Benchmarking Parallel K-Means Cloud Type Clustering from Satellite Data

Carlos Barajas[1], Pei Guo[2], Lipi Mukherjee[3,4], Susan Hoban[4], Jianwu Wang[2], Daeho Jin[5], Aryya Gangopadhyay[2], and Matthias K. Gobbert[1]

[1] Dept. of Mathematics and Statistics, University of Maryland, Baltimore County
[2] Dept. of Information Systems, University of Maryland, Baltimore County
[3] Dept. of Physics, University of Maryland, Baltimore County
[4] Joint Center for Earth Systems Technology, University of Maryland, Baltimore County
[5] GESTAR, USRA and NASA GSFC
{barajasc, peiguo1, lipimuk1, hoban, jianwu, gangopad, gobbert}@umbc.edu, daeho.jin@nasa.gov

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. Clustering is a common machine learning technique used in atmospheric science to classify cloud types. Many parallelism techniques e.g., MPI, OpenMP and Spark, could achieve efficient and scalable clustering of large-scale satellite observation data. In order to understand their differences, this paper studies and compares three different approaches on parallel clustering of satellite observation data. Benchmarking experiments with K-means clustering are conducted with three parallelism techniques, namely OpenMP, OpenMP + MPI, and Spark, on a HPC cluster using up to 16 nodes.

Keywords: Parallel computing · High performance computing · MPI · OpenMP · Spark· K-means Clustering

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 [10, 18]. 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. However 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. This process consequently warms 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, 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" effect. 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 five years (2005-2009), and we employ K-Means clustering to identify the prominent cloud types.

K-means clustering is a widely applied unsupervised machine learning algorithm. When the input data is large, the computation speed of K-means clustering should be considered. In our study, we applied three different implementations of parallelized computation of K-means clustering: OpenMP, OpenMP + MPI, and Spark. We explain the parallelization and compare the clustering results and performance of the three implementations. The contributions of this paper are: 1) implementations of three different parallelization techniques on K-means clustering 2) using performance comparisons of these three different parallelized techniques.

2 Background

2.1 Cloud Joint Histograms

COT and CTP are recorded by a satellite from the snapshot of a cloud which we visualize with the 2-D joint histogram [13]. The International Satellite Cloud Climatology Project (ISCCP) cloud type is used in order to interpret the histogram [17]. With this categorization, it is easy to link the joint histogram data to real world clouds as shown in Figure 1.

It is natural that multiple cloud types occur in the same $1^\circ \times 1^\circ$ grid cell. Consequently individual joint histogram data (representing one time and one location) has 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.

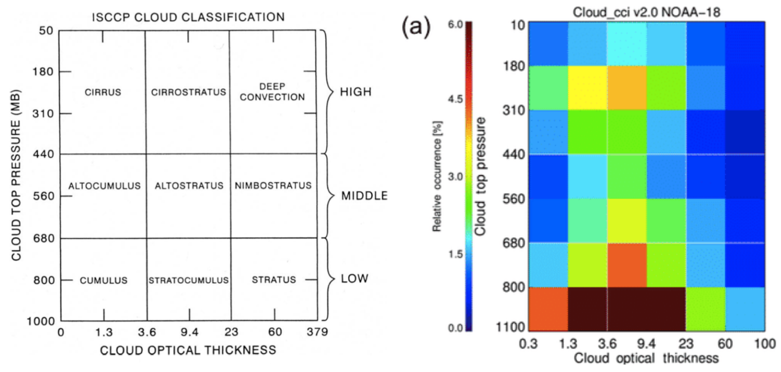

Fig. 1. Left: Cloud type definitions can be extrapolated using joint histograms where the joint-histogram is broken up into regions which are blocked according to cloud-type. Additional information on this technique can be seen in [17]. Right: The joint histogram of cloud top pressure and cloud top thickness suggesting high frequency of stratocumulus clouds.

2.2 K-means Clustering

In order to cluster the cloud types based on their properties (COT, CTP) as shown in Figure 2, we used K-means clustering. The general idea behind K-means clustering is grouping data according to distance where distance is a measure of similarity [9].

K-means is an unsupervised clustering algorithm. It starts with choosing k cluster centers (centroids) in the space representing the data objects. Next each data object is assigned to a cluster center with the closest Euclidean distance. After assigning all data to some centroid a new position for the k centroids are calculated. If the centroids move such that they have a smaller mean distance the new clusters are kept and the old centroids are discarded. Then the previous steps of assigning and calculating are repeated until the centroids' movement is negligible [14, 15].

The K-means algorithm is sensitive to the initialization of randomly selected cluster centers [9]. 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.

3 Implementation Details

We have three different approaches to K-means clustering in this section. Two were our own implementations and one was provided by Dr. Jin as a baseline to be improved and compared against. Our source code can be found on GitHub [6].

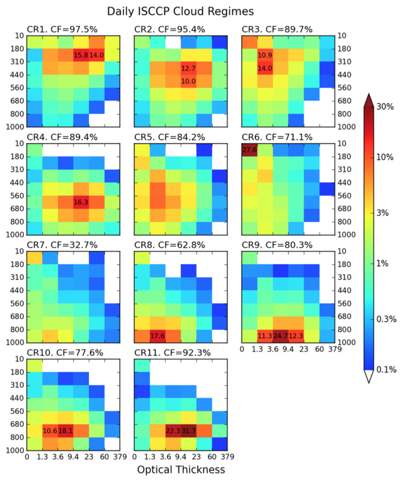

Fig. 2. The cloud regime (CR) centroids of daily ISCCP joint histograms. The cloud fraction (CF) of each regime, the sum of 42 bin values, is also provided. When bin values are larger than 10%, they are explicitly colored [13].

3.1 OpenMP based Implementation

Our initial baseline for improvement was code provided to us by Dr. Jin which uses Python for pre-processing and post-processing of data while leveraging OpenMP enabled FORTRAN for computationally heavy tasks such as the K-means clustering algorithm. The bindings were generated using `f2py`. We refer to this approach as the OpenMP approach.

The code takes in a binary data file that is a $n \times 42$ multi-dimensional array where the n dimension is the total number of histograms to be used for the K-means algorithm whereas 42 is the number of cloud fraction bins within each histogram. Concisely each row is one joint histogram. The binary data is produced using level 3 MODIS data that is provided in the HDF format. The binary format is more compact on disk and is loaded directly into an array using NumPy. Note that each joint histogram(s) is a data point in the K-means clustering algorithm and will be referred frequently as "record" or "records".

As is typical of OpenMP code the number of threads is set *a priori* with the environment variable `OMP_NUM_THREADS`. First Python calculates the $k = 10$ initial centroids for K-means clustering using the same idea as the `k-means++` initialization algorithm. This attempts to make the initial centroids sparse so that they can each encompass the largest amount of data with minimal, if any, overlap. The first iteration uses the initial centroids as a 0^{th} iteration. All data and the previous iteration's centroids are then passed to the first FORTRAN subroutine, `assign_and_get_new_sum`, which determines a new centroid and computes the Euclidean distance of each record from the new centroids. The newly generated centroids and respective distances are returned to Python from FORTRAN as two NumPy arrays. To prevent performance loss that comes with using Python, NumPy's array vectorization is used to compute the mean distances. A vectorized check is implemented with NumPy to determine if the mean distances of the new centroids are superior to the previous iteration's centroids. The centroid set with the best mean distances is kept for the next iteration. This process continues until either the maximum number of iterations is reached, 40, or the mean distance between the previous iteration's centroids and the newly computed centroids is smaller than the given threshold of 0.125 which was provided by Dr. Jin. Once a stopping criterion has been met the final centroids are written to disk in a binary format so that may be post-processed at a later time. A Python script then reads in these binary centroids to produce the several joint histograms seen in Figure 2.

3.2 OpenMP and MPI based Implementation

Our first approach uses Cython, Python, OpenMP, and MPI. The total number of records r_t are split as evenly as possible between the p MPI processes such that no process has more than one record compared to any other process. Whereas OpenMP is used in hot computational C loops for increased parallelism. We refer to this approach as OpenMP + MPI.

The load balancing scheme for MPI and OpenMP is discussed on a per node basis as follows. The environment variable `OMP_NUM_THREADS` is set *a priori* to run time. The Intel OpenMP environment variable `KMP_AFFINITY` is set to `scatter` so that threads are distributed as evenly as possible among the cores. Given our HPC testbed the cores per node $c = 16$ in conjunction with some number of processes per node p_n the number of threads per MPI process is computed by $t_p = c/p_n$. This balancing system allows for all node resources to be used, even if $p_n < c$.

Before any K-means calculations begin, each MPI process determines its own process rank and the total number of processes running. The processes use the total number of records and total number of processes to determine their local number of records r_l as $r_l = r_t/p$. 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 process reads in its respective records from the same binary data as mentioned in Section 3.1. This means that each process knows only of its own records and no data is duplicated across the processes.

First the initial centroids are calculated as mentioned in Section 3.1. All data and the previous iteration's centroids are passed to the Cython `def` function `assign_and_get_new_sum` , which calls the `cdef` functions `calculate_cl` and `calculate_outsum`. The deterministic behavior of K-means promises that the new cluster produced by `calculate_cl` is the same on every process. The Euclidean distance computation is where the parallelism plays a role. Figure 3 represents just one of the k many centroids where $p = 2$. Process 0 and Process 1 compute the Euclidean distance from their respective records to the centroid independently of each other. Then the mean distance for all records to the centroid would be computed using a `MPI_Allreduce` followed by a local division by r_t.

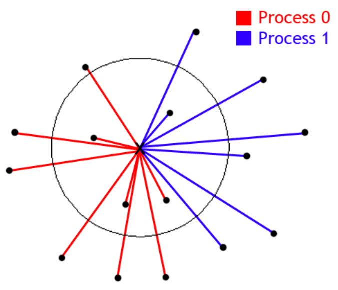

Fig. 3. The general idea for parallelization over a large data set with the repeated calculation. Each black dot is a record and the colored lines tell which process would be handling that Euclidean distance from the current center of the cluster.

OpenMP is implemented with a `pragma omp parallel for` around the record distance calculation loop. Thus the most expensive computation of the K-means algorithm is sped up by splitting r_t into r_l with MPI and multi-threading the record distance calculation with OpenMP.

In the code these distances and clusters are returned from Cython to Python as two NumPy arrays. In actuality the processes collectively compute a global mean distance for each cluster using a `MPI.allreduce` in Python. While the MPI command could have taken place inside the Cython code the idea is to keep the same data transaction style as the FORTRAN code. The MPI call happens in Python rather than Cython. All processes have the same newly calculated centroids, previous iteration's centroids, and respective mean distances to the centroids. So all processes make the same choice on which set of centroids have the better mean distances and discard the other. The stopping criterion and post-processing is the same as in Section 3.1.

3.3 Spark based Implementation

Our second approach is implemented in Python using Apache Spark's scalable machine learning library Spark MLib and the associated API. We utilized Spark 2.3.0 and the built-in K-means algorithm for the cloud regime [1, 2]. There are four steps in our applied Spark machine learning workflow: load our data, extract the features, train the model, and evaluate the results.

First we load our data into a Spark DataFrame which is organized as a distributed collection of data by name columns [4]. Upon the creation of the DataFrame it is apparent that our data contained 42 columns which are the bins of the joint histogram. We extracted the 42 features and assembled a features vector in preparation for the clustering. In the clustering process, we set $k = 10$. We changed set the Spark variable `max.iteration` to 40 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 concluded that 40 iterations are enough in our case. We executed the program many times and output the silhouette with squared Euclidean distance to make sure that our result was relatively stable [14].

The results of the clustering are dumped in a binary format and post-processed using the same Python script in Section 3.1.

4 Results

In this section three different aspects of the results are highlighted. Code validity is for testing whether parallelism is implemented correctly. Computation may proceed successfully but the application results could be incorrect. To check the validity of our two implementations we compare our results against the results that are produced by the provided implementation. Performance contains wall-clock times with various environment conditions as cataloged in their respective sections for each of the code implementations. Cross-comparison compares all implementations to one another in both qualitative and quantitative measures.

The experiments are conducted on the UMBC High Performance Computing Facility (HPCF) `hpcf.umbc.edu`. Each node used in our experiments has two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory. These nodes are connected by a high-speed quad-data rate (QDR) InfiniBand network.

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, rounding errors can occur, images can degrade in quality, and values that serial code correctly computes 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. Each of the implementations were run using the same initial parameters in order to mimic the run environment of the OpenMP approach. Additionally all the of the implementations were post-processed using the same Python script so that the images are comparable qualitatively and quantitatively.

First consider Figure 4. The OpenMP and OpenMP + MPI joint histograms are identical in their order, shape, and colorings. Since the algorithms in the OpenMP approach were recoded line by line in the OpenMP + MPI approach using Cython, it makes sense that the results should be identical. The only fundamental difference between the two coding schema was the major ordering of the data and record splitting via MPI. More importantly, the OpenMP approach and the OpenMP + MPI both used the same Python functions to calculate the initial centroids. The underlying numerical differences between each of the results is inevitable as there is no promise that the FORTRAN compiler and the C compiler will make the same sort of optimizations. Thus the FLOP round off error is most certainly different between each of the three implementations. However the accuracy of COT and CTP need only be accurate within 10^{-3} for the results to be consider good enough in the scope of the problem. The post-processing script only uses decimals on the order of 10^{-2}. Beyond the quantitative results produced, the qualitative results are seen as the more important use of the joint

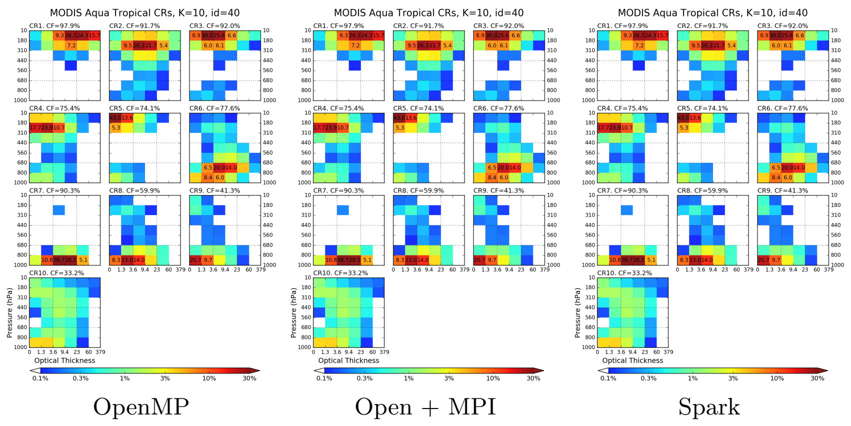

Fig. 4. Post-processed joint histogram results of the k-means final stable clusters for all three implementations. The images are qualitatively identical
histogram model as discussed in [13]. This means that the scale, color, shape, and ordering of the histograms play an integral role in determining the accuracy of the implementation compared to the original.

While the implementations are fundamentally different the underlying algorithm is still the K-means clustering algorithm with sparse initialization of the first set of centroids. Even though the Spark code uses open-source libraries rather than personally coded algorithms the qualitative results are identical to the OpenMP approach which was programmed from scratch. The numerical values between each of the post-processed results are functionally identical and as stated qualitatively identical as well.

The major difference is the approach of parallelism. Spark's parallelism uses a completely different methodology than the the typical operation of one compute node with OpenMP enabled code. Additionally Spark's data handling is vastly different than the OpenMP+MPI code, yet the results are the same. These differences are irrelevant because the application results computed by all approaches are the within acceptable margins. Therefore both of the alternative implementations can be regarded as accurate parallelized representations of the OpenMP approach, as they show no signs of result degradation.

4.2 Performance

Table 1. OpenMP wall clock results with total number of threads used in HH:MM:SS.

Threads	1	2	4	8	16
Wall clock	00:14:59	00:07:10	00:03:47	00:02:58	00:02:38

OpenMP Table 1 presents the recorded times for the varying number of OpenMP threads in the OpenMP approach. Clearly as we use more threads the time improves slightly but there appears to be bottleneck. Even though we're using 16 threads (see the final column) the time is not 16 times faster. We can use the best speed possible from these results as a baseline to compare other results to. There is a clear improvement in the timings as we increase the number of threads used. This indicated that the OpenMP parallelism is having a positive on the performance. However as the number of threads double the timing is not halved. This then implies that the implementation has a bottleneck beyond the OpenMP components. Thus the 1-node, 1-process-per-node, 16-thread timing in Table 1 shall be the timing that all other timings are compared too.

Table 2. OpenMP + MPI wall clock results with Nodes and Processes Per Node in HH:MM:SS.

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

OpenMP + MPI The MPI results in Table 2 show that as the number of processes per node increase the performance decreases. Consider the 8 node column of the table. As the number of processes per node increase the times gradually worsen at an increasing rate until the timing from eight processes per node to sixteen processes per node doubles. This same behavior is consistent for all node columns. Thus we can say that there is an optimal load balancing issue that must be addressed. The most optimal way to take advantage of all cores on a node in this case is to use the minimal amount of MPI processes and maximum number of threads per process. This cuts down on the communication required between processes and allows for a collection of nodes to be used mainly for threads. These threads are lightweight and require no intercommunication of data to function. For all rows as the number of nodes used increases the performance also increases which is the expected strong scalability outcome.

The data set fits comfortably within the total memory capacity available. Meaning that there is less memory contention and one process per node performs more optimally than expected. On dual socket nodes the minimal number of processes required for optimal performance of memory bound code has been concluded to be two processes per node. This allows one process and its respective threads to be placed on their own processor [5,16]. Once larger data sets approach the node memory limit of $\approx$ 62GB MPI should start to demonstrate a clear performance improvement as the communication time becomes a small player in the overall timing results.

Spark Table 3 is the run time table of our Spark implementation. In Table 3 by increasing nodes from 1 to 4 our spark program wallclock time decreases significantly from 9 minutes to less than 3 minutes. However when scaling up from 4 nodes to 8 nodes, the timings do not change significantly, despite the continued decrease from just under 3 minutes to around 2 minutes. The reason is that during most of the run time Spark is working on loading data into the Spark DataFrame. The actual calculating time of the centroids in Spark with 4 nodes is around 7 seconds, and with 8 nodes, it is only 4 seconds. We conclude that performance did not improve much by increasing the number of nodes. This is because the size the data set (3 GB) is not big enough to make a significant difference and there's an overhead when loading the data into the DataFrame.

Table 3. Spark wall clock results with total number of nodes used in HH:MM:SS.

Nodes	1	2	4	8
Wall clock	00:09:03	00:06:16	00:02:51	00:02:09

4.3 Cross Comparison

Implementation Comparisons The first step in implementing MPI was to convert the FORTRAN code into C code to maintain high performance and ease the MPI parallelization. MPI is better equipped to handle C's native ordering (row major). In constrast CPython API is rather terse and unwieldy. Thus 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 and properly optimize for 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 using the CPython API. 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`. A `Memoryview` provides a closer interface to the heap than NumPy arrays. This allows 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 [7].

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

The Spark code is so fundamentally different from the other two implementations, a comparison would just be reiterating the implementation described in Section 3.3.

Wall Timings All but the bottom left three timings in Table 2 are better than the best timing in Table 1. Consider the best timing from the OpenMP approach. This OpenMP timing is 2× 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 2 use the same amount of resources as the best timing in Table 1. This indicates that the benefits of Cython, rather than MPI, are to thank for the jump in performance. By enabling MPI and using 8 nodes we get a mere 8 second run time. This is 18× faster than the OpenMP performance time and approximately 7.5× faster than the single node OpenMP + MPI code.

Consider the timings in Table 3 compared to the timings in Table 1. Observe the single node performance of Spark in this case the OpenMP approach is 3.4× faster than the Spark approach. It is not until Spark uses 8 full nodes before it is able to compete with the single node performance of OpenMP. Even then it is only 1.2× faster.

The main reason for the under performance of Spark is that the data set is very small and the communication time and initial overhead of Spark far outweigh the actual computation needed to solve the problem. Similarly as we increase the number of MPI processes it is clear that the communication time is a large price to pay despite very minimal amounts of communication. The problem size is small enough that communication still plays a big role in performance timings and OpenMP + MPI has the least amount of overheard when using only one process per node is used which is why this row of timings dwarf all other results.

5 Related Work

The reasons for running benchmarks vary considerably. One may wish to test the capability of new hardware as seen in [5]. The idea of transcoding a problem into multiple languages and use different underbelly computation code is commonplace in the sphere of development. Even on the exact hardware we utilized for our implementations, there have been several transcoding performance studies. For example: the performance of numerical solvers in Julia, R, and Matlab which is found in [16]. In [12], K-means clustering is used as a comparison of other machine learning techniques on Hadoop using their benchmarking suite HiBench. OpenMP applications and K-means clustering are tested in [8]. Another benchmarking work on parallel computing among different parallel programming approaches includes Hadoop, Spark, and Hive database. This proved that different programming methods could cause more than 100 times difference in running speed [11]. However there are no specific combinations that reflect our language choice and application problem.

6 Conclusions

Both parallel implementations managed to correctly compute the same clusters as the original code. Only OpenMP + MPI implementation managed to outperform the original code with the same amount of resources at its disposal. Only OpenMP + MPI 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 3 GB. These results are not indicative of what would happen given 20+ GB of data. In our Spark application, we basically use only its default level of parallelism. By configuring higher parallel level to load data, or upload data to HDFS might improve the speed of our Spark program. Moreover, Spark application utilizes Python, and the programming in Python itself is slower than programming in FORTRAN and C. So we cannot conclude that Spark is an inferior implementation in this current stage. We only can conclude that it might need more tuning work to make it optimized and competitive.

When MPI scaled is scaled to multiple nodes always the performance always proved. One point is that when MPI is run with multiple nodes using one process per node the total number of threads increase proportionally. However when the number of processes increased beyond one process per node, performance decreased indicating that the data set is also too small for MPI communication. Ordinarily this would be a smaller price to pay for increased parallelism but was not in our case.

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.

Acknowledgment

This work is supported by NSF grant OAC–1730250 and NASA grant 80NSSC17K0366. The hardware used, the UMBC HPCF, is supported by NSF grants (CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (DMS–0821311), with additional substantial support from UMBC.

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. Kritesh Arora, Carlos Barajas, and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the Stampede2 cluster and comparison of networks. Technical Report HPCF–2018–10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018.
6. Carlos Barajas, Pei Guo, Lipi Mukherjee, and Jin Daeho. `https://github.com/big-data-lab-umbc/cybertraining/tree/master/year-1-projects/team-2`. Source Code.
7. 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.
8. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.
9. J. Fauld. Unsupervised learning: Association rule learning and clustering, March 2018.
10. Steve Graham. `https://earthobservatory.nasa.gov/Features/Clouds/?src=share`, March 1999.
11. Pei Guo, Jianwu Wang, and Zhiyuan Chen. A comparison of big data application programming approaches: A travel companion case study. *2017 IEEE International Conference on Big Data (Big Data)*, pages 2869–2878, 2017.
12. S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51, March 2010.
13. Daeho Jin, Lazaros Oreopoulos, and Dongmin Lee. Regime-based evaluation of cloudiness in cmip5 models. *Climate Dynamics*, 48(1):89–112, Jan 2017.
14. 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.
15. Polytechnic University of Milan. A Tutorial on Clustering Algorithms k-means clustering. `https://home.deib.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html`. Accessed: 2018-06-15.
16. Sai K. Popuri and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, R, and Julia on Maya. Technical Report HPCF–2017–3, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2017.
17. William B. Rossow and Robert A. Schiffer. Advances in understanding clouds from isccp. *Bulletin of the American Meteorological Society*, 80(11):2261–2288, 1999.
18. J. M. Wallace. *Atmospheric science: An introductory survey.* Academic Press, 1977.