This work was written as part of one of the author's official duties as an Employee of the United States Government and is therefore a work of the United States Government. In accordance with 17 U.S.C. 105, no copyright protection is available for such works under U.S. Law. Access to this work was provided by the University of Maryland, Baltimore County (UMBC) ScholarWorks@UMBC digital repository on the Maryland Shared Open Access (MD-SOAR) platform.

Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it’s important to you. Thank you.
More information about this series at http://www.springer.com/series/7409
BenchCouncil: Benchmarking and Promoting Innovative Techniques

The past decade witnessed many innovative techniques as they evolved our economy and society. The innovative techniques in Datacenter, AI, dedicated processors, and IoT are especially changing our world. For example, new wave AI techniques have changed every part of our daily life, such as facial recognition, self-driving cars, robots, financial services, personal assistants, etc. In recent years, this trend has accelerated, and there is an increasing interest in generating values from these emerging technologies. However, benchmarking and promoting innovative techniques are the very first steps in making the innovation successful, which is the mission of the International Open Benchmark Council (BenchCouncil).

BenchCouncil is a non-profit research institute, which aims to promote multi-disciplinary benchmarking research and practice and foster collaboration and interaction between industry and academia. The major goals of the BenchCouncil are as follows:

– Establish and maintain a repository of benchmark specifications for quantitative system and algorithm evaluation and analysis
– Review, shepherd, and release open-source benchmark implementations
– Organize conferences, workshops, and teleconferences fostering the transfer of knowledge between industry and academia in the areas of benchmarking
– Build a testbed to verify and promote the innovative techniques
– Organize challenges and competition using released benchmarks
– Publish the performance numbers to evaluate competitive techniques using the benchmarks

In order to fulfill its mission, the BenchCouncil organized the First International Symposium on Benchmarking, Measuring, and Optimizing (Bench 2018) (http://prof.ict.ac.cn/Bench18/), which was co-located with IEEE BigData 2018 (http://cci.drexel.edu/bigdata/bigdata2018/)—an IEEE International Conference on Big Data. The symposium solicits papers that address hot topical issues in benchmarking, measuring, and optimizing systems. This book includes 15 papers from the Bench 2018 conference, and 5 invited papers which report on the state-of-the-art AI benchmarking research and engineering efforts of the BenchCouncil. In addition, 6 Benchmark proposals were presented at the conference, but were not included in this book.
The call for papers for Bench 2018 attracted a number of high-quality submissions. During a rigorous review process, each paper was reviewed by at least three experts. In addition, we invited three keynote speakers:

- Prof. Geoffrey Fox (Indiana University): “Big Data Benchmarking: Applications and Systems”
- Prof. Vijay Janapa Reddi (Harvard University): “MLPerf: the Vision Behind an ML Benchmark Suite for Measuring the Performance of ML Software Frameworks, ML Hardware Accelerators, and ML Cloud and Edge Platforms”

During the conference, we had 6 benchmark proposals covering the topics in Big Data, AI, Datacenter, Scalable Transaction, and Database. In order to discuss the implementation of benchmark and BenchCouncil workgroup specification, we also held the BenchCouncil Open Meeting to format the working groups.

We are very grateful for the efforts of all authors in relation to writing, revising, and presenting their papers at Bench 2018. We appreciate the indispensable support of the Bench 2018 Program Committee and thank them for their efforts and contributions in maintaining the high standards of the Bench 2018 Symposium.

June 2019

Chen Zheng
Jianfeng Zhan
Organization

Program Chair

Jianfeng Zhan
ICT, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, China

Program Committee

Lizy K. John
The University of Texas at Austin, USA

Vijay Janapa Reddi
University of Texas at Austin and Google, USA

Wenguang Chen
Tsinghua University, China

Woongki Baek
UNIST, South Korea

Yunji Chen
ICT, Chinese Academy of Sciences, China

Piotr Luszczek
University of Tennessee, USA

Rui Hou
Institute of Information Engineering, Chinese Academy of Sciences, China

Yueguo Chen
Renmin University of China, China

H. Peter Hofstee
IBM, USA

Cheqing Jin
East China Normal University, China

Trevor E. Carlson
National University of Singapore (NUS), Singapore

Jun-zhao Du
Xidian University, China

Hyogi Sim
Oak Ridge National Laboratory, USA

Weining Qian
East China Normal University, China

Zheng Cao
Alibaba, China

Matthias Nicola
IBM, Germany

Zhihui Du
Tsinghua University, China

Chunjie Luo
ICT, Chinese Academy of Sciences, China

Bin Ren
College of William and Mary, USA

Hua Chen
ICT, Chinese Academy of Sciences, China

Jiaquan Gao
Nanjing Normal University, China

Jiannan Ouyang
Facebook, USA

Xueming Si
Fudan University, China

Yanjun Wu
Institute of Software Chinese Academy of Sciences, China

Lucas Mello Schnorr
Federal University of Rio Grande do Sul (UFRGS), Brazil

Yunquan Zhang
ICT, Chinese Academy of Sciences, China

Shengzhong Feng
Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China

Zhen Jia
Princeton University, USA

Zhibin Yu
Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
<table>
<thead>
<tr>
<th>Name</th>
<th>Institution and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dapeng Wang</td>
<td>Institute of Software Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Bo Wu</td>
<td>Colorado School of Mines, USA</td>
</tr>
<tr>
<td>Shaoliang Peng</td>
<td>National University of Defense Technology, China</td>
</tr>
<tr>
<td>Yong Qi</td>
<td>Xi’an Jiaotong University, China</td>
</tr>
<tr>
<td>Gwangsun Kim</td>
<td>Arm Inc., Austin, USA</td>
</tr>
<tr>
<td>Rui Ren</td>
<td>ICT, Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Zujie Ren</td>
<td>Hangzhou Dianzi University, China</td>
</tr>
<tr>
<td>Xiaoyi Lu</td>
<td>The Ohio State University, USA</td>
</tr>
<tr>
<td>Lei Wang</td>
<td>ICT, Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Jianhui Li</td>
<td>Computer Network Information Center, Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Jianwu Wang</td>
<td>University of Maryland, USA</td>
</tr>
<tr>
<td>Kenli Li</td>
<td>Hunan University, China</td>
</tr>
<tr>
<td>Suzanne Rivoire</td>
<td>Sonoma State University, USA</td>
</tr>
<tr>
<td>Rui Han</td>
<td>ICT, Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Jungang Xu</td>
<td>Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Xu Liu</td>
<td>College of William and Mary, USA</td>
</tr>
<tr>
<td>Tong Wu</td>
<td>National Institute of Metrology, China</td>
</tr>
<tr>
<td>Li Zha</td>
<td>University of the Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Lei Liu</td>
<td>ICT, Chinese Academy of Sciences, China</td>
</tr>
<tr>
<td>Chen Zheng</td>
<td>ICT, Chinese Academy of Sciences, China</td>
</tr>
</tbody>
</table>
Contents

AI Benchmarking

AIBench: Towards Scalable and Comprehensive Datacenter
AI Benchmarking ........................................ 3
  Wanling Gao, Chunjie Luo, Lei Wang, Xingwang Xiong, Jianan Chen,
  Tianshu Hao, Zihan Jiang, Fanda Fan, Mengjia Du, Yunyou Huang,
  Fan Zhang, Xu Wen, Chen Zheng, Xiwen He, Jiahui Dai, Hainan Ye,
  Zheng Cao, Zhen Jia, Kent Zhan, Haoning Tang, Daoyi Zheng,
  Biwei Xie, Wei Li, Xiaoyu Wang, and Jianfeng Zhan

HPC AI500: A Benchmark Suite for HPC AI Systems ................. 10
  Zihan Jiang, Wanling Gao, Lei Wang, Xingwang Xiong, Yuchen Zhang,
  Xu Wen, Chunjie Luo, Hainan Ye, Xiaoyi Lu, Yunquan Zhang,
  Shengzhong Feng, Kenli Li, Weijia Xu, and Jianfeng Zhan

Edge AIBench: Towards Comprehensive End-to-End Edge
Computing Benchmarking .................................. 23
  Tianshu Hao, Yunyou Huang, Xu Wen, Wanling Gao, Fan Zhang,
  Chen Zheng, Lei Wang, Hainan Ye, Kai Hwang, Zujie Ren,
  and Jianfeng Zhan

AIoT Bench: Towards Comprehensive Benchmarking Mobile
and Embedded Device Intelligence .......................... 31
  Chunjie Luo, Fan Zhang, Cheng Huang, Xingwang Xiong, Jianan Chen,
  Lei Wang, Wanling Gao, Hainan Ye, Tong Wu, Runsong Zhou,
  and Jianfeng Zhan

A Survey on Deep Learning Benchmarks: Do We Still Need New Ones? .... 36
  Qin Zhang, Li Zha, Jian Lin, Dandan Tu, Mingzhe Li, Fan Liang,
  Ren Wu, and Xiaoyi Lu

Cloud

Benchmarking VM Startup Time in the Cloud .......................... 53
  Samiha Islam Abrita, Moumita Sarker, Faheem Abrar,
  and Muhammad Abdullah Adnan

An Open Source Cloud-Based NoSQL and NewSQL Database
Benchmarking Platform for IoT Data ........................... 65
  Arjun Pandya, Chaitanya Kulkarni, Kunal Mali, and Jianwu Wang
Scalability Evaluation of Big Data Processing Services in Clouds

Xin Zhou, Congfeng Jiang, Yeliang Qiu, Tiantian Fan, Yumei Wang, Liangbin Zhang, Jian Wan, and Weisong Shi

PAIE: A Personal Activity Intelligence Estimator in the Cloud

Yingjie Shi, Fang Du, Yanyan Zhang, Zhi Li, and Tao Zhang

DCMIX: Generating Mixed Workloads for the Cloud Data Center

Xingwang Xiong, Lei Wang, Wanling Gao, Rui Ren, Ke Liu, Chen Zheng, Yu Wen, and Yi Liang

Machine-Learning Based Spark and Hadoop Workload Classification Using Container Performance Patterns

Mikhail Genkin, Frank Dehne, Pablo Navarro, and Siyu Zhou

Testing Raft-Replicated Database Systems

Guohao Ding, Weining Qian, Peng Cai, Tianze Pang, and Qiong Zhao

Big Data

Benchmarking for Transaction Processing Database Systems in Big Data Era

Chunxi Zhang, Yuming Li, Rong Zhang, Weining Qian, and Aoying Zhou

UMDISW: A Universal Multi-Domain Intelligent Scientific Workflow Framework for the Whole Life Cycle of Scientific Data

Qi Sun, Yue Liu, Wenjie Tian, Yike Guo, and Bocheng Li

MiDBench: Multimodel Industrial Big Data Benchmark

Yijian Cheng, Mengqian Cheng, Hao Ge, Yuhe Guo, Yuanzhe Hao, Xiaoguang Sun, Xiongpai Qin, Wei Lu, Yueguo Chen, and Xiaoyong Du

Modelling and Prediction

Power Characterization of Memory Intensive Applications: Analysis and Implications

Yeliang Qiu, Congfeng Jiang, Tiantian Fan, Yumei Wang, Liangbin Zhang, Jian Wan, and Weisong Shi

Multi-USVs Coordinated Detection in Marine Environment with Deep Reinforcement Learning

Ruiying Li, Rui Wang, Xiaohui Hu, Kai Li, and Haichang Li

EC-Bench: Benchmarking Onload and Offload Erasure Coders on Modern Hardware Architectures

Haiyang Shi, Xiaoyi Lu, and Dhabaleswar K. Panda
Algorithm and Implementations

Benchmarking SpMV Methods on Many-Core Platforms ........................................ 233
  Biwei Xie, Zhen Jia, and Yungang Bao

Benchmarking Parallel K-Means Cloud Type Clustering
from Satellite Data ........................................................................................................ 248
  Carlos Barajas, Pei Guo, Lipi Mukherjee, Susan Hoban, Jianwu Wang,
  Daeho Jin, Aryya Gangopadhyay, and Matthias K. Gobbert

Author Index .................................................................................................................. 261
Benchmarking Parallel K-Means Cloud Type Clustering from Satellite Data

Carlos Barajas¹(✉), Pei Guo², Lipi Mukherjee³,⁴, Susan Hoban⁴, Jianwu Wang², Daeho Jin⁵, Aryya Gangopadhyay², and Matthias K. Gobbert¹

¹ Department of Mathematics and Statistics, University of Maryland, Baltimore County, USA
{barajasc,gobbert}@umbc.edu
² Department of Information Systems, University of Maryland, Baltimore County, USA
{peiguo1,jianwu,gangopad}@umbc.edu
³ Department of Physics, University of Maryland, Baltimore County, USA
lipimuk1@umbc.edu
⁴ Joint Center for Earth Systems Technology, University of Maryland, Baltimore County, USA
hoban@umbc.edu
⁵ GESTAR, USRA and NASA GSFC, Columbia, USA
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 use MODIS data for five years (2005–2009) and 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 speed of k-means clustering should be considered. In our study, we apply three different implementations of parallelized computation of k-means clustering: OpenMP, OpenMP+MPI, and Spark. 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 Fig. 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.

### 2.2 K-means Clustering

In order to cluster the cloud types based on their properties (COT, CTP) as shown in Fig. 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].
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 Fig. 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 \textit{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 Sect. 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 Sect. 3.1. All data and the previous iteration’s centroids are passed to the Cython \texttt{def} function \texttt{assign_and_get_new_sum}, which calls the \texttt{cdef} functions \texttt{calculate_cl} and \texttt{calculate_outsum}. The deterministic behavior of k-means promises that the new cluster produced by \texttt{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 \texttt{MPI_Allreduce} followed by a local division
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. (Color figure online)

by $r_t$. 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 Sect. 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 Sect. 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 Fig. 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
Fig. 4. Post-processed joint histogram results of the k-means final stable clusters for all three implementations. The images are qualitatively identical.

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 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 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

OpenMP. Table 1 presents the recorded times for 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 1.** OpenMP wall clock results with total number of threads used in HH:MM:SS.

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall clock</td>
<td>00:14:59</td>
<td>00:07:10</td>
<td>00:03:47</td>
<td>00:02:58</td>
<td>00:02:38</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Nodes</th>
<th>1 ppn</th>
<th>2 ppn</th>
<th>4 ppn</th>
<th>8 ppn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00:01:01</td>
<td>00:00:34</td>
<td>00:00:17</td>
<td>00:00:08</td>
</tr>
<tr>
<td>2 ppn</td>
<td>00:01:23</td>
<td>00:00:41</td>
<td>00:00:20</td>
<td>00:00:11</td>
</tr>
<tr>
<td>4 ppn</td>
<td>00:01:50</td>
<td>00:00:54</td>
<td>00:00:28</td>
<td>00:00:16</td>
</tr>
<tr>
<td>8 ppn</td>
<td>00:02:42</td>
<td>00:01:22</td>
<td>00:00:45</td>
<td>00:00:29</td>
</tr>
<tr>
<td>16 ppn</td>
<td>00:04:47</td>
<td>00:02:32</td>
<td>00:01:29</td>
<td>00:01:07</td>
</tr>
</tbody>
</table>

**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 ≈62 GB, 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 wall clock time decreases significantly from 9 min to less than 3 min. However when scaling up from 4 nodes to 8 nodes, the timings do not change significantly, despite the continued decrease from just under 3 min to around 2 min. 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 s, and with 8 nodes, it is only 4 s. 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>
<thead>
<tr>
<th>Nodes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall clock</td>
<td>00:09:03</td>
<td>00:06:16</td>
<td>00:02:51</td>
<td>00:02:09</td>
</tr>
</tbody>
</table>

### 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 contrast 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 Sect. 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 $2 \times$ 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 8s run time. This is $18 \times$ faster than the OpenMP performance time and approximately $7.5 \times$ 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 $3.4 \times$ 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 $1.2 \times$ 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 overhead 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 with number OAC–1730250 and NASA grant 80NSSC17K0366. The hardware used the UMBC High Performance Computing Facility, which 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

Author Index

Abrar, Faheem 53
Abrita, Samiha Islam 53
Adnan, Muhammad Abdullah 53
Bao, Yungang 233
Barajas, Carlos 248
Cai, Peng 131
Cao, Zheng 3
Chen, Jianan 3, 31
Chen, Yueguo 172
Cheng, Mengqian 172
Cheng, Yijian 172
Dai, Jiahui 3
Dehne, Frank 118
Ding, Guohao 131
Du, Fang 91
Du, Mengjia 3
Du, Xiaoyong 172
Fan, Fanda 3
Fan, Tiantian 78, 189
Feng, Shengzhong 10
Gangopadhyay, Aryya 248
Gao, Wanling 3, 10, 23, 31, 105
Ge, Hao 172
Genkin, Mikhail 118
Gobbert, Matthias K. 248
Guo, Pei 248
Guo, Yike 159
Guo, Yuhe 172
Hao, Tianshu 3, 23
Hao, Yuanzhe 172
He, Xiwen 3
Hoban, Susan 248
Hu, Xiaohui 202
Huang, Cheng 31
Huang, Yunyou 3, 23
Hwang, Kai 23
Jia, Zhen 3, 233
Jiang, Congfeng 78, 189
Jiang, Zihan 3, 10
Jin, Dachao 248
Kulkarni, Chaitanya 65
Li, Bocheng 159
Li, Haichang 202
Li, Kai 202
Li, Kenli 10
Li, Mingzhe 36
Li, Ruiying 202
Li, Wei 3
Li, Yuming 147
Li, Zhi 91
Liang, Fan 36
Liang, Yi 105
Lin, Jian 36
Liu, Ke 105
Liu, Yue 159
Lu, Wei 172
Lu, Xiaoyi 10, 36, 215
Luo, Chunjie 3, 10, 31
Mali, Kunal 65
Mukherjee, Lipi 248
Navarro, Pablo 118
Panda, Dhabaleswar K. 215
Pandya, Arjun 65
Pang, Tianze 131
Qian, Weining 131, 147
Qin, Xiongpa 172
Qiu, Yeliang 78, 189
Ren, Rui 105
Ren, Zujie 23
Sarker, Moumita 53
Shi, Haiyang 215
Shi, Weisong 78, 189
Shi, Yingjie 91
Sun, Qi 159
Sun, Xiaoguang 172
Tang, Haoning 3
Tian, Wenjie 159
Tu, Dandan 36

Wan, Jian 78, 189
Wang, Jianwu 65, 248
Wang, Lei 3, 10, 23, 31, 105
Wang, Rui 202
Wang, Xiaoyu 3
Wang, Yumei 78, 189
Wen, Xu 3, 10, 23
Wen, Yu 105
Wu, Ren 36
Wu, Tong 31

Xie, Biwei 3, 233
Xiong, Xingwang 3, 10, 31, 105
Xu, Weijia 10
Ye, Hainan 3, 10, 23, 31

Zha, Li 36
Zhan, Jianfeng 3, 10, 23, 31
Zhan, Kent 3
Zhang, Chunxi 147
Zhang, Fan 3, 23, 31
Zhang, Liangbin 78, 189
Zhang, Qin 36
Zhang, Rong 147
Zhang, Tao 91
Zhang, Yanyan 91
Zhang, Yuchen 10
Zhang, Yunquan 10
Zhou, Aoying 131
Zheng, Chen 3, 23, 105
Zheng, Daoyi 3
Zhou, Aoying 147
Zhou, Runsong 31
Zhou, Siyu 118
Zhou, Xin 78