Benchmarking parallel implementations of cloud type clustering from satellite data

CyberTraining: Big Data + High-Performance Computing + Atmospheric Sciences

Carlos Barajas[1], Lipi Mukherjee[2], Pei Guo [3], Susan Hoban[4]
Faculty mentors: Dr. Daeho Jin [5], Dr. Aryya Gangopadhyay [2] , Dr. Jianwu Wang[2]

[1]Department of Mathematics and Statistics, UMBC
[2]Department of Information Systems, UMBC
[3]Department of Physics, UMBC
[4] Joint Center for Earth Systems Technology, UMBC
[5] GESTAR, USRA, and NASA GSFC

Technical Report HPCF–2018–11, hpcf.umbc.edu > Publications

Abstract

The study of clouds, i.e., where they occur and what are their characteristics, plays a key role in the understanding of climate change. The aim of this project is to use machine learning in conjunction with parallel computing techniques to classify cloud types. Experiments with k-means clustering are conducted with two parallelism techniques.

1 Introduction

The climate of Earth tends to maintain a balance between the energy reaching the Earth from the Sun and the energy leaving the Earth to space. This is also known as Earth's "radiation budget." The components of the Earth system contributing to the radiation budget include Earth's surface, atmosphere, and clouds [8, 13, 14]. The study of clouds, including their frequency of occurrence, location and characteristics, plays a key role in the understanding of climate change. Thick clouds in the lower atmosphere primarily reflect the incoming solar radiation and consequently cool the surface of the Earth. On the other hand, thin clouds in upper atmosphere easily transmit the incoming solar radiation and also trap some of the outgoing infrared radiation emitted by the Earth's surface and radiate it back downward, consequently warming the atmosphere and surface of the Earth. Usually, the clouds in the upper atmosphere have a colder cloud top that traps the energy in form of outgoing longwave emission. As a result of the trapped energy, the temperature of the Earth's atmosphere and surface increases until the longwave emission to space is balanced by the incoming solar shortwave radiation.

Two parameters that are directly related to the heating and cooling effects of clouds are cloud optical thickness (COT) and cloud top height (CTH) (which is related to cloud top pressure (CTP)). COT is a measure of the thickness of cloud which largely determines the reflection of sunlight (shortwave), i.e., the cooling effects of clouds. The thicker the cloud the stronger the reflection. The CTP also plays a role in the warming of clouds in the thermal infrared region (greenhouse effect). For example a cloud with high CTP and low COT would

result in warming affect but a cloud with a high CTP and high COT would result in a net 0 or "neutral" affect. For this reason, the satellite retrievals of the cloud COT and CTP are often portrayed in a joint histogram of COT and CTP.

We can study these variables using NASA satellite data such as Moderate Resolution Imaging Spectroradiometer (MODIS) and Cloud-Aerosol Lidar and Infrared Pathfinder Satellite Observation (CALIPSO). The clouds can be studied through atmospheric modelling, where computer simulations are used in conjunction with field measurements and lab studies to further our understanding of cloud physics. In this work we are using MODIS data for one year (2005), and we employ K-Means clustering to identify the cloud types.

2 Background

2.1 Joint Histograms

A snapshot of cloud at one location and time is detected by the satellite, and COT and CTP from these measurements maybe reported as the 2-D joint histogram. [9]. The International Satellite Cloud Climatology Project (ISCCP) cloud type is employed in order to interpret the histogram [12]. With this categorization, it is easy to link the joint histogram data to real world clouds as shown in Figure 2.1 and Figure 2.2.

At the $1^\circ \times 1^\circ$ grid cell, it is natural that multiple cloud types occur together. Consequently individual joint histogram data (representing one time and one location) have great variability. This is the reason why the concept of 'cloud regime' was created. In short, the cloud regime is the concept representing the domain mixtures of cloud types.

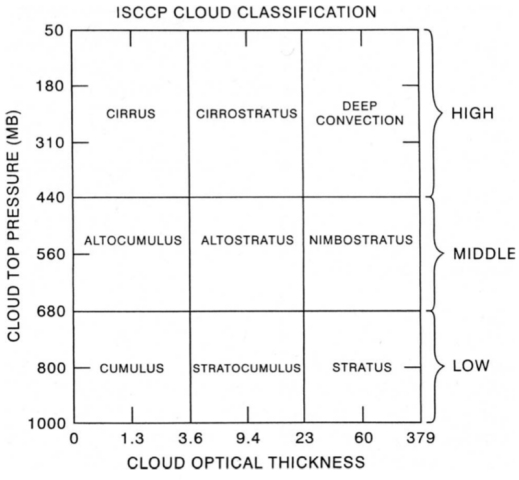


Figure 2.1: Cloud type definitions can be extrapolated using joint histograms where the joint-histogram is broken up into regions which are blocked according to cloud-type. Additional information on this technique can be seen in [12].

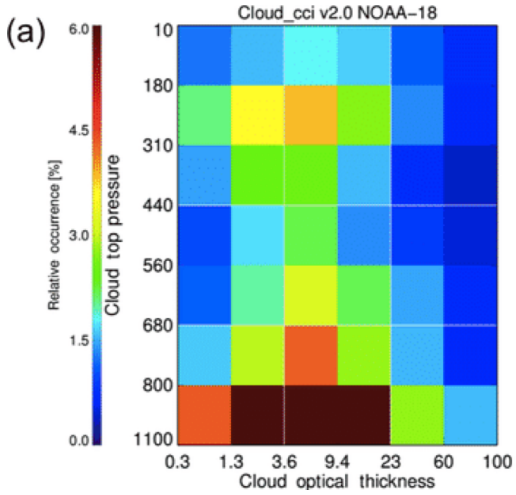


Figure 2.2: Joint histogram of cloud top pressure and cloud top thickness suggesting high frequency of stratocumulus clouds.

2.2 K-means

In order to cluster the cloud types based on their properties (COT, CTP) as shown in Figure 2.3, we used K-means clustering algorithm.

The general idea behind k-means clustering is grouping data according to similarities typically using distance as a similarity measure [7].

k-means is an unsupervised clustering algorithm. The procedure includes classifying a given data set through a certain number of clusters (assume k clusters), where k is fixed *a priori*. It starts with choosing k cluster centers (centroids) in the space representing the data objects. Next, each data object is assigned to its closest cluster center, based on Euclidean distance. After assigning all data to a a set of cluster centers, the new positions of the k centroids are calculated. If the centroids are found to move, the previous steps of assigning and calculating are repeated until the centroids move no more [10][11].

The K-means algorithm is sensitive to the initialization of randomly selected cluster centers [7]. To reduce the randomness in the cluster results, it is better to initialize the centroids as sparse as possible. To get stable clustering results, the algorithm can be made to run multiple times, and the within-cluster-variance and Euclidean distance can be used as clustering criteria to be optimized.

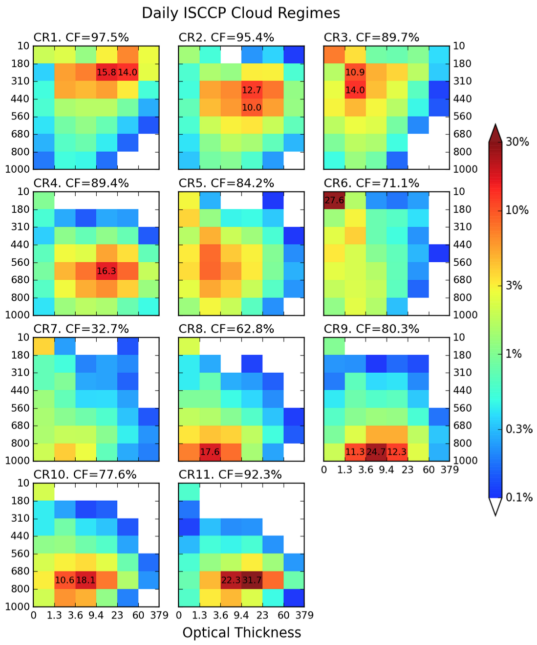


Figure 2.3: The cloud regime (CR) centroids of daily ISCCP joint histograms. The cloud fraction of each regime (=sum of 42 bin CF values) is also provided. When bin values are larger than 10%, they are explicitly shown [9].

3 Implementation Details

All implementations discussed in this section can be found on GitHub [5].

3.1 Python and FORTRAN

The original source code that was provided to our team used two programming languages to achieve maximum performance while also maintaining an acceptable level of readability. The calculation-heavy sections of the code were written in FORTRAN90 to leverage the computational performance of FORTRAN. The high level functions, such as reading data from disk, writing to disk, correct algorithm ordering, and other smaller, computationally cheap tasks, were written in Python to keep the remainder of the code readable and take advantage of the large open-source libraries that are commonly available through GitHub or PyPy. The original source code pulled on the popular numerical package NumPy for the usage of arrays, allowing for compact memory management of large scale data. By only using the arrays from NumPy the code is capable of using any modern version of the NumPy distribution available from PyPy or GitHub. The FORTRAN aspects of the code were compiled to a dynamically linked library using the `f2py` executable that comes bundled with NumPy 1.14+, which allowed for the FORTRAN code to be directly callable in Python. For the best performance the arrays were arranged in FORTRAN's native column-major ordering. Lastly OpenMP was integrated into FORTRAN hot loops using pragmas

to allow the code to make use of all resources on the host machine.

The code takes in a binary data file that is an $n \times 42$ multi-dimensional array where the n dimension is the number of total histograms to be used for the k-means algorithm and 42 is the number of cloud fraction bins associated with each histogram. Concisely: each row is one joint histogram. The binary data are produced using level 3 MODIS data that are provided in an HDF format. The binary format is more compact on disk and is loaded directly into an array using NumPy.

First the code sets the total number of threads that FORTRAN's OpenMP can create by using OpenMP pragmas. Then Python calculates the k initial centroids that will be used to start K-means clustering. The algorithm for calculating the initial centroids is similar to `k-means++` in that it attempts to make the initial centroids sparse so that they can each encompass the largest amount of data with minimal, if any, overlap. At this point the program starts the iteration process for finding a new centroid. The first iteration uses the initial centroids, and the data and the centroids are passed to the FORTRAN subroutine `assign_and_get_new_sum` that attempts to find a new centroid and then computes the Euclidean distance of each record from the post-iteration centroid. Python receives the the post-iteration centroids and computed distances from FORTRAN, whereby it uses NumPy's array arithmetic to compute the mean distances for the new centroids. A quick check is performed to determine if the post-iteration mean distances are superior to the pre-iteration mean distances. If the post-iteration distances are superior, the new centroids are adopted to be the current centroids, and the next iteration begins. This process continues until either the maximum number of iterations is reached or the mean distance between the pre-iteration centroid and the post-iteration centroid is smaller than the given threshold. After the iterations have ceased, the final centroids are written to disk in a binary format so that may be post-processed at a later time.

Lastly a Python script written by Dr. Jin reads in the binary version of the centroids, which were output by the k-means code, to produce the several joint histograms seen in Figure 2.3.

3.2 MPI and Cython

The first step in implementing MPI was actually to convert the FORTRAN code into C code to maintain the high performance and ease the MPI parallelization, since MPI is better equipped to handle row-major ordering for arrays, which is C's native ordering. However the CPython API is rather terse and unwieldy so when trying to implement a simple interface a great deal of boilerplate code has to be written. The use of Cython removes a large amount of the API complexities because Cython will automatically generate the CPython API compatible C code from the Cython code while, when properly optimized, maintaining C-like performance. Fortunately the Cython handler is an executable that comes bundled with a modern NumPy distribution at or beyond 1.14+. The Cython handler converts the Cython code into C that uses the CPython API and then the generated C code is compiled to a dynamically linked library which can be imported directly into Python. This process is similar to how `f2py` works for the original FORTRAN implementation. One benefit is that

Cython allows any C function to be used inside the Cython code. The major benefit is that Cython also allows for C speed memory accesses via `Memoryviews`. `Memoryviews` provide a closer interface to the heap than NumPy arrays and actually allow the block of memory controlled by the NumPy array to be changed as if it were created using `malloc`. With all these tools in place the FORTRAN code was converted line by line into Cython code and all original NumPy arrays were converted into row-major format so that they are compatible with the C-style arrays that MPI prefers. Importantly Cython allows for easy integration of OpenMP into the `cdef` functions, which means that portions of the code needed to be refactored into `cdef` and `def` portions. [6]

Lastly `mpi4py` is used to integrate MPI into the Python portion of the code since Cython handles the computation efficiently, MPI will only be tasked with chopping the data into smaller portions and sharing minor amounts of data. An `MPI.allreduce` is usedf for reducing integers and simple datatypes. Whereas we used `MPI.Allreduce` for reducing NumPy arrays efficiently.

The environment variable `OMP_NUM_THREADS` is set to control the number of threads that OpenMP is allowed to use during pragma loops. Additionally the Intel OpenMP environment variable `KMP_AFFINITY` is set to `scatter` so that threads are distributed as evenly as possible among the cores. The number of threads used is $t_p = c_{total}/p_{ppn}$. Where t_p is the threads per node, c_{total} is the total cores per node, and p_{ppn} is the processes per node. On HPCF–2013 $c_{total} = 16$. This balancing system allows for all node resources to be used, even if $p_{ppn} < c_{total}$.

Before any K-means calculations begin, each MPI process determines its own process rank $rank$ and the total number of processes p_t running. The processes then take $\text{rec}_{local} = \text{rec}_{total}/p_t$, where rec_{local} is the number of records for which each process is responsible for computing the Euclidean distance, and rec_{total} is the total number of records. In the event that the total number of records cannot be evenly distributed, the remaining records will be distributed such that no processes have more than one record compared to any other process. Then each processes reads in its respective records from the same data binary data as mentioned in Section 3.1. This means that all processes have no access to any records beyond their own as no records are duplicated.

First the initial centroids are calculated in Python as mentioned in Section 3.1. The data and the current centroids are passed to the Cython `def` function `assign_and_get_new_sum` , which calls the `cdef` functions `calculate_cl` and `calculate_outsum`. Since k-means is deterministic, the new cluster produced by `calculate_cl` is the same on every process. The Euclidean distance, however, is calculated differently compared to the FORTRAN code. Consider Figure 3.1 such that the image represents just one centroid of k many with $p_t = 2$. Process 0 computes the Euclidean distance from the centroid to all its respective records. Likewise Process 1 computes the euclidean distance from the centroid to all of its respective records. These distances and clusters are returned from Cython to Python. Since the clusters were computed identically by all the processes, they are able to merge their own local distances for their records with each cluster into mean distance for each cluster using an `MPI.allreduce`. At this point all processes have the identical post-iteration centroid, pre-iteration centroid, and mean distances to the centroids; therefore, all processes make the

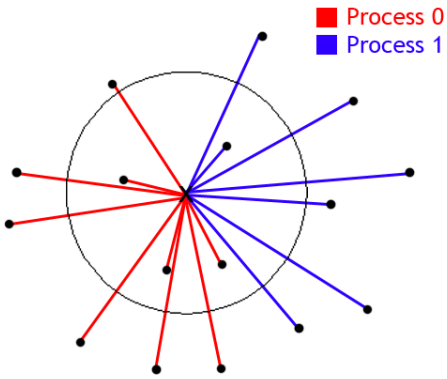


Figure 3.1: The general idea for parallelization over a large data set with the repeated calculation. One record is represented as a black dot and the colored lines tell which process would be handling that Euclidean distance from the current center of the cluster.

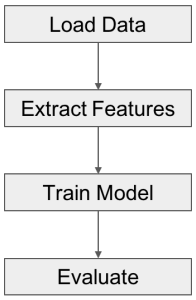


Figure 3.2: Spark Machine Learning Workfolow

same choice whether the post-iteration centroid should become the new centroid or not. After the maximum number iterations has been reached or the within-cluster-variance threshold is satisfied, Process 0 writes the final centroids to disk in a binary format to post-processed later.

The post-processing is then done using Dr. Jin's original Python script similar to Section 3.1.

3.3 Python and Spark

We used Spark MLlib, which is Apache Spark's scalable machine learning library, in our program to apply the K-means algorithm to cluster cloud regime [1][2]. We utilized the latest version of Spark, 2.3.0 with Python. The workflow is as Figure 3.2. There are four steps in the workflow: load data, extract features, train model and evaluate. We will discuss these steps in detail.

We first loaded data into Spark to create a Spark DataFrame, which is organized as a distributed collection of data by named columns [4]. In the load data step, since the raw data were in .dat file format, we first tried to extract the binary file to a csv format spreadsheet

to allow Spark to load it. However, there were two problems with this approach: 1) file size: The size of the csv file was larger than the binary file. It increased from 550 MB to more than 770 MB. In the future, if we explorer larger data files, the increase in raw data size could be unmanageable and should be avoided. 2) efficiency: It took time to transform binary file to csv, and Spark took some time to load the csv file and create the DataFrame. Thus, the csv approach was not an ideal solution. Instead, we extracted the data from the binary file using Numpy in Python, and then used Pandas to save the raw data to a Pandas DataFrame in Python.

After creating Spark DataFrame, we could see that our data contained 42 columns. Since the 42 columns were the 42 bins of the joint histogram, we extracted the 42 features and assembled them as a features vector. The features vector was preparation for clustering.

In the clustering process, we used K = 10, which was selected by Dr. Jin. Also, we changed the variable value of max.iteration to 40, which in Spark default setting was 20, to make sure that a sufficient number of iterations occurred before the algorithm stopped [3]. We also tried to set larger iteration limits such as 2000, but the run time and clustering result remained similar. So we conclude that 40 iterations are enough in our case.

To evaluate our clustering results, we generated the result plots and compared it to Dr. Jins. We also executed the program many times and output the silhouette with squared euclidean distance to make sure that our result was relatively stable [10].

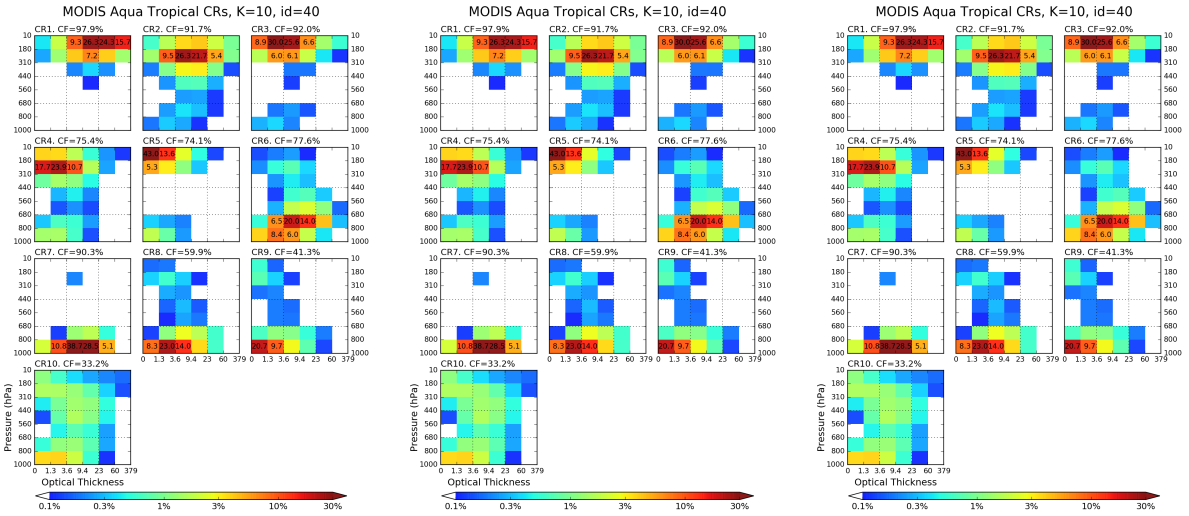
4 Results

4.1 Code Validity

When parallelism is involved, we commonly assume that there has to be some numerical drawback. For example, if parallelism is implemented incorrectly, values can become improperly rounded, images can degrade in quality, and values that serial code correctly computed are now no longer within an acceptable margin of error. Any code which produces incorrect results in order to improve performance cannot be accepted as correct code. In order to demonstrate how the accuracy of the serial was preserved, each of the implementations was run using the same initial parameters and post-processed using the script provided by Dr. Jin, to test that the images are comparable qualitatively and quantitatively.

First consider Table 4.1. The FORTRAN and Cython joint histograms are identical in their order, shape, and colorings. Since the FORTRAN algorithms were recoded line by line in Cython it makes sense that the results should be identical. The only fundamental difference between the two coding schemas was the major ordering of the data and record splitting via MPI, which still uses all records for computation in the end. More importantly, the Cython and the FORTRAN both used the same Python functions to calculate the initial centroids, thus there shouldn't be any reason for deviation, aside from small rounding errors which we accept will be within the margin of error, as they beyond the control of the programmer in this case.

The difference between how the Spark implementation and the FORTRAN implementa-



FORTRAN/Python MPI/Cython Spark/Python

Figure 4.1: Post-processed joint histogram results of the k-means final stable clusters for all three implementations with initialization magic set to 40.

Table 4.1: FORTRAN/Python OpenMP wall clock results with total number of threads used in HH:MM:SS format.

PPN	1	2	4	8	16
Time Elapsed	00:14:59	00:07:10	00:03:47	00:02:58	00:02:38

tion work is significant. The Spark code uses open-source libraries designed by a community, rather than Dr. Jin. Also, Spark's parallelism uses a completely different methodology than the the typical operation of 1 compute node with OpenMP enabled code.

Despite these differences the Spark code also correctly computed the same clusters with the same centroids and same ordering.

Therefore both of the alternative implementations can be regarded as accurate parallelized representations of the original code, as they show no signs of result degradation.

4.2 Performance

4.3 FORTRAN/Python

Table 4.1 presents the recorded times for the varying number of OpenMP threads for the original FORTRAN/Python code. Clearly, as we use more threads the time improves slightly, but there seems to be a bottleneck somewhere, because even though we're using 16 threads (see the final row) the time is not 16× faster! We can use the best speed possible from these results as a baseline to compare other results to. Additionally the fact that using more threads shows a clear improvement in speed means that the OpenMP parallelism is having a positive effect on the performance. However as the threads double, the timing is not halved,

Table 4.2: MPI/Cython wall clock results with Nodes and Processes Per Node in HH:MM:SS format.

Nodes	1	2	4	8
1 ppn	00:01:01	00:00:34	00:00:17	00:00:08
2 ppn	00:01:23	00:00:41	00:00:20	00:00:11
4 ppn	00:01:50	00:00:54	00:00:28	00:00:16
8 ppn	00:02:42	00:01:22	00:00:45	00:00:29
16 ppn	00:04:47	00:02:32	00:01:29	00:01:07

implying that the implementation has a bottleneck beyond the OpenMP components.

So the 1-node, 1-process-per-node, 16-thread timing in Table 4.1 shall be the number that all other timings are compared too.

4.4 MPI/Cython

The MPI results in Table 4.2 demonstrate a counter intuitive concept. As the number of processes per node increases the performance decreases. Looking at the 8 node column as the number of processes per node increases the time gradually worsen at an increasing rate until the slight doubling of the time from eight processes per node to sixteen processes per node. This same behavior is consistent for all node columns. However for all rows as the number of nodes used increases the performance also increase which is the expected outcome.

All but the bottom left three timings in Table 4.2 are better than the best timing in Table 4.1. Consider the best timing from the FORTRAN code. This FORTRAN timing is twice as fast as the slowest single node performance time for the MPI enabled code. However this timing takes twice as long as the fastest single node performance time. The 1 node 1 process per node timings in Table 4.2 uses the same amount of resources as as the best timing in Table 4.1 meaning that the benefits of Cython, rather than MPI, are to thank for the initial jump in performance. Enabling MPI and using 8 nodes allows for a mere 8 second run time which is 18× faster than the FORTRAN performance time and approximately 7.5× faster than the single node MPI code.

An overall evaluation of the table implies that the best use of MPI, for the problem size, is to enable the use of more OpenMP threads rather than distributing the records for a smaller per-process problem size. The data set fits comfortably within the total memory capacity available. With larger data sets approaching the node memory limit of ≈ 62GB MPI should start to demonstrate a clear performance improvement as the communication time becomes a small player in the overall timing results.

4.5 Spark/Python

Table 4.3 is a run time table of our Spark implementation. Before load the data into Spark program, we first convert the original binary data to .csv format, then load the .csv data into Spark as Spark DataFrame.

Table 4.3: Spark wall clock results with total number of nodes used in HH:MM:SS format.

Nodes	1	2	4	8
Run Time	00:09:03	00:06:16	00:02:51	00:02:09

The Spark implementation program run time is faster compared to the original implementation in FORTRAN when running in one or two nodes. We also found that with node number increasing, Spark performance didn't change much, although there was a decreasing trend. We conclude that performance didn't improve much with increasing number of nodes because the size the data set (550MB) is not big enough to make significant difference, and there's an overhead when loading the data into Spark as DataFrame.

5 Conclusions

Both parallel implementations managed to correctly compute the same clusters as the original code. Only MPI/Cython implementation managed to outperform the original code with the same amount of resources at their disposal.

Only MPI/Cython managed to outperform the original implementation when using more resources than the original code was capable of using.

However, the demonstration of increased performance of both parallel implementations was severely limited by the lack of data. Spark is designed to handle data on the TB scale, yet we only used less than 1 GB. These results are not indicative of what would happen given 20+ GB of data or larger.

So we cannot conclude that Spark is an inferior implementation, only that it is not right for such a small scale test.

Likewise when MPI scaled to multiple nodes, performance started to decrease, indicating that the data set is also too small for MPI to effectively slice it into distributed portions. The saving grace of MPI was the fact that using one node, one process per node, with 16 threads per node is essentially just a language change from the original FORTRAN code meaning that Cython out performed `f2py`.

In the future we would like to test these parallel implementations with much larger data sets. We propose that both Spark and MPI will have significant increases in performance beyond the original code once scaled up to 20+ GB.

Acknowledgments

This work is supported by the grant CyberTraining: DSE: Cross-Training of Researchers in Computing, Applied Mathematics and Atmospheric Sciences using Advanced Cyberinfrastructure Resources from the National Science Foundation (grant no. OAC–1730250). Co-author Lipi Mukherjee additionally acknowledges the NASA Earth and Space Science Fellowship Program - supported by NASA Headquarters. The hardware in the UMBC High

Performance Computing Facility (HPCF) is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

References

[1] Apache Software Foundation. Apache spark – unified analytics engine for big data. https://spark.apache.org/. Accessed: 2018-06-15.

[2] Apache Software Foundation. MLlib | Apache Spark. https://spark.apache.org/mllib/. Accessed: 2018-06-15.

[3] Apache Software Foundation. Spark mllib python api docs. https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.clustering.KMeans. Accessed: 2018-06-15.

[4] Apache Software Foundation. Spark sql, dataframes and datasets guide. https://spark.apache.org/docs/2.3.0/sql-programming-guide.html. Accessed: 2018-06-15.

[5] Carlos Barajas, Pei Guo, Lipi Mukherjee, and Jin Daeho. https://github.com/AmericanEnglish/K-means_Clustering4CloudHistogram. Source Code.

[6] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, March–April 2011.

[7] J. Fauld. Unsupervised learning: Association rule learning and clustering, March 2018.

[8] Steve Graham. https://earthobservatory.nasa.gov/Features/Clouds/?src=share, March 1999.

[9] Daeho Jin, Lazaros Oreopoulos, and Dongmin Lee. Regime-based evaluation of cloudiness in cmip5 models. *Climate Dynamics*, 48(1):89–112, Jan 2017.

[10] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[11] Polytechnic University of Milan. A Tutorial on Clustering Algorithms k-means clustering. https://home.deib.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html#macqueen. Accessed: 2018-06-15.

[12] William B. Rossow and Robert A. Schiffer. Advances in understanding clouds from isccp. *Bulletin of the American Meteorological Society*, 80(11):2261–2288, 1999.

[13] J. M. Wallace. *Atmospheric science: An introductory survey.* Academic Press, 1977.

[14] Z. Zhang. Class lectures:big data+hpc+atmospheric science, module 4: Overview of earth's atmosphere and radiative energy budget, March 2018.