# CMPE- 275: ENTERPRISE APPLICATION DEVELOPMENT (SECTION 01)

## PROFESSOR: JOHN GASH

# CLIMATOLOGICAL WIND ROSE GENERATION FOR DATA MINING


**Submitted By:**

**Abhranil Naha (009395555)**
**Aditi Rajawat (010021635)**
**Divya Bitragunta (010125011)**
**Harsh Vyas (010095410)**
**Sandyarathi Das (009994036)**

**Date: 10/28/2015**

## Introduction:

The objective of the project is to design a solution to parallelize data extraction for generation of Wind Rose plots and to provide language binding to call the created compiled library used for the parallelization.

A wind rose, also known as polar rose plot, is a special diagram for representing the distribution of meteorological data, typically wind speeds by class and direction. The data used for plotting the wind rose is huge and has large accumulated data sets spread over years. As this is a highly serial task, a better alternative i.e., parallelization needs to be done for effective utilization and plotting of the data.

The primary goal is to learn about the effects of serial and parallel (intra-process) performance and to explore different technologies that provide the same. Also, a binding needs to be provided, as the project needs to be integrated well and used with the modern day languages like Python, Java etc.

## Technologies Used:

Programming Languages – C++, Java, Python

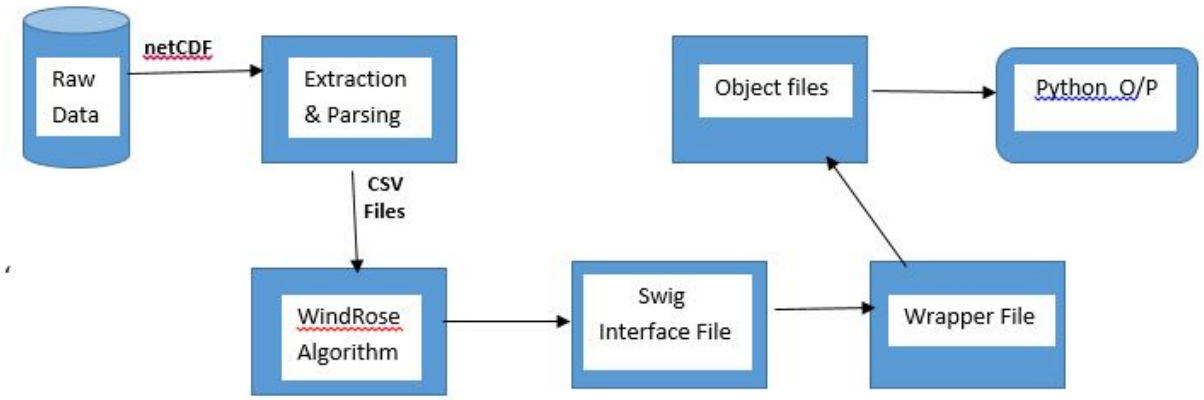
Open MP: Open MP provides a way to standardize directive-based multi-language high-level parallelism that is performant, productive and portable. The Open MP API is a portable, scalable model that gives a simple and flexible interface for developing parallel applications on different platforms.

Open MPI: Open MPI is open source high performance computing which provides the best message passing interface implementation with features like thread safety & concurrency, dynamic process spawning, support for network heterogeneity etc.,

SWIG: Simplified Wrapper and Interface Generator (SWIG) is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. It is an interface compiler used to build more powerful C, C++ programs, rapid prototyping & debugging, systems integration and construction of scripting language extension modules.

## System Architecture:

The raw data from netCDF is extracted and parsed using JAVA code. The CSV files produced are run-through the algorithm to give output in C++. This is passed through the SWIG Interface file, which creates the wrapper (_wrap) file. Compiling the wrapper file we get the .o and .so object files which in turn are imported and executed in Python to give the Python output.




## Implementation:

**Overview:**

High-level flow of the implementation is as follows:

**Source folder: src_OpenMP**

1. Serial01.cpp: Algorithm# 1 Storing Filtered Raw Data in a Local Container
2. Parallel01_01.cpp: Strategy# 1 Parallelizing data processing using local containers3
3. Parallel01_02.cpp: Strategy# 2 Parallelizing data read and processing
4. Parallel01_03.cpp: Strategy# 3 Parallelizing data processing by accessing elements according to the fixed strip width.
5. Serial02.cpp: Algorithm# 2 Without Using a Local Container for Storing Input Raw Data
6. Parallel02_01: Strategy # 1 Parallelizing data read and processing using 2-D local containers
7. Parallel02_02: Strategy# 2 Parallelizing data read and processing using 3-D shared container

**Source folder: src_MPI**

1. Serial.cpp: Procedural serial code with the usage of local containers to store intermediary data in vectors and 2Darrays.
2. SerialModified.cpp: Procedural serial code without the usage of local containers to store intermediary data.

3. MPITaskDivision.cpp: Parallelized code using OpenMPI Send and receive prototypes that spawns two processes to process data for two station ids.

4. MP+MPI.cpp: Hybrid parallelized code optimized using Open MPI send and receive prototypes and open MP parallel for pragma directives.

**Source folder: src_SWIG**

1. Parallel01_03.cpp: C++ Open MP optimized algorithm.

2. windrose.i : Interface file for SWIG.

3. myHeader.h: Header file for the interface file.

**Source folder: src_Parser**

1. extractdata.java: Extraction file to give output csv file containing station id, timestamp, latitude, longitude, wind speed and wind direction.

**Wind rose Data Decomposition Strategies using On-Node Shared Memory ( Open MP )**

We have implemented two algorithms for extracting the raw data and producing the corresponding output array, which can be used for plotting wind rose plot. The first algorithm uses a local container for saving the filtered raw data whereas the second algorithm doesn't stores the filtered raw data in any local container.

**Algorithm # 1 : Storing Filtered Raw Data in a Local Container**

Algorithm #1 uses a structure for storing the filtered raw data. The raw data is read using ifstream from files stored on the hard drive of the system. As the data is read from files, it is filtered using the station Id. Data of the required station Id is stored in a C++ structure with the following specification:

```
struct MesoData
{
        int maxDataPoints;
        int numDataPoints;
        float* windDir;
        float* windSpd;
};
```

As shown above, the structure contains two pointers to point to 1-D arrays, each for wind direction and wind speed. The property numDataPoints is used for total no. of data elements in 1-D arrays pointed by windDir and windSpd pointers. The property maxDataPoints is used to set the maximum storage capacity of each array. Each array is allocated memory on heap using calloc. The value of maxDataPoints is kept equal to a large value i.e. 227 so that the input local container

doesn't fit into the cache memory. As the data is read from the external files and stored into the local container, the corresponding bucket nos. for wind speed and direction are computed.

We have used 5 speed buckets and 16 direction buckets. The speed buckets have following range:

| Speed | Speed Bucket No. |
|---|---|
| =0 | 0 |
| >0 and <=5 | 1 |
| >5 and <=15 | 2 |
| >15 and <=25 | 3 |
| >25 | 4 |

Table 1: Wind Speed Bucket Range

Wind direction is in the range [0, 360o] and is divided into 16 uniform direction buckets.

The output array is a 2-D array of the following structure, where NUM_OF_SECTORS is no. of direction buckets and is equal to 16 and NUM_OF_SPEED is no. of speed buckets and is equal to 5.

```
typedef int outputData[NUM_OF_SECTORS][NUM_OF_SPEED];
```

The output array is computed incrementing the value of its element at the speed and direction bucket nos. corresponding to each input data element.

This serial code takes 83.71 secs to compute the output 2-D array on a single core machine for ~ 200GB of total input data.

The various parallelization strategies applied on this algorithm are described next.

**Parallelization Strategy# 01:**

The strategy# 1 focuses on applying on-node shared memory parallelization strategies in the function used to compute the output array i.e. aggData. The function calculates the direction and speed buckets for each data point stored in the local container. As the data stored in local container is large, the function consumes time.

In order to reduce the time consumed, we divided the for loop iterations, used to iterate over 1-D arrays of the input local container, among 4 threads. In following code snippet, NUM_OF_THREADS is set as 4. Each thread has a local output array of the type of actual output array and each thread stores the partial result in its local output array, corresponding to the data points over which it iterates.

Each thread adds its partial data into global output data after iterating over all data points assigned to it. This is called as reduction. Since more than one thread may access the same location of the output array, this operation is made atomic by using #pragma statement, as shown below.

```
void aggData(MesoData & inputData, outputData & outData)
{
omp_set_num_threads(NUM_OF_THREADS);
int globalPrint = 0, globalNoPrint=0;
#pragma omp parallel
{
      //initialising local output array for each thread
       outputData localOutData;
       int print=0, noPrint=0;
       for(int i=0; i< NUM_OF_SECTORS; i++){
             for(int j=0; j< NUM_OF_SPEED; j++)
                   localOutData[i][j] = 0;
              }
              #pragma omp for
              for(int i=0; i< inputData.numDataPoints;i++){
                  int d = calcDirectBin(inputData.windDir[i]);
                  int s = calcSpeedsBin(inputData.windSpd[i]);
                  if((d<NUM_OF_SECTORS && d>=0)&&(s<NUM_OF_SPEED && s>=0)){
                         localOutData[d][s]++;
                         print++;
                  }
                  else
                        noPrint++;
              }
              for(int i=0; i< NUM_OF_SECTORS; i++){
                  for(int j=0; j< NUM_OF_SPEED; j++){
                        #pragma omp atomic
                        outData[i][j] += localOutData[i][j];
                  }
              }
       }
}
```

If local output arrays are not used, then each thread would have stored its result in the global output array. This operation had to be made atomic using #pragma statement. In this case atomic statement would have been executed N times, where N is the no. of data points stored in input arrays. As N is large, this approach consumes time.

With use of local output arrays, the no. of times atomic operation is executed is reduced from N to T, where T is the no. of threads used. As such using local output arrays followed by a reduction section is more efficient way.

This code executed in 77.21 secs and showed the speed up of 1.08x for ~ 200GB of data processed on single core machine with four threads.

**Parallelisation Strategy# 02:**

Strategy# 2 focuses on parallelizing the function used for reading data i.e. readData and uses aggData same as explored in strategy# 1. This strategy divides the total no. of input files among various threads. Each thread reads data from allocated file and stores it in a local container of type MesoData (local input container). A global container array of type MesoData and size equal to the no. of threads is used. Each thread stores its local input container in global array at index equal to its thread Id. As each thread has a unique thread Id, each thread accesses different memory location of array. As such this operation is not considered atomic. Once each thread stores its corresponding local container in the global array, each local container is added to the input global container of type MesoData.

Using this strategy the algorithm executed in 77.69 secs and showed the performance speed up of 1.07x for ~ 200GB data on a single core machine using four threads.

**Parallelisation Strategy# 03:**

This strategy uses the same algorithm for parallel reading, as described in the previous section and intends to improve aggData function. As the input data, read by function aggData is very huge, the 1-D arrays of the input container (windDir and windSpd) contains large no. of data points. Iterating over these arrays and computing corresponding wind direction and speed bucket is having overhead of N, where N is the total no. of filtered data points. In strategy# 1, we divided the total iterations among total no. of threads. Each task of these threads was consisting of a single iteration. In this strategy, we divided the total iterations in chunks of STRIP_WIDTH, where STRIP_WIDTH was set as 32. Each task assigned to a thread at a time has 32 consecutive iterations to be completed. The no. of iterations may not be exactly divisible by 32 and so the remaining no. of iterations are handled separately by a remainder loop. The remainder loop handles the remaining iterations and divides them among threads as a task of single iteration.
Similar to the strategy# 1, we have used local containers for each thread to accumulate the partial result and the reduction section follows this.

This code executed in 40.99 secs and showed the speed up of 2.04x for ~ 200GB of data processed on single core machine with four threads.

**Algorithm # 2 : Without Using a Local Container for Storing Input Raw Data**

In algorithm# 2 we are reading data from input files stored on the hard disk and filtering the data while reading using ifstream. If the data corresponds to the required station Id then the data is processed to calculate the wind speed and direction bucket.

In comparison to the algorithm# 1, algorithm# 2 eliminates the time of storing the input data into a local container and then accessing the container for data processing.

The algorithm uses the following structure for temporarily storing the data of each line of input file while data reading and processing.

```
struct measurements

{     float windSpd;

      float windDir;

};
```

This serial algorithm took 10.43 secs to process ~ 200GB data on a single core machine.

**Parallelisation Strategy# 01:**
Strategy# 1 divides the unified task of reading data from external files and processing it to compute the output array, into sub tasks. These sub-tasks are divided among threads. The no. of threads used are four. Each of these threads accumulates partial data into local containers, which is of the type of the global output array. The data from the local containers is added to the global container using #pragma atomic statement.
The code executed in 5.43 secs to process ~ 200GB data on a single core machine using four threads. The strategy showed performance speedup of 1.92x.

**Parallelisation Strategy# 02:**
Similar to strategy#1, strategy#2 divides the task of reading data from external files and processing to produce the output array among various threads. The difference is that in strategy# 2 we have used a shared container among the threads. The container is a 3-D array as shown below:

local_wr[NUM_OF_SECTORS][NUM_OF_SPEED][NUM_OF_MAX_THREADS] ;

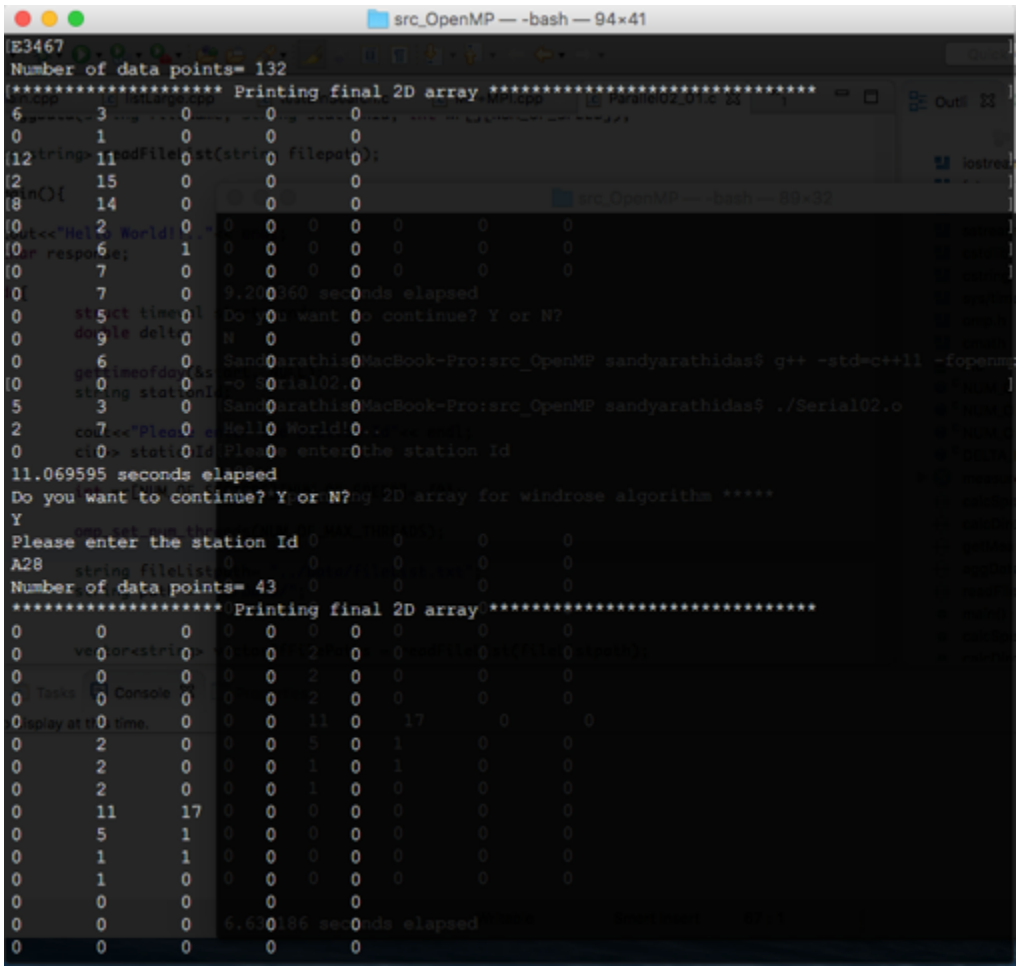
Each thread stores its partial data at local_wr[ ][ ][tid], where tid is the unique thread Id. As each thread has its unique thread id, no two threads access the same memory location. This eliminated the need of using #pragma atomic statements.

For reduction, following for loop is used, which collapse the first and second for loops and divide it among various threads. As a result each element [m,n] of the output array is accessed by only one thread, which adds all values of local_wr[m][n][ ] to it.

```
#pragma omp for collapse(2) schedule(guided)
for(int m=0; m< NUM_OF_SECTORS; m++){
      for(int n=0; n< NUM_OF_SPEED; n++){
            for(int t=0; t< NUM_OF_MAX_THREADS; t++)
                  wr[m][n] += local_wr[m][n][t];
      }
}
```

The code executed in 5.77 secs for processing ~ 200GB data on a single core machine using four threads and showed performance speed up of 1.81x.

**Output for serial code:**



**Output for OPEN MP optimized code:**

```
E3467
Total number of data points= 132
******************** Printing final 2D array ******************************
6       3       0       0       0
0       1       0       0       0
12      11      0       0       0
2       15      0       0       0
8       14      0       0       0
0       2       0       0       0
0       6       1       0       0
0       7       0       0       0
0       7       0       0       0
0       5       0       0       0
0       9       0       0       0
0       6       0       0       0
0       0       0       0       0
5       3       0       0       0
2       7       0       0       0
0       0       0       0       0

5.499086 seconds elapsed
Do you want to continue? Y or N?
Y
Please enter the station Id
A28
Total number of data points= 43
******************** Printing final 2D array ******************************
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       2       0       0       0
0       2       0       0       0
0       2       0       0       0
0       11      17      0       0
0       5       1       0       0
0       1       1       0       0
0       1       0       0       0
0       0       0       0       0
0       0       0       0       0
```

**Observation Charts:**

The chart below demonstrates the comparison of execution time for various code strategies used for algorithm# 1.

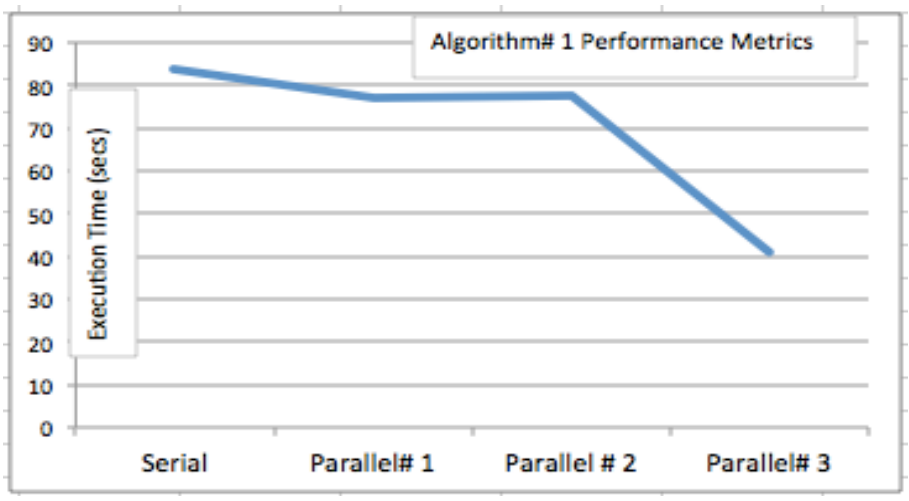



Figure 1: Plot showing comparison of execution time for Algorithm# 1 code strategies

The chart below demonstrates the execution time for various code strategies used for algorithm# 2.

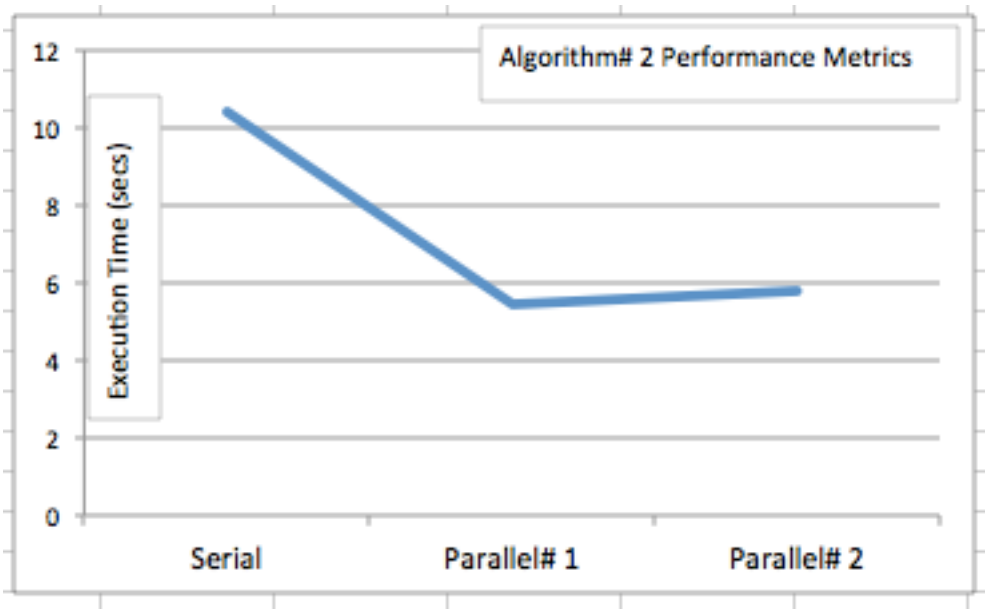



Figure 2: Plot showing comparison of execution time for Algorithm# 2 code strategies

We used the following specifications:

- 2.5GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB L3 cache
- C++ Library for Time Measurement: sys/time.h

**Concepts of MPI design:**

**Communicator**: A communicator defines a group of processes that have the ability to communicate with one another. Each process is assigned a unique rank using which it explicitly communicates with other processes.

**Point-to point communication**: The foundations of communication are built upon send and receive operations among processes. A process may send a message to another process by providing the rank of the receiver process and a unique tag to identify the message. The receiver then posts a receive for the message with or without a tag and handles the data accordingly.

**Collective communication**: When a master process needs to broadcast information to all of its worker processes. MPI handles both point-to-point and collective communications that involves all processes.

**Send and receive directives:**

MPI Send Prototype:

```
MPI_Send( void* data,
        int count,
        MPI_Datatype datatype,
        int destination,
        int tag,
        MPI_Comm
communicator)
```

MPI Receive Prototype:

```
MPI_Recv(void* data,
        int count,
        MPI_Datatype datatype,
        int source,
        int tag,
        MPI_Comm
communicator,
        MPI_Status* status)
```

**Parameter description:**

- The first argument is the data buffer.
- The second and third arguments describe the count and type of elements that reside in the buffer. MPI_Send sends the exact count of elements, and MPI_Recv will receive **at most** the count of elements.
- The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message.
- The sixth argument specifies the communicator and the last argument (for MPI_Recv only) provides information about the received message.

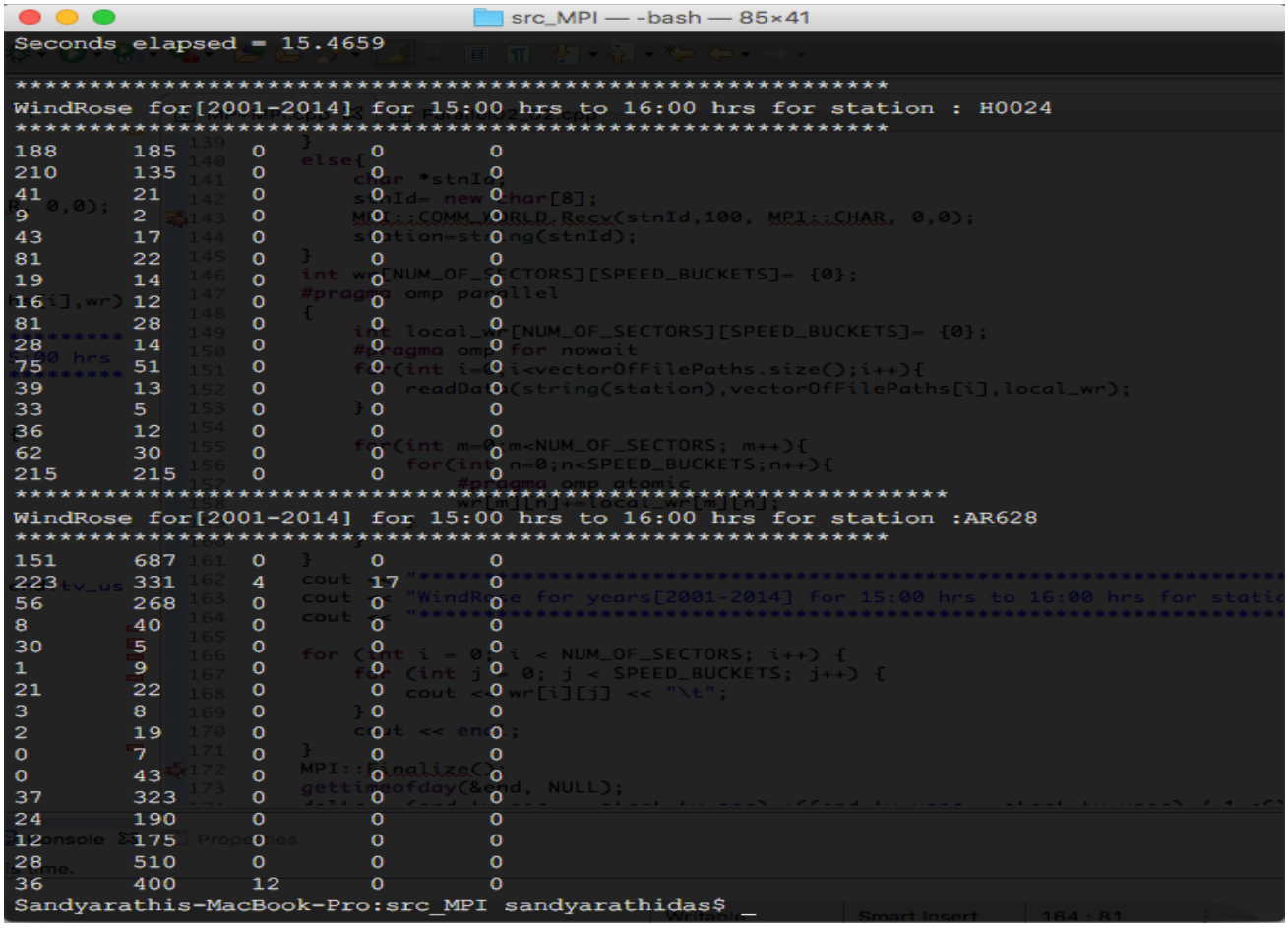
**Strategies used for MPI optimization of Wind rose algorithm**:

- Parallelization using openMPI
- Hybrid parallelization using open MP + openMPI

The flow of the base serial code:

1. Create a list of file-paths to be accessed by the algorithm.
2. Parse through each file in the list to extract the wind speed and wind direction data based on one station id and calculate the matrix.
3. Repeat step 2 for the second station id and calculate the second matrix.
4. Note the time taken for the processing of data of two station ids.
5. Use the matrices to plot two separate wind-rose plots.

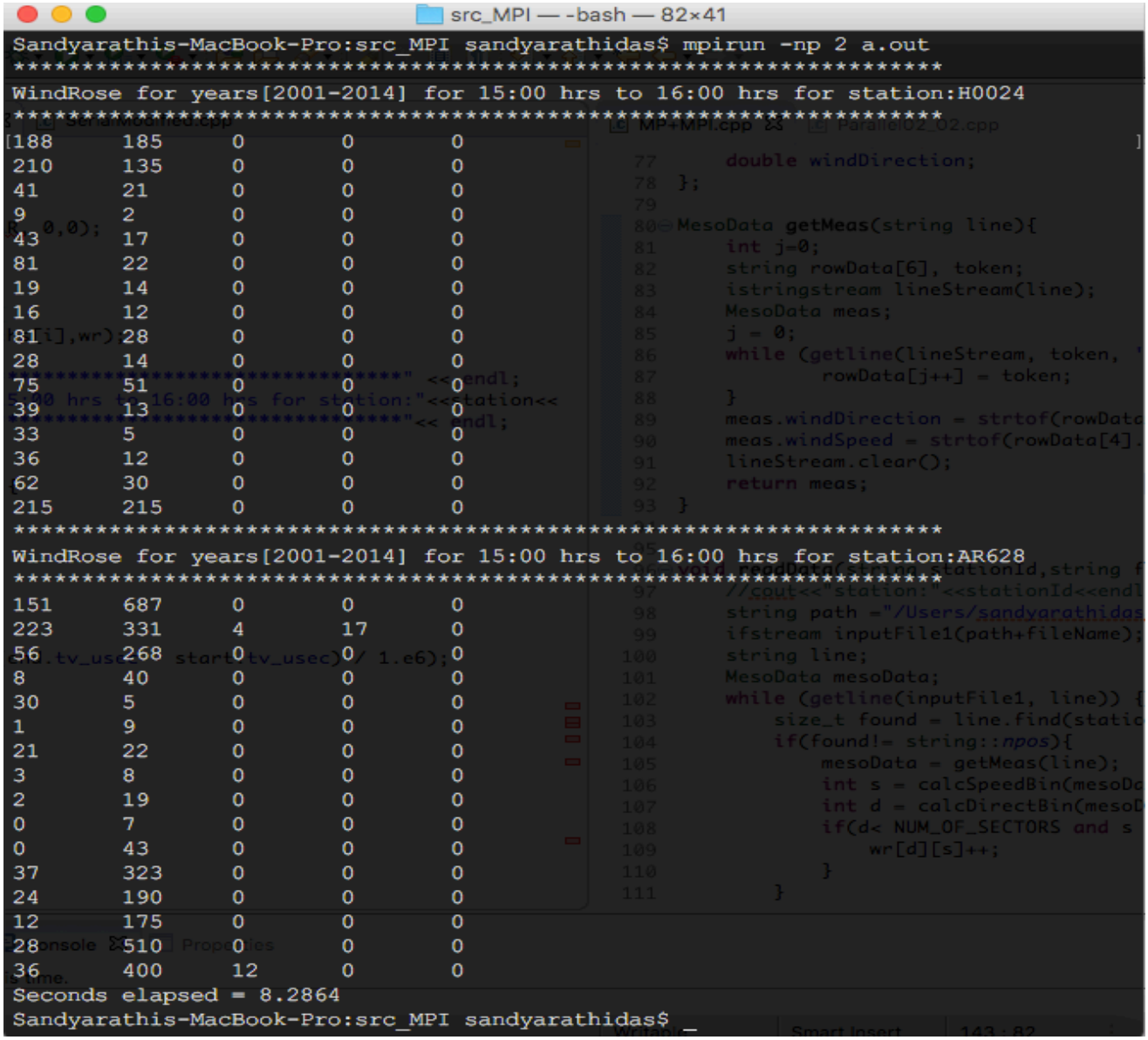
**Output for serial code:**




The flow of the optimized parallel MPI code:

1. Create a list of file-paths to be accessed by 2 processes.
2. MPI_Init – Initialize the parallel processes communication environment.
3. Procure the number of processes spawned in the environment and current process thread rank.
4. Process 0 MPI_Sends the station ids as data to each process:
    a. Process 0 sends stationid1 to process1.
    b. Process 0 works on stationid0 data
5. Parallel processing of file list with respective station ids begin.
    a. Process 0 continues processing of the file list with stationid0.
        i. Parse through each file in the list to extract the wind speed and wind direction data based on one station id0 and calculate the matrix.
        ii. Time taken to compute the plot-matrix is recorded.
    b. Process 1 receives its stationid1 and processes the file list.
        i. Parse through each file in the list to extract the wind speed and wind direction data based on one station id0 and calculate the matrix.
        ii. Time taken to compute the plot-matrix is recorded.
6. Note the time taken for the processing of data of two station ids.
7. Use the matrices to plot two separate wind-rose plots.

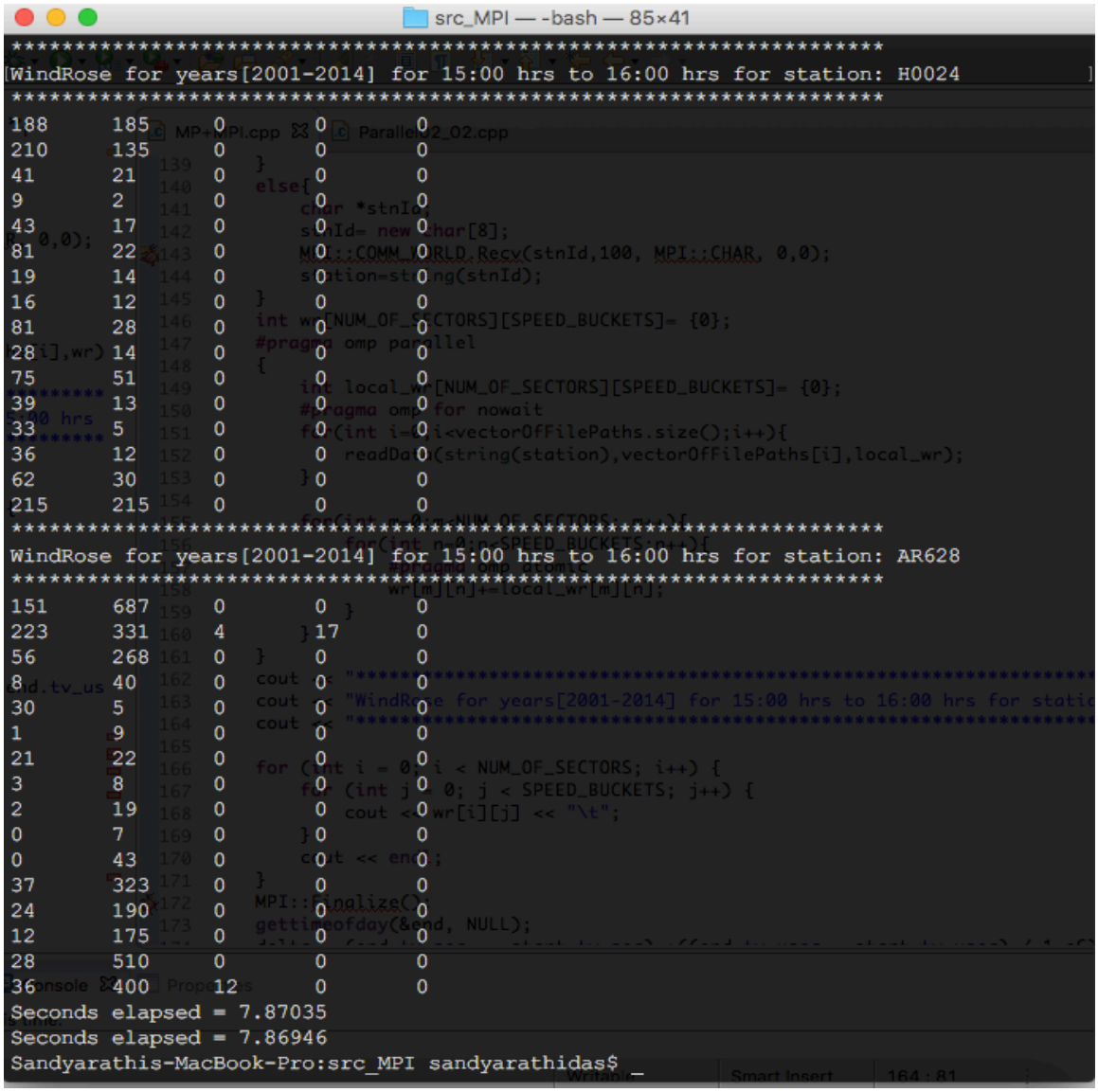
**Output for MPI optimized code:**




**Strategy for a hybrid of Open MP and Open MPI optimization:**

To explore another level of optimization, we have employed both open MP and OpenMPI in the algorithm to run parallel processes along with loop level multi-threading optimization technique using #pragma omp for directive to parallelize the iteration through the file-list.

## Output for openMP+MPI optimized code:




## Observations:

| | Serial.cpp | MPI optimized | OpenMP+MPI optimized |
|---|---|---|---|
| **Execution Times in seconds** | 15.4659 | 8.6404 | 7.86946 |
| | 15.7985 | 8.727 | 7.84125 |
| | 15.5504 | 8.384 | 7.7516 |
| | 15.5493 | 8.3856 | 7.805 |
| | 16.0916 | 8.307 | 8.215 |

| | | | |
|---|---|---|---|
| **Average Time** | 15.69114 | 8.4888 | 7.896462 |

Table1. Readings of execution times for each of the optimization strategy

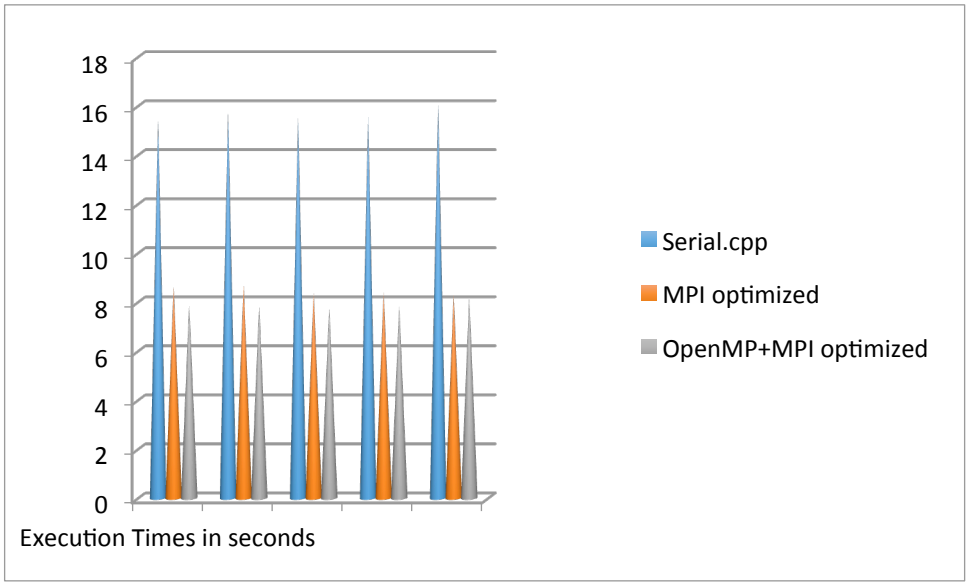



Fig1: Chart representing the execution times of the strategies employed

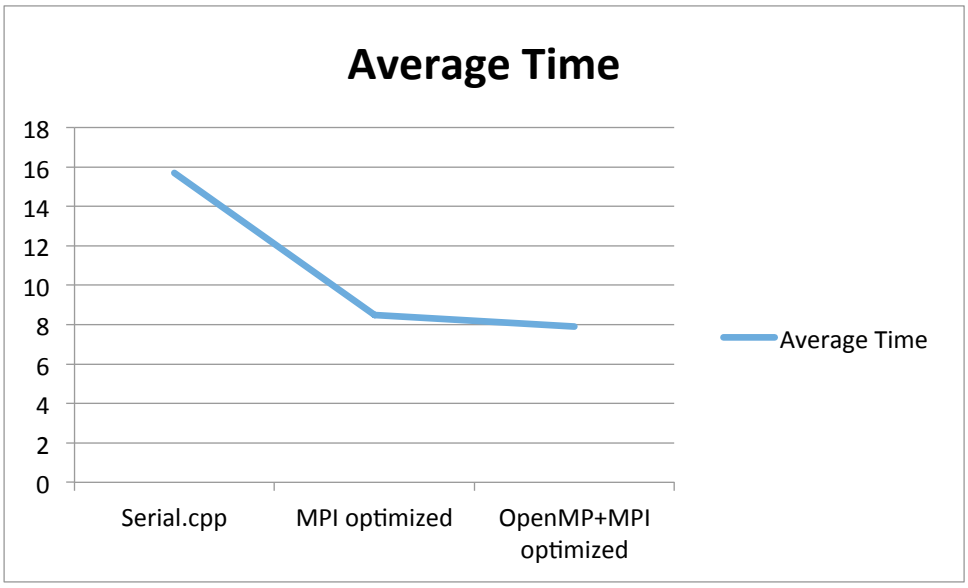



Fig2: Chart representing the 2x optimization achieved by employing parallelizing techniques.

System specifications: 2.5GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB L3 cache

## SWIG IMPLEMENTATION:

Swig Instructions:

- Make an interface file named "filname.i".
- Add the module name that you want to use in your higher-level language(like python).
- Add required header files and functions that you want to use in the %{""""""""" %} section.
- Now compile the file using swig.

Compilation with Swig Instructions:

Use the following commands for compilation (Don't include "" in your commands)

swig -c++ -python "filename.i"

g++ -c -fpic -fopenmp "filename.cpp"

g++ -c -fpic -fopenmp "filename_wrap.cxx" -I "path to "

g++ -shared "cpp object file" "wrapper object file" -o "_outpufile.so" –fopenmp

After compiling filename.i file it will generate the wrapper file filename_wrap.cxx. The wrapper file has a code, which is converted line by line to be used in python. Now the following 3 commands will compile and link the object files. At the end it will generate the output file, which can be used in python or any higher level language. Now Go to Python console(use command python).Type the following commands.

```
import outputfile

outputfile.callFunction()
```

**Swig output:**

```
harshvyas@harshvyas-VirtualBox:~/Desktop/WindRosePOC/SWIG$ swig -c++ -python win
drose.i
harshvyas@harshvyas-VirtualBox:~/Desktop/WindRosePOC/SWIG$ g++ -c -fPIC -fopenmp
 Parallel01_03.cpp -std=c++11
harshvyas@harshvyas-VirtualBox:~/Desktop/WindRosePOC/SWIG$ g++ -c -fPIC -fopenmp
 windrose_wrap.cxx  -std=c++11 -I /usr/include/python2.7
harshvyas@harshvyas-VirtualBox:~/Desktop/WindRosePOC/SWIG$ g++ -shared Parallel0
1_03.o windrose_wrap.o -o _windrose.so -fopenmp
harshvyas@harshvyas-VirtualBox:~/Desktop/WindRosePOC/SWIG$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import windrose
>>> windrose.callFunction()
Hello World!!..
Please enter the station Id
E3467
Total number of data points= 132
******************** Printing final 2D array ********************************
6       3       0       0       0
0       1       0       0       0
12      11      0       0       0
2       15      0       0       0
8       14      0       0       0
0       2       0       0       0
0       6       1       0       0
0       7       0       0       0
0       7       0       0       0
0       5       0       0       0
0       9       0       0       0
0       6       0       0       0
0       0       0       0       0
5       3       0       0       0
2       7       0       0       0
0       0       0       0       0

60.287541 seconds elapsed
Do you want to continue? Y or N?
Y
Please enter the station Id
A28
Total number of data points= 43
******************** Printing final 2D array ********************************
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       2       0       0       0
0       2       0       0       0
0       2       0       0       0
0       11      17      0       0
0       5       1       0       0
0       1       1       0       0
0       1       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0

8.867412 seconds elapsed
Do you want to continue? Y or N?
N
>>>
harshvyas@harshvyas-VirtualBox:~/Desktop/WindRosePOC/SWIG$
```

## Conclusion:

We have achieved approximately 2x performance improvement using openMP, openMPI and a hybrid technique (Multi-process-multi thread) for optimizing the serial Wind Rose algorithm. We have also bound the Windrose cpp code to Python using Swig.

**GitHub Repository Link:**

https://github.com/Sandyarathi/WindRosePOC

**Future Enhancements:**

The output from python using SWIG can be used to plot the WindRose.

## References:


HTTP://SOURCEFORGE.NET/PROJECTS/WINDROSE/

HTTPS://WWW.MESONET.ORG/INDEX.PHP/WEATHER/DAILY_DATA_RETRIEVAL

HTTP://WWW.SWIG.ORG/

HTTP://OPENMP.ORG/WP/

HTTP://WWW.OPEN-MPI.ORG/

HTTP://COLFAXRESEARCH.COM/CATEGORY/TUTORIALS/