15618: Spring 2023 - Final Project Report

Wei-Lun Chiu (weilunc) (weilunc@andrew.cmu.edu)
Jhao-Ting Chen (jhaoting) (jhaoting@andrew.cmu.edu)

May 4, 2023

1 Title

Supercomputer or Cloud Computing: Large-scale, High-computation Data Clustering Tasks with PyOMP on single machine and PySpark with AWS EC2 instances.

2 URL

https://jtchen0528.github.io/cmu-15618-project/

3 Summary

We compared the performance and costs of executing data clustering algorithms on a multi-core machine with PyOMP and on a set of workers with comparable resource constraints of AWS instances with PySpark. We analyzed the serial, parallelized, and distributed implementation of two clustering algorithms: K-means and DBSCAN, evaluated their trade-off between performance, financial cost, and different algorithmic strategies. Our implementation showed 5.41x speedup with a 6-core CPU, out-performed NumPy-optimized version by 1.44x, and 10.41x speedup with 12-node cloud cluster.

For DBSCAN, we explores ways to improve the runtime of the grid-based DBSCAN algorithm through parallelization techniques. We evaluated Python-based frameworks and found that developing a complex parallel framework on Python is not feasible. With only a 1.7x speedup achieved with six threads in Python, we turned our attention to C++. Implementing parallelism techniques in C++ resulted in a significant 9.86x speedup with 8 threads, which involved using C++ Concurrency API, a lock-free Union-find data structure, work-stealing mechanism with fine-grained lock, and SIMD/AVX instructions for critical floating-point arithmetic functions.

4 Background

4.1 Data Clustering

The dataset size for machine learning tasks grows larger by days, and self-supervised tasks become more popular than before. The performance and effectiveness for data clustering tasks such as dataset reduction [1] or pseudo-labeling [2] hence raise more research awareness. For our project, we choose to analyze the k-means data clustering [3], DBSCAN(Density-Based Spatial Clustering of Applications with Noise) [4], and different parallel approaches (grid-based, partition-based, and task-based). We will implement serially and in parallel, on multi-core machines and multi-node clusters.

4.1.1 K-means Data Clustering

K-means is a data clustering algorithm used for machine learning tasks including image segmentation, anomaly detection and data preprocessing. The algorithm works by first assigning a set of centroids representing a set of initial clusters, iteratively minimizing the Euclidean distance between data points and the closest centroids, then updating the centroids of each cluster until convergences. Since the cost function is a non-increasing function of the iterations, and there is a finite number

of data points and clusters, the algorithm is guaranteed to converge to a local minimum of the cost function. The convergent outcome, iterative computation, straight-forward algorithm makes it best for us to compare its performance and correctness of our serial, PyOMP parallelized, and PySpark parallelized program.

4.1.2 DBSCAN Data Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) was proposed by Ester et al [4]. The algorithm is a density-based clustering algorithm that can discover clusters of arbitrary shape in spatial data and is robust to noise and outliers. There are three advantages that make DBSCAN stands out. First, DBSCAN automatically detect the number of clustering. It does not require the predefined number of clusters to be and can find the appropriate number of clusters based on the density and separation of the data. Second, DBSCAN is noise-tolerant. It can effectively identify outliers as noise during clustering expansion. Third, DBSCAN is effective in handling datasets that have non-uniform density levels, where the clusters may not be clearly distinguishable or well-separated from each other.

Despite its effectiveness in handling datasets with varying density levels, DBSCAN has limitations when it comes to high-dimensional data due to two main reasons. First, as the number of dimensions increases, the computational complexity of the algorithm grows, leading to a higher computational cost. Chen et al [5] prove that the time complexity is $O(n^{2(1-1/(d+2)}polylog(n)$ for d dimension. The second reason why DBSCAN is not well-suited for high-dimensional data is that the amount of data required to produce accurate and meaningful results grows exponentially with the dimensionality of the dataset, which is commonly referred to as the curse of dimensionality [6]. In contrast, K-means is often faster and more scalable than DBSCAN, particularly when dealing with large high-dimensional datasets.

4.2 Cloud Computing

Cloud computing provides a cost-efficient alternative to maintaining expensive high-performance computing infrastructure. With on-demand computing resources, scalability, and pay-as-you-go services, one may access computing power as needed without the need for investing in specialized hardware. Cloud computing is especially beneficial for those who don't have access to resources like PSC or multi-core machines at home. By executing large-scale, high-computing tasks on several cloud service instances in parallel with distributed technologies, organizations can achieve comparable performance and accurate results while also significantly reducing costs.

4.3 PyOMP

PyOMP [7] caught our eyes when we're researching through our project. PyOMP, released by *Intel Corp.*, provides an easy-to-use API that allows developers to parallelize their Python code with minimal modifications, by adding OpenMP directives to their code. Instead of embedded parallelism inside Numpy [8], which has overhead when creating and destroying threads, PyOMP library was implemented in Numba [9]. PyOMP then offers users an OpenMP-similar syntax for accessibility. Programs written in C with OpenMP performs only 2.8% faster than programs in PyOMP.

4.3.1 Numba

Numba is just-in-time(JIT) compiler that translates Python into optimized machine code at runtime, just before execution, providing significant speedups for computationally intensive tasks. Numba uses LLVM as the backend for it JIT compiler, capable of many optimization techniques including vectorization, automatic parallelization, and loop unrolling.

4.4 PySpark

PySpark [10] is a Python API for Apache Spark [11], a powerful open-source distributed computing system used for big data processing and analytics. Apache Spark is created to address the limitation of Hadoop MapReduce framework. Spark is designed to work with large-scale data processing tasks that require high-speed data processing and distributed computing capabilities. PySpark provides

even more easy-to-use interface for users to handle data analytic tasks in Python. For our project, we aim to implement our program in PySpark, so that data can be clustered in parallel on multiple cloud instances.

4.5 Python Multiprocessing Module

The multiprocessing module is the official parallel processing module in Python. It is part of the standard library and is included with all versions of Python since version 2.6. It create multi processes instead of threads to workaround the intervention of Global Interpreter Lock (GIL). The GIL is a process-level lock that is used to ensure the integrity and thread-safety of Python. The multiprocessing module supports variant parallel models such as shared address models, message-passing models, master-worker models, and common synchronization primitives, making it a versatile tool for implementing parallel algorithms in Python.

4.6 C++ Concurrency API

The C++ Concurrency API was first introduced with the C++11 standard, it provide C++ developers a means for fine-grained control over parallelism and resource management at the thread level. The API is highly flexible and allows for the development of customized parallel frameworks. Additionally, the std::async function enables the compiler to handle thread and resource management. Also, the API's portability and efficiency make it a popular choice, although it requires more attention to detail compared to OpenMP and OpenMPI.

5 Approaches

5.1 K-means Serial Implementation

The K-means algorithm includes three parts: (i) centroids initialization, (ii) clustering data points, and (iii) updating centroids. The centroids of clusters are first initialized in (i), then iterate (ii) and (iii) until the centroids converge to means of the clustered data points.

5.1.1 Centroids initialization

The algorithm starts by assigning k data points to a set of centroids as initial clusters. There are several ways of initializing centroids, including (i) random and (ii) k-means++ [12]. K-means++ initialization partitions a set of data points into k clusters, tends to produce better clustering than randomly selected centroids. The algorithm is shown in Algorithm 1. Since the centroids are initialized only once, we separate our analysis between initialization and latter parts.

Algorithm 1 k-means++(*data_points*, *n_clusters*) (Centroids Initialization)

Assign the 1_{st} centroid randomly
for *cluster_idx* = 1 to *n_clusters* **do**
  Calculate the Euclidean distances *dists* of every data point to every assigned centroids
  Normalize all *dists*
  Choose the next centroid from data points with probability proportional to *dists*
end for

5.1.2 Clustering Data

K-means clusters data by finding the nearest centroid of each data points. Algorithm 2 demonstrates the implementation. The algorithm iterates through all data points to calculate distance between each centroid, then write the index of the closest centroid for the data point. For each iteration, the data points are assigned to the clusters of that iteration. The time complexity of 1 iteration of clustering data stage is $O(P * K * S)$ where K is the number of clusters, P is the amount of data points and S is the size of each data point.

Algorithm 2 kmeans_serial(*data_points*, *centroids*) (Clustering Data)

Allocate an array of length *data_points*: *nearest_cluster_idx*
for *idx*, *data* in *data_points* **do**
  Calculate the Euclidean distances *dists* of *data* every centroids
  Find the *centroid_idx* with minimum *dists* (*np.argmin*)
  Write *centroid_idx* to *nearest_cluster_idx*[*idx*]
end for

5.1.3 Updating Centroids

The centroids need to converge to their optimal positions by minimizing the distance between data points and their assigned centroids. To achieve this, the centroids are updated iteratively based on the newly clustered data points and the previous centroids. Algorithm 3 demonstrates the process. The time complexity of 1 iteration of updating centroids stage is $O(K * P)$.

Algorithm 3 update_centroids(*data_points*, *n_clusters*, *nearest_cluster_idx*, *centroids*)

for *cluster_idx* in *n_clusters* **do**
  $sum = 0$, $cnt = 0$
  for *idx*, *data* in *data_points* **do**
    if *nearest_cluster_idx*[*idx*] is *cluster_idx* **then**
      $cnt+ = 1$, $sum+ = data$
    end if
  end for
  centroids[*cluster_idx*] = *sum*/*cnt*
end for

5.1.4 NumPy Optimization v.s. Loops

Certain operations in the above mentioned algorithms can be represented as arrays and executes with NumPy [8] operations. NumPy is optimized through a combination of several techniques including (i) efficient data storage, (ii) C-level implementation, (iii) broadcasting and (iv) optimized algorithms. Specifically, some iterations of datapoints in the above algorithms can be replaced with `np.where`, `np.take`, `np.argmin`, etc, resulting massive speedups. The experiments of NumPy-optimized and PyOMP parallelized program is shown in Sec 6.1.5. However, Numba only supports a certain set of NumPy operations, such as basic arithmetic of `np.array` and `np.argmin`. The algorithms we illustrated above are the ones we mainly used for serial-parallel comparisons.

5.2 K-means PyOMP Implementation

PyOMP is dedicated to offer similar syntax with OpenMP in C. For instance, `#pragma omp parallel` is replaced with `with openmp("parallel"):`; `#pragma omp parallel for reduction(+:sum)schedule(static, 32)` is replaced with `with openmp("for␣reduction(+:sum)␣schedule(static,␣32)"):`. A table of supported PyOMP operations and its OpenMP references is shown in Table 1. An example code snippet of PyOMP is shown in Fig 1.

For Clustering Data stage in Sec 5.1.2, the PyOMP parallelized program is rather simple since entries of NumPy array are stored in contiguous but different memory locations, and all entries are written only once. There are no race condition. The code snippet is shown in Fig 1. There are two reason of specifying static scheduler in this parallel loop: (i) all subtasks have exact amount of computation, and (ii) PyOMP does not support dynamic scheduler yet.

For Updating centroids stage in Sec 5.1.3, the PyOMP parallelized version includes atomically adding data points in each cluster. A problem arose during our implementation: PyOMP does not support `reduction` for an array, only single variables are allowed. We came up with a work-around by specifying an integer for `reduction`, and wrapping the array addition within the computation. For every experiments, we ensured that the centroids are exactly correct with single-thread program.

PyOMP	OpenMP
`with openmp("parallel"):`	`#pragma omp parallel`
`with openmp("for"):`	`#pragma omp parallel for`
`with openmp("single"):`	`#pragma omp single`
`with openmp("task"):`	`#pragma omp task`
`with openmp("taskwait"):`	`#pragma omp taskwait`
`with openmp("barrier"):`	`#pragma omp barrier`
`with openmp("critical"):`	`#pragma omp critical`
`with openmp("for␣schedule(static␣[,chunk])"):`	`#pragma openmp for schedule(static, [chunk])`
`with openmp("for␣reduction(op:List)"):`	`#pragma omp for reduction(op:List)`
`with openmp("for␣private(List)"):`	`#pragma omp for private(List)`
`with openmp("for␣shared(List)"):`	`#pragma omp for shared(List)`

Table 1: PyOMP supported operations and references to OpenMP

```
@njit
def iter_kmeans(X_train, centroids):
    sorted_points = np.zeros(len(X_train))
    with openmp("parallel␣for␣schedule(static)"):
        for i in range(len(X_train)):
            dists = euclidean(X_train[i], centroids)
            centroid_idx = np.argmin(dists)
            sorted_points[i] = centroid_idx
    retyrb sorted_points
```

Figure 1: A code snippet of PyOMP parallelized kmeans iteration

5.3 K-means PySpark Implementation

Apache Spark offers solutions to limited hardware resource by sending data across a set of machines and execute concurrently. Spark is developed upon MapReduce [13]. For large machine learning tasks, large-sized dataset cannot fit in memory or disk of a single machine, and utilizing CPUs on multiple machine is for efficient. Therefore with parallelized cloud computing, data and operations are mapped by a driver through a series of executors in a cluster, the computations are executed in parallel, then results are reduced back to the driver. The concept of spreading data across nodes can be known as **Data Parallelism**, and a cluster that broadcasts and updates the parameters on each executor in parallel can be known as a **Parameter Server**. For comparison with single-machine, multi-thread implementation, the executors in a Spark cluster is acting as cores in a CPU. For continuity of our notation with

5.3.1 Spark Cluster Setup

To setup the Spark cluster efficiently on Amazon Web Services, we leveraged a python package, flintrock. Flintrock offers CLI for launching and terminating Spark clusters on AWS, configuring the cluster settings, installing Spark and its dependencies, and managing SSH access to the cluster nodes. We use flintrock to install required packages on all nodes for better performance, such as NumPy.

5.3.2 Data Parallelism

The data are evenly distributed across all executors before the clustering starts. We first distribute the dataset into several files in Hadoop Distributed File System (HDFS), to mimic the scenario of not able to store all data in the driver node. The data are then download in each executor's storage space, and loaded by each executor into its memory. Now the data are distributed into partitions on each executors. To ensure that the partitions are even, we assigned IDs to every data, then repartition the data by the remainder of ID divided by number of executors. The partition function is shown in equation 1. The action of data transmitting between executors through network is called data shuffling. The performance of data shuffling depends on network bandwidth between nodes in a cluster.

```
partitionFunc = lambda x : x % nThreads
```
(1)

5.3.3 Parameters Synchronization

The centroids of the clusters in k-means clustering are required by every data points, in another word, the centroids need to be broadcast to all executors. We're leveraging the built in function of pyspark, `SparkContext.broadcast()`, to broadcast the centroids to every executor. The parameter synchronization is also a form of data shuffling.

5.3.4 Implementation Details

Taking the efficiency of data shuffling into design, the Spark implementation is very different to multi-thread parallelization. Since the data are distributed across nodes, a calculated result of data across clusters requires a more cautious design. Algorithm 4 demonstrated the Updating Centroids stage in k-means clustering. As shown in line 4 and 5, the local mean of the data in each cluster is calculated in parallel, the individual results are reduced to the driver, then the final mean is calculated as the new centroid.

Algorithm 4 spark_update_centroids(*data_points*, *n_clusters*, *nearest_cluster_idx*, *centroids*)

```
for cluster_idx in n_clusters do
    cluster_indices = nearest_cluster_idx.filter(x == i)
    cluster_datapoints = data_points.filter(id in cluster_indices)
    local_means = cluster_datapoints.mapPartitions(calculate_local_means)
    new_centroid = np.mean(local_means)
    new_centroids.append(new_centroid)
end for
SparkContext.broadcast(new_centroids)
```

5.4 K-means parallelization experiment setup

5.4.1 Datasets

We're leveraging the simplicity of MNIST [14] dataset as our accuracy, correctness, and performance experiments. MNIST contains 70,000 grayscale images of handwritten digits, each of size 28x28 pixels. The images can be clustered into 10 clusters as a metric of algorithm accuracy test. The small-sized dataset is suitable for initial performance tests.

OpenImages.

5.4.2 Number of Clusters, Iterations, Random Seeds

The number of clusters for all experiments were set to be the number of target class to be classified. The iterations were set between iterations to convergence. The seed for randomness were always set as the course number, 15618.

For MNIST dataset, the number of clusters is 10, k-means converges after 92 iterations, with seed 15618.

5.4.3 Hardware setup for PyOMP experiments

For k-means PyOMP parallelization, we ran the experiments on one of our personal computer with AMD Ryzen5 5600x as CPU. AMD 5600x has 6 cores (12 threads), maximum clock speed of 4.6 GHz, and 35 MB cache size. Due to disk quota constraint, we are unable to install required packages for PyOMP on GHC and PSC both. The pricing of AMD Ryzen5 5600x is 299.00 $USD.

5.4.4 Hardware setup for PySpark experiments

For k-means PySpark parallelization, we are using Elastic Compute Cloud (EC2) on Amazon Web Services (AWS). The instance type we chose is `t3.large`. `t3.large` uses Intel Xeon Platinum 8000 series CPU with 2 threads per core and clock speed up to 3.0 GHz. Our experiment showed the maximum clockspeed for each executor is 2.5GHz, and the same program (Py_serial in Fig 5) runs two times longer on `t3.large` (466.23 seconds v. 232.52 seconds). For comparison between PyOMP, we take 1 node (with 2 vCPU/threads) as 1 thread in AMD Ryzen5 5600x. The spot instance price of `t3.large` is $0.0637 USD/hr on April 16_{th}, 2023.

Note: We cannot find AWS instance type with the same performance as AMD Ryzen5 5600x CPU, and the installation of PyOMP failed on AWS EC2 instances. It is a compromise between experiment consistency and costs.

5.5 DBSCAN Serial Implementation

In this section, we begins by discussing the hyper-parameters used to determine the connectivity between data points and the quality of clustering in DBSCAN. We then provide an overview of the naive DBSCAN algorithm and highlight its limitations. Next, we introduce the k-dimensional tree, which is used to speed up the process of finding the closest pairs in the DBSCAN algorithm. Finally, we introduce grid-based DBSCAN, a variant of the naive DBSCAN algorithm that offers better performance.

5.5.1 Hyper-parameters

DBSCAN has two key hyper-parameters: epsilon and minimum samples. Epsilon is used to define the connectivity threshold between any two points. Minimum samples is a threshold value that determines the minimum number of cluster points. Together, these hyper-parameters are used to classify points into one of three categories: core, border, or noise, based on their connectivity with other points in the dataset. Following is a the definition of each category.

- core: #connections $\geq$ minimum samples
- border: #connections $<$ minimum samples, and clustering group contain at least one core point.
- noise: #connections $<$ minimum samples, and clustering group contain no core point.

5.5.2 naive DBSCAN

The naive DBSCAN algorithm consists of two main steps: marking (Algorithm 5) and expanding (Algorithm 6). In the marking step, the algorithm identifies core points based on the epsilon and minimum sample specified by the user. Once all core points have been identified, a traversal algorithm can be used to expand the connections and mark the connected components as clusters. During expansion, the boundary points are also marked, and any cells that do not have any connections are classified as noise cells.The primary challenge of DBSCAN lies in its connection-based algorithm, which requires calculating the distance between every pair of points in the dataset. This leads to a computational complexity of $O(n^2)$ for the naive approach, making it impractical for large datasets.

Algorithm 5 DBSCAN($\mathcal{D}$(Dataset), *eps*, *min_samples*) (Marking)

Require: $eps \in \mathbf{R}^+$
Require: $min\_samples \in \mathbf{N}^+$
  for p_i in $\mathcal{D}$ **do**
    $conn_i \leftarrow \sum[dist(p_i, p_j) \leq eps \ \forall j \neq i, j \in 1, 2, \ldots, n]$
    if $conn_p \geq min\_samples$ **then**
      Add p_i into $\mathcal{C}(core\_point)$
    end if
  end for

Algorithm 6 DBSCAN($\mathcal{D}$(Dataset), *eps*, *min_samples*) (Expanding)

Require: $eps \in \mathbf{R}^+$
Require: $min\_samples \in \mathbf{N}^+$
  $\mathcal{G}$(clustering group) $\leftarrow \{\emptyset\}$
  for p_i in $\mathcal{D}$ **do**
    if $p_i \in \mathcal{C}$ and $p_i \notin g\ \forall g \in \mathcal{G}$ **then**
      $g' \leftarrow$ BFS from p_i
      Add p_j into $\mathcal{B}$(border points) $\forall p_j \in g'$ and $p_j \notin \mathcal{C}$
      Add g' into $\mathcal{G}$
    end if
    Add p into $\mathcal{N}$(Noise points) $\forall p \in \mathcal{D}$ and $p \notin \mathcal{C}$, $p \notin \mathcal{B}$
  end for

5.5.3 Closest Pair and K-Dimensional Tree

Calculating the closest pair (CP) using the naive approach has a time complexity of $O(n^2)$, which is not feasible for large datasets. To address this issue, there are three alternative approaches: KDTree, BallTree, and Brute Force. In this phase, we have adopted the KDTree strategy, but we may explore other methods in subsequent phases.

The KDTree was first proposed by Jon Bentley [15]. The concept of a "k-d tree" is an extension of the binary search tree, which can be used for searching multidimensional data. Using a KDTree for closest pair (CP) searching offers a significant advantage as it can reduce the number of computations required to identify the nearest neighbors. By utilizing the KDTree traversal approach, it can identify a subset of closer points, and the actual distances only need to be computed for this subset.

During the construction and operation of the KDTree, the pivot selection rotates across the different dimensions. Initially, the pivot is chosen as the median point on the first dimension (axis-0), and the points are divided into left and right parts. To ensure that the tree is split on every dimension, we pass the $depth + 1$ to the next depth recursively. This process continues until the minimum leaf size is reached. In the current phase, the minimum leaf size is set to one for simplicity.

Algorithm 7 constructKDTree($\mathcal{P}$(*point set*), *depth*)

  if $\mathcal{P} \neq \emptyset$ **then**
    $axis \leftarrow depth$ mod $dimension(\mathcal{P})$
    $m \leftarrow$ median($\mathcal{P}$) on dimension $axis$
    $left \leftarrow \forall p \in \mathcal{P}\ p < m$
    $right \leftarrow \forall p \in \mathcal{P}\ p > m$
    $n \leftarrow newNode()$
    $n.point \leftarrow m$
    $n.leftChild \leftarrow$ constructKDTree($left$, $depth + 1$)
    $n.rightChild \leftarrow$ constructKDTree($right$, $depth + 1$)
    return n
  end if
  return $\emptyset$

To handle the complexity of the k dimension, KDTree searching also involves rotating the axis. Initially, the algorithm collects the node point if it satisfies the distance threshold. Then, it checks the boundary condition on both the left and right child to determine whether to continue or not.

Algorithm 8 searchKDTree(*node*, *point*, *radius*, *depth*, *result*)

if $node \neq \emptyset$ **then**
  $dist \leftarrow$ distance($node.point$, $point$)
  if $dist \leq radius$ **then**
    Add $node.point$ into $result$
  end if
  $axis \leftarrow depth$ mod $dimension(point)$
  if $point - radius \leq node.point[axis]$ **then**
    searchKDTree($node.leftChild$, $point$, $radius$, $depth + 1$, $result$)
  end if
  if $point + radius \geq node.point[axis]$ **then**
    searchKDTree($node.rightChild$, $point$, $radius$, $depth + 1$, $result$)
  end if
end if
return $\emptyset$

5.5.4 Grid-based DBSCAN Algorithm

Grid-based DBSCAN is a modification of the traditional density-based clustering algorithm DBSCAN proposed by Ester et al [4]. The grid-based optimization was introduced to reduce the time complexity of DBSCAN for large datasets by partitioning the data space into a grid of cells and processing only the cells that contain data points, rather than processing every point in the dataset.

We adopt the method proposed by [4] to divide the space on $eps/\sqrt{d}$ where d is dimension. This method ensures that points within each cell are in the epsilon threshold. After partitioning particles into the associated cell, we identify those cells c with points more than the minimum sample as the in-grid cells. Next, we mark the out-grid cells which have less than minimum sample points but meet the following requirement.

$$\forall p \in c : \exists p : conn(p) \geq min\_sample$$

Once all the core cells have been identified, the clustering algorithm proceeds by expanding from each core cell. If a Bichromatic Closest Pair (BCP) is found and it is less than the epsilon threshold with a neighboring cell, the two cells are connected and grouped as the same cluster. The non-core cells that are connected to other cells are known as border cells, while the non-core cells without any connections are classified as noise cells.

The grid-based DBSCAN algorithm depends on solving the Bichromatic Closest Pair (BCP) problem, which can be implemented using the multi-point closest pair method. To optimize neighbor cell searching time, we construct a KDTree for all existing grids. During the identification of out-grid cells, we construct multiple KDTrees. These BCP algorithms include $findNeighbor()$ and $BCP()$ in algorithm 11 and algorithm 12.

Algorithm 9 grid-based DBSCAN(Partitioning)($\mathcal{D}$, *eps*, *min_samples*)

$l \leftarrow \epsilon/\sqrt{d}$
$Grid \leftarrow \emptyset$
for $p \in \mathcal{D}$ **do**
  $gridID \leftarrow getGridID(p, l)$
  Add p into $Grid[gridID]$
end for

Algorithm 10 grid-based DBSCAN(Mark in-grid cells)($\mathcal{D}$, eps, $min\_samples$)

$cores \leftarrow \{\emptyset\}$
for $cell \in grid$ **do**
  if $|p| \geq min\_samples \forall p \in cell$ **then**
    Add $cell$ into $cores$
  end if
end for

Algorithm 11 grid-based DBSCAN(Mark out-grid cells)($\mathcal{D}$, eps, $min\_samples$)

for $cell \in grid$ **do**
  $neighborCells \leftarrow findNeighbor(cell)$
  $neighborTree \leftarrow KDTreeConstruction(neighborCells)$
  if $\exists\ p \in cell : |KDTreeSearch(p, neighborTree, \epsilon)| \geq min\_sample)$ **then**
    Add $cell$ into $cores$
  end if
end for

Algorithm 12 grid-based DBSCAN(clustering)($\mathcal{D}$, eps, $min\_samples$)

for $cell \in cores$ **do**
  $neighborCells \leftarrow findNeighbor(cell)$
  for $neighborCell \in neighborCells$ **do**
    if $BCP(cell, neighborCell) \leq \epsilon$ **then**
      Mark $cell$ and $neighborCell$ as same cluster
    end if
  end for
end for

5.6 From Serial to Parallel: From BFS to Union-Find

In order to implement parallel grid-based DBSCAN, it is necessary to modify the expansion method used in the serial algorithm. The serial program utilizes a BFS method with a queue, which severely restricts the parallelism potential and leads to numerous race conditions. To address this challenge, previous studies have utilized the union-find data structure, which has demonstrated exceptional performance. Therefore, we have also adopted this strategy.

5.7 DBSCAN Parallel Implementation

The use of PyOMP in parallelizing DBSCAN was unsuccessful due to restrictions posed by Numba and JIT (Just-In-Time) compilation. JIT is unable to support two important data structures, dictionaries and sets, due to limitations in type deduction. Additionally, the complexity of the DBSCAN algorithm and its reliance on complex data structures such as KDTree and grid-based data make development without these datas structure impractical. As an alternative to PyOMP, we have implemented parallelism in DBSCAN using the multiprocessing module. It is noteworthy that we experimented with three parallelism models in the multiprocessing module and have observed that the master-slave model outperforms the other models. In this phase, we have implemented the following parallelisms:

Parallel Grid Construction In this process, the master distributes data points to workers and collects the corresponding grid IDs when workers return. The workers calculate the grid ID for each data point received and return the ID to the master. However, since this action has low arithmetic intensity (1/2), we did not observe any parallel benefit and therefore turned off the parallelism for this function.

Parallel in-grid core cell identification The master passes each cell to workers, who then compare the size of the cell to the minimum sample requirement and return the result to the master. However, since this is a low arithmetic intensity action (1/2), we observed that the overhead outweighs the benefits and decided to turn off the parallelism for this function.

Parallel out-grid core cell identification The master passes each non-core cell and collects its associated identity. For each received cell, workers collect the neighboring cells and construct a neighbor tree. They apply the BCP algorithm to determine if there is a point with number of connections exceeding the threshold, and if so, return the cell as a core cell. This is a more complex action with a computation of $O(cnd)$, where c is the number of points in the cell, n is the number of points in the neighboring cells, and d is the dimension. However, the real computation time is much lower due to the early stop mechanism. Once a point satisfying the constraint is found, the process can be stopped earlier. Our experiments show a 1.27x speed-up.

Parallel clustering(expanding) The master distributes core cells among workers, who then apply the BCP algorithm to each neighboring cell and return the connection list to the master. Once all workers have finished, the master uses a union-find algorithm to construct the clustering. This is the most computationally complex part of the process, with a time complexity of $O(cnd)$, where c is the number of points in the cell, n is the number of points in the neighbors, and d is the dimension. Since BCP has to be applied to every neighbor, the early stop mechanism is limited compared to out-grid core cell identification. However, our experiments showed a speed-up of 1.73x. Due to the limitation of parallelism on Python, we move our parallelism development on C++.

5.8 DBSCAN Parallel Implementation on C++

Dataset Developers working in C++ can simplify the testing process for datasets of different sizes by utilizing a dataset generated with random library. This enables them to produce data of varying dimensions and sizes by employing various distribution functions that can create diverse data distributions through their combination. In this particular section, three clusters with normal distribution are randomly created with sizes ranging from 500 to 1,000,000 in a 2-dimensional space. The following sections will describe the environment settings, including the number of data points (n), number of clusters (cluster), epsilon value (eps), and minimum number of points (minPts).

Data Structure Even the most advanced C++ DBSCAN implementation [16] failed to meet the efficiency on high dimensional DBSCAN algorithms, we move the development of KDTree and high dimensional DBSCAN algorithms in future work. This will allow us to prioritize parallelism and optimization. As a result, we will only be using the Grid-based data structure and Union Find data structure from the previous section.

Identify performance-critical function we first analyze the performance bottleneck on the serial implementation. From Table 2, it can be observed that the "expand"(clustering) procedure takes up the majority of the runtime with 98.7861%. This indicates that improving the performance of the "expand" procedure should be the primary target for optimizing the overall performance of the system. Meanwhile, the other procedures such as "assignPoints" and "mark_outgrid_corecell" take up a relatively small portion of the runtime, at 0.9538% and 0.2599% respectively. The "mark_ingrid_corecell" procedure takes up a negligible amount of runtime at 0.0001%. Therefore, optimizing the "expand" procedure is likely to yield the most significant improvements in the system's overall performance.

Procedure	Runtime	Percentage
assignPoints	0.011691	0.9538%
mark_ingrid_corecell	1.68E-06	0.0001%
mark_outgrid_corecell	0.00318535	0.2599%
expand	1.21085	98.7861%

Table 2: Performance analysis of serial grid-based DBSCAN implementation.

Parallelism with OpenMP We implemented the grid-based DBSCAN algorithm in a parallel manner by processing each core cell independently. The only dependency exists in the Union-Find data, but we have eliminated race conditions by avoiding bidirectional unions and compare and swap(CAS). Initially, we used loop-based parallelism due to its simplicity, but we have also experimented with task-based parallelism, which did not yield a significant performance difference.

Detail Performance Anaylsis with Perf We have conducted a detailed examination of the performance bottleneck in the 'expend' program and have utilized the perf tool to analyze its performance, as shown in Table 3 and Table 4. The first perf report highlights the percentage of CPU cycles utilized by different functions during the execution of the dbscan program. The dist function in the dbscan code accounts for 27.50% of the CPU cycles, while the malloc function in the libc library accounts for 29.23% of the CPU cycles, followed by _int_free and isConnect. Additionally, the report indicates that the operator new function in the libstdc++ library and the cfree function in the libc library consume a significant amount of CPU cycles. This suggests that memory allocation and deallocation may be a bottleneck in the program's performance. The second perf stat report contains standard performance counter statistics, including information on time elapsed, CPU utilization, context-switches, page-faults, cycles, stalled-cycles-frontend, stalled-cycles-backend, instructions, branches, and branch-misses. Based on these metrics, we have concluded that there are no unusual performance patterns in the program. (The original Perf report can be found at Appendix 8.3.)

Table 3: Performance report for the DBSCAN algorithm - 1

Ratio	Event
29.23%	[.] malloc
27.50%	[.] dist
16.92%	[.] _int_free
9.06%	[.] isConnect
5.84%	[.] operator new
3.78%	[.] cfree@GLIBC_2.2.5
3.09%	[.] __memmove_avx_unaligned_erms
1.53%	[.] operator delete@plt
1.12%	[.] memmove@plt
0.89%	[.] malloc@plt

Table 4: Performance report for the DBSCAN algorithm - 2

Value	Counter	Note
87,602.85 msec	task-clock:u	1.000 CPUs utilized
0	context-switches:u	0.000 K/sec
0	cpu-migrations:u	0.000 K/sec
11,303	page-faults:u	0.129 K/sec
284,992,059,313	cycles:u	3.253 GHz (50.00%)
13,553,401	stalled-cycles-frontend:u	0.00% frontend cycles idle (50.00%)
4,873,430,201	stalled-cycles-backend:u	1.71% backend cycles idle (50.00%)
925,491,792,090	instructions:u	3.25 insn per cycle, 0.01 stalled cycles per insn (50.00%)
226,426,881,223	branches:u	2584.698 M/sec (50.00%)
1,155,082	branch-misses:u	0.00% of all branches (50.00%)

C++ Concurrency We have also utilized the C++ Concurrency API to implement an additional parallel framework to determine if we could further improve performance. We tested this framework on the following environment settings: (n:100000, cluster:3, eps:0.12, minPts:5). For this implementation, we adopted loop-based parallelism, which breaks down the workload into similar small blocks for each thread. We observed a slight improvement in performance compared to OpenMP, but we also encountered a workload imbalance issue. When the number of threads exceeds half the number of tasks, there is a significant workload imbalance where each thread, except for the last one, is assigned two core cells, while the last thread is assigned half the number of tasks. Similarly, when the number of threads exceeds the number of tasks, the last thread is assigned all the workload, while the others have none. This highlights the downside of static scheduling. As a result, we developed a dynamic scheduling approach, which utilizes a work-stealing mechanism.

Dynamic Scheduling Development: A work-stealing mechanism In order to tackle workload imbalances, we have introduced a work-stealing technique that allocates a task queue to every thread. This method allows the master thread to keep track of each worker thread's progress. When the master identifies a vacant task queue, it reassigns a task from a neighboring queue to the idle thread for processing. The work-stealing approach guarantees effective utilization of threads, fostering equitable workload distribution and boosting overall performance. The impact of this strategy is illustrated in the accompanying figure. While there is a minor overhead with a lower thread count, the work-stealing method improve the workload balancing issue.

Accelerating Parallel Programs with SIMD and AVX Instructions In our previous section(Detail Performance Anaylsis with Perf), we discovered that the primary performance bottleneck in our parallel program was related to memory allocation and floating-point arithmetic operations. In an effort to further optimize our program, we turned to the study of SIMD and AVX instructions. We implemented a new version of our parallelism framework that enabled AVX instructions for the most critical floating-point arithmetic functions. As a result, we were able to achieve significant performance gains - up to 1.78x on a single thread, 3.50x on 32 threads, and 12.79x compared to the serial program, details can be found in the following table and figure. These results demonstrate the tremendous potential of using SIMD and AVX instructions to accelerate parallel programs, particularly those with significant floating-point arithmetic computations.

Algorithm 13 DynamicScheduling(TASK, nthreads)

```
Distribute the task to work queue of each threads.
for each thread do
    Execute the sub-functions with its work queue.
end for
while TASK is not completed do
    for each thread do
        if work queue is empty and next thread's work queue is not empty then
            Steal the last element from neighbor and execute it.
        end if
    end for
end while
```

6 Results

6.1 K-means Multi-thread Parallelization with PyOMP

6.1.1 Serial Implementation Analysis

To ensure the correctness of our algorithm, we conducted preliminary experiments on clustered accuracy with number of clusters. For MNIST, which has 10 clusters, clustering data points into more clusters then inference the clusters into true labels yields better accuracy. The accuracy of our implementation is shown in Table 5. The centroids of k-means with $k = 10$ is shown in Fig 2.

The following experiments has the setup of $k = 10$ for k-means clustering.

Clusters(k)	64		36		10	
Implementation	Serial	PyOMP	Serial	PyOMP	Serial	PyOMP
Iterations	100	100	255	255	92	92
Accuracy	83.74%	83.74%	77.73%	77.73%	59.42%	59.42%
Homogeneity	77.62%	77.62%	71.40%	71.40%	50.88%	50.88%

Table 5: Correctness of serial and PyOMP implementation of K-means clustering

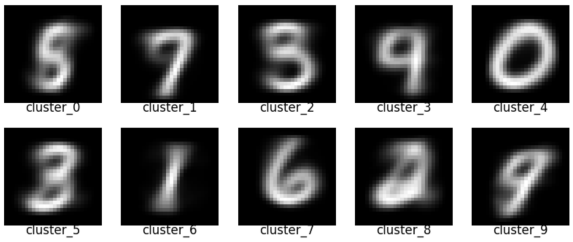


Figure 2: k-means clustered centroids with $k = 10$

6.1.2 Runtime decomposition with different nThreads

The runtime for the three stages in k-means are shown in Fig 3. As demonstrated in our experiments, Clustering Data stage accounts for most of the runtime. This is expected since clustering data points involves heavy computation on Euclidean distances. We leave our analysis of speedups in Sec 6.1.3.

6.1.3 Clustering Data Analysis

Fig 4a illustrated the speedups of Clustering Data stage with respect to number of threads executing. From $nThread = 1$ to 6, the speedups increases almost linearly. An interesting phenomenon occurred when $nThread = 7$. The performance decreased but later increased with $nThread$ increased to 12. The reason may be the chip has only 6 cores, each with 2 threads. Synchronizing and communicating across the cores had less overheads. When adding threads that shares the resource in a core, the communication overheads and contentions are more apparent. Nevertheless, the speedup of 12 threads still out-performed the speedup of 6 threads.

We also experimented on the supported static scheduler. However, the result seems similar. Considering the fact that they have not implemented dynamic scheduler yet, we assume the implicit scheduler is the static scheduler.

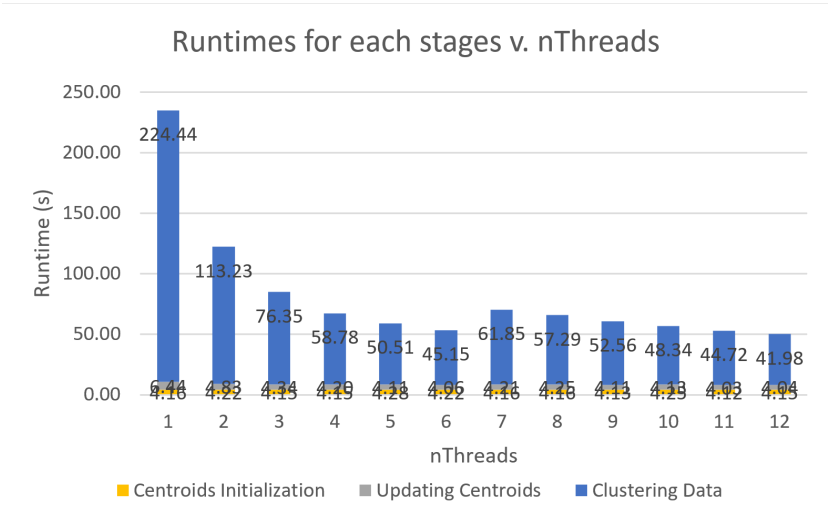
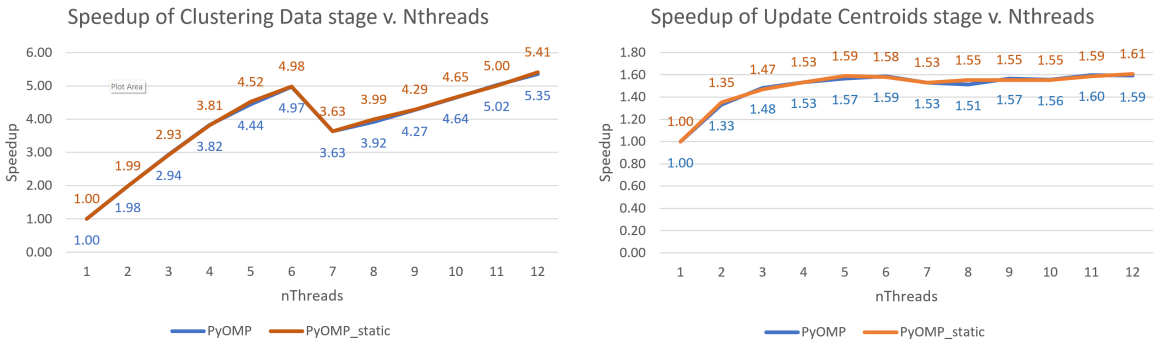


Figure 3: PyOMP K-means total runtime w.r.t. nThreads



(a) Clustering Data stage

(b) Updating Centroids stage

Figure 4: PyOMP K-means stage speedup w.r.t. nThreads

6.1.4 Atomic Implementation Analysis

Fig 4b illustrated the speedups of Updating Centroids stage with respect to number of threads executing. Observed in our experiment, the speedup were limited at around $1.6x$ when $nThread$ increases. The reason is that there were a necessary atomic action mentioned in Sec 5.1.3. The atomic operations in each threads contented for execution of writes to the summarized data points. Thus caused the speedup ceiling in the stage.

6.1.5 Different Optimization Approaches

We also implemented our serial algorithm with NumPy and the parallelized scikit-learn [17]. Fig 5 demonstrated the runtimes of PyOMP with 1 thread, PyOMP with 12 threads, pure serial implementation, NumPy optimized implementation, and scikit-learn implementation. We see that the same program but optimized with NumPy operations achieves quite a speedup comparing with plain implementation. However, our PyOMP implementation with 12 threads speedup still out-performed the NumPy-optimized program. Scikit-learn implements a more optimized K-means algorithm with built-in OpenMP support on multi-threading, which makes it the fastest. A more detailed analysis on CPU core utilization is shown in Sec 6.1.6.

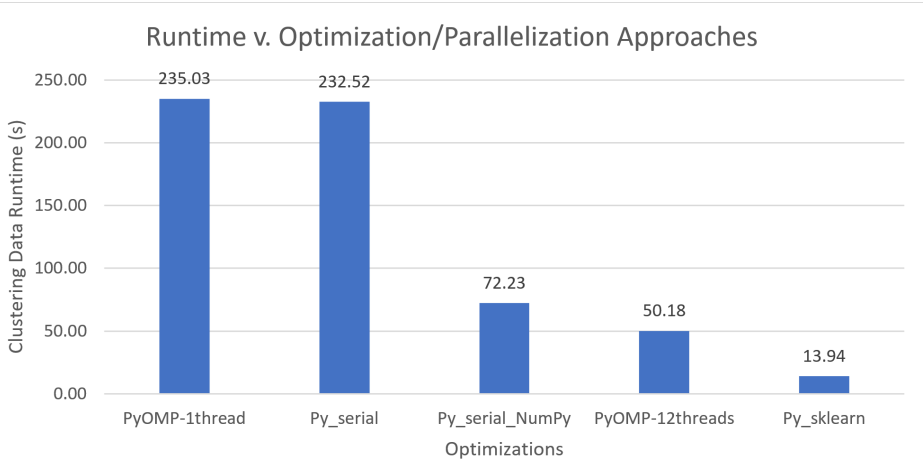


Figure 5: PyOMP K-means runtime with different optimization approaches

(a) PyOMP with 1 thread

(b) PyOMP with 6 threads

(c) PyOMP with 8 thread

(d) PyOMP with 12 threads

Figure 6: PyOMP K-means CPU Utilizations

6.1.6 CPU Utilization Analysis

To better inspect the threads utilization during execution, we logged the CPU clock rate.

For PyOMP parallelized implementation, some logs are shown in Fig 6. As demonstrated in Fig 6a, there is always 1 core with the clock rate at about 4.5 GHz, close to the maximum clock speed of 4.6 GHz. The reason of threads changing is due to context switching between processors. As shown in Fig 6b and Fig 6c, with more threads running the program, the cores were utilized more heavily. Finally, Fig 6d demonstrated the cores were utilized in the same way, since every threads were executing exact same computation. One phenomenone worth noticing is the maximum clock speed differ between Fig 6a and 6d. We concluded two reasons for this: (i) communication overhead, and (ii) heterogeneous core design. The communication overhead between each threads may decrease the maximum clock speeds with more executing threads. The heterogeneous design of the CPU chip may benefit the utilization of single-thread process.

6.1.7 Conclusion

As shown in the above analysis, our PyOMP parallelized k-means clustering achieved a $5.41x$ **speedup** with 6 cores (12 threads) comparing with the serial implementation. Our PyOMP parallelized implementation performed 44% better than NumPy-optimized version. Although it is difficult to surpass scikit-learn's efficient algorithm and parallelized execution, we consider our PyOMP implementation with 12 threads a success.

6.2 K-means Distributed Parallelization with PySpark

6.2.1 Implementation Correctness Analysis

Clusters (k)	10		
Implementation	Serial	PyOMP	PySpark
Iterations	92	92	100
Accuracy	59.42%	59.42%	64.62%
Homogeneity	50.88%	50.88%	55.93%

(a) Accuracy of PySpark K-means clustering

(b) K-means clustered centroids with $k = 10$

Figure 7: PySpark Accuracy Analysis

To ensure the correctness of our algorithm, we tried to align every steps between PySpark and PyOMP. However, there are floating points error between AWS EC2 `t3.large` instance and AMD Ryzen5 5600x. Through experiments, we discovered that the error is about 10^{-9} for every data points each iteration. The slight difference caused about 5 in 60000 datapoints to be clustered in different cluster per iteration. Since the accuracy does not effect the performance analysis, we allowed it. The accuracy of PySpark k-means clustering is shown in Fig 7a and an example clusters shown in Fig 7b. During execution, we can see the execution time of each executor through PySpark server. A screenshot of a job executed on 12 executors is shonw in Fig 8.

Figure 8: PySpark job executed on 12 executors

6.2.2 Runtime decomposition with different nThreads

We set up each experiments where $nThread$ is the number of `t3.large` nodes in a cluster. The dataset were partitioned into $nThread * 2$ since each node has 2 threads, each with half the clockd speed of AMD Ryzen5 5600x.

As we learned, Spark packs the several operations (stages) into a job. Jobs end with reduce tasks, such as `.collect()`, `.sum()`. Since Clustering Data stage is a map task which computes together with the later Updating Centroids stage, the two stages are analysis together.

The runtime for the two stages in k-means are shown in Fig 9. There is a drastic time difference between PySpark and PyOMP, where a 40 seconds runtime with 12 threads PyOMP turned to 24 minutes with 12 nodes PySpark.

Nonetheless, as demonstrated in the figure, the execution time of k-means with PySpark decrease linearly with number of executors. The same script executed on only 1 node takes about 4.2 hours, and 24 minutes on 12 nodes. The runtime includes execution, data synchronization, and Spark execution overhead, which is why the task took this long.

There is an anomaly when executing on 5 nodes, so we neglected the result. A detailed analysis is in Appendix 8.2.

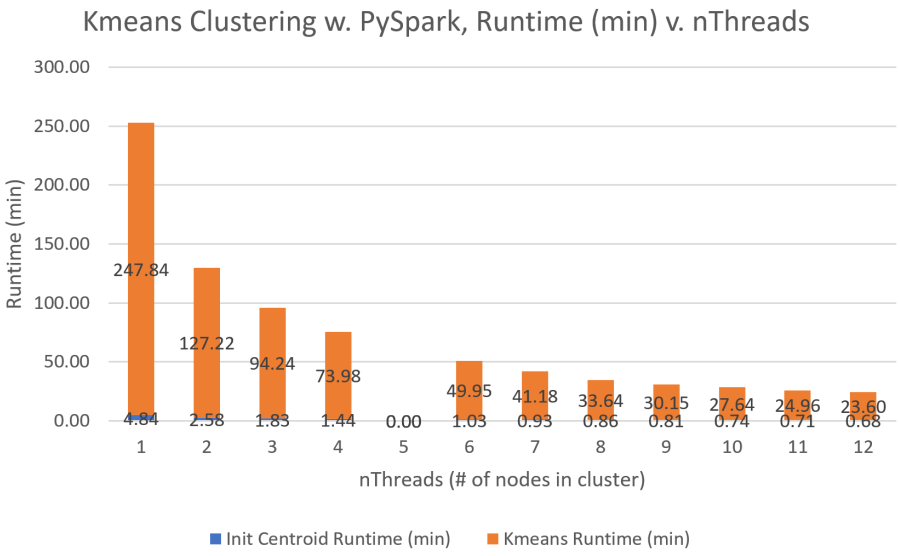
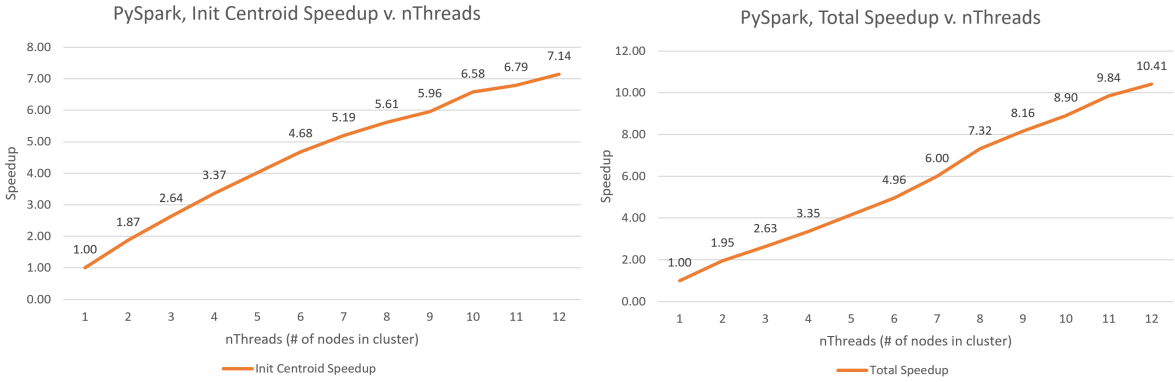


Figure 9: PySpark K-means total runtime w.r.t. nThreads



(a) Init Centroid Stage Speedup

(b) Total Speedup

Figure 10: PySpark K-means speedups w.r.t. nThreads

6.2.3 Speedup Analysis

In terms of speedup, Fig 10 illustrated the speedup of PySpark k-means implementation with respect to number of nodes in the cluster. The speedup between PySpark and PyOMP is quite similar from 1 thread to 6 threads. However, from 7 to 12 threads, the individual nodes do not suffer from issue mentioned in Sec 6.1.3. The speedup achieved $10.41x$ with 12 nodes executing at a time.

6.3 grid-based DBSCAN Parallelization with multiprocessing

To ensure the correctness of our DBSCAN implementation, we compared our accuracy with sklearn kit on the iris dataset in Table 6. We align our default value of hyper-parameters to sklearn, $\epsilon = 0.5$, and $min\_sample = 5$. Our implementation has higher accuracy compared to the implementation from sklearn, which could be due to differences in floating-point sizes(32bit, 16bit, and 8bit are commonly used in machine learning) and the sensitivity of floating-point calculations and comparisons.

Metrics	Sklearn DBSCAN	our DBSCAN
Accuracy(%)	0.56	0.63
Homogeneity(%)	0.50	0.66

Table 6: Accuracy of our DBSCAN and sklearn DBSCAN on Iris dataset

To evaluate the performance, we used our parallelism DBSCAN algorithm on the MNIST dataset with PCA dimensionality reduction applied, resulting in a reduced dimension of 2. The results are in Fig 11. Python's parallelism support is limited, posing challenges for parallelizing DBSCAN using PyOMP or multiprocessing. PyOMP is only capable of parallelizing simple numerical algorithms, while multiprocessing has limited support for shared-address models and incurs high overhead compared to C++. We developed grid-based DBSCAN using both PyOMP and multiprocessing and evaluated three approaches for multiprocessing: shared address models, message-passing models, master-worker models. The message-passing models involves queueing tasks and results, but its runtime increases with the number of workers. The shared address models passes shared resources to each worker but results in longer runtime compared to task-queue approach. The master-worker models performed better than the others, though not as well as expected.

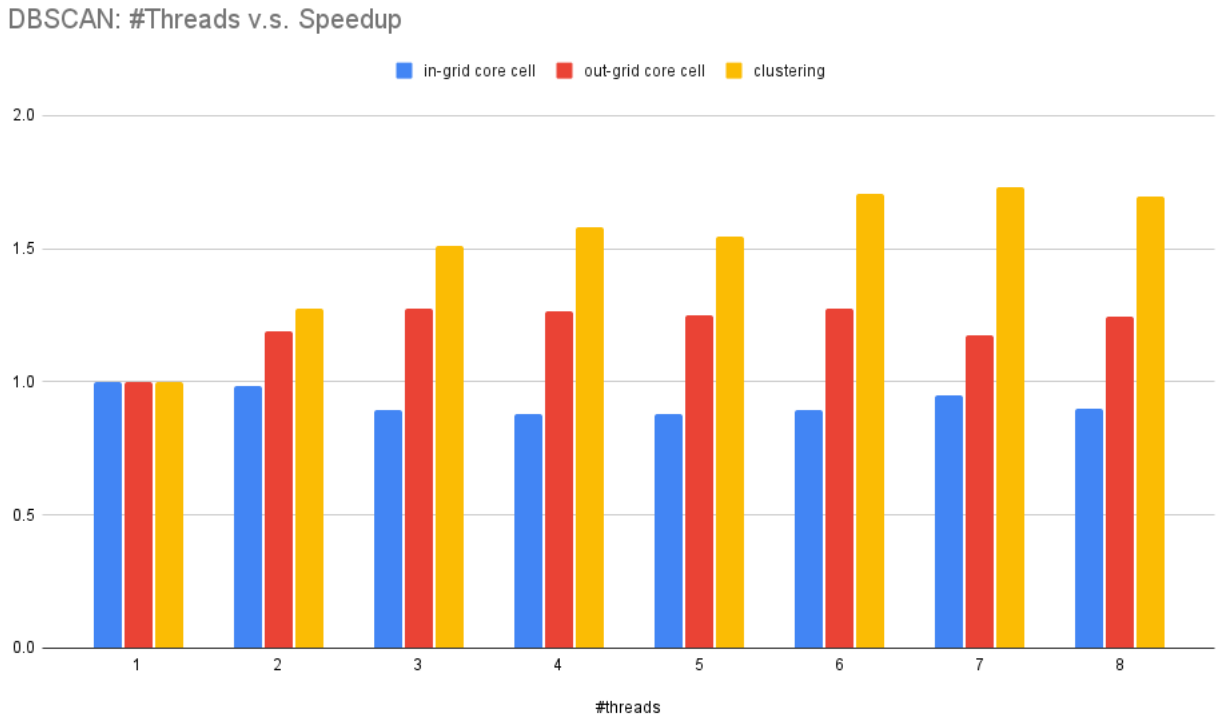


Figure 11: DBSCAN number of threads v.s. speedup

6.4 gird-based DBSCAN Parallelization with C++ Concurrency API and Work-Stealing

We evaluated the performance of our C++ parallelization implementation on the PSC using 1 to 128 threads, and a test environment with parameters of (n:100000, cluster:3, eps:0.12, minPts:5). The results, shown in Fig 12, indicate that the C++ Concurrency API provides slightly better performance compared to OpenMP, which aligns with our expectations since the former is a lower-level implementation. However, as we mentioned in the previous chapter, the C++ Concurrency API suffers from a workload balancing issue, which we addressed by developing a dynamic scheduling mechanism that enables work stealing. Our optimized implementation using the C++ Concurrency API with work stealing achieved the most reliable and best performance, particularly when the number of threads was large.

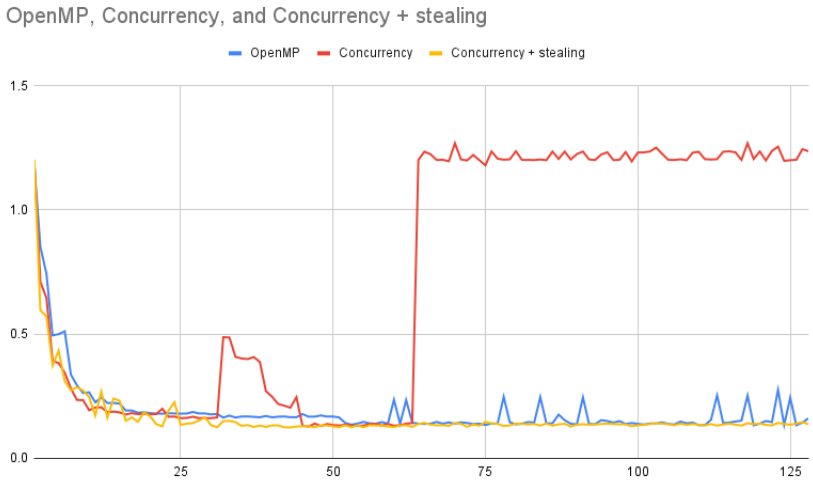


Figure 12: DBSCAN number of threads v.s. runtime

6.5 gird-based DBSCAN Parallelization Optimized with AVX Instructions

Additionally, we explored the use of SIMD and AVX instructions to further optimize our implementation, to further test the ability of our parallel framework, we use another environment setting(n:10000000, cluster:3, eps:0.12, minPts:5). Our final parallel implementation resulting in significant performance gains of up to 1.75x on a single thread, 3.5x on 32 threads, and 12.79x compared to the serial program. Details can be found from Table 7. Overall, our approach achieved impressive results and highlights the potential for using parallelism techniques and optimization strategies to accelerate performance in numerical computations.

Threads	OpenMP	Concurrency + stealing	Concurrency + stealing + AVX
1	60.5908 (0.99x)	59.9386 (1.00x)	33.683 (1.78x)
4	45.5341 (1.32x)	25.7059 (2.33x)	11.5379 (5.20x)
8	24.6547 (2.43x)	16.8995 (3.55x)	6.08641 (9.86x)
16	19.4427 (3.09x)	15.2388 (3.94x)	5.57829 (10.76x)
32	20.3203 (2.95x)	16.4425 (3.65x)	4.69244 (12.79x)
64	17.8841 (3.35x)	16.1772 (3.71x)	5.05127 (11.88x)
78	19.089 (3.14x)	15.9645 (3.76x)	4.80074 (12.50x)

Table 7: Comparison of runtime performance between OpenMP, Concurrency + stealing, and Concurrency + stealing + AVX on a multi-threaded program. The runtime is measured in seconds and compared to the serial program.

6.6 The limitation of parallelism on gird-based DBSCAN

Upon observation, we have determined that workload balancing is a primary concern, which is difficult to predict due to the intricate nature of DBSCAN. To address this issue, we have employed a straightforward method of workload monitoring through a basic counter that tracks the number of distance calculations. However, we have discovered that certain core cells necessitate over 50,000 computations during expansion, while others may only require 10. This limitation is inherent in the dataset and the choice of hyperparameters.

7 Conclusion

7.1 PyOMP K-Means Clustering

Our implementation of k-means clustering parallelized with PyOMP achieved **5.41x speedup** on a 6 core, 12 threads CPU; and **4.98x speedup** with 6 cores. The experiment was done with AMD Ryzen5 5600x with 6 cores and 12 threads total. The speedup out-performed the NumPy-optimized implementation by **1.44x**. We examined the maximum speedup of **1.6x** for atomic operations. We experimented that for AMD Ryzen5 5600x, the speedup with over 6 threads declined due to resource sharing within a core. We inspected the execution of PyOMP on switching context with CPU clock speed.

7.2 PySpark K-Means Clustering

Our implementation of k-means clustering parallelized with PySpark achieved **10.41x speedup** on a 12-node cluster, with each node having 2 vCPUs, each having half of the computation power of AMD Ryzen5 5600x. The speedup increased linearly with respect to the number of node executing. However, due to the heavy overhead of Spark job and synchronization across nodes, the runtime is drastically slower than PyOMP implementation of the same code.

7.3 Supercomputer or cloud computing

In this project, we compared the performance between using parallelizing across cores in a CPU and across nodes in a cluster, simulating the scenario of executing on a supercomputer and across cloud clusters. The same workload executed 50.18s with a 299.00 $USD AMD Ryzen5 5600x, whereas executed 1440s with 13 (with driver) AWS EC2 `t3.large` instance total of 0.33 $USD. We concluded that, multi-processing on a single-machine is definitely faster, since synchronization with internet is much slower than with interconnect network. There is too much overhead for Spark jobs. However, cloud instance is drastically cheaper than having a physical computer with simple configuration and pay-as-you-go service. Executing across nodes in a cluster surpasses the limitation of disk space and memory constraint. We thus conclude that the two use cases has its suitable scenario.

7.4 Parallization grid-based DBSCAN

We evaluated the feasibility of PyOMP and found that its lack of support for complex data structures in JIT and Numba resulted in an inevitable bottleneck. As a result, we opted to use the multiprocessing module to parallelize grid-based DBSCAN. We examined three parallel models: Queue-based message-passing, shared-address, and master-slave. The Queue-based message-passing and shared-address models resulted in significant overhead, and their performance degraded linearly as the number of workers increased. In contrast, the master-slave model yielded a slight improvement in performance for high-arithmetic functions, achieving 1.7x speedup with six workers and 1.25x with two workers.

After our unsuccessful attempt at Python parallelism, we implemented three new parallelism frameworks in C++: the OpenMP approach, the C++ Concurrency API approach, and the C++ Concurrency API with dynamic scheduling approach. By implementing these parallelism techniques and optimizing our most critical floating-point arithmetic functions with SIMD and AVX instructions, we were able to achieve a significant improvement in runtime - up to 1.78x on a single thread, 3.50x on 32 threads, and 12.79x compared to the serial program. Our approach involved profiling the runtime and identifying performance hotspots that accounted for 99% of the total runtime. We utilized OpenMP to

obtain a preliminary understanding of parallelism performance and further optimized using the C++ Concurrency API and a work-stealing mechanism. Our optimized implementation using the C++ Concurrency API with work stealing achieved the most reliable and best performance, particularly when the number of threads was large. Overall, our approach achieved impressive results and highlights the potential for using parallelism techniques and optimization strategies to accelerate performance in numerical computations.

References

[1] Amro Abbas, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S. Morcos, "Semdedup: Data-efficient learning at web-scale through semantic deduplication," 2023.

[2] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed, "Hubert: Self-supervised speech representation learning by masked prediction of hidden units," 2021.

[3] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967, vol. 1, pp. 281–297.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. 1996, KDD'96, p. 226–231, AAAI Press.

[5] Danny Chen, Michiel Smid, and Bin Xu, "Geometric algorithms for density-based data clustering," 08 2002, vol. 15, pp. 284–296.

[6] Richard Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[7] Todd Anderson and Tim Mattson, "Multithreaded parallel Python through OpenMP support in Numba," in *Proceedings of the 20th Python in Science Conference*, Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, Eds., 2021, pp. 140 – 147.

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sept. 2020.

[9] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[10] "Pyspark," https://github.com/apache/spark/tree/master/python, 2017.

[11] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al., "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[12] David Arthur and Sergei Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2007, pp. 1027–1035.

[13] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2004.

[14] Li Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[15] Jon Louis Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, sep 1975.

[16] Yiqiu Wang, Yan Gu, and Julian Shun, "Theoretically-efficient and practical parallel dbscan," 2021.

[17] "scikit-learn/_kmeans.py," https://github.com/scikit-learn/scikit-learn/blob/9aaed4987/sklearn/cluster/_kmeans.py#L1161.

8 Appendix

8.1 Work Distribution

8.1.1 Jhao-Ting Chen (jhaoting)

1. K-means Serial implementation & experiments
2. K-means PyOMP implementation & experiments
3. K-means PySpark implementation & experiments
4. Report writing on the above 2 parts
5. Poster: Supercomputer or Cloud Computing? Large-scale, High-computation Data Clustering Task with PyOMP on single machine and with PySpark on AWS EC2 instances

8.1.2 Wei-Lun Chiu (weilunc)

1. Python DBSCAN Serial implementation & experiments, including grid-based, KD-Tree
2. Python DBSCAN multi-processing implementation & experiments
3. C++ DBSCAN Serial implementation & experiments, including grid-based
4. C++ DBSCAN OpenMP implementation & experiments, including lock-free Union-Find
5. C++ DBSCAN C++ Concurrency API implementation & experiments
6. C++ DBSCAN Work-stealing implementation & experiments, including fined-grain queue lock
7. C++ DBSCAN AVX implementation & experiments
8. Poster: From Python to C++: A Journey to Efficient DBSCAN
9. Proposal: initiate the project framework
10. Milestone report

8.2 Anomaly when running PySpark with 5 nodes

We experienced an odd behavior when executing the same PySpark script with a 5-node clusters. With the same partition function of Eq 1, the data seems to be repartitioned to only 3 of the 5 nodes, as we observed in Fig 13. The issue caused the runtime to be 2.1 hrs. This only experienced on running with 5 nodes. The issue falls beyond the scope of the project.

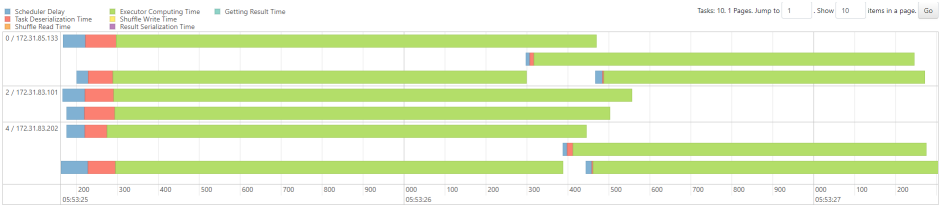


Figure 13: PySpark job executed on 5 executors

8.3 Performance Analysis Report for Perf

The original performance analysis report is Figure 14, and Figure 15.

```
Samples: 347K of event 'cycles:u', Event count (approx.): 282010497999
Overhead Command Shared Object Symbol
29.23% dbscan libc-2.28.so [.] malloc
27.50% dbscan dbscan [.] dist
16.92% dbscan libc-2.28.so [.] _int_free
9.06% dbscan dbscan [.] isConnect
5.84% dbscan libstdc++.so.6.0.28 [.] operator new
3.78% dbscan libc-2.28.so [.] cfree@GLIBC_2.2.5
3.09% dbscan libc-2.28.so [.] __memmove_avx_unaligned_erms
1.53% dbscan libstdc++.so.6.0.28 [.] operator delete@plt
1.12% dbscan dbscan [.] memmove@plt
0.89% dbscan libstdc++.so.6.0.28 [.] malloc@plt
```

Figure 14: Performance report for the DBSCAN algorithm - 1

```
 Performance counter stats for './dbscan 0.2 2':

         87,602.85 msec task-clock:u              #    1.000 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
            11,303      page-faults:u             #    0.129 K/sec
   284,992,059,313      cycles:u                  #    3.253 GHz                      (50.00%)
        13,553,401      stalled-cycles-frontend:u #    0.00% frontend cycles idle     (50.00%)
     4,873,430,201      stalled-cycles-backend:u  #    1.71% backend cycles idle      (50.00%)
   925,491,792,090      instructions:u            #    3.25  insn per cycle
                                                  #    0.01  stalled cycles per insn  (50.00%)
   226,426,881,223      branches:u                # 2584.698 M/sec                    (50.00%)
         1,155,082      branch-misses:u           #    0.00% of all branches          (50.00%)

      87.603927362 seconds time elapsed

      87.006668000 seconds user
       0.022818000 seconds sys
```

Figure 15: Performance report for the DBSCAN algorithm - 2