Parallel BIRCH Clustering

High Performance Computing for Data Science Project 2023/2024

Loris Lorenzini
mat.239944
University of Trento
38123 Povo TN, Italy
loris.lorenzini@studenti.unitn.it

Federico Sentineri
mat.247375
University of Trento
38123 Povo TN, Italy
federico.sentineri@studenti.unitn.it

***Abstract*—Clustering, a fundamental task in exploratory data analysis, involves categorizing data objects based on inherent similarities to form clusters with high intra-group similarity and low inter-group similarity. Despite their utility across diverse applications, clustering algorithms have historically struggled with large datasets due to hardware limitations and inefficiencies in processing extensive data volumes. The BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm addresses these challenges by incrementally clustering incoming multi-dimensional data points, optimizing quality within memory and time constraints. BIRCH stands out for its capability to deliver high-quality clustering with a single data pass, with further refinements possible through additional scans. This paper explores the potential for parallelizing the BIRCH algorithm via the *MPI* framework to enhance its performance further. We provide a comprehensive analysis of our parallelization approach, aiming to improve BIRCH's efficiency in handling large-scale datasets, thereby contributing to the broader goal of optimizing clustering methodologies in resource-constrained environments.**

I. Introduction

Clustering, also known as Cluster analysis, involves the process of categorizing a collection of data objects based on their inherent similarities. The goal is to create clusters, or groups, where the objects within a cluster exhibit greater similarity to one another, as defined by the analyst, than to objects in different clusters [1]. This type of task stands as a fundamental undertaking in exploratory data analysis, serving as a prevalent technique in statistical data analysis with widespread applications across various domains. Its versatility finds expression in numerous fields, including market segmentation, social network analysis, grouping of search results, medical imaging, image segmentation and anomaly detection.

Several approaches to clustering exist [2], each best suited to a particular data distribution. There are four prominent classes of clustering algorithms:

- Centroid-based: This algorithm efficiently organizes data into non-hierarchical clusters. However, it is sensitive to initial conditions and outliers, which may impact its performance.
- Density-based: This algorithm identifies clusters by connecting high-density areas, allowing for accommodation of arbitrary shapes with drawbacks when handling varying densities and high dimensions.
- Distribution-based: This algorithm assumes that data follows specific distributions, such as Gaussian distributions. As the distance from the distribution's center increases, the likelihood of a point belonging to the distribution decreases. It is recommended to choose a different algorithm when the data's distribution is unknown.
- Hierarchical: This algorithm constructs a cluster tree, making it ideal for hierarchical data structures like taxonomies. It provides flexibility in determining the number of clusters by cutting the tree at an appropriate level.

By the late 1990s, regardless of the approach employed, clustering algorithms faced significant challenges in accommodating the need for analyzing increasingly voluminous datasets within the constraints of limited hardware resources. The existing clustering algorithms encountered notable inefficiencies when applied to very large databases, often struggling to address scenarios where a dataset exceeded the capacity of the main memory. Consequently, the maintenance of high clustering quality incurred considerable overhead, given the need to also minimize the cost of additional input/output operations. Moreover, during this period, many algorithms failed to adequately adapt to the scale of the data, treating all data points (or existing clusters) uniformly in each clustering decision. Notably absent was a heuristic weighting mechanism that took into account the distance between the data points, further limiting the algorithms' ability to optimize their performance in the face of extensive and intricate datasets.

In response to the aforementioned challenge, the *BIRCH* (*Balanced Iterative Reducing and Clustering using Hierarchies*) algorithm was proposed in 1996 [3]. Functioning as an efficient and scalable hierarchical clustering method, BIRCH addresses the issue by incrementally and dynamically clustering incoming multi-dimensional metric data points, aiming to achieve optimal clustering quality within the constraints of available resources (i.e., available memory and time constraints). Remarkably, BIRCH is adept at achieving a high-quality clustering outcome with just a single pass through the data, and further enhancements can be realized with a few additional scans [4].

This paper seeks to present an analysis and evaluation of a potential parallelization implementation for the BIRCH algorithm. The objective is to enhance the clustering performance achieved through BIRCH, thereby contributing to overall improvements in algorithmic efficiency.

II. Related Works

This section serves as an introduction to the original Balanced Iterative Reducing and Clustering using Hierarchies algorithm [3] [4], referred to as BIRCH in the following parts of this paper. Specifically, we delve into the serial workflow of the algorithm, exploring the computational intricacies and calculations essential for effectively clustering large and dense datasets. Furthermore, the complexity of BIRCH is discussed.

A. Implementation

The BIRCH algorithm takes as input a set of N data points, each represented as **real-valued** vectors, alongside a user-specified number of clusters denoted as K. At its core, BIRCH operates by incrementally building a hierarchical data structure, known as a *CF* tree (*Clustering Feature* tree), which captures the distribution of data points in the dataset thanks to the concept of Clustering Feature.

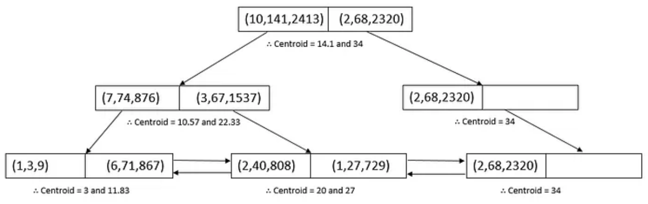
Definition II.1 (Clustering Feature)**.** Given a set of N d-dimensional data points, the Clustering Feature CF of the set is defined as the triple $CF = (N, \overrightarrow{LS}, SS)$, where:

- $\overrightarrow{LS} = \sum_{i=1}^{N} \overrightarrow{X_i}$ is the linear sum.
- $SS = \sum_{i=1}^{N} (\overrightarrow{X_i})^2$ is the square sum of data points.

The *CF* tree efficiently organizes data into clusters by considering their closeness in feature space, all the while ensuring minimal memory usage and computational requirements. This tree structure relies on two main parameters: the branching factor B and the threshold T. Each non-leaf node contains at most B entries, denoted as $[CF_i$,$child_i]$, where $child_i$ points to the ith child node and CF_i represents the clustering feature for the associated subcluster. Leaf nodes, on the other hand, have a maximum of L entries, each in the form of $[CF_i]$. Additionally, leaf nodes are linked together through pointers $prev$ and $next$. The size of the tree depends on the threshold parameter T, since specifies the maximum diameter of subclusters stored at the leaf node, and each node must fit within a page of size P. The values of B and L are determined based on the page size P, offering flexibility for performance optimization. Notably, this structure ensures compact representation of the dataset, as each entry in a leaf node represents a subcluster rather than an individual data point

The algorithm unfolds in four distinct phases, with the second and fourth phases being optional.

1) CF Tree Construction Phase: The algorithm begins by initializing an empty *CF* tree. Then, it iteratively processes each data point in the dataset, updating the *CF* tree accordingly. When a new data point is inserted into the *CF* tree, BIRCH follows a two-step process. First, it traverses the tree to find the leaf node that is closest[1] to the new data point. If



CF tree with a branching factor (B) of 2 and threshold (T) of 5 with data points consisting of a single feature.

Fig. 1: Example of CF tree [5].

the distance to the closest leaf node is less than the threshold T, the algorithm inserts the new data point into that leaf node. However, if the distance exceeds the threshold, indicating that the point is too far from every already existing clusters, a new leaf node is created, consisting only of that single data point. After determining the appropriate leaf node, BIRCH updates the *CF* of the corresponding subcluster with the new data point. If the insertion causes the number of points in the leaf node to exceed the threshold L, BIRCH may split the leaf node into two, redistributing the points between them while ensuring that each resulting subcluster maintains its compact representation through *CF*s.

This iterative insertion and splitting process continues until all data points have been processed and the *CF* tree structure is fully constructed[2]. At this point, the *CF* tree represents a hierarchical clustering of the dataset, with clusters organized at different levels of the tree based on their proximity.

2) CF Tree Refining (Optional): In the second phase, the algorithm systematically examines each leaf entry within the initial CF tree to reconstruct a more compact CF tree. During this process, outliers are eliminated, and densely populated subclusters are merged to form larger ones. This step is considered optional in the original BIRCH presentation as its purpose is solely to enhance the performance and quality of subsequent steps.

3) Global Clustering Phase: In step three, an existing clustering algorithm is utilized to cluster all leaf entries. This is necessary because the fixed size of each node can lead to anomalies with respect to the actual clustering patterns of the data. Specifically, an agglomerative hierarchical clustering algorithm is employed directly on the subclusters represented by their CF vectors. This clustering approach offers flexibility, allowing also users to specify the desired number of clusters or the desired diameter threshold for clusters. The outcome is a set of clusters that capture the major distribution patterns in the data. However, minor and localized inaccuracies may remain, which can be addressed in an optional step four.

4) Cluster Refining Phase (Optional): The fourth step of the algorithm, although optional, involves additional passes

[1] Various distance metrics can be utilized to determine the closest subcluster to a specific point: Euclidean Distance, Manhattan Distance, Average Inter-cluster Distance, Average Intra-cluster Distance and Variance Increase Distance [3] [4].

[2] If the algorithm exhausts memory before completing data scanning (which is rare with modern hardware), it activates a contingency plan. It increases the threshold value and constructs a smaller CF tree by re-inserting leaf entries from the old CF tree. Once all old leaf entries are re-inserted, data scanning and insertion into the new CF tree resumes from the interruption point.

over the data to correct inaccuracies and enhance cluster refinement. The centroids of clusters generated in phase three serve as seeds, and data points are redistributed to their closest seed to form new clusters. This redistribution not only allows points to move between clusters but also ensures that all copies of a particular data point are assigned to the same cluster. Additionally, during this phase, each data point can be labeled with its corresponding cluster, facilitating data point identification within clusters. Moreover, this step offers the option to discard outliers, data points that are significantly distant from their closest seed that can be excluded from the result.

B. Complexity

Let's delve into the CPU cost of Phase 1 (II-A1). Considering memory size M and page size P, the tree's maximum size is $\frac{M}{P}$. To insert a point, one traverses a path from root to leaf, passing through approximately $1+\log_B \frac{M}{P}$ nodes. At each node, B entries are examined, costing proportionally to dimension d. Hence, the total insertion cost for all data points is $O(d*N*B*(1+\log_B \frac{M}{P}))$. If the tree requires rebuilding, the cost is $O(d*\frac{M}{C*d}*B*(1+\log_B \frac{M}{P}))$, where $C*d$ represents the *CF* entry size. The frequency of rebuilding is determined by a threshold heuristic, approximately $\log_2 \frac{N}{N_0}$, where N_0 is the number of data points loaded into memory with a specific threshold T_0. Thus, the total CPU cost for Phase 1 is $O(d*N*B*(1+\log_B \frac{M}{P})+\log_2 \frac{N}{N_0}*d*\frac{M}{C*d}*B*(1+\log_B \frac{M}{P}))$[3]. CPU cost for Phase 2 (II-A2) mirrors Phase 1.

For I/O, Phase 1 entails scanning data once, while Phase 2 doesn't require scanning. However, with outlier-handling and split-delaying, there's a cost associated with writing outliers to disk and reading them back during a rebuild. Despite this, considering disk space and rebuild frequency, Phase 1's I/O cost is comparable to reading the original dataset. Phase 3 (II-A3) has no I/O, and its CPU cost is bounded by a constant depending on input size range and chosen algorithm. Therefore, Phases 1, 2, and 3 scale linearly with N. In Phase 4 (II-A4), each data point is assigned to its cluster, with time proportional to $N*K$. However, advanced nearest neighbor techniques significantly reduce time by only considering nearby cluster centers ($K \sim 1$). Thus, BIRCH's computational complexity is $O(N)$[3][4].

III. Methodology and Implementation

After examining the original version of BIRCH, this section focuses on our parallelized approach. Alongside detailing the parallelized algorithm implemented via the *MPI API* [6], this part analyzes the data dependencies of our solution and elucidates the common structures between the parallelized version of BIRCH with its serial counterpart. Furthermore, this section delves into the PBS directives employed to execute the developed algorithm and the datasets utilized.

[3]Given that $B=\frac{P}{C*d}$, the overall cost can be further simplified to $O(N*\frac{P}{C}*(1+\log_{\frac{P}{C*d}} \frac{M}{P})+\log_2 \frac{N}{N_0}*\frac{M}{C*d}*\frac{P}{C}*(1+\log_{\frac{P}{C*d}} \frac{M}{P}))$.

A. Common Methods

Before discussing the parallel implementation we have proposed, it's essential to identify the common structures of the algorithm that could be used both in sequential and parallel execution.

Intuitively, both approaches replicate all distinctive steps of the BIRCH clustering in the same way, from initializing the *CF* tree, to constructing and refining the *CF* tree (see Pseudocode 1), all the way to outputting information regarding the clustering performed. This is justified by the fact that the parallel version of the algorithm is achieved by creating N processes that basically apply the serial algorithm locally to distinct portions of the user-entereed dataset (more on this in the following).

Algorithm 1 *CF* Tree Construction

```
function TREE_INSERT(tree, data_sample)
    entry ← create_entry(data_sample)
    dont_split ← insert_entry(tree → root,
                              entry)
    if dont_split = $False$ then
        tree_split_root(tree)

function INSERT_ENTRY(node, entry)
    if node has no entries then
        add_new_entry(node → entries, entry)
        return $True$
    closest ← find_closest_entry(node,
                                 entry)
    if closest has at least a child node then
        dont_split ← insert_entry(closest
                                  → child, entry)
        if dont_split = $True$ then
            update_entry(closest, entry)
            return $True$
        else
            split_entry(node, closest)
            if |node → entries| > B[4] then
                return $False$
            else
                if merging refinement is required then
                    merging_refinement(node)
                return $True$
    else if distance(closest, entry) ≤ T[5] then
        update_entry(closest, entry)
        return $True$
    else if |node → entries| < B[4] then
        add_new_entry(node → entries, entry)
        return $True$
    else
        add_new_entry(node → entries, entry)
        return $False$
```

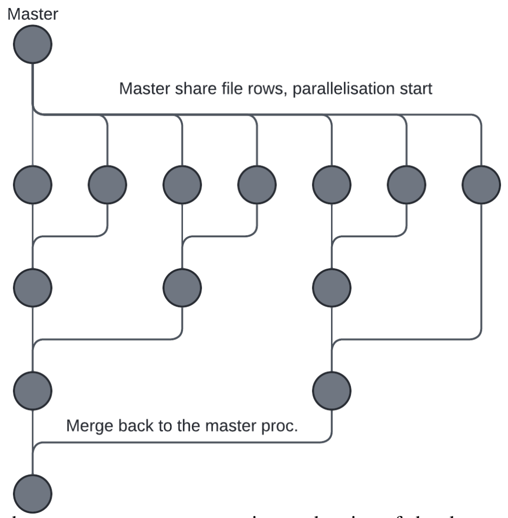
[4]Branching factor.
[5]Threshold.

Obviously, the parallel implementation of the algorithm requires some additional steps, detailed in the continuation of this section. Through those steps, processes get the portion of the dataset to focus on (Pseudocode 2) and the *CF* trees obtained by each process are gradually merged together to create a single one (Pseudocode 3 and Pseudocode 4).

B. Parallelization with MPI

Our parallel approach relies on the *MPI API*. This interface facilitates programming on distributed memory systems, streamlining the parallelization process by allowing each parallel process to operate within its own memory space, independent of others. These characteristics eliminate complexities associated with memory allocation across processes, however, the automated nature of parallelization may introduce some inefficiencies.

As previously mentioned, our parallel algorithm is based on dividing the dataset evenly among *N* processes. Each process then independently applies the serial BIRCH algorithm to generate partial *CF* trees and these partial trees are progressively merged to form the final *CF* tree, encapsulating the resultant clustering (Figure 2). Due to the above workflow, our code heavily relies on two pivotal components: the communication primitives provided by the MPI API for inter-process communication, and the process 0, denoted also as the master, which orchestrates and supports other processes during non-local operation.


Master
Master share file rows, parallelisation start
Merge back to the master proc.

After the master process communicates the size of the dataset, each process applies the sequential version of BIRCH to a specific portion of the dataset. Subsequently, the resulting *CF* trees are merged.

Fig. 2: Workflow of the parallel algorithm.

After starting a new instance of the algorithm, process 0 takes on the responsibility of coordinating the dataset assignment step to all other processes, including itself. To do this, the master determines the number of rows in the .csv file received as input and communicates it to the other processes using MPI's broadcast function, *MPI_Bcast()*. Once this information is obtained, all processes can identify the portion of the dataset on which to focus their work, extract the related data samples and apply BIRCH clustering locally on those samples (see Pseudocode 2).

Algorithm 2 Dataset Partition and BIRCH Local Application

```
1: if this is process 0 then
2:     filesize ← size(dataset.csv)
3: MPI_Bcast(&filesize, 1, MPI_INT, 0,
4:             MPI_COMM_WORLD)
5: stream ← fopen(dataset.csv)
6: count ← 0
7: while count < $\frac{\text{filesize}}{\text{numProc}} * \text{proc\_id}$ do
8:     fgets(line, stream)          ▷ Skip data samples not
9:     count ++             ▷ belonging to process's partition
10: tree ← initialize_new_tree()
11: fgets(line, stream)
12: do
13:     tree_insert(tree, line)  ▷ BIRCH local application
14: while fgets(line, stream) &&
15:         count < $\frac{\text{filesize}}{\text{numProc}} * (\text{proc\_id} + 1)$[6]
16: print_intermediate_output(tree, output_file)
```

Once the various processes have completed applying BIRCH on their respective data portions and have generated the initial intermediate *CF* trees and outputs, the parallel algorithm proceeds to its second phase. This phase focuses on merging together the partial outcomes from the different processes to construct a *CF* tree equivalent to the one obtained by applying the serial algorithm on the entire dataset. During this phase, the objective is to progressively merge the intermediate *CF* trees by pairing trees obtained from distinct processes. Hence, each merge phase requires identifying the process pairs that need to communicate for this operation, along with deciding which process shares its partial results and which integrates the results into its *CF* tree. Process 0 once again takes charge of coordinating these operations. It is tasked with both determining the parallel workflow for the merge operation and assigning roles to the various processes, communicating the schedule via *MPI_Bcast()*. In Pseudocode 3, the code segment that enables the master to comprehend how to manage and coordinate the merging phase is presented.

Once all processes have acquired an understanding of the merging process organization, the actual merging phase of the intermediate results begins. Leveraging the information stored in *senders[]* and *receivers[]*, each process is able to comprehend its role in each merging round. When two processes need to collaborate on merging intermediate results, they rely on the *sendClustersTo()* and *receiveClustersFrom()* methods, based on the MPI *MPISend()* and *MPIRecv()* functions, to send and receive the information necessary for the correct execution of the operations. In detail, the process tasked with

[6]To be precise, condition $\text{count} < \frac{\text{filesize}}{\text{numProc}} * (\text{proc\_id} + 1)$ applies to all processes except the last one, which continues reading the dataset until its end. Thus, even if the dataset cannot be evenly partitioned, all data samples of the dataset are considered.

Algorithm 3 Scheduling Merging of Partial Results

```
1: senders[]
2: receivers[]
3: nMerge ← 0
4: if this is process 0 then
5:     step ← 0
6:     i ← 0
7:     for step; step < ⌈log_2 NumProc⌉; step++ do
8:         incr ← $2^{step+1}$
9:         for i; i + $2^{step}$ < NumProc; i ← i + incr do
10:            senders[nMerge] ← i + $2^{step}$
11:            receivers[nMerge] ← i
12:            nMerge++
13: MPI_Bcast(&nMerge, 1, MPI_INT, 0, MPI_COMM_WORLD)
14: MPI_Bcast(senders, 1, MPI_INT, 0, MPI_COMM_WORLD)
15: MPI_Bcast(receivers, 1, MPI_INT, 0, MPI_COMM_WORLD)
```

providing the partial results sends the following information to the other process via the *sendClustersTo()* method: the number of clusters present in its *CF* tree, the linear sum of the various numerical features of each cluster and the number of data samples present in each cluster[7]. The receiving process, once it receives this information via the *receiveClustersFrom()* method, must update its own *CF* tree by considering all the information related to the received clusters. To do so, the process proceeds to create a new entry for each received cluster, which essentially represents the centroid of all points belonging to a particular cluster, and proceeds to insert them into its own *CF* tree. This phase proceeds iteratively until all merging rounds have been completed, so until process 0 (the master) contains the final *CF* tree, obtained by merging all the *CF* trees initially created by the various processes.

The merging process is also observable in Pseudocode 4.

Algorithm 4 Merging of Partial Results

```
1: i ← 0
2: for i; i < nMerge; i++ do
3:     if senders[i] = proc_id then
4:         c_info ← tree_get_clusters_info(tree)
5:         sendClustersTo(c_info, receivers[i])
6:     else if receivers[i] = proc_id then
7:         c_info ← receiveClustersFrom(senders[i])
8:         n ← 0
9:         for n; n < c_info → nClusters; n++ do
10:            e ← create_entry_from_cluster_info(c_info
11:                                → clusters[n])
12:            tree_insert_entry(tree, e)[8]
13:        update_output_after_merge(tree, senders[i]
14:                           , output_file)
```

[7]All Clustering Features associated with a specific cluster are transmitted during this phase, with the exception of the square sum. This decision was made to minimize the data exchange between two processes when they communicate, as the square sum can be retrieved using the linear sum.

The algorithm described in this part can be found in the corresponding GitHub repository [7]. In the following GitHub repository [8], instead, it is possible to find the code for the serial BIRCH algorithm, which we used as the basis for developing our solution.

C. Data Dependencies

Data dependency analysis in parallel code context is used to identify and manage dependencies between different parts of a program that may affect execution order. Its primary purpose is to ensure that parallel tasks do not interfere with each other, thus preventing race conditions, ensuring data consistency, and optimizing performance. By analyzing these dependencies, it is possible to determine which tasks can safely run in parallel and which must be executed sequentially, enabling efficient and correct parallel execution.

Given the complexity and size of the parallel algorithm we developed, we decided to focus exclusively on analyzing the data dependencies in a crucial part of our BIRCH implementation. Specifically, this section discusses the dependencies within the code responsible for identifying the *CF* entry containing the cluster closest to a given data sample, showed in Listing 1, during the creation of the *CF* Tree[9]. This analyzed code not only plays a fundamental role in the operation of BIRCH but is also particularly suitable for further parallelization through threads, enabling the development of a hybrid solution based on both MPI and *OpenMP* [9].

```c
double distance(Entry* e1, Entry* e2){
    double dist = 0;
    int i;
    for (i = 0; i < e1->dim; ++i){
        double diff = (e1->ls[i] / e1->n) - (e2->ls[i] / e2->n);
        dist += diff * diff;
    }
    return sqrt(dist);
}

Entry* find_closest_entry(Node *node, Entry* entry)
{
    int i;
    double min_dist, curr_dist;
    Entry* curr_entry, closest_entry;
    closest_entry = NULL;
    min_dist = DBL_MAX;
    for (i = 0; i < array_size(node->entries); ++i){
        curr_entry = (Entry*) array_get(node->entries, i);
        curr_dist = node->distance(curr_entry, entry);
        if (curr_dist < min_dist){
            min_dist = curr_dist;
            closest_entry = curr_entry;
        }
    }
    return closest_entry;
}
```

Listing 1: Finding the Closest Entry During CF Tree Creation.

[8]*tree_insert_entry()* is similar to *tree_insert()* (Pseudocode 1) but works directly with an entry, while *tree_insert()* takes a data sample and constructs an entry during *CF* tree creation.

[9]Please note that the *find_closest_entry*() method, depicted in Listing 1, is referenced in Pseudocode 1, which indeed outlines the logic behind constructing *CF* trees.

The data dependencies for the proposed piece of code are presented in Table I. Starting from the *distance()* method, the only data dependency that emerges is with the variable *dist*. Specifically, on line 6, there is a flow dependency due to the accumulation operation *dist += diff * diff*, as this variable is read and updated at each iteration. In an uncontrolled parallelism scenario, this behavior can cause errors. For example, due to parallelism, during iteration *i+1*, instead of reading the value of *dist* calculated at iteration *i* to update the distance, the value obtained at iteration *i-1* might be read, thus compromising the correctness of the final *dist* value. In contrast, there is no dependency for the variable *diff*, as it is calculated independently for each dimension at each iteration. Therefore, the calculation of *diff* in one iteration (*i*) does not depend on the calculation of *diff* in another iteration (*i+1*).

Focusing on the *find_closest_entry()* method reveals an higher number of data dependencies, primarily because the analyzed code was not developed for use in multi-threaded parallelism scenarios. Specifically, for all four memory locations used in this method, there is an output dependence issue. When threads are used, a race condition arises for all these locations since they are shared across all iterations. Consequently, two or more threads could attempt to modify one of these four variables simultaneously, leading to inconsistencies or incorrect results. Furthermore, in concluding the analysis of this method's dependencies, the *min_dist* variable presents an additional data dependency, specifically an anti-dependence. Each iteration reads (line 21) and potentially writes (line 22) to *min_dist* to advance the search for the closest entry. However, because these operations are not atomic, without proper precautions, different threads working on different iterations of the for loop could read an outdated value of *min_dist* and write inconsistent values for both *min_dist* and *closest_entry*, compromising the correctness of the final results.

Memory Location	Earlier Statement			Later Statement			Loop Carried?	Kind of Dataflow
	Line	Iteration	Access	Line	Iteration	Access		
dist	6	i	write	6	i+1	read	yes	flow
min_dist	21	i	read	22	i	write	no	anti
min_dist	22	i	write	22	i+1	write	yes	output
curr_dist	20	i	write	20	i+1	write	yes	output
curr_entry	19	i	write	19	i+1	write	yes	output
closest_entry	23	i	write	23	i+1	write	yes	output

TABLE I: Data Dependencies.

To address the highlighted dependencies and ensure compatibility with multithreaded execution, leveraging the pragma directives provided by OpenMP becomes necessary. Specifically, to handle the dependency of the variable *dist* within the *distance()* method, we can utilize a reduction operation. This operation efficiently accumulates results from different iterations in a parallel-friendly manner. This approach capitalizes on the associativity of the operation *dist += diff * diff*, which enables the distribution of *dist* computation among various threads and the combination of the results only at the conclusion.

Regarding the data dependencies identified in the *find_closest_entry()* method, two changes are required. Before initiating the for loop, the scopes of the four manipulated variables need to be defined. *curr_entry* and *curr_dist* must be privatized for each thread since they serve functions exclusively local to individual threads. This ensures that each thread possesses its own copy, thereby eliminating output dependence for those variables. Conversely, *min_dist* and *closest_entry* are defined as shared among threads due to their global functionality. Consequently, it is imperative to establish a critical section in the part of the code where it is assessed whether a new minimum distance has been found and the information about the current minimum distance and the closest entry are updated. The critical section ensures that only one thread at a time can execute that block of code, thus preventing race conditions when checking and updating *min_dist* and *closest_entry*.

The discussed changes are illustrated practically in Listing 2.

```
1  double distance(Entry* e1, Entry* e2){
2      double dist = 0;
3      int i;
4      #pragma omp parallel for reduction(+:dist)
5      for (i = 0; i < e1->dim; ++i){
6          double diff = (e1->ls[i] / e1->n) - (e2->ls[
   i] / e2->n);
7          dist += diff * diff;
8      }
9      return sqrt(dist);
10 }
11
12 Entry* find_closest_entry(Node *node, Entry* entry)
13 {
14     int i;
15     double min_dist, curr_dist;
16     Entry* curr_entry, closest_entry;
17     closest_entry = NULL;
18     min_dist = DBL_MAX;
19     #pragma omp parallel for private(curr_entry,
   curr_dist) shared(min_dist, closest_entry)
20     for (i = 0; i < array_size(node->entries); ++i){
21         curr_entry = (Entry*) array_get(node->
   entries, i);
22         curr_dist = node->distance(curr_entry, entry
   );
23         #pragma omp critical
24         {
25             if (curr_dist < min_dist){
26                 min_dist = curr_dist;
27                 closest_entry = curr_entry;
28             }
29         }
30     }
31     return closest_entry;
32 }
```

Listing 2: Finding the Closest Entry During *CF* Tree Creation - Multithread Compliant.

D. PBS Directives

To execute the parallel version of BIRCH on the cluster, whose specification can be found in IV-A, it necessary to run the bash file presented in Listing 3.

As can be seen from the PBS directives, for optimal performance in all tests, a single node with $\$NUMCORE$ cores was exclusively allocated. This decision was made following observed inconsistencies in execution times when utilizing 64 cores spread across multiple nodes, likely attributable to communication and latency issues. Each execution was assigned a maximum runtime of one hour.

```bash
#!/bin/bash
#PBS -l select=1:ncpus=$NUMCORE:mem=2gb -l place=
    pack:excl
# Set max execution time
#PBS -l walltime=1:00:00
# Set execution queue
#PBS -q short_cpuQ
# Load the required module
module load mpich-3.2
# Run algorithm
mpirun.actual -n $NUMCORE ./BIRCH_Clustering/
    birch_clustering $BRANCHING_F $THRESHOLD
    $REFINEMENT $DATASET $DELIMITER $LAST_COL
```

Listing 3: PBS Job Submission.

The algorithm expects six input arguments:

- $\$BRANCHING\_F$: an integer representing the branching factor;
- $\$THRESHOLD$: a floating-point value representing the threshold for consideration;
- $\$REFINEMENT$: a binary value indicating whether to apply the merging refinement (1 for yes, 0 for no);
- $\$DATASET$: the name of the CSV file containing the input dataset;
- $\$DELIMITER$: the delimiter used in the dataset (e.g., ”,”);
- $\$LAST\_COL$: a binary value indicating whether to ignore the last column of the dataset (1 for yes, 0 for no).

E. Benchmark Datasets

Both the serial and parallel algorithms were evaluated using five different artificial datasets generated using the *make_blobs()* method from *scikit-learn* [10]. Each dataset consists of shuffled data samples characterized by 20 distinct features and grouped into 15 different clusters[10]. To enable an optimal comparison of the performance of the two BIRCH approaches analyzed in this paper, the datasets were created with an increasing number of data samples. Specifically, the five datasets used comprised 1.000.000, 2.000.000, 4.000.000, 8.000.000 and 16.000.000 elements, respectively.

```python
import argparse
import numpy as np
import pandas as pd

from sklearn.datasets import make_blobs

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="
    Generator of clusterized artificial data samples
    ")
    parser.add_argument("--seed",type=int,...)
    parser.add_argument("--n_samples",type=int,...)
    parser.add_argument("--n_features",type=int,...)
    parser.add_argument("--n_clusters",type=int,...)
    parser.add_argument("--std",type=float,...)

    args = parser.parse_args()

np.random.seed(args.seed)

X, labels_true = make_blobs(n_samples=args.n_samples
    , n_features=args.n_features, centers=args.
    n_clusters, cluster_std=args.std, shuffle=True)
labels_true = labels_true.reshape(args.n_samples,1)

artificial_dataset = pd.DataFrame(np.append(X,
    labels_true,axis=1))
artificial_dataset[args.n_features] =
    artificial_dataset[args.n_features].astype(int)
artificial_dataset.to_csv("ArtificialDataset.csv",
    index=False, header=False)
```

Listing 4: Python Script for Artificial Dataset Generation.

```
$ python3 artificial_generation.py --seed=85 --
  n_samples=16000000 --n_features=20 --n_clusters
  =15 --std=0.75
```

Listing 5: Example of Artificial Dataset Generation.

[10]Clusters standard deviation set to 0.75

IV. Experimental Evaluation

This section presents the experimental results obtained from the implementation of our parallel algorithm. The main objective is to evaluate the performance and efficacy of the proposed algorithm. By examining various metrics and comparing the results with those obtained with a serial version of BIRCH, this part aims to provide insights into the algorithm's strengths, weaknesses and overall effectiveness.

A. Hardware

To evaluate the performance of our algorithm, we leveraged on the computing resources provided by the *High-Performance Computing* (HPC) cluster at the University of Trento, which operates on the *Altaris PBS Professional* cluster management software.

Outlined below are the primary specifications of the cluster:

- Operating System of the nodes: *Linux CentOS7*
- Number of nodes: 126
- CPU cores: 6092
- CUDA cores: 37.376
- RAM: 53 TB
- Connectivity: The nodes are interconnected via a 10Gb/s network. Additionally, select nodes are equipped with high-speed connectivity options, including *Infiniband* at 40Gb/s and *Omnipath* at 100Gb/s.

B. Experimental Setup and Parameters

The parallel algorithm we have developed underwent testing using the artificial datasets outlined in Section III-E. Those datasets consist of an increasing number of data samples, segmented into 15 distinct clusters and defined by 20 different features. Throughout the testing phase, we executed the algorithm on these datasets employing six different core counts (2, 4, 8, 16, 32 and 64). For every run, we configured the algorithm parameters as follows:

- $\$BRANCHING\_F$: 100;
- $\$THRESHOLD$: 7.0;
- $\$REFINEMENT$: 1;
- $\$DELIMITER$: ”,”;
- $\$LAST\_COL$: 1.

C. MPI Parallelization Results

Table II presents the results of both the parallel and serial algorithms for each execution conducted during the testing phase. As clearly shown, the table correlates the number of parallel processes used during the tests with the average execution time required to perform clustering on datasets of varying sizes. Therefore, our performance analysis focuses exclusively on execution time, as it allows us to accurately assess the benefits and drawbacks of adding parallelization to the BIRCH clustering case study.

N_{cores} \ $N_{samples}$	**1 M**	**2 M**	**4 M**	**8 M**	**16 M**
1	23.13	46.34	96.19	203.43	437.44
2	12.52	25.36	50.95	103.16	204.04
4	6.16	11.81	26.80	50.39	107.63
8	2.93	6.58	15.50	26.38	64.19
16	2.12	4.13	8.36	16.90	34.11
32	2.11	3.78	7.37	14.28	32.70
64	2.11	3.76	7.37	13.84	30.83

TABLE II: Average Execution Time (s).

Table II already reveals several key aspects of the impact that the parallel algorithm has on performing clustering tasks. Firstly, as expected, there is a clear trend: increasing the number of cores leads to a reduction in execution time across all dataset sizes. This clearly demonstrates the potential of parallelism in such algorithms. The collected data also emphasizes the efficiency of parallel processing for larger datasets. For instance, the execution time for the largest dataset drops dramatically from 437.44 seconds on a single core to just 30.83 seconds on 64 cores. This substantial decrease underscores again the effectiveness of utilizing multiple cores to handle extensive computational tasks. However, the benefits of adding more cores are subject to diminishing returns. In fact, while the initial increase from 1 to 8 cores seems always to result in a significant speedup, further increases beyond 16 cores yield progressively smaller improvements. This may indicate an optimal range for core usage, probably based on dataset size, where the balance between performance gain and resource allocation is most efficient.

Speaking of dataset size, it is interesting to note that for a fixed number of cores doubling the dataset size almost doubles the execution time. This is, of course, consistent with what was mentioned about the algorithm's complexity in Section II-B, namely, that the computational complexity of the algorithm grows linearly with the number of data samples to be handled.

In summary, the table highlights significant advantages derived from the application of parallelization on the BIRCH algorithm, especially for large datasets. The algorithm's scalability is readily apparent; as the number of cores increases, there is a marked reduction in execution time. However, the table seems also to reveal some practical limits of the scalability of the algorithm, showing diminishing returns as the number of cores increases. For this reason, it is important to have strong evaluation metrics to clearly understand how effectively the parallelism speeds up the process.

D. Algorithm Evaluations

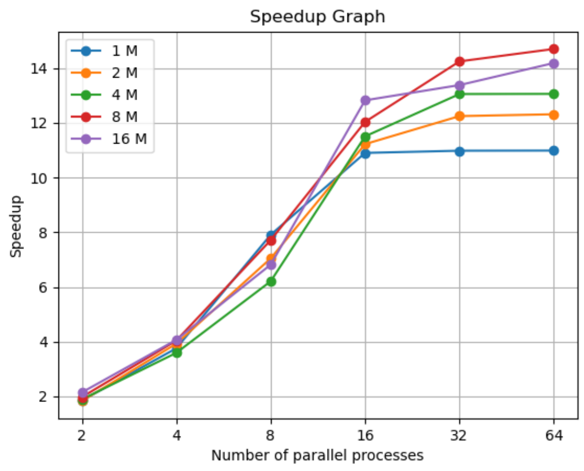
To better understand and evaluate the performance of the parallel algorithm, it is appropriate to define metrics that allow for a more detailed visualization of the trends emerging from the average execution times recorded during the tests. In this section, the following metrics will be analyzed:

- **Speedup** $S = \frac{T_{serial}}{T_{parallel}}$: Speedup represents the ratio between the execution time of the serial program and that of the parallel program. Ideally, the speedup should be linear with respect to the number of cores used, indicating a proportional improvement in performance.
- **Efficiency** $E = \frac{S}{N_{core}}$: Efficiency measures how effectively the cores are utilized by the parallel algorithm. High efficiency indicates that the algorithm makes good use of the available resources and, ideally, should remain constant.
- **Scalability**: the scalability of the algorithm is analyzed in two ways:
  - *Strong scalability*: measures how efficiency varies while keeping the problem size constant as the number of cores increases. An algorithm is defined as strongly scalable if efficiency remains constant in the described situation.
  - *Weak scalability*: evaluates how efficiency varies while keeping the workload per core constant, increasing the total number of cores and the problem size proportionally. In this case as well, if efficiency remains constant, the algorithm is considered weakly scalable.

These metrics will provide a detailed view of the parallel algorithm's performance, allowing for the identification of strengths and areas for improvement.

1) Speedup: Observing the graph in Figure 3, several key insights emerge. First, there is a general trend of increasing speedup as the number of parallel processes increases. This is consistent with expectations, as parallelizing computations should lead to performance gains by distributing the workload across multiple processors. However, this trend does not continue indefinitely, and the speedup gains begin to plateau beyond a certain point for each dataset size. This plateauing effect is likely due to the overhead associated with managing parallel tasks outweighing the benefits of additional parallelism.

For the smallest dataset, the speedup increases steadily up to 16 parallel processes, reaching a factor of approximately 12. Beyond that point, the speedup plateaus, indicating that adding more processes does not significantly improve performance. The 2M dataset follows a similar pattern, but with a slightly higher initial speedup and a plateau beginning around 16 parallel processes as well. The speedup for the 4M dataset exhibits a more pronounced increase, before leveling off. The larger datasets, 8M and 16M, show even more dramatic improvements. From these observations, it is clear that larger datasets benefit more from parallelization, achieving higher speedup factors and sustaining them across more processes. This is likely due to the increased computational load, which makes the overhead of parallel management relatively smaller. Conversely, smaller datasets reach their parallel efficiency limits more quickly, as the overhead becomes a more significant factor in the overall performance. Consequently, the differences in speedup across dataset sizes highlight the importance of tailoring parallel processing strategies to the specific workload, as supposed in the previous part of this section (IV-C). For smaller datasets, fewer parallel processes may be optimal, whereas larger datasets can leverage more extensive parallelization to achieve significant performance improvements.



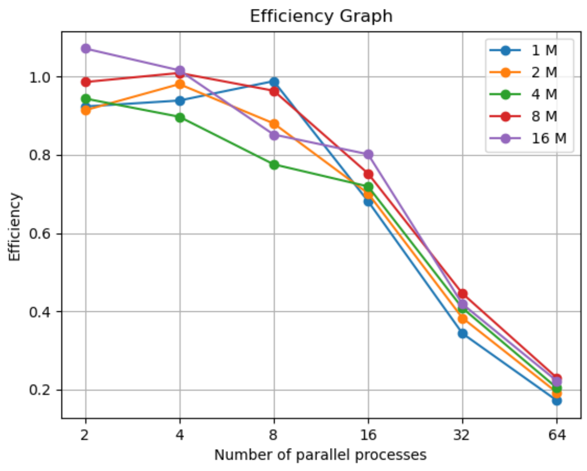
As expected, increasing the number of parallel processes significantly improves speedup. However, this trend plateaus beyond a certain point due to managing parallel processes' overhead. Smaller datasets show lower returns with an high number of processes, while larger ones benefit more, sustaining higher speedup factors.

Fig. 3: Speedup Performance of the Parallel Execution.

To conclude the speedup analysis, it is noteworthy that the graph indicates a slightly higher average speedup for the dataset composed of 8 million elements compared to the dataset with 16 million elements, especially when considering parallel executions with 32 and 64 distinct processes. However, this result is likely influenced more by the fluctuating times recorded during the tests, caused by latency and cluster memory allocation issues, than by any particular characteristic of our proposed solution.

2) Efficiency: Now, let's shift our focus to the efficiency graph depicted in Figure 4. As depicted, initially, with a smaller number of parallel processes (2, 4, or 8), the efficiency remains relatively high across all datasets. This observation indicates that BIRCH can effectively leverage parallel processing at lower scales without incurring significant overhead. However, as the number of parallel processes increases to 16 and beyond, a noticeable decline in efficiency occurs across all datasets. This decline becomes more pronounced with higher degrees of parallelism, particularly evident when parallelization involves 32 or 64 processes. For instance, in the dataset composed of 1 million elements, there is a steep drop in efficiency, falling below 0.5 at 32 processes and further plummeting close to 0.2 at 64 processes.

It is noteworthy that the algorithm maintains slightly higher efficiency when handling the two largest datasets compared to the others at higher levels of parallelism. This observation suggests, once again, that larger datasets may derive more benefit from parallel processing up to a certain threshold before overhead and synchronization costs become dominant. In our opinion, this phenomenon can be attributed to the inherent characteristics of larger datasets, which provide more work per process, thus reducing the relative overhead associated with synchronization and communication between processes[11].



With a smaller number of parallel processes (2, 4, or 8), the efficiency remains relatively high. As the number of parallel processes increases to 16 and beyond, a noticeable decline in efficiency occurs across all datasets, particularly evident when parallelization involves 32 or 64 processes. The algorithm maintains slightly higher efficiency when handling the two largest datasets compared to the others at higher levels of parallelism.

Fig. 4: Efficiency Performance of the Parallel Execution.

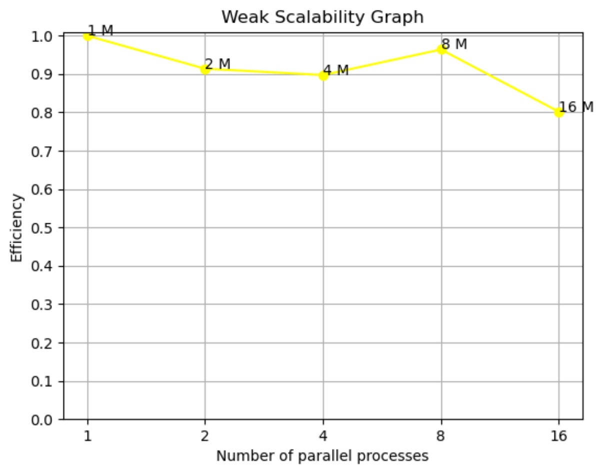
Therefore, the results obtained from this analysis reinforce the idea that for our parallel version of BIRCH, there exists an optimal range of parallelism that balances performance gains with overhead costs. Based on our experiments, this range appears to lie within 2 to 16 parallel processes when handling datasets consisting of tens of millions of elements, beyond which the efficiency losses outweigh the benefits of additional parallelism.

3) Scalability: From the graph in Figure 4, it is also possible to evaluate the strong scalability of the parallel algorithm we developed. In fact, as requested by the definition of strong scalability, each line within the diagram illustrates the trend of parallelization efficiency as the number of processes varies, while maintaining a constant problem size[12]. As previously discussed, the efficiency does not remain stable

[11]Specifically, most of the overhead arises when two processes need to communicate in order to merge their respective partial *CF* trees, owing to the substantial volume of data involved in the process (number of identified clusters, linear sum of all features within each cluster and number of data samples present in each cluster).

[12]In this case the size of the dataset utilized.

once the number of processes increases beyond 8; instead, it consistently diminishes across all five scenarios, with a particularly pronounced decline once surpassing 16 parallel processes. Hence, our proposed solution does not exhibit strong scalability.



The efficiency remains around 90% in all cases, except for the clustering of a dataset with 16 million elements using 16 processes, where it reaches approximately 80%. This indicates that the algorithm exhibits weak scalability.

Fig. 5: Weak Scalability Evaluation.

In Figure 5 is shown the graph upon which the weak scalability analysis is based. Before delving into this analysis, it's crucial to note that executions involving 32 and 64 parallel processes were omitted. This exclusion stems from the inability to utilize datasets comprising 32 million and 64 million elements during testing. Consequently, including these datasets would have compromised the adherence to the definition of weak scalability, since involves scrutinizing efficiency in correlation with proportional variations in both problem size and the number of processes employed.

The graph reveals evident differences compared to the previous analysis on strong scalability. Despite proportional variations in the problem size and the number of parallel processes used, the measured efficiency remains essentially constant, fluctuating around 90%. The only exception is the clustering of a dataset with 16 million elements, where parallelization with 16 processes achieves an efficiency of approximately 80%. This analysis suggests that the algorithm provides weak scalability, particularly considering that the results are also influenced by the fact that the execution times considered are averages from all the runs conducted during the testing phase and by occasional overheads not strictly related to the algorithm, but rather to the hardware on which the tests were performed.

V. Conclusion

This paper thoroughly explores, explains, and discusses a potential parallelization strategy using the MPI framework for the BIRCH algorithm. Our goal was to leverage modern computational resources to achieve faster and more efficient clustering of numerical datasets.

Our analysis demonstrates that the proposed parallelization approach significantly enhances the performance of BIRCH, ensuring both good speedup relative to the serial implementation and weak scalability. Although the algorithm does not provide strong scalability and levels off when using a large number of parallel processes relative to the dataset size, we believe our solution represents a valuable tool for clustering large datasets.

VI. Future Works

In our opinion, the parallel version of the BIRCH algorithm is already an effective tool for clustering large datasets. However, there are several areas for improvement and extension. One significant enhancement could be integrating a hybrid approach combining MPI and threads (OpenMP). Initially, we explored this solution, but due to performance concerns with applying threads in the nearest *CF* entry search discussed in Section III-C, we decided not to pursue also a hybrid implementation. Nonetheless, a well-designed hybrid solution could further improve efficiency by better leveraging multicore systems.

Moreover, our experiments have been limited to small datasets. Therefore, testing the algorithm on larger datasets is essential to better assess its scalability and performance in more realistic scenarios with the presence of high data volumes.

To conclude, exploring alternative approaches to improve the performance reported in this work would be beneficial. In particular, reducing the overhead associated with the communication of partial results between processes during the merge phase, in our opinion the main bottleneck of the algorithm, could significantly reduce latency and improve the overall performance of the parallel BIRCH.

References

[1] Wikipedia, Feb 2024. [Online]. Available: https://en.wikipedia.org/wiki/Cluster_analysis

[2] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, p. 165–193, 2015. [Online]. Available: https://doi.org/10.1007/s40745-015-0040-1

[3] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," *SIGMOD Rec.*, vol. 25, no. 2, p. 103–114, Jun 1996. [Online]. Available: https://doi.org/10.1145/235968.233324

[4] ——, "Birch: A new data clustering algorithm and its applications," *Data Mining and Knowledge Discovery*, vol. 1, p. 141–182, 1997. [Online]. Available: https://doi.org/10.1023/A:1009783824328

[5] medium, Feb 2023. [Online]. Available: https://medium.com/@vipulddalal/birch-algorithm-with-working-example-a7b8fe047bd4

[6] M. P. I. Forum, "Mpi: A message-passing interface standard," *Technical Report*, 1994. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[7] L. Lorenzini and F. Sentineri, 2023. [Online]. Available: https://github.com/FedericoSentineri00/BIRCH_Clustering

[8] D. Monteiro, 2022. [Online]. Available: https://github.com/douglas444/birch

[9] R. Chandra, R. Menon, L. Dagum, D. K. abd Dror Maydan, and J. McDonald, "Parallel programming in openmp," *Morgan Kaufmann*, Oct. 2000.

[10] scikit learn, June 2024. [Online]. Available: https://scikit-learn.org/stable/