-Apr-2022].

Appendices

This section contains the source code for the project.

Sequential Python Prototype

weather.py

```
from operator import indexOf
import pandas as pd
import math
from pprint import pprint
import time
import matplotlib.pyplot as plt
import numpy as np

data_files_2020 = [
    pd.read_csv("data/2020_tor_data.csv"),
    pd.read_csv("data/2020_van_data.csv"),
    pd.read_csv("data/2020_mtl_data.csv"),
    pd.read_csv("data/2020_cal_data.csv"),
    pd.read_csv("data/2020_edm_data.csv"),
    pd.read_csv("data/2020_ham_data.csv"),
    pd.read_csv("data/2020_hfx_data.csv"),
    pd.read_csv("data/2020_kel_data.csv"),
    pd.read_csv("data/2020_kit_data.csv"),
    pd.read_csv("data/2020_lon_data.csv"),
    pd.read_csv("data/2020_osh_data.csv"),
    pd.read_csv("data/2020_ott_data.csv"),
    pd.read_csv("data/2020_peg_data.csv"),
    pd.read_csv("data/2020_qbc_data.csv"),
    pd.read_csv("data/2020_reg_data.csv"),
    pd.read_csv("data/2020_sas_data.csv"),
    pd.read_csv("data/2020_stcath_data.csv"),
    pd.read_csv("data/2020_stjn_data.csv"),
    pd.read_csv("data/2020_vic_data.csv"),
    pd.read_csv("data/2020_win_data.csv"),
]

data_files_2021 = [
    pd.read_csv("data/2021_tor_data.csv"),
    pd.read_csv("data/2021_van_data.csv"),
    pd.read_csv("data/2021_mtl_data.csv"),
    pd.read_csv("data/2021_cal_data.csv"),
```

```
pd.read_csv("data/2021_edm_data.csv"),
pd.read_csv("data/2021_ham_data.csv"),
pd.read_csv("data/2021_hfx_data.csv"),
pd.read_csv("data/2021_kel_data.csv"),
pd.read_csv("data/2021_kit_data.csv"),
pd.read_csv("data/2021_lon_data.csv"),
pd.read_csv("data/2021_osh_data.csv"),
pd.read_csv("data/2021_ott_data.csv"),
pd.read_csv("data/2021_peg_data.csv"),
pd.read_csv("data/2021_qbc_data.csv"),
pd.read_csv("data/2021_reg_data.csv"),
pd.read_csv("data/2021_sas_data.csv"),
pd.read_csv("data/2021_stcath_data.csv"),
pd.read_csv("data/2021_stjn_data.csv"),
pd.read_csv("data/2021_vic_data.csv"),
pd.read_csv("data/2021_win_data.csv")
]
```

```
PARAMS = [
    "Max Temp (°C)",
    "Min Temp (°C)",
    "Mean Temp (°C)",
    "Heat Deg Days (°C)",
    "Cool Deg Days (°C)",
    "Total Rain (mm)",
    "Total Snow (cm)",
    "Total Precip (mm)",
    "Snow on Grnd (cm)",
]
```

```
s1time = 0
s2time = 0
s3time = 0
s4time = 0
s5time = 0
s6time = 0
```

```
NUM_OF_PARAMS = len(PARAMS)
```

```
def get_mean(arr):
    return [val / 7 for val in arr]
```

```
# Predicting any day in 2021
def forecast_day(day, data_21, data_20):
    if day < 7 or day > 360:
        return None, None
```

```
s1 = time.time()
```



```

present_days = data_21.loc[day-7:day-1]
present_days = present_days[PARAMS]
present_days = present_days.fillna(0)
CD = present_days.to_numpy()
s1e = time.time()
global s1time
s1time += s1e - s1

s2 = time.time()
prev_days = data_20.loc[day-7:day+6]
prev_days = prev_days[PARAMS]

prev_days = prev_days.fillna(0)
PD = prev_days.to_numpy()
s2e = time.time()
global s2time
s2time += s2e - s2

s3 = time.time()
windows = []
for i in range(len(PD) - 6):
    windows.append(PD[i: i+7])
s3e = time.time()
global s3time
s3time += s3e-s3

# calculating means for present year 7 days
s4 = time.time()
present_sums = [0] * NUM_OF_PARAMS
for line in CD:
    for i in range(len(line)):
        if not math.isnan(line[i]): present_sums[i] += line[i]
present_means = get_mean(present_sums)

# mean of parameters for each window
ed_list = []
for window in windows:
    prev_sums = [0] * NUM_OF_PARAMS
    for vals in window:
        for i in range(len(prev_sums)):
            prev_sums[i] += vals[i]
    prev_means = get_mean(prev_sums)

    euclid_distance = sum([(present_means[i] - prev_means[i]) ** 2 for
i in range(len(present_means))])
    ed_list.append(euclid_distance)

```

```

s4e = time.time()
global s4time
s4time += s4e-s4

s5 = time.time()
min_ed = min(ed_list)
index_min_ed = indexOf(ed_list, min_ed)
selected_prev_window = windows[index_min_ed]
s5e = time.time()
global s5time
s5time += s5e - s5

s6 = time.time()
present_variation_vector = []
for i in range(len(CD[0])):
    vector = []
    for j in range(len(CD)-1):
        vector.append(CD[j+1][i] - CD[j][i])
    present_variation_vector.append(vector)

prev_variation_vector = []
for i in range(len(selected_prev_window[0])):
    vector = []
    for j in range(len(selected_prev_window)-1):
        vector.append(selected_prev_window[j+1][i] -
selected_prev_window[j][i])
    prev_variation_vector.append(vector)

mean_present_var = []
for i in range(len(present_variation_vector)):
    mean_present_var.append(sum(present_variation_vector[i]) /
len(present_variation_vector[i]))

mean_prev_var = []
for i in range(len(prev_variation_vector)):
    mean_prev_var.append(sum(prev_variation_vector[i]) /
len(prev_variation_vector[i]))

mean_variation_vector = []
for i in range(len(mean_present_var)):
    mean_variation_vector.append((mean_present_var[i] +
mean_prev_var[i]) / 2)

prev_day = CD[-1]
for i in range(len(prev_day)):
    prev_day[i] += mean_variation_vector[i]
pred_arr = prev_day.tolist()
    
```

```

    for i in range(len(pred_arr[5:])):
        if pred_arr[5+i] < 0:
            pred_arr[5+i] = 0

    actual_data = data_21.loc[day]
    actual_data = actual_data.fillna(0)
    actual_values = actual_data[PARAMS].to_numpy().flatten()
    s6e = time.time()
    global s6time
    s6time += s6e - s6

    return actual_values, pred_arr

if __name__ == "__main__":
    t0 = time.time()
    day_range = np.arange(0, 365)
    actual_list = [[] for i in range(20)]
    pred_list = [[] for i in range(20)]

    for day in day_range:
        for i in range(len(pred_list)):
            temp_actual, temp_pred =
forecast_day(day, data_files_2021[i], data_files_2020[i])

            if temp_pred:
                actual_list[i].append(list(temp_actual))
                pred_list[i].append(list(temp_pred))

            else:
                actual_list[i].append([0] * NUM_OF_PARAMS)
                pred_list[i].append([0] * NUM_OF_PARAMS)

    print(f"\nTotal time: {time.time() - t0}")
    print(f"\n Step 1 done in:{s1time}")
    print(f"\n Step 2 done in:{s2time}")
    print(f"\n Step 3 done in:{s3time}")
    print(f"\n Step 4 done in:{s4time}")
    print(f"\n Step 5 done in:{s5time}")
    print(f"\n Step 6 done in:{s6time}")

    toronto_actual_data = np.transpose(actual_list[0])
    toronto_pred_data = np.transpose(pred_list[0])
    
```

```
plt.figure()
plt.subplot(1, 3, 1)
plt.gcf().set_size_inches(20,12)

index = 0
for param in PARAMS:
    plt.subplot(2, 5, index+1)
    plt.plot(day_range, toronto_actual_data[index], label='Toronto
Actual', color = "red")
    plt.plot(day_range, toronto_pred_data[index], label='Toronto
Prediction', color = "blue")
    plt.ylim(-20, 40)
    plt.title(f"{param}")
    plt.legend(['Toronto Actual', 'Toronto Prediction'])
    index += 1
plt.show()
```

Parallel Python Prototype

weather_mpi.py

```
from operator import indexOf
import pandas as pd
import math
from pprint import pprint
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data_20 = pd.read_csv("data/2020_tor_data.csv")
    data_21 = pd.read_csv("data/2021_tor_data.csv")
    data_21.head()
    present_days = data_21[data_21["Month"] == 6]
    prev_days = data_20[data_20["Month"] == 6]

    for i in range(size-1):
        comm.send(data_20, dest=i+1)
        comm.send(data_21, dest=i+1)
        comm.send(present_days, dest=i+1)
        comm.send(prev_days, dest=i+1)
else:
    start = MPI.Wtime()
```

```

data_20 = comm.recv(source=0)
data_21 = comm.recv(source=0)
present_days = comm.recv(source=0)
prev_days = comm.recv(source=0)

present_days = present_days[(present_days["Day"] < 15 + rank) &
(present_days["Day"] >= 8 + rank)]
present_days = present_days[["Max Temp (°C)", "Min Temp (°C)", "Total
Rain (mm)"]]
CD = present_days.to_numpy()

prev_days = prev_days[(prev_days["Day"] < 15 + rank) &
(prev_days["Day"] >= 1 + rank)]
prev_days = prev_days[["Max Temp (°C)", "Min Temp (°C)", "Total Rain
(mm)"]]
PD = prev_days.to_numpy()

w_dct = {}

for index in range(8):
    w_dct[f"W{index+1}"] = PD[index: index+7]

ed_list = []
# calculating means for present year 7 days
present_max = present_min = present_rain = 0

for line in CD:
    present_max += line[0]
    present_min += line[1]
    present_rain += line[2]

present_max = present_max / 7
present_min = present_min / 7
present_rain = present_rain / 7

# mean of parameters for each window
for line in w_dct:
    max_sum = min_sum = rain_sum = 0
    for values in w_dct[line]:
        max_sum += values[0]
        min_sum += values[1]
        rain_sum += values[2]
    max_mean = max_sum / 7
    min_mean = min_sum / 7
    rain_mean = rain_sum / 7

```

```

        euclid_distance = math.sqrt((present_max - max_mean)**2 +
(present_min - min_mean)**2 + (present_rain - rain_mean) ** 2)
        ed_list.append(euclid_distance)

min_ed = min(ed_list)
index_min_ed = indexOf(ed_list, min_ed)

selected_prev_window = w_dct.get(f"W{index_min_ed+1}")

present_variation_vector = []
for i in range(len(CD[0])):
    vector = []
    for j in range(len(CD)-1):
        vector.append(CD[j+1][i] - CD[j][i])
    present_variation_vector.append(vector)

prev_variation_vector = []
for i in range(len(selected_prev_window[0])):
    vector = []
    for j in range(len(selected_prev_window)-1):
        vector.append(selected_prev_window[j+1][i] -
selected_prev_window[j][i])
    prev_variation_vector.append(vector)

mean_present_var = []
for i in range(len(present_variation_vector)):
    mean_present_var.append(sum(present_variation_vector[i]) /
len(present_variation_vector[i]))

mean_prev_var = []
for i in range(len(prev_variation_vector)):
    mean_prev_var.append(sum(prev_variation_vector[i]) /
len(prev_variation_vector[i]))

mean_variation_vector = []
for i in range(len(mean_present_var)):
    mean_variation_vector.append((mean_present_var[i] +
mean_prev_var[i]) / 2)

prev_day = CD[-1]
for i in range(len(prev_day)):
    prev_day[i] += mean_variation_vector[i]
pred_arr = prev_day.tolist()

pred_max = pred_arr[0]
pred_min = pred_arr[1]

```

```

pred_rain = pred_arr[2]

actual_data = data_21[(data_21["Month"] == 6) & (data_21["Day"] == 15
+ rank)]
actual_values = actual_data[["Max Temp (°C)", "Min Temp (°C)", "Total
Rain (mm)"]].to_numpy().flatten()

print(f'''\nPrediction    for    June    {15    +    rank},    2021
\n-----
\tMax Temp | Min Temp | Rainfall
Actual:      {actual_values[0]}      |      {actual_values[1]}      |
{actual_values[2]}
Predicted:    {round(pred_max,  2)}    |      {round(pred_min,  2)}    |
{round(pred_rain, 2)}
''')
end = MPI.Wtime()
print(f"Time taken to predict weather by process {rank}: {end}")

```

Sequential C Program

CSV.C

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Reads a weather CSV file and returns an array of strings
// containing the values in the CSV file
void readCSV(
    char result[][50][50],
    char * filename
) {
    char line[1024];
    char *linePtr;
    int i = 0;
    int j = 0;
    FILE *file = fopen(filename, "r");

    // Read row by row
    int lineNumber = 0;
    while (fgets(line, sizeof(line), file)) {
        // Skip the first column
        if (lineNumber == 0) {
            lineNumber++;
            continue;
        }
    }
}

```

```

    }

    // Read column by column
    char *token;
    linePtr = line;
    while ((token = strsep(&linePtr, ",")) != NULL) {
        // Copy cell from CSV to array
        strcpy(result[i][j], token);
        // Increment Columns
        j++;
    }

    // Increment Rows
    i++;
    // Reset to first column
    j = 0;
}

fclose(file);
}

```

utils.c

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>

// Accepts a matrix and filters it to a specified range of
// rows and columns
void filterMatrix(
    char matrix[][50][50],
    char result[][50][50],
    int startRow,
    int endRow,
    int *colsToFilter,
    int numColsToFilter
) {
    int i = 0;
    int j = 0;

    // For any weather data matrix
    int numRows = 400;
    int numCols = 50;

```



```

// Read row by row
int lineNumber = 0;
for (int row = 0; row < numRows; row++) {
    // Constrain lines to row range
    if (lineNumber < startRow) {
        lineNumber++;
        continue;
    }
    // Stop reading once specified rows are read
    else if (lineNumber > endRow) {
        break;
    }

    // Read column by column
    int columnNumber = 0;
    int columnsRead = 0;
    for (int col = 0; col < numCols; col++) {
        // Constrain columns to the ones specified
        int shouldReadColumn = 0;
        for (int k = 0; k < numColsToFilter; k++) {
            if (columnNumber == colsToFilter[k]) {
                shouldReadColumn = 1;
                break;
            }
        }
        if (!shouldReadColumn) {
            // Stop reading once specified columns are read
            if (columnsRead == numColsToFilter) {
                break;
            }
            columnNumber++;
            continue;
        }

        // Copy cell from CSV to array
        strcpy(result[i][j], matrix[row][col]);

        // Increment Columns
        j++;
        columnNumber++;
        columnsRead++;
    }

    // Increment Rows
    i++;
    lineNumber++;
}

```

```

    // Reset to first column
    j = 0;
}
}

// Calculate the euclidian distance of two arrays
double calcEuclidianDistance(
    double *presentMeans,
    double *prevMeans,
    int numPresentMeans
) {
    double euclidianDistance = 0;
    for (int i = 0; i < numPresentMeans; i++) {
        euclidianDistance += pow(presentMeans[i] - prevMeans[i], 2);
    }
    return sqrt(euclidianDistance);
}

// Determine the minimum value from an array
double min(double *arr, int len) {
    double minVal = DBL_MAX;
    for (int i = 0; i < len; i++)
        if (arr[i] < minVal)
            minVal = arr[i];
    return minVal;
}

```

weather.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "csv.h"
#include "utils.h"

char presentCSV[][50] = {
    "./data/2021_tor_data.csv",
    "./data/2021_van_data.csv",
    "./data/2021_mtl_data.csv",
    "./data/2021_cal_data.csv",
    "./data/2021_edm_data.csv",
    "./data/2021_ham_data.csv",
    "./data/2021_hfx_data.csv",
    "./data/2021_kel_data.csv",

```

```

    "/data/2021_kit_data.csv",
    "/data/2021_lon_data.csv",
    "/data/2021_osh_data.csv",
    "/data/2021_ott_data.csv",
    "/data/2021_peg_data.csv",
    "/data/2021_qbc_data.csv",
    "/data/2021_reg_data.csv",
    "/data/2021_sas_data.csv",
    "/data/2021_stcath_data.csv",
    "/data/2021_stjn_data.csv",
    "/data/2021_vic_data.csv",
    "/data/2021_win_data.csv"
};

char prevCSV[][50] = {
    "/data/2020_tor_data.csv",
    "/data/2020_van_data.csv",
    "/data/2020_mtl_data.csv",
    "/data/2020_cal_data.csv",
    "/data/2020_edm_data.csv",
    "/data/2020_ham_data.csv",
    "/data/2020_hfx_data.csv",
    "/data/2020_kel_data.csv",
    "/data/2020_kit_data.csv",
    "/data/2020_lon_data.csv",
    "/data/2020_osh_data.csv",
    "/data/2020_ott_data.csv",
    "/data/2020_peg_data.csv",
    "/data/2020_qbc_data.csv",
    "/data/2020_reg_data.csv",
    "/data/2020_sas_data.csv",
    "/data/2020_stcath_data.csv",
    "/data/2020_stjn_data.csv",
    "/data/2020_vic_data.csv",
    "/data/2020_win_data.csv"
};

// Predicts the weather of a particular day of a year
void predictWeatherForGivenDay(
    double *prediction,
    double *actual,
    char presentData[][50][50],
    char prevData[][50][50],
    int day,
    int *cols,
    int numParams
) {

```

```
// Get current data for the past 7 days
char CD[7][50][50];
filterMatrix(presentData, CD, day - 7, day - 1, cols, numParams);

// Get previous data (1 year old) for a 14 day period
char PD[14][50][50];
filterMatrix(prevData, PD, day - 7, day + 6, cols, numParams);

// 8 sliding windows based on previous data, 7 days per window
double windows[8][7][numParams];
memset(windows, 0, 8*7*numParams*sizeof(double));
for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 7; j++) {
        for (int k = 0; k < numParams; k++) {
            windows[i][j][k] = atof(PD[j+i][k]);
        }
    }
}

// Sum the present values
double presentSums[numParams];
memset(presentSums, 0, numParams*sizeof(double));
for (int i = 0; i < 7; i++)
    for (int j = 0; j < numParams; j++)
        presentSums[j] += atof(CD[i][j]);

// Using the present sums, determine the means
double presentMeans[numParams];
memset(presentMeans, 0, numParams*sizeof(double));
for (int i = 0; i < numParams; i++)
    presentMeans[i] = presentSums[i] / 7;

// Determine the means of the previous values based on the
// sliding windows and calculate the Euclidian Distances
double edList[8];
memset(edList, 0, 8*sizeof(double));
for (int i = 0; i < 8; i++) {
    double prevSums[numParams];
    memset(prevSums, 0, numParams*sizeof(double));
    for (int j = 0; j < 7; j++)
        for (int k = 0; k < numParams; k++)
            prevSums[k] += windows[i][j][k];

    double prevMeans[numParams];
    memset(prevMeans, 0, numParams*sizeof(double));
    for (int j = 0; j < numParams; j++)
        prevMeans[j] = prevSums[j] / 7;
```

```

        int numPresentMeans = sizeof(presentMeans) / sizeof(presentMeans[0]);
        double euclidDistance = calcEuclidianDistance
        (
            presentMeans,
            prevMeans,
            numPresentMeans
        );

        edList[i] = euclidDistance;
    }

    // Determine the minimum Euclidian Distance and find its index
    double minEd = min(edList, sizeof(edList) / sizeof(edList[0]));
    int indexMinEd = 0;
    for (int i = 0; i < 8; i++) {
        if (edList[i] == minEd) {
            indexMinEd = i;
            break;
        }
    }

    // Isolate the window with the minimum Euclidian Distance
    double selectedPrevWindow[7][numParams];
    memset(selectedPrevWindow, 0, 7*numParams*sizeof(double));
    for (int i = 0; i < 7; i++)
        for (int j = 0; j < numParams; j++)
            selectedPrevWindow[i][j] = windows[indexMinEd][i][j];

    // Find variation vector based on present data
    double presentVariationVector[numParams][6];
    memset(presentVariationVector, 0, numParams*6*sizeof(double));
    for (int i = 0; i < numParams; i++)
        for (int j = 0; j < 6; j++)
            presentVariationVector[i][j] = atof(CD[j+1][i]) - atof(CD[j][i]);

    // Find variation vector based on previous data
    double prevVariationVector[numParams][6];
    memset(prevVariationVector, 0, numParams*6*sizeof(double));
    for (int i = 0; i < numParams; i++)
        for (int j = 0; j < 6; j++)
            prevVariationVector[i][j] = selectedPrevWindow[j+1][i] -
            selectedPrevWindow[j][i];

    // Determine the mean of the present variation vector
    double meanPresentVar[numParams];
    memset(meanPresentVar, 0, numParams*sizeof(double));
    
```

```

    for (int i = 0; i < numParams; i++) {
        double presentVarSum = 0;
        for (int j = 0; j < 6; j++)
            presentVarSum += presentVariationVector[i][j];
        meanPresentVar[i] = presentVarSum / 6;
    }

    // Determine the mean of the previous variation vector
    double meanPrevVar[numParams];
    memset(meanPrevVar, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++) {
        double prevVarSum = 0;
        for (int j = 0; j < 6; j++)
            prevVarSum += prevVariationVector[i][j];
        meanPrevVar[i] = prevVarSum / 6;
    }

    // Determine the mean of both present and previous mean variation
    vectors
    double meanVariationVector[numParams];
    memset(meanVariationVector, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++)
        meanVariationVector[i] = (meanPresentVar[i] + meanPrevVar[i]) / 2;

    // Retrieve values from the day previous to the day being predicted
    double previousDay[numParams];
    memset(previousDay, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++)
        previousDay[i] = atof(CD[6][i]);

    // Add mean variation vector values to the previous day to get final
    prediction
    for (int i = 0; i < numParams; i++)
        previousDay[i] += meanVariationVector[i];

    // Assign the final prediction values
    for (int i = 0; i < numParams; i++)
        prediction[i] = previousDay[i];

    // Retrive the actual values for comparision purposes
    char actualValues[1][50][50];
    filterMatrix(presentData, actualValues, day, day, cols, numParams);
    for (int i = 0; i < numParams; i++)
        actual[i] = atof(actualValues[0][i]);
}

int main() {

```

```

clock_t t;
t = clock();

// 9 - Max Temp (°C)
// 11 - Min Temp (°C)
// 13 - Mean Temp (°C)
// 15 - Heat Deg Days (°C)
// 17 - Cool Deg Days (°C)
// 19 - Total Rain (mm)
// 21 - Total Snow (cm)
// 23 - Total Precip (mm)
// 25 - Snow on Grnd (cm)
int cols[] = {9, 11, 13, 15, 17, 19, 21, 23, 25};
int numParams = sizeof(cols) / sizeof(cols[0]);
int numLocations = sizeof(presentCSV) / sizeof(presentCSV[0]);

for (int location = 0; location < numLocations; location++) {
    // Get data from CSV files
    char presentData[400][50][50];
    char prevData[400][50][50];
    readCSV(presentData, presentCSV[location]);
    readCSV(prevData, prevCSV[location]);

    for (int i = 8; i <= 359; i++) {
        double prediction[numParams];
        double actual[numParams];
        predictWeatherForGivenDay(
            prediction,
            actual,
            presentData,
            prevData,
            i,
            cols,
            numParams
        );

        // Comment out these print statements for a more accurate time
        measurement
        printf("\nDay %d\nActual: ", i);
        for (int j = 0; j < numParams; j++) {
            printf("%.2lf ", actual[j]);
        }
        printf("\nPrediction: ");
        for (int j = 0; j < numParams; j++) {
            printf("%.2lf ", prediction[j]);
        }
        printf("\n");
    }
}

```

```

    }
}

t = clock() - t;
double timeTaken = ((double)t)/CLOCKS_PER_SEC;
printf("\nTime taken: %f seconds\n", timeTaken);

return 0;
}

```

Parallel C Program

weather_mpi.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "mpi.h"
#include "csv.h"
#include "utils.h"

char presentCSV[][50] = {
    "./data/2021_tor_data.csv",
    "./data/2021_van_data.csv",
    "./data/2021_mtl_data.csv",
    "./data/2021_cal_data.csv",
    "./data/2021_edm_data.csv",
    "./data/2021_ham_data.csv",
    "./data/2021_hfx_data.csv",
    "./data/2021_kel_data.csv",
    "./data/2021_kit_data.csv",
    "./data/2021_lon_data.csv",
    "./data/2021_osh_data.csv",
    "./data/2021_ott_data.csv",
    "./data/2021_peg_data.csv",
    "./data/2021_qbc_data.csv",
    "./data/2021_reg_data.csv",
    "./data/2021_sas_data.csv",
    "./data/2021_stcath_data.csv",
    "./data/2021_stjn_data.csv",
    "./data/2021_vic_data.csv",
    "./data/2021_win_data.csv"
};

```



```

char prevCSV[][50] = {
    "./data/2020_tor_data.csv",
    "./data/2020_van_data.csv",
    "./data/2020_mtl_data.csv",
    "./data/2020_cal_data.csv",
    "./data/2020_edm_data.csv",
    "./data/2020_ham_data.csv",
    "./data/2020_hfx_data.csv",
    "./data/2020_kel_data.csv",
    "./data/2020_kit_data.csv",
    "./data/2020_lon_data.csv",
    "./data/2020_osh_data.csv",
    "./data/2020_ott_data.csv",
    "./data/2020_peg_data.csv",
    "./data/2020_qbc_data.csv",
    "./data/2020_reg_data.csv",
    "./data/2020_sas_data.csv",
    "./data/2020_stcath_data.csv",
    "./data/2020_stjn_data.csv",
    "./data/2020_vic_data.csv",
    "./data/2020_win_data.csv"
};

// Predicts the weather of a particular day of a year
void predictWeatherForGivenDay(
    double *prediction,
    double *actual,
    char presentData[][50][50],
    char prevData[][50][50],
    int day,
    int *cols,
    int numParams
) {
    // Get current data for the past 7 days
    char CD[7][50][50];
    filterMatrix(presentData, CD, day - 7, day - 1, cols, numParams);

    // Get previous data (1 year old) for a 14 day period
    char PD[14][50][50];
    filterMatrix(prevData, PD, day - 7, day + 6, cols, numParams);

    // 8 sliding windows based on previous data, 7 days per window
    double windows[8][7][numParams];
    memset(windows, 0, 8*7*numParams*sizeof(double));
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 7; j++) {
            for (int k = 0; k < numParams; k++) {

```

```

        windows[i][j][k] = atof(PD[j+i][k]);
    }
}

// Sum the present values
double presentSums[numParams];
memset(presentSums, 0, numParams*sizeof(double));
for (int i = 0; i < 7; i++)
    for (int j = 0; j < numParams; j++)
        presentSums[j] += atof(CD[i][j]);

// Using the present sums, determine the means
double presentMeans[numParams];
memset(presentMeans, 0, numParams*sizeof(double));
for (int i = 0; i < numParams; i++)
    presentMeans[i] = presentSums[i] / 7;

// Determine the means of the previous values based on the
// sliding windows and calculate the Euclidian Distances
double edList[8];
memset(edList, 0, 8*sizeof(double));
for (int i = 0; i < 8; i++) {
    double prevSums[numParams];
    memset(prevSums, 0, numParams*sizeof(double));
    for (int j = 0; j < 7; j++)
        for (int k = 0; k < numParams; k++)
            prevSums[k] += windows[i][j][k];

    double prevMeans[numParams];
    memset(prevMeans, 0, numParams*sizeof(double));
    for (int j = 0; j < numParams; j++)
        prevMeans[j] = prevSums[j] / 7;

    int numPresentMeans = sizeof(presentMeans) / sizeof(presentMeans[0]);
    double euclidDistance = calcEuclidianDistance
    (
        presentMeans,
        prevMeans,
        numPresentMeans
    );

    edList[i] = euclidDistance;
}

// Determine the minimum Euclidian Distance and find its index
double minEd = min(edList, sizeof(edList) / sizeof(edList[0]));

```

```

    int indexMinEd = 0;
    for (int i = 0; i < 8; i++) {
        if (edList[i] == minEd) {
            indexMinEd = i;
            break;
        }
    }

    // Isolate the window with the minimum Euclidian Distance
    double selectedPrevWindow[7][numParams];
    memset(selectedPrevWindow, 0, 7*numParams*sizeof(double));
    for (int i = 0; i < 7; i++)
        for (int j = 0; j < numParams; j++)
            selectedPrevWindow[i][j] = windows[indexMinEd][i][j];

    // Find variation vector based on present data
    double presentVariationVector[numParams][6];
    memset(presentVariationVector, 0, numParams*6*sizeof(double));
    for (int i = 0; i < numParams; i++)
        for (int j = 0; j < 6; j++)
            presentVariationVector[i][j] = atof(CD[j+1][i]) - atof(CD[j][i]);

    // Find variation vector based on previous data
    double prevVariationVector[numParams][6];
    memset(prevVariationVector, 0, numParams*6*sizeof(double));
    for (int i = 0; i < numParams; i++)
        for (int j = 0; j < 6; j++)
            prevVariationVector[i][j] = selectedPrevWindow[j+1][i] -
selectedPrevWindow[j][i];

    // Determine the mean of the present variation vector
    double meanPresentVar[numParams];
    memset(meanPresentVar, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++) {
        double presentVarSum = 0;
        for (int j = 0; j < 6; j++)
            presentVarSum += presentVariationVector[i][j];
        meanPresentVar[i] = presentVarSum / 6;
    }

    // Determine the mean of the previous variation vector
    double meanPrevVar[numParams];
    memset(meanPrevVar, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++) {
        double prevVarSum = 0;
        for (int j = 0; j < 6; j++)
            prevVarSum += prevVariationVector[i][j];
    }

```

```

        meanPrevVar[i] = prevVarSum / 6;
    }

    // Determine the mean of both present and previous mean variation
    vectors
    double meanVariationVector[numParams];
    memset(meanVariationVector, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++)
        meanVariationVector[i] = (meanPresentVar[i] + meanPrevVar[i]) / 2;

    // Retrieve values from the day previous to the day being predicted
    double previousDay[numParams];
    memset(previousDay, 0, numParams*sizeof(double));
    for (int i = 0; i < numParams; i++)
        previousDay[i] = atof(CD[6][i]);

    // Add mean variation vector values to the previous day to get final
    prediction
    for (int i = 0; i < numParams; i++)
        previousDay[i] += meanVariationVector[i];

    // Assign the final prediction values
    for (int i = 0; i < numParams; i++)
        prediction[i] = previousDay[i];

    // Retrive the actual values for comparision purposes
    char actualValues[1][50][50];
    filterMatrix(presentData, actualValues, day, day, cols, numParams);
    for (int i = 0; i < numParams; i++)
        actual[i] = atof(actualValues[0][i]);
}

int main(int argc, char **argv) {
    int rank, numProcs;
    double startTime = 0.0;
    double endTime = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

    if (rank == 0)
        startTime = MPI_Wtime();

    // 9 - Max Temp (°C)
    // 11 - Min Temp (°C)
    // 13 - Mean Temp (°C)

```

```

// 15 - Heat Deg Days (°C)
// 17 - Cool Deg Days (°C)
// 19 - Total Rain (mm)
// 21 - Total Snow (cm)
// 23 - Total Precip (mm)
// 25 - Snow on Grnd (cm)
int cols[] = {9, 11, 13, 15, 17, 19, 21, 23, 25};
int numParams = sizeof(cols) / sizeof(cols[0]);
int numLocations = sizeof(presentCSV) / sizeof(presentCSV[0]);

for (int location = 0; location < numLocations; location++) {
    char presentData[400][50][50];
    char prevData[400][50][50];
    if (rank == 0) {
        // Get data from CSV files
        readCSV(presentData, presentCSV[location]);
        readCSV(prevData, prevCSV[location]);
    }

    // Retrieve data from rank 0
    MPI_Bcast(&presentData, 400*50*50, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Bcast(&prevData, 400*50*50, MPI_CHAR, 0, MPI_COMM_WORLD);

    // Divide days evenly between processes
    int count = 344 / numProcs;
    int remainder = 344 % numProcs;
    int startDay, endDay;
    if (rank < remainder) {
        startDay = rank * (count + 1);
        endDay = startDay + count;
    } else {
        startDay = rank * count + remainder;
        endDay = startDay + (count - 1);
    }

    // Add 8 since first 7 days cannot be predicted
    // printf("Rank %d starting from %d and ending at %d\n", rank,
startDay+8, endDay+8);
    for (int i = startDay+8; i < endDay+8; i++) {
        double prediction[numParams];
        double actual[numParams];
        predictWeatherForGivenDay(
            prediction,
            actual,
            presentData,
            prevData,
            i,

```

NM08: Parallel Computing For Weather Forecasting Using MPI or OpenMP

```
        cols,
        numParams
    );
    // Comment out these print statements for a more accurate time
measurement
    printf("\nFROM PROCESS %d Day: %d\nActual: ", rank, i);
    for (int j = 0; j < numParams; j++) {
        printf("%.2lf ", actual[j]);
    }
    printf("\nPrediction: ");
    for (int j = 0; j < numParams; j++) {
        printf("%.2lf ", prediction[j]);
    }
    printf("\n");
}
}

MPI_Barrier(MPI_COMM_WORLD);

if (rank == 0) {
    endTime = MPI_Wtime();
    double timeTaken = endTime - startTime;
    printf("\nTime taken: %f seconds\n", timeTaken);
}

MPI_Finalize();
return 0;
}
```