Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Parallel Flexible Clustering

Edoardo Pastorino

Master Thesis

Università di Genova, DIBRIS Via Dodecaneso, 35, 16146 Genova GE, Italy
https://www.dibris.unige.it/

Università
di Genova

MSc Computer Science
Data-centric Computing

Parallel Flexible Clustering

Edoardo Pastorino

Supervisors: **Matteo Dell'Amico**, **Daniele D'Agostino**
Reviewer: **Vito Paolo Pastore**

November, 2023

Abstract

The thesis presents the parallelization of a state-of-the art clustering algorithm, the FISHDBC. This target has been achieved by improving the creation of the main data structures and components of the algorithm: the HNSW, a graph-based data structure used in approximate nearest neighbor search; the MST, a tree that spans all the vertices in the graph while minimizing the total weight of the edges; the HDBSCAN clustering, designed to perform robust clustering of data points based on their density. My contribution is based on a lock-free strategy parallel implementation with shared memory, made feasible because FISHDBC provides an approximated solution, and provides good performance figures. It is worth to note that the Parallel Flexible Clustering algorithm is completely written in Python, without dependencies to other languages. This represents an important feature making it user-friendly and highly customizable, considering that user defined distance metrics, for computing similarity among data, are mostly written in this language.

Table of Contents

Chapter 1

Introduction

In exploratory data analysis (EDA), data are often large, complex, and arrive in a streaming fashion, therefore it is not simple to manage these data for finding useful insights. Clustering, however, is an important tool for EDA, because it summarizes datasets, making them more amenable to human analysis, by grouping similar items. Data can be complex because of heterogeneity: consider, e.g., a database of user data as diverse as timestamps, IP addresses, user-generated text, geolocation information, etc.

Clustering helps a lot to deal with these heterogeneous data: it identifies groups of data points that share similar patterns or characteristics, regardless of the variable types; it can handle both numerical and categorical data, making them suitable for heterogeneous datasets; it also can be used to reduce dimensionality by grouping similar data points together, so you can focus on the essential characteristics of the data, making it more manageable for analysis; Clustering can also handle missing data effectively. In fact, when dealing with heterogeneous data, missing values may be present in different variables.

Clustering structure can be complex as well, involving clusters within clusters. Complexity requires clustering algorithms that are flexible, in the sense that they can deal with arbitrarily complex data, and are able to discover hierarchical clusters. Large datasets call for scalable solutions, and streaming data benefits from incremental approaches where the clustering can be updated cheaply as new data items arrive. In addition, it is desirable to distinguish signal from noise with algorithms that do not fit isolated data items into clusters.

But what is clustering? It is the main task in the exploratory data analysis field and it is a kind of unsupervised learning method. Unsupervised learning refers to algorithms that learn patterns from unlabeled data, in which you have the input data, but you don't know the output a priori. On the contrary, in the case of supervised machine learning methods, the known association between input and output is the guide for the algorithm. Clustering

can also be explained as the task of dividing the data points into a number of groups such that the data points in the same groups are more similar between them and dissimilar to the data points in other groups. Actually, there is a kind of supervision due to the usage of a distance metric applied for establishing the similarity between the data and for creating the final clusters.

A good method to perform clustering, that is indeed able to help us to deal with heterogeneous data providing good features, is the FISHDBC algorithm. FISHDBC acronym refers to the namesake Dell'Amico's algorithm and it stands for Flexible Incremental Scalable Hierarchical Density Based Clustering. In a few words, it is a clustering algorithm that, for obtaining the final result, ensures all the aforementioned good properties. The FISHDBC clustering algorithm is based on three main concepts:

- **The HNSW graph**, which stands for Hierarchical Navigable Small World, is a graph-based data structure used in approximate nearest neighbor search. It is designed to efficiently find approximate nearest neighbors in high-dimensional spaces. HNSW builds on the principles of the "small world" network structure and hierarchical organization to achieve fast search times.
- **The MST**, which stands for Minimum Spanning Tree, is a subset of the edges of an undirected, connected graph, that connects all the vertices together with the minimum possible total edge weight. In other words, an MST is a tree that spans all the vertices in the graph while minimizing the total weight of the edges.
- **The HDBSCAN** (Hierarchical Density-Based Spatial Clustering of Applications with Noise) is an extension of the traditional DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm, designed to perform robust clustering of data points based on their density in high-dimensional spaces.

The main goal of the thesis is to implement a parallel version of the FISHDBC algorithm, taking inspiration from the original Dell'Amico's implementation. The main question, now, is why did we pose the problem of paralleling it? We wanted to parallelize such an algorithm, at first simply because we desired to improve the running time of the algorithm to obtain an even faster version. Additionally, we implement the concurrent methodology also for showing the possibility of reaching a very nice performance of a complex algorithm with only the usage of the Python language and environment, proving the fact that, even if better languages exist, in Python we can implement mechanism to get great time results. Last but not least, with such multi-process implementation we show that the FISHDBC is not a type of algorithm that lends itself to the usual types of parallelization, such as partitioning, hence the direction is not trivial, in fact we had to resort to parallelize the work inside the same machine, sharing what we need thanks to the shared memory, therefore using a different paradigm.

The structure of the thesis is the following:

After a careful review of some existing clustering approaches and HNSW creation methods (some of them are also parallel procedures) found in the literature, exposed in the chapter 2, we focused our efforts to study and understand the main concepts on which the FISHDBC is based, as explained in the chapter 3. These two chapters are fundamental to understand well in which scenario the algorithm is inserted and what are the data structure and methods that it inherits to compute its solution.

In the chapter 4 we analyze the original single process FISHDBC in order to have a complete view of the algorithm and to figure out what are the main parts to be improved in terms of performance. After a careful study and analysis, the main bottlenecks of such algorithms emerged: the HNSW construction and MST creation. Before digging into the implementation details, we wanted to offer an overview of some ways and mechanisms that could be used to speed-up the running time, or in general to write efficient code in Python for what concerns the performance. The chapter 5, in fact, elucidates these high-performance computing methods, including some basic instructions and explanations to help the user choose the most suitable technique for his/her use case.

The chapter 6, after a general explanation of the different methods that can be used to perform program profiling, points out what are the main hot-spots. In general, this section informs us about the time distribution of the algorithm's execution over blob and text data set and it could be very useful for orienting the next steps of the work.

The real implementation of our parallel FISHDBC version is deeply analyzed in the chapter 7. This section of the thesis allows the user to comprehend how the different parts of the concurrent algorithm are written in Python, starting from the parallel creation of the HNSW until the MST computation.

Looking at the implementation, the key aspect that emerged in this chapter is the possibility of obtaining a quick parallel implementation, since no blocking operations are performed during the execution of processes, which however maintains a very nice result's accuracy, obviously not perfect, since we are talking about an approximated solution, that is not already 100% accurate by definition. Therefore, if we do not lose a lot of accuracy, while we save a lot of running time, it could be great to use such an idea. Additionally, analyzing the implementation it turns out that in this specific case, we cannot distribute the computation over many machines, but instead, we push the single machine to the limit using parallelization. We cannot split the space of the problem because the input data could have a free format, so we can use, together with many processes, shared data structures to find our solution.

The chapter 8 shows the experimental results and an evaluation of the effective improvements. In this portion of the thesis, many timing results are shown, proving how much is performance enhancement brought by our implementation, with respect to the original ver-

sion, for the different parts of the algorithm (HNSW, MST, and the complete FISHDBC). We want also to point out what are the differences between our parallel HNSW and the state-of-the-art approach, the Hnswlib, to highlight why our version is better than the library and when it is true the opposite. This section demonstrates also what is the speed-up factor based on the different numbers of used cores. In this experimental results chapter, also the accuracy test results are illustrated, confirming all the discussion about the avoidance of sync usage, since they are very similar to the reference ground truth, used for comparison.

At the end of the document, in the chapter 9, we collect all the results and observations to highlight the good aspects of our work and, of course, mentioning all the drawbacks and the possible future upgrades. The final message that we want to send is that it is possible to have a good only Python-dependent fast parallel implementation, very easy to use, that allows to have an approximated, but nice solution, without the effort of thinking and implementing a synchronization system.

Chapter 2

Related Works

The preexisting FISHDBC algorithm, and consequently my new parallel version of the same, could fit in the very general clustering problem, since in the end what the FISHDBC algorithm wants to perform is to separate the input data in different output clusters, based on a certain dissimilarity function. It uses, to compute the clustering result, three main important algorithms and data structures, i.e. the Hierarchical Navigable Small World Network (HNSW), the Minimum Spanning Tree (MST), and the final Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN). We will see more in detail what are these data structures and approaches and how they are used inside the overall procedure in the chapter 3, now we will focus on what are some existing clustering alternatives and HNSW construction methods.

2.1 Clustering Approaches

The cluster analysis can be done with many algorithms that could differ a lot between them, in terms of cluster definition and in terms of optimal way to find clusters. The appropriate clustering method, and consequently the right distance function to be used, strictly depends on the type of data set it will be dealt with. To better understand how to collocate the FISHDBC clustering algorithm, let's focus us on some of these existing different clustering approaches, since several algorithms have a subset of the desirable properties mentioned in the chapter 1, even though no other algorithm embodies at once all the properties that FISHDBC satisfies, being flexible, incremental, scalable, and providing hierarchical density-based clustering:

- **Relational and Spectral Clustering**. The algorithms that belong to this class of clustering methods take as input a matrix D, containing all the pairwise distances of

the input data points. Among these approaches, some are specialized to deal with arbitrary distance functions. The problem here is that they are not scalable, because computing the pairwise distance matrix requires a computational cost of $O(n^2)$. Spectral clustering approaches, instead, could be faster thanks to the Nystrom method ([FBCM04]), a classical method for the solution of the integral eigenvalue problem. In short, with Nystrom, the approximation works by first solving the grouping problem for a small random subset of data (like pixels) and then extrapolating this solution to the full set of, for instance, pixels in the image. This provides the flexibility of pairwise grouping with a computational complexity comparable to that of central grouping: rather than compare all pixels to a set of cluster centers, we compare them to a small set of randomly chosen samples. The approach is simple and has the appealing characteristic that, for a given number of sample points, its complexity scales linearly with the resolution of the image. But spectral clustering will produce a poor approximation of the nodes' neighborhood and for this reason, they are not suitable for a density-based approach like the FISHDBC.

- **Density Based Clustering**, the basic idea behind the density-based clustering approach is derived from a human-intuitive clustering method. For instance, one can easily identify clusters along with several points of noise, because of the differences in the density of points, by simply looking at them. The main existing method part of the density-based approaches is called the DBSCN (Density-Based Spatial Clustering and Application with Noise), which can be used to identify clusters of any shape in a data set containing noise and outliers. It requires only one parameter and supports the user in determining an appropriate value for such input. Additionally, it is efficient even for large spatial databases ([EKSX96]). The DBSCAN approach was generalized to deal with arbitrary input data thanks to the GDBSCAN, in which the clusters are defined as connected dense areas, that are (given an item's neighborhood and given a distance function to determine such neighborhood) areas in which all the neighborhood of the points that belong to them contain, at least, a specific number of elements (usually called MinPts). With GDBSCAN, instead of simply counting the items in the neighborhood of a point, we can use other measures, e.g., considering the non-spatial attributes, thus, such a method can cluster point objects as well as spatially extended objects according to both, their spatial and their non-spatial attributes ([SEK98]). However, for density-based clustering, the complexity remains $O(n^2)$ and to reduce it we need to use some strategies like indexing structure or specialized filter functions based on the used distance metric. With the FISHDBC, there is no necessity to implement such practices to improve the algorithm's complexity. Another approach belonging to this class of clustering is the NG-DBSCAN, a distributed approximate DBSCAN implementation that discovers neighbors in arbitrary spaces with an approach inspired by the NN-Descent approximate nearest neighbor algorithm. NG-DBSCAN gives instead the flexibility of specifying any symmetric

distance function on the original data and computes an approximation to the exact DBSCAN clustering. Rather than partitioning an Euclidean space, which is impossible with arbitrary data and has problems with high dimensionality, this algorithm is based on a vertex-centric design, whereby we compute a neighbor graph, a distributed data structure describing the "neighborhood" of each piece of data. In fact, in NG-DBSCAN we compute the clusters based on the content of the neighbor graph ([LDR16]). Other similar algorithms to the NG-DBSCAN exist, like scalable k-NN-based text clustering, a clustering approach that produces high-quality clusters that are easy to interpret, and that accommodates any kind of similarity metrics providing scalability. The aim of this other approach consists of building an approximate k-NN graph of the input text data and computing its connected components, which identify data clusters ([LDDR15]). The main drawback of such approaches is that, differently from FISHDBC, are not incremental and so when data changes a re-computation of the whole process is required, with an expensive cost. Always for what concerns the density-based approaches, also some incremental density-based clustering exists, such as an incremental clustering algorithm based on OPTICS ([FLC15]), but, unlike the FISHDBC, has a quadratic complexity in non-metric space and has a quite small speed-up factors.

- **Hierarchical Clustering**, it is an alternative approach to a classical partitioning clustering method for identifying groups in the data set. In fact, it does not require to pre-specifying the number of clusters to be generated. The result of hierarchical clustering is a tree-based representation of the groups, which is also known as a dendogram. It is possible to decide at which level of this tree we want to stop to obtain the desired groups until that level. There are two main ways to follow for performing a hierarchical clustering:

  - *Agglomerative*: each point is considered as a cluster at the beginning of the procedure and at each further step we merge together the clusters that are more similar at the previous level until we reach the final cluster containing all the data points.
  - *Divise*: we start with a big single cluster that represents all the data points and at each further step we split it considering the most heterogeneous groups of points of the previous level until we reach a situation in which clusters are the single data points.

  One of the best hierarchical algorithms is the HDBSCAN, an optimization of the DBSCAN method, that will be explained in a more detailed way in the dedicated section 3.1 inside the chapter 3, because it is a pillar for the FISHDBC procedure. Anyway, using the HDBSCAN* alone does not allow for obtaining the incremental feature, since when new data arrives, results have to be recomputed from scratch and it has $O(n^2)$ complexity in the generic case of arbitrary distance functions.

Searching in the literature, no parallel implementation, like mine, of the aforementioned FISHDBC clustering algorithm exists. Of course, different parallel clustering procedures, based on other clustering methods, are already used to perform fast clustering, but all of them are based on algorithms that do not have good properties as Dell'Amico's FISHDBC, hence it doesn't exist a parallel implementation that comprehends all the FISHDBC's features. Some of the existing parallel clustering approaches are based on the K-Means algorithm, like: [FRCC08] (that exploit GPU for the parallelization and the speed-up), [BDG16] (using both Cilk Plus and OMP on the CPU, and CUDA on the GPU), [AAA+23] (using instead the OpenMP tools to reach a fast execution) and others that try to not use specific tools and library like [KSG08] and [ZWS21]. The main drawback of these parallel methods is that the K-Means algorithm, despite being a well-known algorithm, suffers from linear or, in the worst case, super-polynomial complexity, while parallel FISHDBC could have a poly-logarithmic complexity. Other parallel clustering approaches, for instance [AWW+23] (tailored for trajectory data) and [XJK02]), use a parallel DBSCAN version to perform the clustering operation, offering a density-based approach, but again not offering all the feature of the parallel FISHDBC.

2.2 HNSW Approaches

Since, as we will see further in the profiling chapter (chapter 6) and in the implementation chapter (chapter 7), we have spent a lot of effort to understand and improve the HNSW creation, one of the main parts of the FISHDBC algorithm, we should mention the most common state-of-the-art HNSW implementation, the Hnswlib (we can find such implementation on GitHub: `https://github.com/nmslib/hnswlib`).

Hnswlib is a library for approximate nearest neighbor search using the HNSW graph structure. With Hnswlib a user can exploit HNSW by creating an index, adding data points to it, and then she/he can perform nearest neighbor searches on the above-inserted data. It is particularly useful when you need to find approximate nearest neighbors in a high-dimensional space efficiently, which is a common requirement in machine learning and data analysis. But Hnswlib does not provide direct methods for exporting or retrieving the internal graph structure of the index and, even more importantly, retrieving all the computed distances during the HNSW creation. If you have a specific use case where you need to access or manipulate the HNSW graph, you cannot do it with Hnswlib. My parallel HNSW version allows more flexibility because it provides direct access to the graph structure and to the saved distance results. In fact, there are cases in which it is needed to use this data structure along with all the calculated distances for other operations, as the paper [HGJ+23] and the article [ZNFVW22] demonstrates.

But also in my implementation (and more in general in the original FISHDBC procedure)

the HNSW with its "distance cache" is fundamental, since it is used to compute the MST and consequently the HDBSCAN. Actually, in my parallel HNSW version, such graph-based structure is represented as two shared numpy arrays:

- the first (shared_adjs) represents the edges between the nodes
- the second (shared_weights) listing the dissimilarity values used to measure the distances between pairs of edges.

Another difference is that Hnswlib manages only numerical data. It is however possible to use it with textual data, but there is the need to represent these text data as numerical vectors by implementing a specific conversion procedure. With this conversion, it is possible to lose the original data's quality. The choice of text embeddings is essential to achieve efficient and accurate similarity searches.

FISHDBC, instead, manages these data in a native way, since it is not necessary to convert the text data, hence the data can be passed directly to the HNSW creation as string data type, also because it is possible to use the right dissimilarity to measure the distance for such kind of data type.

At last, Hnswlib requires the user to modify the source C++ code to use a custom distance metric, which can be a complex and non-trivial task for non-skilled developers in both C++ and Python. On the contrary, FISHDBC requires only to add a line of code calling the desired distance function provided and already implemented by some scientific Python libraries, such as Scipy and Scikit-learn. However, Hnswlib is much faster than our implementation (see the Table 8.5 and Table 8.6), both sequential and parallel. The main reason is that it is written in C++, thus the resulting code can exploit vectorization and other optimization performed by the compiler, while FISHDBC adds also the overhead due to the Python interpreter. In addition, Hnswlib uses the very efficient multithreading approach instead of multiprocessing, the opposite of our parallel HNSW creation, since in Python it is not possible to exploit the multithreading parallelism (as we will explain in the chapter 5).

But if a user wants to exploit a custom-specific function, it is not straightforward to use it in Hnswlib. Moreover, if this function is computationally expensive (like in the work of the paper [CVD+20]), Hnswlib will provide almost the same performance as our implementation because the distance function evaluation becomes the main hot-spot. In fact, Hnswlib works very well only with two defined and optimized functions written directly inside the C++ code. Anyway, using different and custom functions is a frequent case, considering that most of the users of the lib write Python applications, thus the distance functions should be written in this language.

As regards the quality of results, Hnswlib does not always return the exact nearest neighbor, it provides an approximation that is often sufficiently accurate for many applications (see

the Table 8.12), especially when dealing with high-dimensional data where exact nearest neighbor search can be computationally expensive. Also, our parallel HNSW creation returns a solution that could be affected by some errors, but the numbers of such errors are usually low and acceptable (take a look at the Table 8.9), even if the Hnswlib provides better-approximated solutions since the accuracy is better than our version. The similarity is true because we use an algorithm that is very similar to the one used by the Hnswlib, even if it is not the same. However, we have to point out that the HNSW data structure is not our main result, while the final FISHDBC clustering result is.

Chapter 3

Background

As will be explained in detail in such section, Dell'Amico's FISHDBC uses three main concepts as support for the algorithm's functioning:

- the so-called **HDBSCAN** as the final method to perform the hierarchical cluster.
- HDBSCAN is, in turn, based on another well-known subject, the **MST**, using it to perform the hierarchical result, removing edges in decreasing order of weight (following a divisive approach).
- the **HNSW** graph-based data structure to represent the whole dataset efficiently for performing the MST creation, thanks to its good search complexity.

3.1 HDBSCAN*

As we have seen until now, there are many ways to perform clustering and some of them are density-based, but they only provide a flat representation of the clusters (like the DBSCAN), based only on a single density threshold that doesn't characterize properly data set with very different densities clusters. Actually, some methods that provide a hierarchical result exist, but, with respect to the HDBSCAN* approach used by the FISHDBC, they aren't able to provide a representation involving only the most significant clusters, or they work well only for some specific classes of problems, or they need to take as input also multiple critical parameters.

The HDBSCAN* clustering ([CMS13]) is based on an optimized version of the classical density-based approach, DBSCAN. This better version of DBSCAN (called DBSCAN*) defines clusters as the connected components of a graph in which the data points are

the vertices and each pair of vertices are connected by edges only if the data points are E-reachable (two core objects x1 and x2 are E-reachable if x1 belongs to the x2's neighborhood and x2 belongs to the x1's neighborhood) with respect of two parameters E and mPts defined by the user (mPts specifies the number of points that a node at least should have in its neighborhood to be considered a core object). HDBSCAN* can also be viewed as an improvement over OPTICS (Ordering Points To Identify the Clustering Structure), another density-based method similar to DBSCAN, but better suited for usage on large datasets.

HDBSCAN* is widely used for many clustering operations:

- **Cluster Analysis in Data Mining:** HDBSCAN* is widely used for clustering in data mining applications, especially in scenarios where clusters have varying shapes and densities. It is effective in identifying clusters of different sizes and shapes.
- **Anomaly Detection:** It can be used for anomaly detection by labeling data points that do not belong to any cluster as noise. This is particularly useful in identifying rare events or outliers in a dataset.
- **Image Segmentation:** HDBSCAN* can be applied to segment images by treating pixels as data points and clustering them based on density. This can help identify distinct regions or objects in images.
- **Natural Language Processing:** In text and document analysis, HDBSCAN* can be applied to cluster similar documents based on the density of words or features. This is useful for document categorization and topic modeling.
- **Density-Based Outlier Detection:** Beyond clustering, HDBSCAN* can be employed for identifying outliers in a dataset based on their low density compared to the surrounding data points.
- **Spatial Data Analysis:** HDBSCAN* is suitable for clustering spatial data, such as geographical data points, to identify spatial patterns or groupings.

HDBSCAN, rather than looking for clusters with a particular shape, looks for regions of the data that are denser than the surrounding space. The mental image you can use is trying to separate the islands from the sea, so we can identify the different clusters as highly dense regions (islands) separated by sparse regions (sea), as we can see in the 1-D cluster representation shown in Figure 3.1. All the images regarding the explanation of the HDBSCAN procedure, are taken from the online article [Scia].

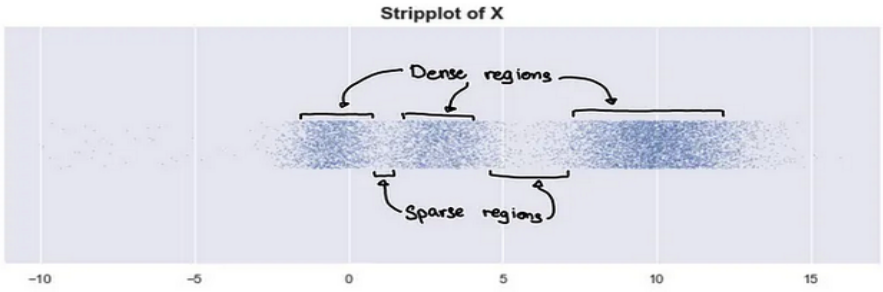

Figure 3.1: 1-D representation of dense areas that correspond to clusters of X

Based on how HDBSCAN recognizes the clusters, we can deduct that it is able to deal with noisy data and even outliers (that in our previous Figure 3.1 correspond to the sparse region), but also with clusters of different sizes (for instance always in the Figure 3.1 there is one dense region that is bigger than the others two) and shapes (since it focuses on the density of the points it doesn't matter what is the shape they form). If we put the clusters problem in terms of probability distribution, printing the probability distribution function (PDF), we are able to state that the peaks, depicted in the Figure 3.2, correspond to area of high probability, this means that they are associated with the dense area, so with our clusters, since clusters are high probable regions, while sparse areas, the valleys, are improbable regions. In this scenario, you can think of clusters as mountains, while the sparse and improbable regions as valleys, rather than islands and sea.

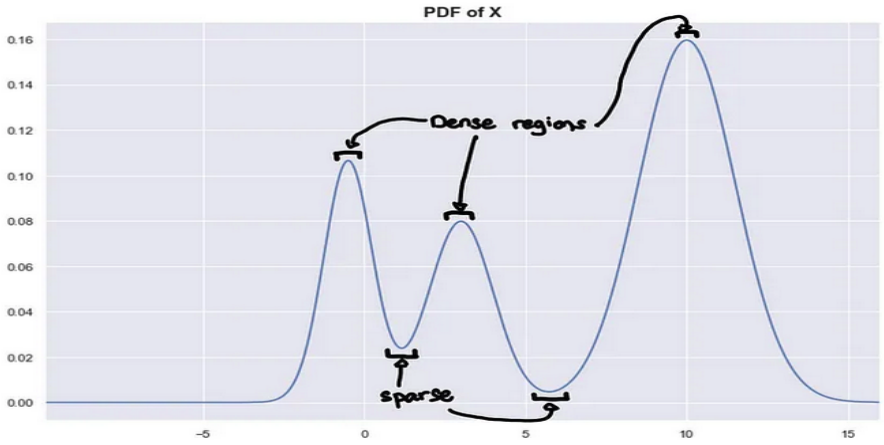

Figure 3.2: Clusters Probability Distribution of X

These two different plots, Figure 3.1 and Figure 3.2, are equivalent because one describes the data through its probability distribution and the other through a random sample from

that distribution. Saying highly dense regions separated by sparse regions is the same as saying highly probable regions separated by improbable regions.

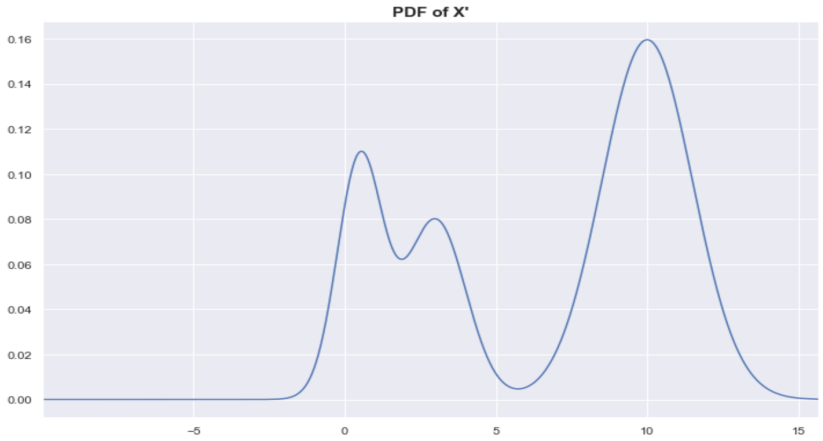

Figure 3.3: Clusters Probability Distribution of X'

But if we consider a different configuration, identified as X', of the probability distribution of X, where the first peak is shifted to the right, as shown in the Figure 3.3 above, we can see that it is no longer clear if the clusters now are 2 or 3, because two peaks are very near to each other.

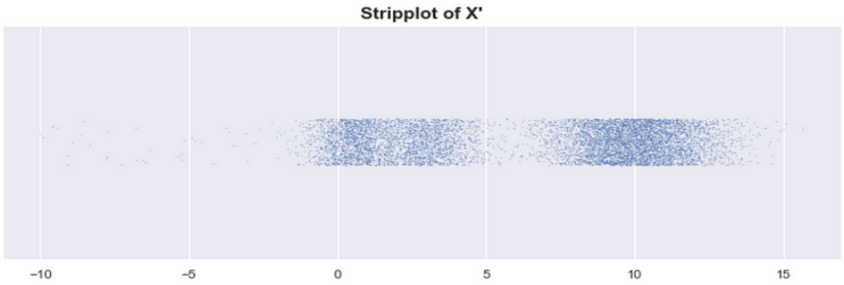

Figure 3.4: 1-D representation of dense areas that correspond to clusters of X'

In fact, looking at the strip plot of the same X' displayed by the Figure 3.4, we can identify only two dense regions, namely two clusters, hence there is a particular value by which the number of clusters has changed between X and X'. This particular value is a global threshold for the PDF. The separation in different clusters depends on the chosen threshold and the relative level-set we are considering, as expressed by the Figure 3.5.

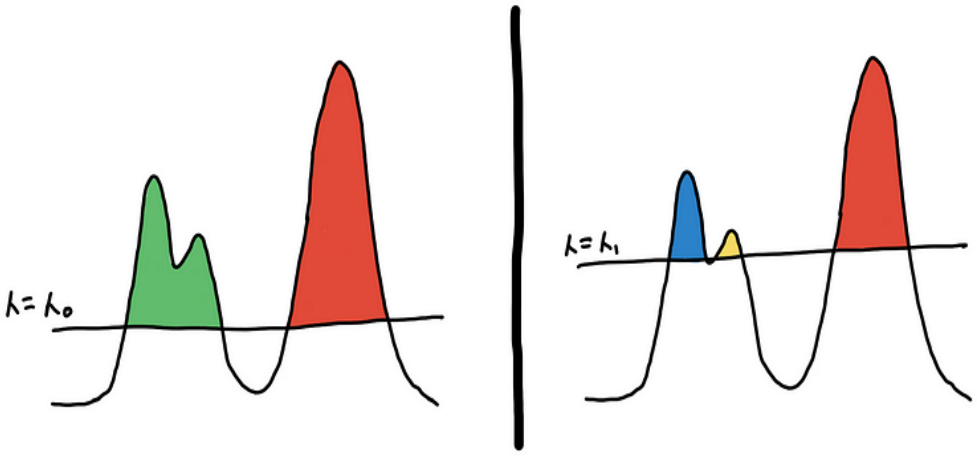

Figure 3.5: Different clustering based on different threshold values

Now, in the current level set, we select the configuration, in such example between the one with 2 or 3 clusters, as illustrated in the following Figure 3.6, based on what is the one that persists more. Saying "persist" refers to the stability of clusters, so in the end, the cluster configuration that will be considered is the one with the highest stability.

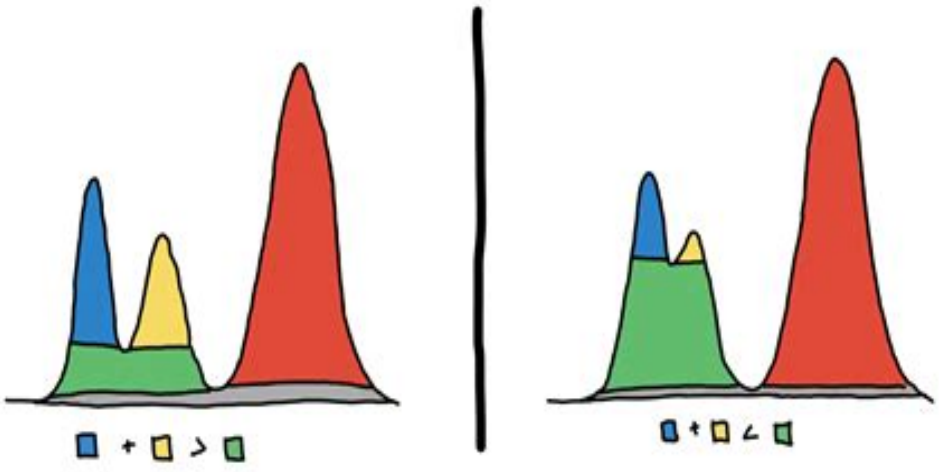

Figure 3.6: Different clustering hierarchy with different stability

HDSBCAN, in fact, solves this issue, maximizing the overall stability of the set of clusters extracted from the HDBSCAN* hierarchy. This new challenge could be viewed as an

optimization problem with the objective of maximizing the sum of the stabilities of the extracted clusters. The solution is a bottom-up traversal of all the clusters in the tree, during which we discard at each iteration the clusters that have worse stability, propagating and updating the total stability. In the end, it will remain only the best clusters in terms of stability, finding an optimal solution. By getting multiple level sets at different values of the threshold, we get a hierarchy. We could again imagine the clusters as islands in the middle of the ocean. As you lower the sea level, the islands will start to "grow" and eventually islands will start to connect with others ([Scia]). Actually, we represent the relationships between clusters as a hierarchy tree, the dendrogram. The Figure 3.7 helps us to show the transition from the cluster hierarchy of the PDF to the equivalent hierarchy tree.

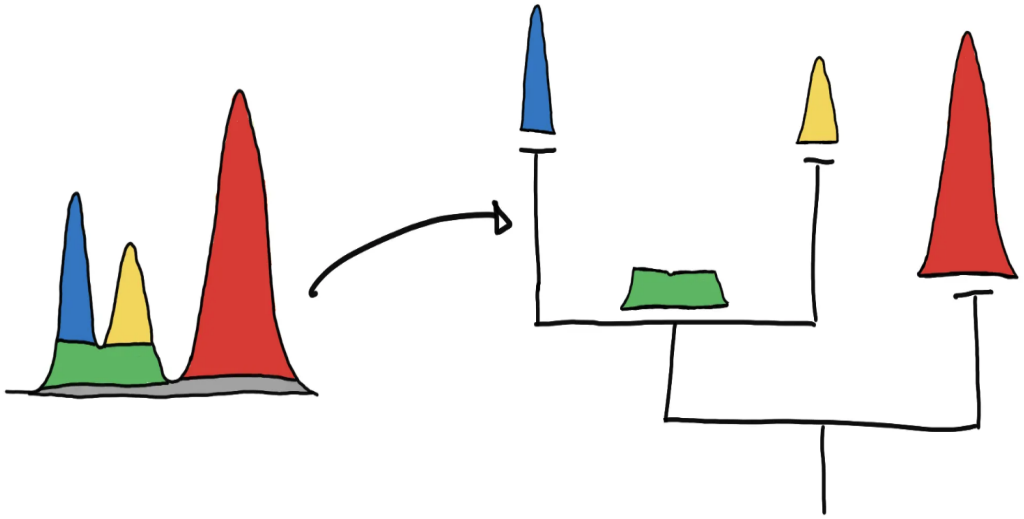

Figure 3.7: Creation of a hierarchy tree from a cluster hierarchy

By convention, the dendrogram is drawn top-down, where the root (the node where everything is just one cluster, i.e. the biggest island) is at the top and the tree grows downward. The Figure 3.8 just demonstrates how the plot of a dendrogram is.

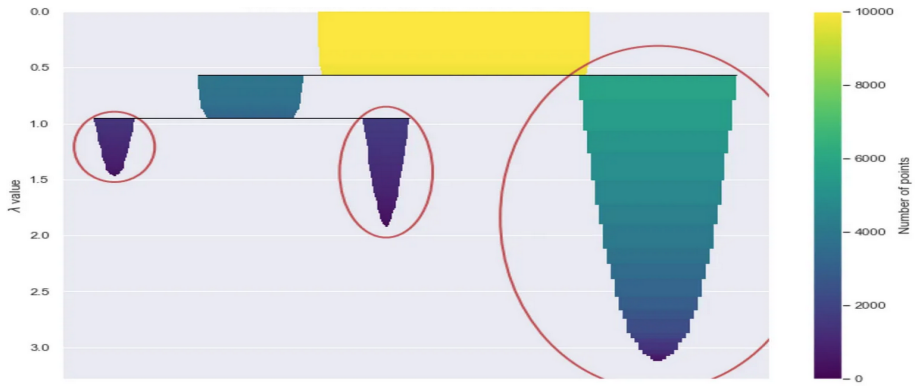

Figure 3.8: Example of the plot of a HDBSCAN's resulting dendrogram

Since creating and visualizing the resulting dendrogram could not be so easy, especially for large and noisy data sets, extracting a summarized tree of only more significant clusters could be fundamental. To apply this simplification to the HDBSCAN it is necessary to select only those hierarchical levels in which new clusters appear after a true split or in which clusters disappear because these are the levels where the most important changes in the cluster structure occur. When a cluster only shrinks it is not so significant. For implementing this idea, the HDBSCAN* uses another parameter called the mClSize (minimum cluster size). The components with less than mClSize are not considered and, adopting this practice, the size of the final hierarchy is reduced. This additional parameter could be set equal to the other aforementioned parameter, the mPts, so, in this way, a single parameter is used both as the mPts, a smoothing factor in density estimates and as cluster size threshold.

We need to clarify that, for the sake of the example, we are considering the true PDF of the underlying distribution. However, the underlying distribution is almost always unknown for real-world data. In a concrete scenario, we must estimate the PDF empirically. The HDBSCAN method gets the empirical PDF finding the distance to the K-th nearest neighbor of each point. The results are what we call core distances in HDBSCAN. Points with smaller core distances are in denser regions and would have a high estimate for the PDF. Points with larger core distances are in sparser regions because we have to travel larger distances to include enough K neighbors, as the Figure 3.9 depicts. Filtering points based on the core distance called λ, is similar to obtaining a level-set from the underlying distribution.

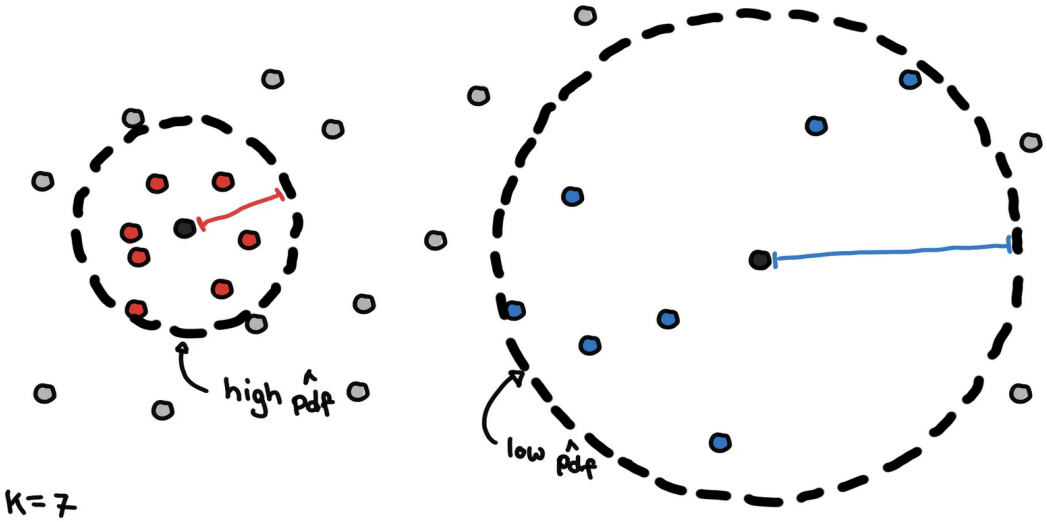

Figure 3.9: Points with core distance less than λ

So imagine now to have a dataset in input with its relative probability distribution function and you want to find the clusters of a certain level set, namely of a certain value of λ (Figure 3.10).

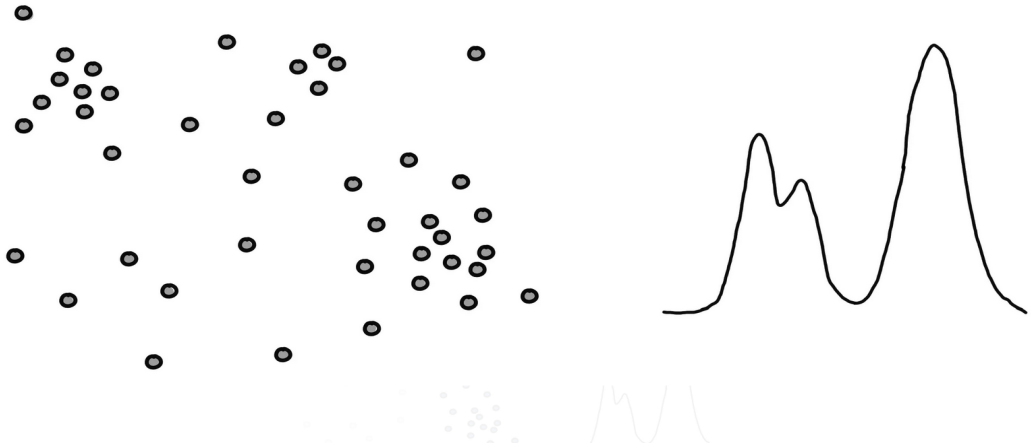

Figure 3.10: Original dataset and its PDF

The first step is to identify the level-set corresponding to the selected value of λ, hence we consider only the point of the dataset that has a core distance <= λ (Figure 3.11).

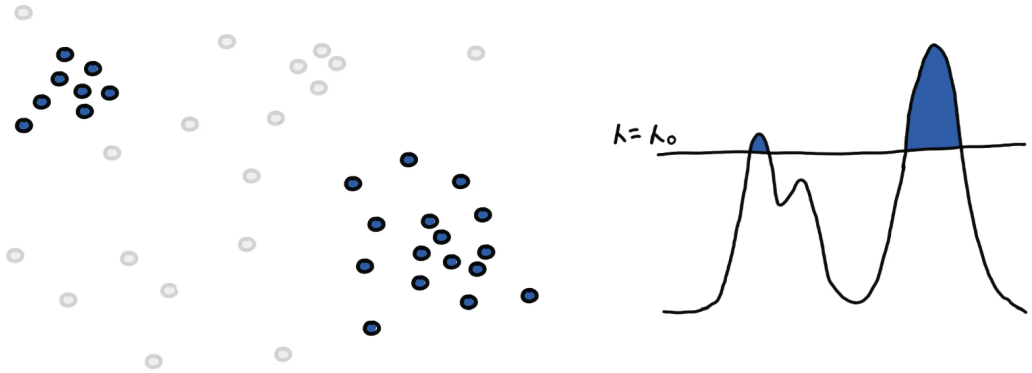

Figure 3.11: Points with core distance less or equal than λ

Now we need to find the different regions. This is done by connecting "nearby" points to each other. "Nearby" means that two points are near enough if their Euclidean distance, for instance, or another metric (e.g. cosine distance, square Euclidean distance, Minkowski distance), is $<$ than λ (Figure 3.12).

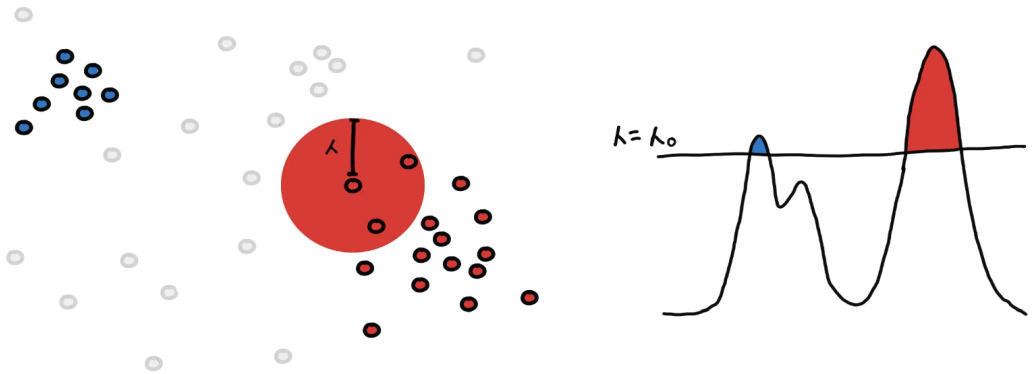

Figure 3.12: λ-sphere of a point

At this moment, we connect the point P to all points that are inside its λ-sphere, this means that we link P to other points for which the computed distance (for example the

Euclidean) is less than λ (Figure 3.13).

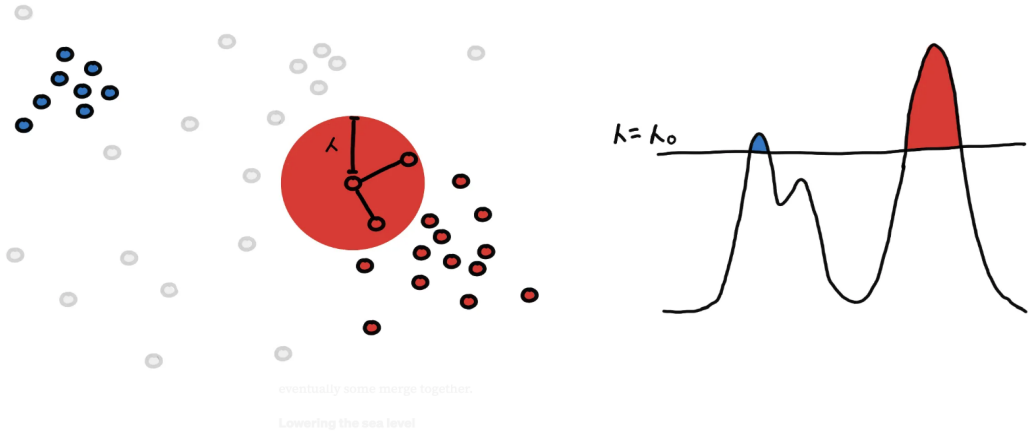

Figure 3.13: Connect the point with selected (Euclidean) distance less than λ

Doing this for every point allows us to obtain the clusters for a certain value of λ (Figure 3.14).

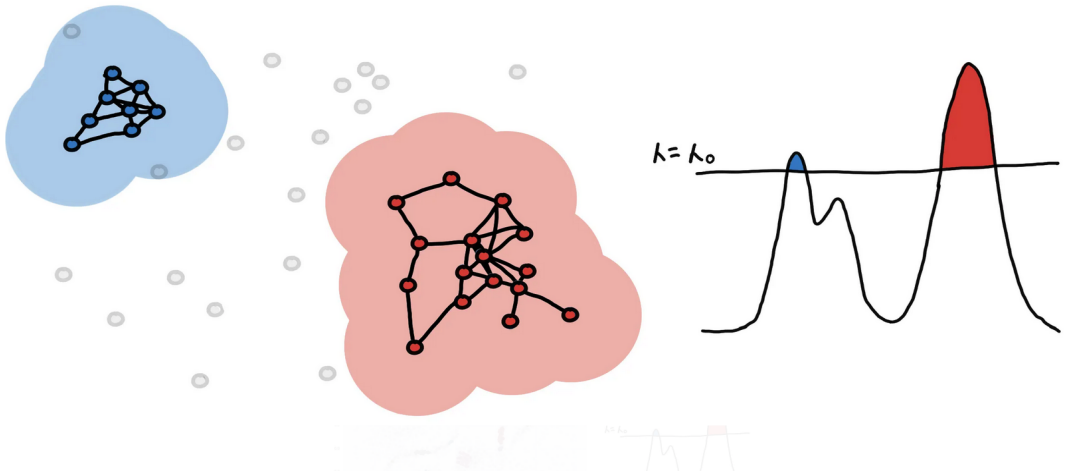

Figure 3.14: Different resulting clusters

Now, changing the value of λ is like lowering the sea level, and later could happen that some clusters could merge together into a bigger one, and so on and so far (hypothetically

until we have only a big cluster, a big island).

More formally, in this approach, new concepts appear, with respect to the classical DBSCAN* method:

- **Core distance** of an object x1, so the distance from x1 to the mPts-nearest neighbor (or considering the previous example of the K-nearest neighbor).
- **E-Core Object**, it is an object Xc for which its core distance is less or equal to every value of E.
- **Mutual Reachability Distance** of two objects x1 and x2, that it is the maximum between the x1's core distance, the x2's core distance and the distance (depending on the chosen dissimilarity) between x1 and x2.
- **Mutual Reachability Graph**, a complete graph in which nodes are connected by weighted edges, in which the weight is determined by the mutual reachability distance defined before.

Before, in the example showing the different steps of HDBSCAN, we have mentioned the fact that we connect a point P with all the other points inside its λ-neighborhood, if a certain distance function is less than λ. Actually, the HDBSCAN procedure uses the mutual reachability distance as a number to compare with the λ value, not directly a distance metric like the Euclidean dissimilarity.

Consequently, all DBSCAN* partitions can be produced in a nested, hierarchical way by removing edges in decreasing order of weight (the weight is the mutual reachability distance) from the mutual reachability graph, which could be seen as an extension of a minimum spanning tree. More precisely, we extend the MST with edges connecting each vertex to itself (self-loops), where the edge weight is set to the core distance of such vertex. These self-edges will then be considered when removing edges. The result is again the HDBSCAN hierarchy.

There are different possible developments of the hierarchical DBSCAN procedure, the state-of-the-art implementation is the hdbscan Python library ([MH17]), used in the FISHDBC algorithm, but exists also a C++ version of the same clustering procedure ([MB19]) or a GO implementation ([Bel]), or also a MatLab version of the algorithm ([Sor18]). All of them are based on the idea and explanation described in the already cited paper ([CMS13]).

3.2 MST

As anticipated at the beginning of this chapter, another well-known data structure that plays an important role, since in our case is related both to the HDBSCAN and HNSW

section, is the Minimum Spanning Tree (MST). An MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight ([Wike]). In other words, it is a spanning tree that has the minimum weight among all the possible spanning trees, as depicted in the 3.15. But what is a spanning tree? A spanning tree T of an undirected graph G is a sub-graph that includes all the vertices of G. In general, a graph may have several spanning trees but also many minimum spanning trees.

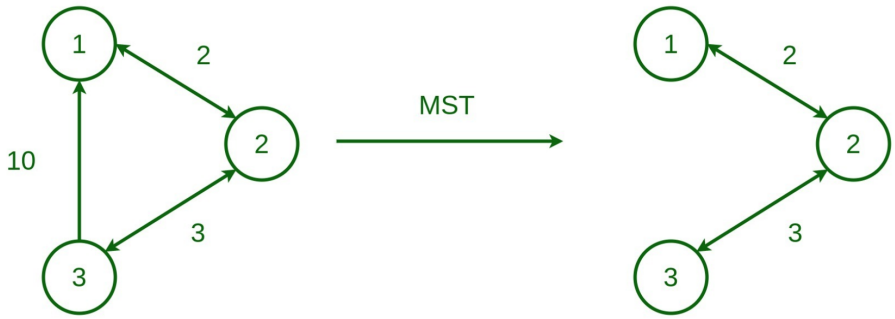

Figure 3.15: MST of a graph

The properties of an MST are the same as a ST:

- The number of vertices in the graph and the associated spanning tree are the same.
- The total number of edges in the spanning tree is equal to the total number of the original graph's vertices minus one.
- The spanning tree should not be disconnected.
- The spanning tree should be acyclic.
- The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.
- Plus, obviously, the constraint of having the minimum possible weights among all possible spanning trees ([Geed]).

Also, the MST is widely used for many applications:

- **Network Design:** MST is commonly used in network design to connect a set of locations with minimum total cost. This is applicable in various scenarios, such as designing communication networks, electrical grids, or transportation systems.

- **Cluster Analysis:** in data analysis, particularly in unsupervised learning, MST can be used for cluster analysis. By treating each data point as a vertex and the distance between points as edge weights, the MST helps identify natural clusters in the data.
- **Approximate Algorithms:** MST is often used as a building block in approximation algorithms for solving optimization problems. For example, the Christofides algorithm uses MST in its solution for the traveling salesman problem.
- **Routing Algorithms:** MST-based routing algorithms are used in computer networks to find efficient paths between nodes while avoiding loops.
- **Image Segmentation:** in image processing, MST can be applied to segment images by treating pixels as nodes and their intensity differences as edge weights. The resulting tree structure helps identify meaningful regions in the image.

There are different possible algorithms to be used for computing the MST of a graph, like the Kruskal ([Geea]) algorithm, the Prim ([Geec]) algorithm and the Boruvka method ([Wikb]).

3.3 HNSW

Before, in this chapter, we have mentioned the fact that the FISDHBC algorithm uses a particular data structure in order to be incremental and scalable with respect to the input data set. This structure is called HNSW ([MY20]).

It is a full graph-based incremental K-ANNS (K Approximated Nearest Neighbor) structure with the main advantage of having a logarithmic complexity scaling for information search. In fact, if we use a classical naive approach like the K-NN (K Nearest Neighbor) the complexity will be linear, making this approach unusable for large data sets. For this reason, an approximated version of the K-NN was designed (the K-ANNS indeed), since it allows better performance on high dimensional data sets, even if it suffers from degradation in case of low dimensional data.

This data structure can be used in many fields and applications:

- **Information Retrieval:** In text or document search applications, you want to find similar documents based on their content.
- **Image Retrieval:** HNSW can be used to find visually similar images in large databases, making it useful in image recognition and retrieval systems.

- **Recommendation Systems:** It can help in building recommendation systems by efficiently finding items or users that are similar to a given item or user.
- **Machine Learning Feature Extraction:** HNSW can be used to efficiently search for similar feature vectors, making it useful in machine learning tasks such as clustering and classification.

Understanding the HNSW approach could be very useful to explain how the probability skip list technique works ([Pin]) since it contributed heavily in the HNSW creation. Probability skip list allows fast search while using a linked list structure for easy (and fast) insertion of new elements. Skip lists work by building several layers of ordered linked lists. On the first layer, we find links that skip many intermediate nodes/vertices. As we move down the layers, the number of "skips" by each link is decreased. To search in a probabilistic skip list structure, we start at the top layer, the one that has the longest skip, and if our current selected element is greater than the element we are searching for (or we reach the end of the current layer), we drop to the next layer. The Figure 3.16, taken from the nice article at the [Pin] website, as all the other pictures of this HNSW section, depicts how the search process works in a skip list structure. For what concerns the HNSW we can think of this skip list method and substitute, in each layer, the linked list with a graph, so we end up with a structure in which in the highest layers we have longer edges, while in the lower layers there are shorter edges.

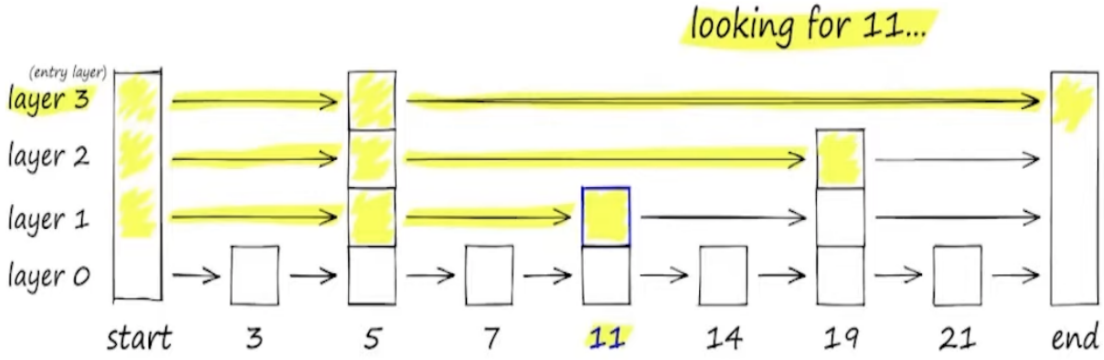

Figure 3.16: Search example on probability skip list structure

Another main foundation on which the HNSW is based, that might be useful to look into, is the proximity graph K-ANNS algorithm called NSW (Navigable Small World), which utilizes the convenient navigable graphs, i.e. graphs that allow to obtain a logarithmic (or poly-logarithmic) complexity considering the number of needed hops during a greedy traversal of the structure. A navigable graph is constructed with the insertion of elements in a random way, each time connecting the current one to the M closest neighbors of the

previously inserted element. These links became bridges between the hubs of the graph and are responsible for the logarithmic scaling of the search. Another good property of a navigable graph is the fact that it could be used for a distributed search system since the construction phase could be parallelized. The problem with NWS is that usually has a poly-logarithmic complexity and that it suffers a performance degradation when it is used for low dimensional data sets.

When searching an NSW graph, we begin at a pre-defined entry point. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there. We repeat the greedy-routing search process (Figure 3.17) of moving from point to point by identifying the nearest neighbor points in each neighborhood. Eventually, we will find no nearer points than our current point, this is a local minimum and acts as our stopping condition. As the real HNSW, also the NSW search is based on two main steps:

- **The Zoom-out** phase is where we pass through low-degree points.
- **The Zoom-in** phase is where we pass through higher-degree points.

Since the NSW could suffer from the local minimum problem, an approach for avoiding this is to start the search on high-degree vertices (zoom-in phase first instead of zoom-out). For NSW, this improves performance on low-dimensional data, as for the HNSW.

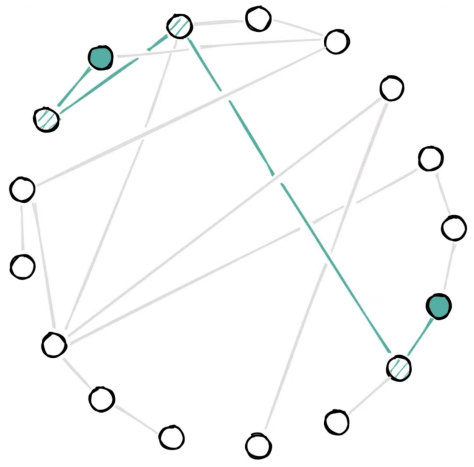

Figure 3.17: Greedy routing inside a NSW

Adding the multilayered approach of skip list structure to the NSW we produce a graph where links, considering their lengths, are separated across different layers, the HNSW. In this optimized approach, the search starts from the upper layer that contains only a

link, the longest link. At each iteration, the algorithm traverses the upper layer until it finds a local minimum for such layer, after, the procedure goes to the lower layer starting from the same element that was the local minimum for the previous layer. We repeat this process until we find the local minimum of our bottom layer, that is layer 0. Following this pattern, the maximum number of connections per element in all the different layers can be made constant, hence this allows to obtain a logarithmic complexity inside the navigable small world network. A schematic search is illustrated in the Figure 3.18.

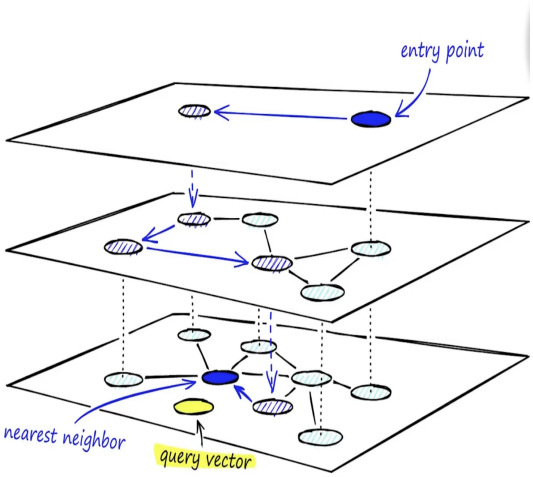

Figure 3.18: HNSW search procedure

The algorithm for the creation of the HNSW structure is divided into two main parts:

- **The first part**, related to the insertion process, starts from the top layer by greedily traversing the graph in order to find a certain number of closest neighbors, called in the literature ef (the ef for the first layer is 1), of the inserted element in the layer L. The layer L to which each element is assigned, usually it is chosen with a particular probabilistic distribution. After that, the algorithm continues the search from the next layer using the closest neighbors from the previous layer as enter points, and the process repeats. Again, a basic schema of the HNSW's creation is depicted in the Figure 3.19.
- **The second part** regards the method used for selecting the best M closest neighbors from the considered candidates (the candidates are the previous ef neighbors). For doing so, instead of using the classical approach of nearest neighbor, the HNSW procedure uses a heuristic that examines the candidates starting from the nearest and creates a connection to a candidate only if it is closer to the base element compared to any of the already connected candidates.

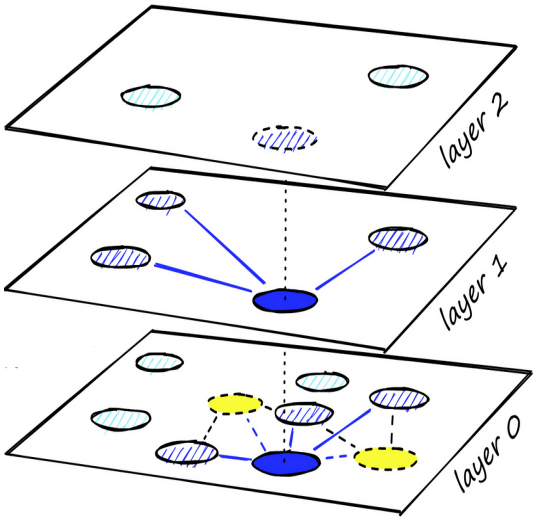

Figure 3.19: HNSW structure creation

In the end, this methodology supports the incremental feature and could also be used as an efficient method to perform an approximated version of K-NN. Also regarding the various ways to perform the HNSW graph, we have to say that all of them are based on the main idea and method explained in the original paper [MY20] and implemented by the Hnswlib. In fact, the HNSW version inside the FISHDBC algorithm is strictly inspired by such state-of-the-art idea. Anyway, there are other versions of the HNSW procedure as the one implemented in Rust ([gro20]) or another C++ version that has the good incremental property of insertion (and removal) of elements inside the structure ([Gor17]).

Chapter 4

FISHDBC

As we have just said in the introduction, my work is based on an existing algorithm created by the professor Matteo Dell'Amico (indeed one of my supervisors), the FISHDBC (Flexible Incremental Scalable Hierarchical Density Based Clustering) algorithm. This method is used to cluster an arbitrary data set, using also an arbitrary distance metric, for obtaining, as result, a hierarchical cluster of the input data. It is strictly based on three, already introduced, main concepts: the HNSW (Hierarchical Navigation Small World) structure and MST (Minimum Spanning Tree) structure to be incremental and scalable with respect to the updates and additions of new data, and the HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise), a procedure to perform the final hierarchical and density-based clustering operation.

4.1 The FISHDBC Algorithm

Now that we have some background knowledge, we can go further into detail about the FISHDBC algorithm ([Del19]). Foremost, we have to answer to the question of why it is called FISHDBC?

So, it is flexible because it is applicable to arbitrary distance functions and arbitrary data, hence there isn't a loss of information; it is incremental because, thanks to some data structures, new input items can be added cheaply and the process can be recomputed quickly; it is scalable, since it avoids, for most of the scenario, the $O(n^2)$ complexity and since it works also quite well for large input data sets; FISHDBC is hierarchical due to the fact that it creates a result composed by clusters within cluster; since this algorithm is inspired by DBSCAN, it belongs to the category of the density based clustering.

The idea of the overall process is to use an approximated version of the MST and updating

it incrementally and at low cost when new input items are considered. The candidate edges to add to the MST are chosen thanks to the HNSW, but instead creating all the HNSW based on our data and after querying it, with the FISHDBC approach, we generate batches of triples composed by two points and the distance between these points, thanks to the calls to the distance function. Since no query is ever performed over the HNSW, we manage to get an improvement in the efficiency of the procedure.

When we have mentioned approximation before, we refer specifically to the fact that we don't want to compute the distance between all the pairs of points, and having only a sub-sampling of the distance matrix is good because it works as a regularization step that avoids over-fitting.

Another interesting aspect of FISHDBC is that, since there are many possible MSTs for the algorithm, it leads to produce MSTs with a lower diameter, thus we obtain as result larger and smaller clusters and consequently less profound hierarchy that again brings an advantage for the algorithm's cost.

Analyzing the code of the FISHDBC written by Professor Dell'Amico, we note that there are four main objects used by the algorithm:

- **The HNSW** graph created using the namesake Python class, heavily inspired by the HNSW implementation of the Hnswlib Python library. The HNSW is used as data structure to memorize the neighbors of elements and after, to make search of these elements, exploiting the logarithmic complexity thanks to the approximated K-NN and the heuristic, both used for the searches. In the FISHDBC implementation, the mPts parameter of the HNSW is set equal to the K parameter of the approximated K-NN.
- **Neighbors** to collect all the mPts neighbors of each node along with their distances, using a heap data structure.
- **The current approximated MST** computed at each new insertion of data. Each MST's instance is composed by the couple of point that forms a connection, the value of the distance (based on the selected dissimilarity) between them and also the value of their reachability distance.
- **Candidates**, a collection of possible candidates edges to be inserted in the current MST.

As one might logically expect, the first phase of the algorithm is the setup phase (algorithm 1), by which all the previous four objects are initialized to be used further inside the different procedures of the algorithm. In this first phase another object is created, called distance_cache, that, as the name explains, works as a cache for the already computed

distances, that it is constantly modified based on the cache misses and hits that occur during the main process. Ut can bring a speed-up in the entire procedure (as a cache in general does).

The second main step is the add phase (algorithm 8), that, as the name suggests, is concerned with the incremental addition of new input items to the data that we are clustering and to the HNSW structure. This procedure updates the neighbors' heap of each current element with the new neighbors discovered in the HNSW. It also goes to modify the candidates' collection of elements to be added to the MST: in fact, for each newly inserted element inside the HNSW, we add to the candidate list all the edges (pairs of points) for which a distance was computed, but not only, it considers as possible candidates even the edges for which the reachability distance is decreased due to new item insertion and consequently the new edge formation.

When the candidates' list reaches a certain dimension (like a threshold) during this adding process, another important function is called to action: the update_MST function (algorithm 1). This method is used for merging the candidates' collection (emptying all the elements from this list) to the current Minimum spanning tree (and inserting the elements in the MST). This procedure is done thanks to Kruskal's algorithm, a minimum-spanning tree algorithm that finds a minimum-spanning forest of an indirect edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. In turn, Kruskal uses a UnionFind approach to perform such a merge operation. Formally, the UnionFind is a data structure that stores a collection of disjoint (non-overlapping) sets and plays a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph. The updated MST is needed for having the incrementality and scalability properties, since it is a data structure that is modified at a low cost when a new element is considered. It is also essential because is the "input" for the final clustering (algorithm 5).

Algorithm's Complexity. The complexity of the algorithm is divided in:

- **Space complexity** is O(n logn), since FISHDBC's state consists of the HNSW (O(n logn) size); neighbors, each node's MinPts closest discovered neighbors and their distance (O(n) size); mst, the current approximated MST stored as a mapping between edges and their weight (n nodes and at most n - 1 edges, hence O(n) size); the temporary set candidates of candidate edges (O(n) size, because each call to add will add to candidates at most n - 1 elements). The union of these four objects has therefore size O(n logn).
- **Time complexity** is ((t+n)logn), where t is the number of calls to distance function performed by the HNSW. The time complexity of FISHDBC depends on HNSW creation: if it requires few distance calls, computation cost remains low.

Algorithm 1 FISHDBC

```
1: procedure SETUP(d, minpts, ef)                          ▷ d is the distance function
2:     self.minpts ← minpts
3:     self.mst ← {}                          ▷ approximate MST: mapping (x, y) edges to weights
4:     self.neighbors ← {} ▷ closest known neighbors per node: maps data to max-heaps of (distance, neighbor) pairs
5:     self.HNSW ← HNSW(d, ...)                                          ▷ HNSW's k is set to
6:     self.candidates ← {} ▷ edges that may be added to the MST: mapping of (x, y) edges to weights
7: end procedure
8: procedure ADD(x)
9:     self.HNSW.add(x)          ▷ Keep note of all results of calling d; we use them in algline:add-outer
10:    self.neighbors[x] ← closest neighbors found
11:    for each time d(x, y) is called by HNSWreturning v do
12:        self.candidates[x, y] ← max(v, core distances of x and y)          ▷ Reachability distance
13:        if x is a new top-neighbor for y then
14:            c_y0 ← old core distance for y
15:            update self.neighbors[y]
16:            c_y1 ← new core distance for y
17:            for all neighbor z of y at distance w < c_y0 do
18:                c_z ← core distance of z
19:                if c_z < c_y0 then          ▷ The reachability distance for (y, z) decreased
20:                    candidates[y, z] ← max(w, c_y1, c_z)
21:                end if
22:            end for
23:        end if
24:    end for
25:    if |candidates| > α · |neighbors| then call UPDATE_MST   ▷ We guarantee that candidateshas (n) size
26:    end if
27: end procedure
```

In the end, we can highlight the fact that FISHDBC, with respect to the HDBSCAN* alone, can manage large data sets and can avoid the O(n^2) complexity, so it outperforms, in most cases, HDBSCAN*. In addition, as we already know, it is possible to define and pass to the algorithm an arbitrary distance function, of course, that makes sense according to the type of data that we are considering. The last good feature of such methodology is that the separation between neighbor discovery and maintenance of incremental insertion of data could lead to an increased simplicity of understanding and implementing such procedure, along with an improvement in terms of execution.

```
1: procedure UPDATE_MST
2:     mst ← Kruskal(mst ∪ candidates)
3:     candidates ← {}
4: end procedure
5: procedure CLUSTER
6:     if candidatesis not empty then call UPDATE_MST
7:     end if
8:     compute clustering from MST        ▷ using the accelerated approach introduced by
   [CMS13]
9: end procedure
10: procedure ADD_BATCH_AND_CLUSTER(elems, )
11:     for all x ∈ elems do
12:         call (x)
13:     end for
14:     return CLUSTER()
15: end procedure
```

Chapter 5

High-Performance Computing in Python

Python is a high-level, interpreted, and general-purpose programming language. Python's design philosophy emphasizes readability and simplicity, and its syntax allows programmers to express concepts in fewer lines of code than might be possible in languages such as C++ or Java. This readability and simplicity, along with a comprehensive standard library, have contributed to Python's popularity for a wide range of applications. In general, we have to be honest and precise that it does not provide necessarily high performance. Anyway, this is not completely true since in these last years the Python community has invested a lot in such high performance field. The current chapter explains many methodologies that could be used to reach a parallel solution for the FISHDBC algorithm. Some of these approaches were considered for the implementation of the thesis.

5.1 Packages & Tools

A first simple aspect to consider when we want to speed up our Python code could be to use the last available version of Python, if you can. This happens since usually the newer version brings compiler optimizations, interpreter enhancements, new language features, optimized libraries and garbage collection improvements, that all together allow to reach faster final running time. For instance, in the remote Linux machine, we can activate a conda environment with the most updated version of Python, going from version 3.8.16 (outside the Python environment) to 3.11.3, and this causes a saving on the final execution time of the program (not so relevant but anyway good).

Another simple good practice to speed-up our Python program is to import and use some

scientific packages, already well-tuned and optimized. The most important ones are:

- **Numpy**. NumPy (Numerical Python) is the fundamental package for scientific computing and the universal standard for working with numerical data in Python. This library provides a homogeneous multidimensional array object, called ndarray, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting and much more. It is better to use a numpy array instead of a classical Python list because a list could be composed by heterogeneous data types and this could slow lots of mathematical operations performed on such a list. The ndarray, since it is homogeneous, cannot have many data types inside, but it can only be created according to one dtype (int, float, ecc...), thus it allows to be faster and more compact. Additionally, an array consumes less memory and so it is convenient to use. A last aspect is that it is written in C language and therefore it exploits the vectorization. For all the information on how to create the numpy array and how to manipulate them, there is a complete guide in the official documentation ([Numc]).

- **SciPy**. SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. It is a collection of mathematical algorithms and convenience functions built on NumPy. It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data. SciPy wraps highly-optimized implementations written in low-level languages like Fortran, C, and C++ and combines them with the flexibility of Python (check the doc [Scib]).

- **SymPy**. SymPy is a Python library for symbolic mathematics. SymPy can simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, and much more, and do it all symbolically ([Sym]). SymPy adds more specialized modules for optimization, integration, interpolation, eigenvalue problems, signal and image processing, statistical functions, and more. It provides very nice features to the Python program, such as optimized algorithms, Cython integration, efficient data structures, parallelism and multi-threading.

- **Pandas**. Pandas is a fast, powerful, flexible and easy-to-use open-source data analysis and manipulation tool, built on top of the Python programming language. In particular, it offers data structures and operations for manipulating numerical tables and time series. Pandas is mainly used for data analysis and associated handling of tabular data in DataFrames. It allows importing data from various file formats and also to perform various data manipulation operations ([Wikf]). The Pandas library is built upon another library, NumPy, which is oriented to efficiently work with arrays. Again to understand better how it works you can read the documentation ([Pan])

Alternatively, you can use tools to directly compile Python code for improving the execution. Some popular ones are:

- **PyPy** is a replacement for CPython. CPython is the default and most widely used implementation of the Python programming language. It is the reference implementation of Python, meaning it serves as the standard against which other Python implementations are measured for compatibility. CPython is written in C and is known for its simplicity, ease of integration with other C code, and robustness. PyPy, instead, is built using the RPython language that was co-developed with it. The most important advantage using it instead of CPython is speed: it runs generally faster. The main reason of this speed up is the so-called JIT (Just in Time) compiler. A JIT compiler runs after the program has started, during the execution, and compiles the code on the fly into a form that is usually faster. JIT compilation combines the speed of compiled code with the flexibility of interpretation ([Wikd]). PyPy works very well when we compile log-running program with many parts of the code that are pure Python code (not run time libraries, for example). After the installation of PyPy, with pip or conda, we can simply run a Python script with the keyword pypy instead than Python ([PyP]):

```
pypy script.py
```

- **Numba** is, instead, an high performance Python compiler that uses again the fundamentals of the JIT compilation. It translates a subset of Python and NumPy code into fast machine code. The main features of Numba is to decorate the functions inside our programs that we want to improve in terms of performance with the @njit keyword. Now, when we compile the script, the Numba compiler is able to identifies what are the part of the code to speed up. The Numba-decorated function can reach the speed of C or FORTRAN. It generates specialized code for different array data types to optimize performance. It is designed to be used with NumPy arrays and Numpy functions, in fact, it can automatically execute NumPy array expressions on multiple CPU cores. Numba also allows you to parallelize explicitly the code, making it easy to write parallel loops, simply adding, to the decorated function, the "parallel=True" feature as in this line of code below ([Numb]):

```
@njit ("paralle=True")
def function(args):
    \\do things
```

- **Cython** is an optimizing static compiler for both the Python programming language and the extended Cython programming language. We can also define it as a superset

of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. Basically, it translates the Python code into an optimized C/C++ version and this allows a fast program execution without losing the high programming features of Python. Cython executes the code within the CPython environment, but differently from CPython, you don't need to write C-like code, you can remain in the interface of the Python source code, but at the speed of compiled C. You can assign C semantics to some part of the code, with type declarations, and translate them into a very efficient C code. All of this makes Cython the ideal language for fast C modules that speed up the execution of Python code. One important difference from pure Python is that a Cython code should be compiled before the execution. A .pyx or .py file is compiled by Cython to a .c file, containing the code of a Python extension module. This could be done for example with the cythonize command utility:

```
cythonize -i script.pyx
```

5.2 Python on Accelerators

GPU CUDA . Numba also supports GPU CUDA programming, compiling a subset of Python code into the kernels and device function, following the CUDA execution model. CUDA (Compute Unified Device Architecture) is a developer toolkit to compile programs and run them easily in heterogeneous systems, using a set of high-level programming language extensions to use a GPU (Graphical Process Unit) as a co-processor to speed up compute-intensive applications, exploiting the power of GPUs for the parallelizable part of the computation. A general CUDA program is divided into many serial sections of code (host) that are performed by the CPU and in a few parallel ones that are instead performed by the GPU (device). Host and device have separate memory, so it is necessary to transfer data between CPU and GPU using a way similar to point-to-point message passing. The CUDA GPUs' parallel execution of code is based on stream computing, that is the subdivision of a dataset into a stream of elements. A single computational function (called kernel function) works on each element. Kernels written in Numba appear to have direct access to NumPy arrays. NumPy's arrays are transferred between the CPU and the GPU automatically ([Numa]).

CuPy, PyCUDA and PyOpenCL. CuPy is an open-source array library for GPU-accelerated computing with Python and it utilizes CUDA toolkit libraries. CuPy's interface is highly compatible with NumPy and SciPy. It acts as a drop-in replacement to run existing NumPy/SciPy code on the NVIDIA CUDA platform. We can use the same API of Numpy and Scipy (like ndarray) and use them together with the associated routines for

GPU devices ([CuP]). Also PyCUDA gives you easy, Pythonic access to Nvidia's CUDA parallel computation API (more details: [pyc]), while PyOpenCL allows you to use all the computational power of the OpenCL parallelization, which in a similar way uses the GPU paradigm to execute the code (more details: [pyo]).

5.3 Explicit Parallel Programming

Python Multiprocessing Another way to speed up and improve the execution time of a certain Python program, that could be used together with other HPC approaches, is to write the code following the multiprocessing approach, and so utilizing the Python multiprocessing module.

With a process-based parallelism, we could split the computation of some specific part of the code, usually the ones that require more effort, improving the final time of execution. A Python process could go through fours steps of its life cycle: the creation phase, the start (and run) phase, the block step (optional) and finally the termination phase, when the process is killed ([SBb]). Each Python program is composed of one process called the main process, that is the one responsible for the normal single process execution. From this process (parent process) we could create a lot of other processes (called child processes) to which we can assign the parallel computation.

When we create child processes, usually it is the operating system that is in charge of selecting the default creation method. In some systems could be the spawn methods (macOS and Windows), and in others the fork one (Linux). With spawn, the parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process objects run() method. While the parent process uses os.fork() to fork the Python interpreter, the child process, when it begins, is effectively identical to the parent. All resources of the parent are inherited by the child process. The following scripts explain two possible methods to use the multiprocessing approach in Python:

```
import multiprocessing
N = 10
def func(args):
    #doing things....
processes = []
for i in range(N):
    process = multiprocessing.Process(target=func, args=(i,))
    processes.append(process)
    process.start()
```

```
for process in processes:
    process.join()
```

The same behavior could be obtained by using the multiprocessing pool instead of the two for loop:

```
import muliprocessing
N = 10
def function(args):
    #doing things...
with multiprocessing.Pool(processes=N) as pool:
    results = pool.map(function, input_values)
```

With the usage of the fork method to create a new process, we can obtain a convenient method for sharing read-only objects that will be accessed by many child processes.

When we use process parallelism, an important aspect to consider is the fact that we need to implement manually a locking system, using for example a mutex object (like a lock) to prevent the so-called race condition. A race condition can occur when two or more processes (or threads) want to update a shared piece of data or resource. A Lock is an object that acts like a hall pass. Only one process (or thread) at a time can have the Lock and any other process that wants the Lock must wait until the owner of the Lock gives it up. Apart from the mutex, other synchronization system exist, such as the semaphore or the barrier. In Python, the multiprocessing.Lock() class allows to create, acquire and release of a lock to avoid race conditions.

Since in the multi-processes approach we don't have a default shared memory, we have to create manually a portion of shared memory (with the shared_memory features of multiprocessing) associated to Python objects, like numpy arrays, that now could be accessed and modified, making visible all the updates to all the processes.

We can follow many directions for creating shared memory objects for performing concurrent and parallel modification, two of the most important are:

- **Shared memory ctypes**
  - *Value*, a shared value of different ctypes, that is shared among all the processes, so the parallel modifications of this value are viewed by all the processes that work simultaneously. Sometimes could be useful to protect the modification of the shared value with a lock ([SBa]). I have implemented an example that shows the mechanism of sharing a value, a float variable that represents the pi, to compute, indeed, the pi with the Monte Carlo's formula, using multi-processes

that run at the same time (see the parallel pi computation in the Appendix A). We protect the modification of the total pi with the lock since each process, that wants to modify the total pi with its simultaneously computed partial pi, needs to wait for other process's modification. In this particular example, very adaptable to a process-based parallelism, the total time to compute the pi is reduced in a good way, for instance with a number of intervals of 100'000'000 we take 6 seconds to complete the computation (with 4 parallel processes), while with the single process method, we spent 20 seconds to calculate the pi.

- *Array*, a shared array of elements (all the elements inside such array should have the same ctype), shared among all the processes, so the parallel modification of this array are viewed by all the process that work simultaneously. Sometimes could be useful to protect the modification of the shared array with a lock. In another program example I have showed how to use the shared Array in an algorithm, the bubble sort, for performing the sort and we can see in the result that the array is sorted as expected using shared memory among processes (see the parallel bubble sort with ctype Array in the Appendix A). We have to notice that in this case the usage of multiprocessing is not an improvement in the execution time of the entire procedure because the overhead and the waiting time with only one lock for all the process could become very significant. With the usage of many locks, like one for each element of the array, the improvement could be better.

- **Shared memory with numpy**. The idea, in this case, is to share among processes a numpy array using the shared memory feature of the multi-processing module ([MS]). The intuition is to create a space of shared memory SM of the same size as the numpy array that we want to share.

```
SM = shared_memory.SharedMemory(create=True, size=size)
```

Then we create the shared numpy array A, with the desired type and shape and, as a buffer, the buffer of the shared memory SM. Now the numpy array A is shared because its buffer is created from the shared memory.

```
A = np.ndarray(shape=shape, dtype=int, buffer=SM.buf)
```

Now we can pass the shared array A to the Pool of processes or to the creation of each child process. This shared numpy idea seems to be the best in our project, since we have to deal with numpy array as shared data structure to create in parallel the HNSW. We can also reproduce the example of the bubble sort with this methodology, and again the resulting array is sorted as expected with an improvement in time with respect to the previous method (see the parallel bubble sort with shared numpy array in the Appendix A).

Python Multithreading At this moment someone could ask why we should use a multiprocessing approach, in which each process has its own memory and it costs a lot to be created and set (also in terms of overhead), especially in comparison with threads. Threads are similar to processes, however, they execute the work within the same process and share the same context. In fact, the multi-threading approach should be lighter and more convenient than using many processes. Threads, in fact, share memory space, global variables, file descriptors, code sections, and process resources, as the Figure 5.1 describes. A thread has its own registers in the sense that it writes inside the data when it is its turn, but actually also the registers are shared.

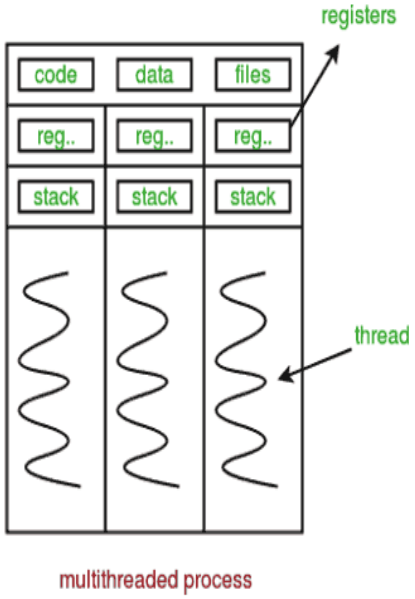

Figure 5.1: Existence of many threads inside a process

Consequently, sharing information or communicating with the other threads is more accessible than if they were separate processes, but in Python, unfortunately, there is an important limitation of using many threads for doing fast computation. This limitation is represented by the Global Interpreter Lock (GIL). In a few words, the GIL is a mutex that allows only one thread to hold the control of the Python interpreter, hence only one thread at a time can be in an execution state, causing a bottleneck in a multi-thread code ([RPA]). The reason why the GIL was introduced in Python is because it provides a lock system to prevent multi-threading programs from having some race conditions or ending up in situations like deadlocks. So, basically, threading may not speed up all tasks. A multi-threading approach could be useful and could improve the performance, especially when the application is I/O bounded.

One method to have threads that are able to release the GIL is wrote them in C language,

but the most popular way to avoid this problem is to use a multi-processing approach where you use multiple processes instead of threads. Each Python process gets its own Python interpreter and memory space, so the GIL won't be a problem. Also for what concern multi-threading, we need to pay attention to the threatening race conditions and also in this case we can implement a basic synchronization system using the Lock() object provided by the threading module. The utilization of these mutexes is almost the same as the previous multi-processing locks. To use the basic features of the multi-threading module, we can write the following lines of Python code ([Geeb]):

```
import threading
def function(args):
    #doing things....
t1 = threading.Thread(function, args)
t2 = threading.Thread(function, args)
t1.start()
t2.start()
t1.join()
t2.join()
```

in this way two threads are created and launched by the main process for executing a function simultaneously. The last thing we need to do is call the join method, which tells one thread to wait until the other thread's execution is complete. If you desire to start and launch many threads and not only two as before, you can do it as in the following code:

([Rea]):

```
import threading
N = 10
def function(args):
    #doing things....
threads = list()
for i in range(N):
    x = threading.Thread(target=function, args=(i))
    threads.append(x)
    x.start()
for thread in threads:
    thread.joiin()
```

Finally, the same behavior could be obtained by using the ThreadPoolExecutor instead of the two for loop:

```
import threading
N = 10
def function(args):
#doing things....
with ThreadPoolExecutor(max_workers=N) as exec:
    exec.map(function, range(N))
```

In both the methodologies, the order in which threads are run is determined by the operating system and cannot be predicted.

OpenMP OpenMP is an industry standard API of C/C++ and Fortran for shared memory parallel programming. OpenMP has three components: compiler directives and clauses, runtime libraries, and environment variables. OpenMP uses the "fork and join" execution model: the master thread forks new threads at the beginning of parallel regions, multiple threads share work in parallel and threads join at the end of parallel regions, as explained by the Figure 5.2.

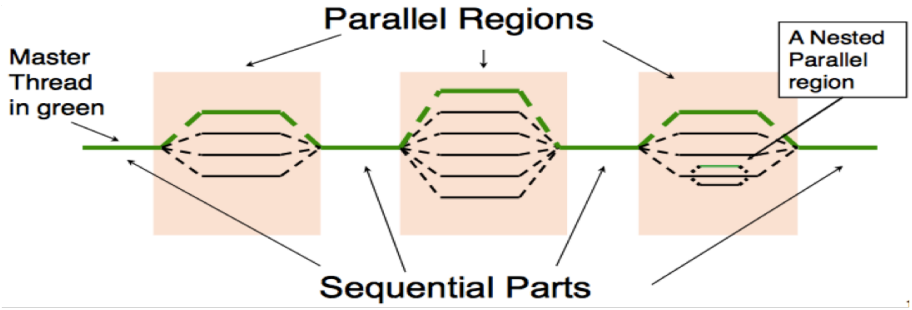

Figure 5.2: OMP Parallel Execution Scheme

In OpenMP, all threads have access to the same shared global memory. Each thread has access to its own private local memory. Threads synchronize implicitly by reading and writing shared variables. No explicit communication is needed between threads ([NERa]). Various Python packages such as Numpy, Scipy and pandas can utilize OpenMP to run on multiple CPUs. Python users can also use OpenMP directives directly in Python via a new project called PyOMP. These directives can be used to achieve low-level thread parallelism in loops, for example. Users can install an experimental development version of Numba with PyOMP support from the conda channel using the following command:

```
conda create -n pyomp python numba -c drtodd13
 -c conda-forge --override-channels
```

To make a simple example on how to use PyOMP for enabling the power of OpenMp parallelism during the execution of a python program we can write something like that:

```
from numba import njit
from numba.openmp import openmp_context as openmp
@njit
def pi(num_steps):
    step = 1.0/num_steps
    sum = 0.0
    with openmp("parallel for private(x) reduction(+:sum)"):
        for i in range(num_steps):
            x = (i+0.5)*step
            sum += 4.0/(1.0 + x*x)
    pi = step*sum
    return pi
pi = pi(1000000000)
print(pi)
```

This script is taken from [NERb] and it computes the pi number following the OMP parallel approach. We declare the for loop to be executed in a parallel way, specifying that the variable x is private for each thread, while the sum is shared.

MPI. The Message Passing Interface (MPI), is a standardized and portable message-passing system designed to work on a wide variety of parallel computers and machines. The standard defines the syntax and semantics of library routines and allows users to write portable programs in the main scientific programming languages (Fortran, C, or C++). Since its release, MPI has become the leading standard for message-passing libraries for parallel computers ([fPb]). MPI is based on communication between processes. Communication could be of two main types:

- a *point to point* communication, when a process p1 sends message to another single process p2.
- a *collective communication* when a process p1 sends message to a collection of processes.

For both of these two kind of communications there are a lot of methods (recv, irecv, send, isend, scatter, gather, broadcast) that could be used. To allows us to use the features of the MPI interface in Python, we can import the mpi4py, a Python wrapper for MPI. Mpi4py provides MPI bindings for the Python programming language, allowing any Python program to exploit multiple processors, but also multiple workstations ([fPa]).

We need to separate who is the process that sends the message and who is the one that receives such message (in a point to point communication) as the following code, taken by the official mpi4py documentation [fPc], demonstrates :

```
from mpi4py import MPI
import numpy
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
```

As we are able to notice, we can also use the numpy array as the content of the message, speeding up the communication with respect to the normal data array. With regard to collective communication, we can structure our program in such way:

```
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = np.arange(100, dtype='i')
else:
    data = np.empty(100, dtype='i')
comm.Bcast(data, root=0)
for i in range(100):
    assert data[i] == i
```

In this particular example, we use the broadcast method of communication, but we can also use one of the other sending approaches, with some small changes to the code. To run a Python script with the mpi4py features, we have to launch this command (the -n options specifies the number of processes to be used simultaneously):

```
mpiexec -n 4 python script.py
```

Chapter 6

Profiling

In this chapter, I want to explain how we can profile our Python program, firstly making an overview of some good practices, accompanied with some results which are intended to be only examples (they do not have the aim of providing insight on our FISHDBC program). After, we collect and report some information about the execution of the original single process FISHDBC, because, understanding how the execution time is distributed among all the functions of the algorithm, could lead us to write the right improvements in the right position to parallelize the computation and reach a correct and efficient parallel version of the FISHDBC, the final target of my thesis.

The main step in the entire profiling process, suggested in the literature and in all the online examples, is program profiling. Program profiling means analyzing many aspects of the code, at first in a very general way, like calculating the total time of execution of the program, and after going deeper into the details, like, for example, profiling line by line some main functions that are responsible for slowing the overall execution. After the profiling, we could be able to understand what are the parts of the code in which most of the running time is spent and so we could be able to modify these program's sections or apply some tools, provided by the Python language, to improve these specific parts.

There are many ways to profile the code, many tools, many libraries, and many terminal commands. Searching online and in some slides and papers, I have found some widespread practices and I have decided to use them to implement the profiling steps of the single process FISHDBC.

6.1 Execution Environment

The following experiments, inside this chapter, have been performed inside a remote Linux machine (Ubuntu2) composed of 18 physical cores (2 threads per core, 36 logical cores) with a Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz processor and 128GB of RAM, with the Python's version 3.10.12 installed. At the beginning of this procedure, a time profiling is needed ([Lisb]). I have launched the Python script of the program for starting the execution of the FISHDBC algorithm with the Unix-like system command time, as follows:

```
$ time python3 fishdbc_example.py --dataset blob
--distance euclidean --nitems 10000 --centers 10
```

In this way, when the execution is completed, the result shows the total running time:

```
real    0m13.718s
```

The real-time, in this and all the following results, represents the time spent inside the program, the elapsed real (wall clock) time used by the process. For obtaining more precise and informative results, the suggestion is to compute an average of the execution times for a certain number of N runs of the algorithm. To do that, the Unix utility multitime exists. Multitime is, in essence, a simple extension to time utility, which executes scripts multiple times and prints the timing averages, standard deviations, medians, mins and maxes. With the usage of the -n option, it is possible to specify how many times a command should be executed ([um]). The following multitime command:

```
multitime -q -n 10 python3 fishdbc_example.py --dataset blob
--distance euclidean --nitems 10000 --centers 10
```

produces such output:

```
==> multitime results
1: -q python3 test_case_single.py
            Mean        Std.Dev.    Min         Median      Max
real        13.942      0.228       13.647      13.959      14.263
```

in which we can know what is the average of the time execution, but also, we can discover if among the different runs there is an important variation, or if the results are similar among all the executions.

For what concerns time profiling there are also other practices that we can adopt, writing some instructions, this time, directly inside the code of our programs. The simplest method is to import the time module and take the time both at the beginning of the program or the part of code that we want to analyze and at the end of that. Subtracting these two different times we obtain the execution time for the interested part. In my profiling implementation I have also used this methodology in a complementary way to the previous method.

```
import time
start = time.time()
    ...
end = time.time()
print('The time of execution of above program is :', (end-start))
```

Another way to take the seconds spent by the run of the algorithm could be the one using the timeit module, more useful for measuring simple expressions inside code.

As second important step in the profiling procedure is the code profiling of our program. We can see this step also as a finer grain profiling since we pass from global time to timing associated with specific functions and even lines of code. To do that we take advantage of a useful Python tool called cProfile ([Lisa]), an optimization, since it is written in C language, of the profile Python standard library. Launching the following Python command with the cProfile module enabled

```
$ python3 -m cProfile -s cumulative fishdbc_example.py
--dataset blob --distance euclidean --nitems 10000 --centers 10
```

we obtain, printed directly in the terminal, a lot of timing details for each single function call. Specifically, the result shows, for each function, the number of calls of that function, the total time spent on it, the time spent on each call, the cumulative time associated with the function and the time spent by the others function called inside it. Of course, also the name of the involved function is displayed, with indications of the belonging file and of the number of code's line in which the procedure is called in that file. These metrics can give us an intiution on where most of the processing of the function is done. The -s cumulative option, in the command above, simply orders the result in a decreasing way according to the cumulative time parameter, for highlighting immediately the most expensive functions. We have also to clarify that when we execute a Python program with the cProfile option, especially in some cases in which there are a lot of different functions to be profiled, like in our scenario, the overhead introduced by the code profiling procedure could be important.

```
26831318 function calls (26795577 primitive calls) in 20.791s

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall
1635/1    0.034    0.000   20.793   20.793
{built-in method builtins.exec}
1     0.000     0.000   20.793   20.793
fishdbc_example.py:1(<module>)
1     0.017     0.017   18.726   18.726
fishdbc.py:177(update)
5000    2.603    0.001   16.144    0.003
fishdbc.py:135(add)
    ...
```

Since the result of the cProfile module, as the one above, is not so user-friendly, there are other tools very useful for the visualization of these outcomes in a proper and more readable way. One of the main among these tools is Snakeviz (installed with pip install Snakeviz, or the conda installer), a browser-based graphical viewer for the output of cProfile module ([Gro]). In fact, first of all, we need to write the output of the program's profiling result in a specific file with a precise extension, .prof, later processable by Snakeviz. To do that, it is necessary only to re-run the program with the option -o out.prof, in addition to -m cProfile. Now we can open the out.prof file on the browser with the following command typed in the terminal:

```
$ snakeviz out.prof
```

Snakeviz will choose, as the default port for starting the local web server that will display the graphical result, port 8080, if it is free (otherwise it will choose 8081, 8082 and so on and so for), and it will open a web page in your default browser, in which it is possible to better understand the information about the profiled code, as the below Figure 6.1, associated to the previous cProfile result, demonstrates. Here we can see that the final execution time is different from the normal execution of the FISHDBC with exactly the same parameter, due to the profiling overhead.

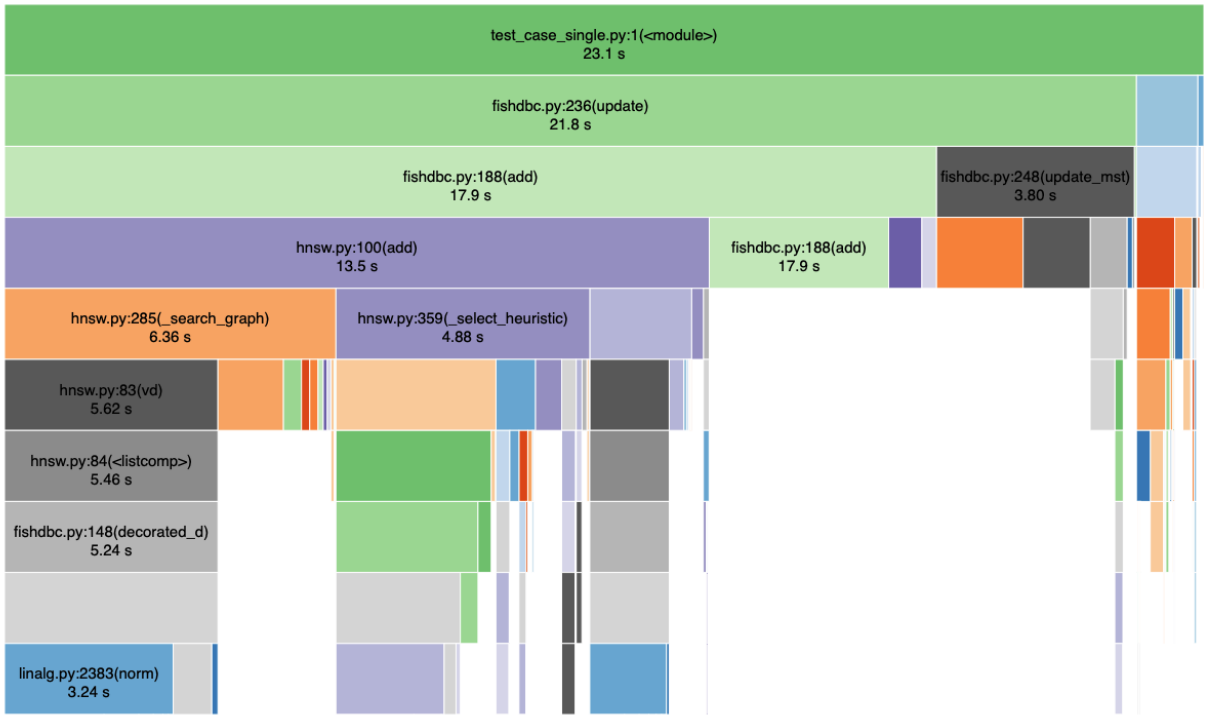

Figure 6.1: Snakeviz graphical visualization of code profiling

A classical Snakeviz graphical representation of the cProfile results, like the one illustrated by the Figure 6.1 above, shows the stack of the top 10 (10 is the default number) function calls according to the decreasing order of the cumulative time. we can also choose to see the top 15 or top 20 functions. Obviously, in this web page are also displayed all the detailed information as before, in the form of a table, represented below the graph. There is the possibility to interact with the graph (there is also another kind of graphical view, called sunburst) by clicking on a specific horizontal bar, to put the clicked function at the top of the stack, analyzing the various function calls. In conclusion, we can say that these graphs are useful to communicate to others information about the execution of a program under test, and it helps to visually identify which part of the process consumes the most computation.

Additionally to the cProfile module another one exists, the line profiler Python module (that can be installed with pip install line_profiler). The line profiler is a very useful tool used for profiling the code of some selected functions line by line. It could be considered as the second step after the analysis with the cProfile, because now that we know what are the most problematic functions, we can go deeper in the analysis of such functions, profiling them in a very meticulous way. There are many ways to use the line profiler, in the simplest case in which you have all the functions that you want to profile inside

the main file, you can decorate such procedures with the special word @profile. After this operation, we can visualize in the bash the result using the kernprof tool, writing the following command:

```
kernprof -l -v --dataset blob --distance euclidean
 --nitems 10000 --centers 10
```

Kernprof, for each of the decorated function, shows, line by line, all the info, as the text below demonstrates. The most useful insight could be the percentage of time that each line spends inside the function because we can discover what part of the already most expensive functions is the one that slows the computation. This is much more descriptive than the results obtained from cProfile, but the idea is to use cProfile to identify which functions you should evaluate with line profiler.

```
File: fishdbc.py
Function: update at line 177

Line#...%Time  Line Contents
==============================================================
177            def update(self,elems,mst_update_rate=100000):
184     0.0      for i, elem in enumerate(elems):
185     90.8       self.add(elem)
186     0.0        if i % mst_update_rate == 0:
187     0.0          self.update_mst()
188     9.2      self.update_mst()
```

Since the implementation of the FISHDBC is composed of many files with different class objects connected by a central main script, we cannot simply write the keyword @profile before each selected procedure. We need instead to use the LineProfiler class for adding, to an instance of this class created in the main file, all the functions to be analyzed, that are situated in other Python modules and therefore enable the profiling, like this excerpt of code exhibits:

```
from line_profiler import LineProfiler
    ...
lp = LineProfiler()
lp.add_function(fishdbc.update)
lp.add_function(fishdbc.add)
lp.add_function(fishdbc.update_mst)
lp.add_function(hnsw.HNSW.add)
    ...
```

```
lp.enable_by_count()
```

A useful practice is to save the output of the kernprof command in a specific output.txt file, instead of in the standard output (the terminal), simply redirecting the output to a desired file in this way:

```
kernprof -l -v --dataset blob --distance euclidean
 --nitems 10000 --centers 10 > out.txt
```

6.2 Blob Data Set

At this point when we know how to profile complex Python code, we can start to collect some profiling results of the single process FISHDBC in order to understand what are the main parts of this algorithm that steal most of the execution time when we run the single process procedure.

In this first section we analyze some profiling results of FISHDBC's executions over a blob data set, generated directly inside the code with the Scikit-learn make_blobs() function. We need to specify that, to obtain a coherent results, every time we created the blob data set, we used the same random seed for the generation (otherwise it will be different), hence the different experiments will use exactly the same data set. To obtain some useful insights we can perform the profiling using different distance functions, as long as the FISHDBC allows us to pass as parameter many metrics.

We need to highlight the fact that, in general, depending on the structure of the data set and the type of distance (some distances work well with numbers, some others with strings, or some with sparse data set), the execution time could change among the different tests. Additionally, we took the exec time using the aforementioned multitime Unix utility to take the average (and also the Standard deviation, which could be useful and interesting) of a certain number of runs. We have decided to consider 10 as a good number of executions for the experiments.

Distance	Mean	Std. Dev.
Euclidean	13.930 s	0.159s
SQEuclidean	14.552s	0.246s
Minkowsky	14.175s	0.336s
Cosine	20.612s	0.325s

Table 6.1: Results with Blob data set, 10 runs, 10'000 data items and 5 centers

As we can see in 6.1, when we run the algorithm over the blob data set, created with 10'000 data items and with 10 centers, so 10 different classes of items inside the data, we can notice that with some distance functions (Euclidean, SqEuclidean and Minkowski) the algorithm perform quite well with a small execution time. The similarity in the execution time between these three metrics depends on the fact that the distances are similar too: the Euclidean distance, also known as the L2 norm or Euclidean norm, is a fundamental concept in geometry and mathematics used to measure the straight-line distance between two points; The squared Euclidean distance, also known as the squared L2 distance, is a distance metric closely related to the regular (non-squared) Euclidean distance but lacks the square root operation, making it computationally less intensive; The Minkowski distance is a generalization of various distance metrics, including the Euclidean distance and the Manhattan distance. The Minkowski distance can be customized with a parameter "p" to emphasize different characteristics of the data.

All the three distances are used to measure the dissimilarity between two points or vectors in a multi-dimensional space. With the Cosine distance, instead, the conclusions are different, in fact the time increase a bit. Looking at the Figure 6.2, that wants to explain the growth behavior of the distance function's execution time with respect to an increase in the number of items of the blob data set, we can observe that the Cosine line (the purple one) is always greater than the others three lines, exactly because the Cosine distance function, when it is called inside the FISHDBC, performs worse than a classical Euclidean distance (the square Euclidean and the Minkowski are very similar to the normal Euclidean).

Why it perform worse? Because Cosine distance is particularly well-suited for data that can be represented as vectors in high-dimensional spaces, where the direction of the vectors is more important than their magnitude. In a 2D space, this means that the angle between two vectors is the primary factor in determining the cosine distance, and this angle is not affected by scaling. Euclidean distance (and its similar metrics), however, is sensitive to the scale of the vectors, as it considers both the x and y components in a 2D space, and since our blob data sets are composed by points in a 2D space in which the X and the Y dimension are both important, the Euclidean, Squared Euclidean and Minkowski distances work better than the Cosine.

These characteristics of the Cosine distance cause it to slow the HNSW creation process and consequently the FISHDBC algorithm (the HNSW creation is the main bottleneck, as we will see immediately after), because the HNSW computation is the one responsible for calling many times the distance function, so its performance strictly depends on the used dissimilarity. The small difference between Euclidean and Squared Euclidean is the fact that, despite in the literature the Squared Euclidean should be less expensive due to the fact that we do not perform the square root, the Scipy distance function associated with the squared Euclidean is a little more costly than the Scipy Euclidean distance (single Euclidean call's time: 0.000002034s, single SQ Euclidean call's time: 0.000003149s). Anyway,

doubling the data size from 5000 to 10000, from 10000 to 20000 and so on and so for, the execution time grows, for all these four metrics, almost linearly (not exponentially) and we are able to state that the algorithm is scalable with respect to the growth of the number of items in the input data set.

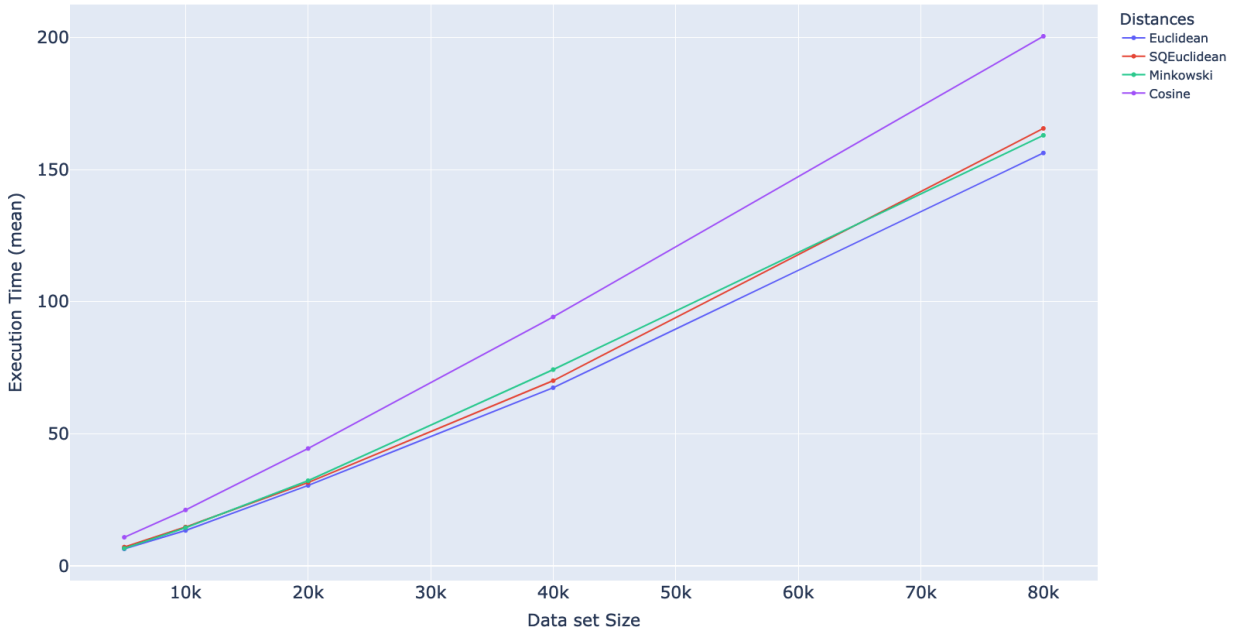

Figure 6.2: Execution Time with various blob dataset's sizes using different distances functions

In this configuration (10'000 items, 10 centers), for all the distances, most of the time is spent inside the add process of the FISHDBC, more specifically the hot-spot is identified as the add operation of the HNSW structure, as shown in the Figure 6.3 below. The other meaningful bottleneck is the creation of the MST (the input of the further HDBSCAN clustering operation), conducted by the update_mst() function. Such Snakeviz picture refers to a profiling operation's result when the Euclidean distance is used as the metric for computing the FISHDBC algorithm, but also for the other distances, despite the differences in the final time, the time's distribution remains very similar (see the Figure 6.4 showing the profiling result with cosine distance).

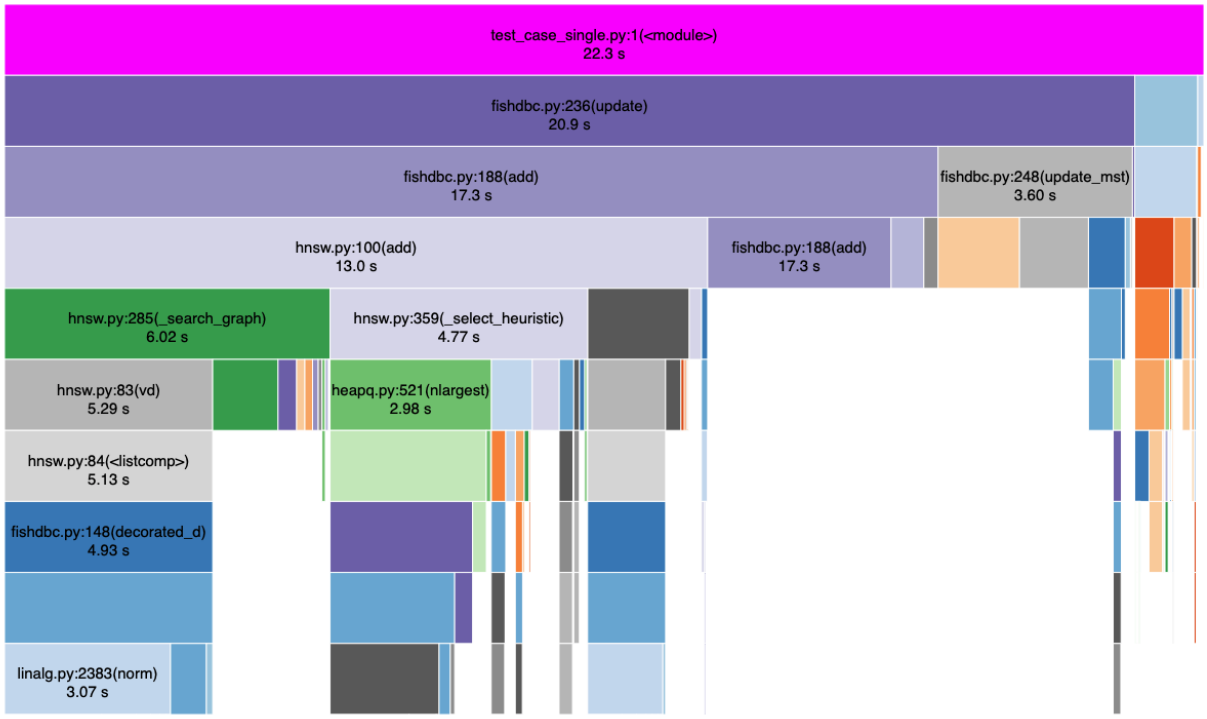

Figure 6.3: Snakeviz visualization of FISHDBC execution over blob dataset with Euclidean distance

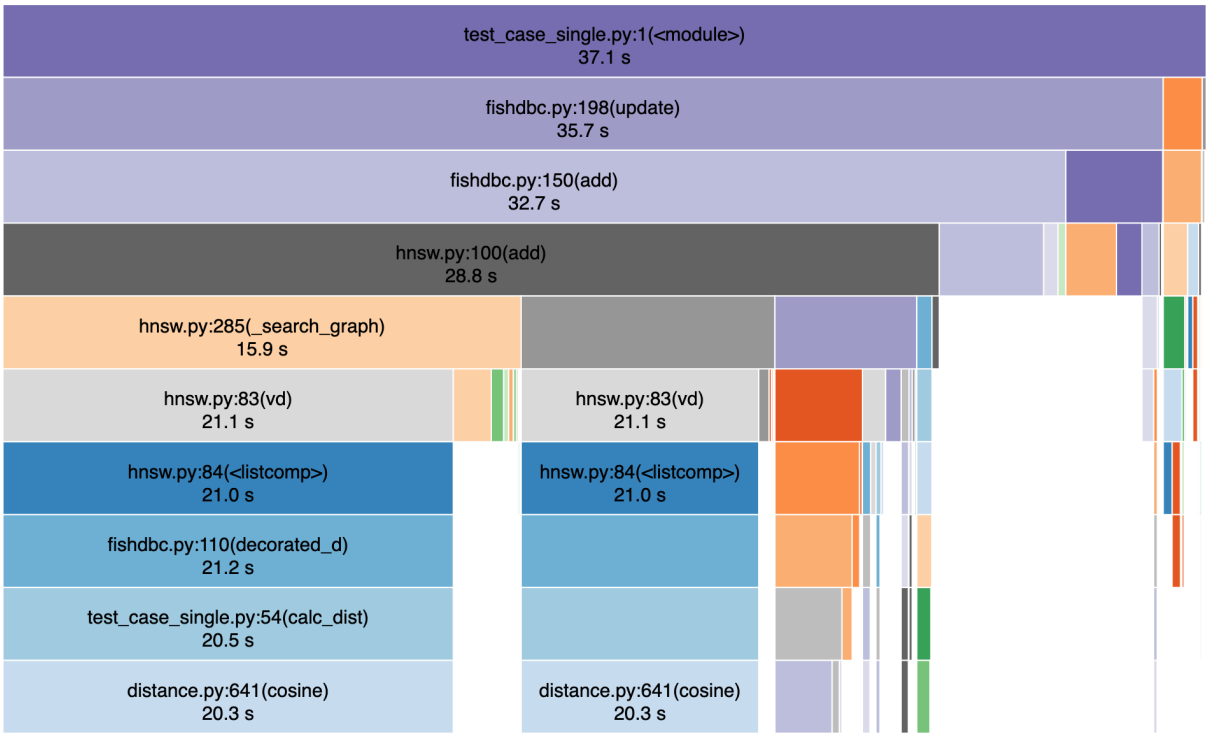

Figure 6.4: Snakeviz visualization of FISHDBC execution over blob dataset with Cosine distance

After collecting lots of profiling results for a blob data set, we can assert what is the overall execution time of the single process FISHDBC, and we can state also what are the hot-spots identified inside the algorithm, that represent most of its running time.

The two main single process FISHDBC's bottlenecks are the HNSW creation (more specifically the add function of the HNSW class) and the MST creation (caused by the FISHDBC's update_mst method). Among these two parts of the code, especially for a blob data set, the section that slows the entire algorithm in a more significant way is the HNSW procedure.

6.3 Text Data Set

In this other section of the profiling chapter, we want to profile and analyze our FISHDBC program, but this time when it runs with a text data set as input of the clustering algorithm. We want to understand if the distribution of the execution time remains similar to the one of the blob data set. Since now we are dealing with strings, using the previous distance metrics makes no sense, because they work well with numbers. Instead, we could use two

dissimilarity functions well suited to compute the distance between elements of a text data set: the Levenshtein and Hamming distance. But, since the Hamming dissimilarity works only with the assumption that two strings should have the same lengths, with our text data sets we can only use the Levenshtein metric.

The Levenshtein distance, also known as the edit distance, is a measure of the similarity, or dissimilarity, between two strings. It quantifies the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. The calculation of the Levenshtein distance involves constructing a matrix where each cell represents the cost of transforming a prefix of one string into a prefix of the other. The goal is to find the minimum cost (or number of edits) to transform one string into the other. This number of edits is indeed the value of the distance between two elements.

Distance	Mean	Std. Dev.
Levenshtein	18.217s	0.775s

Table 6.2: Results of synth text data set, 10 runs: 10,000 data items, 5 centers

The Table 6.2 above shows the results of the FISHDBC execution over a synth text data set. It's worth specifying that for making experiments with a good textual data set, now and after in the chapter 8, we have self-generated, thanks to a Python script, the data set. In fact, it is very complicated to find text data sets well suited for clustering operations. Anyway, in such a table we are able to see that the execution time is quite good, so when the text data set has a medium/small size, even if we are working with text data, the performance remains nice.

For this scenario, the distribution of the execution time becomes similar to the one associated with the previous experiment with blob data, as displayed by the Figure 6.5, where the main expensive part inside the code still remains the HNSW computation rather than the MST calculation. This time, however, there is more balance between the main hot-spot's execution time. For this particular scenario, the FISHDBC add function is another expensive part besides the HNSW creation. Since the FISHDBC add function is also the place in which the computation of the candidate edges for the MST creation occurs, for text data we can say that the MST creation became a hot-spot almost as significant as the HNSW creation.

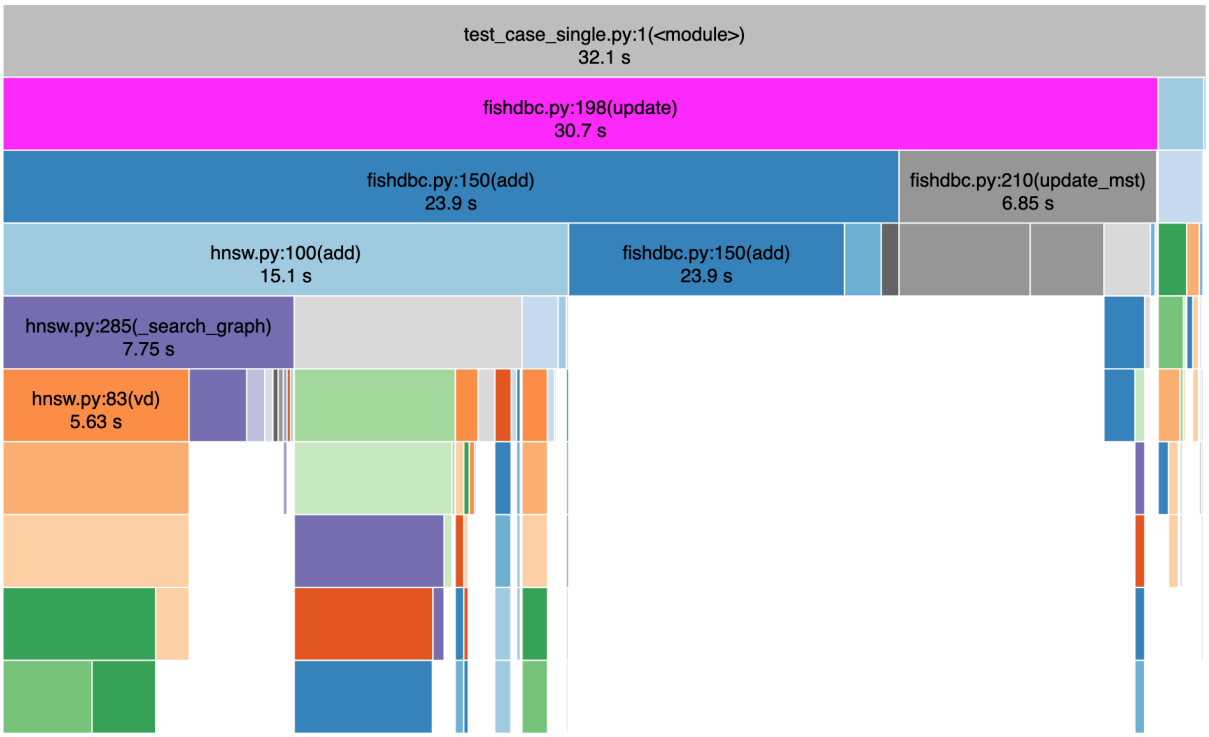

Figure 6.5: Snakeviz visualization of FISHDBC execution over synth text data set with Levenshtein distance

To summarize what we have discovered after the FISHDBC's profiling phase when it has to process textual data, we can state that the execution time, as always, depends on the size of such data sets and also on the distribution of their points, but the behavior of the FISHDBC's running time informs us and confirms that there are two main expensive parts inside the algorithm, where the HNSW procedure remains the most computational-intensive one, even if there are some differences with respect to the blob data set's profiling results.
Additionally, I want to highlight the fact that in all the already presented Snakeviz plots there is overhead, in terms of execution time, introduced by the profiler cProfile. This is why the time on such images differs from the one recorded without the profile option enabled. With the usage of the line profiler, the overhead becomes even more significant, so we have to take this fact into account analyzing the final execution time of a profiled program. Obviously, the profiler and line profiler are useful only for the first runs, to discover what are the hot-spots in the code, therefore this overhead can be not considered for the final time comparison.

Chapter 7

Parallel Implementation

Once you have defined what are the bottlenecks of your code, you can start to modify the implementation accordingly (you can find my associated repository at `https://github.com/edo-pasto/Parallel-Flexible-Clustering.git`), with the aim of improving the final execution time. In our case, after the profiling procedure, two main costly parts emerge: the FISHDBC add phase, more precisely the HNSW add function, introduced in the algorithm 4 (in other words, the creation of the HNSW data structure) and the MST computation (based on the distances computed during the HNSW process), illustrated in the algorithm 9.

Now, we have to focus on which ways can be used to reduce the computational time of these two main parts. Since, between these two expensive components of the FISHDBC procedure, the slowest is the HNSW creation, firstly we decided to spend our effort on understanding how to speed-up this portion. To do that we ask for help from the multiprocessing module, supported by the shared memory feature, to parallelize the execution of the distance function between the elements inside the HNSW data structure. Later, we will see in which way we can integrate the parallel HNSW graph with an efficient creation of the MST, since it will be fundamental for the final HDBSCAN clustering operation.

7.1 Parallel HNSW

Now we can explain the procedure to go from the original sequential HNSW creation of the original FISHDBC to the parallel version of it.

Sharing data. The first important aspect to consider is the need to share data. What are the components to bo shared among many processes?

1. **Original data set.** The original data set could be of two different types, a blob data set: a multidimensional numpy array composed, for each point, by a couple of float numbers. This data set is a synthetic data set in the sense that it is created directly inside the code thanks to the Scikit-learn make_blobs function. The other possible type of data set, available for the execution of the procedure, are the text data set. Since it is very difficult to find a textual data set tailored for making experiments with clustering approaches, we have again generated a synthetic text data set, starting from some string centroids.

2. **Distance function**. It is the dissimilarity that we want to use for the distance computation between data to obtain each node's neighbor list. In our implementation, it is possible to use different functions: Euclidean, Square Euclidean, Minkowski, Cosine, when we are dealing with a blob data set, or Levenshtein, when we are working with a textual data set.

3. **Level list**. It is a Python list of tuples. Each tuple contains two integer numbers. The first one is the index of the element in the original data structure, and the second number represents the level at which the element has been assigned. The choice to decide at which level assign an item is done pseudo-randomly, but in a way in which most of the data items fall in the first levels, which are denser with respect to the higher levels.

4. **Members list**. This time it is a list of lists. Each inner list represents a level of the HNSW, and inside it, there are the indexes of the elements that belong to such a specific level. So Members[0] is the first level of the HNSW structure and inside it, there are all the elements that are in level 0.

5. **Positions list**. This object is a list of dictionaries, each dictionary again corresponds to a level and each couple key-value inside each dictionary is composed as follows: the key is the element's index, while the value is the position of the associated index in a level. In this way, we know what are the positions of each element inside the different levels of the HNSW graph. Having such a data structure allows us to perform retrieval of an element's position in a quick way since, after redirecting our search to the correct level, we need only to perform a simple dictionary lookup by key. If we used the members structure we should search inside a list, looping over it and so in a more inefficient way. For example poisitions[2][4] returns the position of element 4 in level 2 of the HNSW.

6. **HNSW graph**. The most important structure to deal with is indeed the HNSW graph. Originally, this structure was a list of dictionaries, in turn, each dictionary contains other dictionaries. The first inner level of dictionaries represents the different graphs' levels. Inside them, we find as keys the elements that belong to a specific level, while the value is another dictionary. This second inner level of dictionary has

keys that are again the index of elements and the values are distances. So this second dictionaries correspond to each element's neighbor list. To make an example of how to use such HNSW structure, if we want to retrieve the neighbor list of the element 4 at the 2nd level of the structure, we should write: graph[2][4] With graph[2] we retrieve the dictionary corresponding to the second level of the HNSW. In this 2nd level we take the neighbor's list of the element 4, retrieving the dictionary associated to the key 4.

For the distance function, the original data set (blob or textual), the members and levels array, and the positions dictionary array, since they are all read-only objects, we don't need any kind of locks or synchronization mechanism to access them even if they are shared among processes. In addition, it is enough to start every child processes from the main (father) process with fork, as process start method, instead that spawn. With fork, in fact, the child processes have a copy-on-write of all the object owned by the main process. Copy-on-write means that an object is really copied in a process only when it is modified. In our case, since these data structures and functions are read-only, they aren't modified and so not copied really. Of course, this aspect is a first improvement feature for what concerns the performance (time and memory performance), because we shared elements without copying them on shared memory, therefore in a lighter way.

Talking about the HNSW graph, for our purpose we need a HNSW data structure that could be shared between processes, not only to read it, but also to write inside it, so using the fork creation method is not enough anymore. For such reason we could use, for each level, two shared multidimensional numpy arrays, to represent the original graph:

- **Shared numpy array of the neighbors**. This first integer multidimensional shared numpy array is used to track each element's neighbor list. This array has two dimensions: the first dimension is the total number of elements of the original data set, while the second dimension is the defined maximum number of neighbors that each element can have. So if we use tot_neighbors_list[2][10] we could access to the neighbor's list of the 10th element inside the level 2. Simply, this first numpy object represents the edges between nodes.

- **Shared numpy array of the distances**. The second float multidimensional shared numpy array is, instead, the one needed to save the distances associated with each element of the previous array. Also this numpy object is bi-dimensional: the first dimension is again the total number of elements, while the second one is the neighbors' distance list's size. Now, if we use tot_weights_list[2][10] we could access the distances of the neighbor list for the element 10 at the 2nd level. This second numpy object represents the weight of the edges between nodes.

Main's Setting Once you have defined how to share the various objects to be read and written by the multi-processes, we can move further and start to describe the real implementation of the parallel procedure. The first important part that has to be organized in a specific way is the main, responsible for launching the parallel execution. In this script, we need to prepare all the necessary data structures.

1. We define our distance function and we create our data set. We need to precise the fact that in all the remaining parts of the algorithm, except for the real final computation of the distance function between two elements, we are going to use the indexes of the data element (so when we say element we refer to the position of the element in the input data array).

2. After, we create the level list for assigning all the elements to a specific level. The probability of an element insertion at a certain layer is given by a probability function normalized by the level multiplier. A rule of thumb for the optimal value of this level multiplier is 1/ln(M), where M is the number of neighbors that each node has (see the algorithm 2 below).

Algorithm 2 Calculate Levels

$levels \leftarrow []$
for each element of input data **do**
    $level \leftarrow \left\lfloor -\log_2(\text{random}()) \cdot \frac{1}{\log_2(m)} \right\rfloor + 1$
    $levels.append(level)$
end for
$levels \leftarrow$ list of pairs (index, level) from levels list.
$levels \leftarrow$ Sort the list based on the "level" values.

3. Now we can construct the members' list, strictly based on the levels array, to know how each level is composed. Someone now could ask why we need also such an additional list if we already have the levels list. Because if the levels array tells what is the specific level to which an element belongs, it does not inform us on how each level is composed, since when we insert a node inside a higher level it is inserted also in all the lower levels, this is why the level 0 comprehends all the data items. To make a simple example if element 10 is assigned to level 2 inside the levels list, it will be inserted in the list corresponding to level 2 inside the members structure, but not only, it will be inserted also into the lists corresponding to levels 1 and 0.

4. The last indispensable object is the positions dictionary. This time, looping over members, we associate, for each element in each level, its position. In fact, the element 10 at the 2nd level could not be at the 10th position, but at the 2nd position, for instance.

5. After the preparation of all these structures, we have to set the shared memory to be used for the shared HNSW further in the implementation. For each level, we create two regions of shared memory with the correct size, based on the number of elements of the current level. Using the buffer of these created shared memory objects we define also two shared numpy array with the correct parameters: the data type (int for the shared list of neighbors, float for the shared list of weights); the shape, number of items in the current level * M (if we are in higher levels)or M0 (if currently we are processing the shared objects for the level 0, since each element in this level has M0 = M * 2 neighbors). Actually, both the multidimensional array are initialized with some default values. The neighbors numpy array is filled by a so-called MISSING constant value. Due to the integer type of data, this missing value is represented by the largest possible int value that could be used in Python. Following the same reasoning, the weights numpy array initially is composed by MISSING_WEIGHT constant value, that is, this time, the largest float value usable in Python.

6. Another essential shared object is the one used as enter point for each item during the addition inside the HNSW graph. What it means? When we are considering an element for the insertion, we compare (calculating the distance) the current element to a non-static enter point for starting all the search and link procedure to connect the new item with the others elements already present in the HNSW. Since this enter point could change during the procedure, it should be shared among the processes, hence each process should have an up-to-date version of this point. When does it change? It changes the first time that we are dealing with an element belonging to a level that is higher with respect to all the others levels considered until such moment. Due to some implementation choices, we had to create this shared object, representing the enter point, in an intricate way. In fact, the simplest way to share an integer among processes is to create a shared Value ctype, but this is not possible if you are spawning the required processes with the multiprocessing pool method, as in our case. In a situation like that, we can "manipulate the system" and define a shared numpy array as before, based on a shared memory object of size 1 that will represent our shared value.

Of course, lots of these procedures to prepare all the necessary objects could be done directly inside the main or organized if you want in separated functions.

The Parallel HNSW Class. Reached this point we can instantiate the HNSW parallel class object, passing as parameters all the items which we discussed until now, but also the optional parameters M, M0 and EF (optional in the sense that if you don't pass any of these variables, they are set to default values).

Once you have the instance of the parallel HNSW you can start to execute, in parallel,

the add() function for each data item, exploiting the multiprocessing Pool (see algorithm 3). Such pool is created with a customized number of processes (16 as default). Now, we start the pool that spreads the execution of the add function among all the processes. When the parallel add is completed for all the data, we can close the pool and print the resulting HNSW graph. Actually, for the first data item (the element 0) we execute the add function outside the multiprocessing pool, in a single process. For the first item, in fact, the only action that we need to execute is to initialize the enter point with the first element for the next insertion, since the HNSW is void at this moment and we need the right starting enter point for the entire procedure.

Algorithm 3 Multi-process Pool for add execution

```
distances_cache ← []
distances_cache.append(hnsw_add(0))
pool ← multiprocessing.Pool(number of processes)
for each input data item starting from the item 1 (not 0) do
    dist_cache ← hnsw_add(data item)
    distances_cache.append(dist_cache)
end for
```

To dig more into the implementation details, we have to explain how the parallel add works. The HNSW add function takes a single input parameter, the index of the data set's element to be inserted in the hierarchical graph (variable "elem" in the algorithm 4). The first part of such function concerns the initialization of all the useful local variables, but also the reading of the shared variables: the enter point, that after it has been read (to have the up-to-date value), is saved in a local variable used in the rest of the function (now each process has its updated local enter point); the two shared memory objects that will be used inside others inner functions, to obtain the updated version of the two numpy arrays representing the HNSW structure.

Algorithm 4 parallel HNSW add function

```
distance_cache ← {}
level ← calc_level(elem)
sh_point ← initialize with shared memory
enter_point ← sh_point[0]
shared_weights ← initialize with shared memory
shared_adjs ← initialize with shared memory
if enter_point ≠ MISSING then
    dist ← decorated_d(distance_cache, elem, enter_point)
    if enter_point's level is greater than elem's level then          ▷ now we loop over
shared_adjs          for each adjs in the levels which we don't have to insert elem do
        if then enter_point, dist ← search_graph_ef1(adjs, elem, enter_point, dist, distance_cache, M)
            ep ← [(dist, enter_point)]

        end if                                  ▷ now we loop over shared_adjs and shared_weights
        for each adjs and weights in the levels in which we have to insert elem do
            ep ← search_graph(adjs, elem, enter_point, dist, distance_cache, M, ef)
            pos ← calc_position(elem, current level)    ▷ Insert the new node linking it with
the right neighbors
            select_heuristic(ep, pos, current level, elem, adjs, weights, M)
                                                          ▷ Insert backlinks to the new node
            for each idx, dist in adjs and weights do
                select_heuristic((idx, dist), pos, current level, idx, adjs, weights, M)
            end for
        end for

        if enter_point == MISSING or enter_point's level is less than current level then
            acquire the lock
            sh_point[0] ← elem
            release the lock
        end if
        return distance_cache =0
```

After, the function is divided into three main parts:

- **search_graph_ef1() function**. When the shared enter point's level is higher than the current element's level, we start to find the closest neighbor of the element for all the levels in which we don't have to insert it, starting from the current enter point. This function receives the shared memory region associated with the level in which we are going to search for the closest neighbor, since in this way we search and read an updated version of the graph associated with such level. We use ef1 in the function's signature because for this particular case, the size of the candidates for finding the closest neighbor is 1 (we need a single neighbor, not a neighbor list). If, in the current snapshot, we are dealing with element 10 that belongs to level 1, while the enter point is at level 3 (because it has been already considered for the insertion a data item belonging to such level 3), we need to loop over the level 3 and 2 (in which we don't need to insert the 10th element) to find , for each of these two levels, the 10's closest neighbor (with its distance). In this way, we always create a connection with the enter point when we add new items that are not at the same level as this enter point.

- **search_graph() function.** After the previous particular case, we execute the search_graph function for all the levels in which we have to insert the current element. For all these levels, the search finds a heap of candidate items to be taken into account for determining the neighbors list of the current element. Also this function, to execute the search, receives the up-to-date shared memory object related to the current level, to not perform inconsistent reads. For instance, if we are inserting element 9 that belongs to level 3, this time the same level of the enter point, we have to add such data to level 3, of course, but also to the levels 2,1 and 0, finding for all these levels the element 9's neighbors list. For both these two search functions the code of the parallel version is something similar to the original one, except for the fact that after having read the shared objects, we deal with numpy arrays and no more dictionaries.

- **select_heuristic() function**. Strictly correlated to the previous search method, since it is executed in the same loop after the search finds the neighbors list, the select_heuristic() is used to link, with new edges, the current element to the retrieved neighbors, but also the opposite, the discovered neighbors to the new element. This function, together with the HNSW add, has been changed in many parts, because it should get along with the multi-process parallel execution. The main idea remains the same: we want to find the neighbors list of all the items that are inside the current element's neighbor list. Now, we execute a particular inner function called prioritize() that returns, for each item of the current element's neighbor list, the lowest distance value in each item's neighbor list. Among all these items, the ones that we want to really connect to the new element are the first M considering their distance values. The real connection happens just before, looping over these M items and inserting each one of them inside the neighbor's shared array and inserting also their associated distances inside the weights shared array. An element is inserted in these array in the first MISSING value position encountered during the loop (so the index and its distance are at the same position, but each one in the corresponding array). If, during this process, the new element's neighbor list is already full, but there is an item that has now a smaller distance value, we should replace the worst neighbor with this better item. With this quite complex procedure, we create a connection to a node only if it is closer to the newly inserted element compared with all the already connected nodes.

In addition to these main procedures, we have created new small functions, very crucial for parallel multi-processing implementation.

- **decorated_d() function**. Following the original version's idea, the decorated_d function is called every time we have to calculate the distance between the new element and another node. Instead of computing directly the distance, calling the

decorated_d, we firstly check if we have already calculated the distance with the node, maintaining, for each different item, a distance cache (crucial for the second part of the parallel FISHDBC). To make an example, if we call the decorated_d function between the element 10 and the node 15, we check, in the distance cache of the element 10, if we have already computed the distance with 15, if it is true, we return the corresponding value without calling the distance function (and saving time), otherwise we need to compute the metric dissimilarity between 10 and 15 and return its value (see the algorithm 5). Each time that we are considering a different data to be added to the HNSW structure, we clear and recompute its distance cache, in such a way that each item has its own cache.

Algorithm 5 decorated_d

```
1: function DECORATED_D(distance_cache, i, j)
2:     if j is in distance_cache then
3:         return distance_cache[j]
4:     end if
5:     distance_cache[j] ← dist ← distance(data[i], data[j])
6:     return dist
7: end function=0
```

- **calc_level() function.** This simple method is used every time we need to know what is the correct level at which a specific element belongs to, searching the element inside our previous constructed positions structure (look at the algorithm 6. It turns useful when, inside the add procedure, we want to compare the new element's level and the enter point's one.

Algorithm 6 calc_level

```
1: function CALC_LEVEL(elem)
2:     for each level and each dictionary in positions, looping it backwards do
3:         if elem is in dictionary then
4:             return level + 1
5:         end if
6:     end for
7: end function
```

- **calc_position() function.** Such other function, similarly to the previous one, allows us to know what is the position of an item inside a specific level of the HNSW. For doing so, we can use the positions list of dictionaries, exploiting the good property of search by key inside a dictionary (explained in the algorithm 7).

Algorithm 7 calc_position

```
1: function CALC_POSITION(to_find, level_to_search)
2:     return positions[level_to_search].get(to_find)
3: end function
```

The Usage of Locks. In all this creation process of the parallel HNSW we have to use a lock in the specific moment in which we change the shared enter point with another value, because the level of the new element is greater than the one of the actual enter point (when we consider for the first time a point of a new level it becomes the new enter point). We need a lock because, otherwise, we could encounter some inconsistency, since some process could read some enter point's value that is not updated and use it in the remaining part of the procedure.

Theoretically, we should need locks when we modify the two numpy arrays (the HNSW structure), one for each element, or, taking inspiration from the Hnswlib, one lock for each number N of elements (finding the right trade-off between the overhead to manage and create lock objects and the waiting time). The problem here is that the performances are heavily affected by such a synchronization system. For this reason, we applied a lock-free solution to our parallel implementation. Such a solution is acceptable because our algorithm provides an approximated result, hence the eventual loss of precision is tolerated (also because such accuracy loss is not remarkable, as visible in the Table 8.8), while the improvement of the execution time becomes very significantly, as exposed by the Table 8.1

Testing the HNSW results. We need to test the quality of the search result over our HNSW graph to understand if it was constructed in the right way or not, a crucial test for discovering the possibility of not using the locks among the many processes while they are writing simultaneously new values on the two shared numpy arrays.

For doing such a test we have created a particular script that executes both our parallel creation of the HNSW and a state-of-the-art KNN search over a KDTree, created from the same input data of the HNSW. To do that we import in our project the KDTree object from the Scipy library (specifically the scipy.spatial module). When we import our data set, we use a small part of it (like the 10%) as data to pass the search test. For example, if our data set is composed of 110'000 items, we use 10'000 elements as test data and 100'000 for the creation of the HNSW and the KDTree.

Now we create our multilayered graph and the KDTree (with 100'000 items) and we call the parallel HNSW class function search() giving as input the 10'000 items, that it is the same input also for the KDTree.query() method. Both the functions return an array with the result of the search: the K, or M in our case, nearest neighbors for each element. We iterate over these two lists to understand if they are the same or not, counting the number of different elements in each neighbor's list between the state-of-the-art implementation and our version.

Among all the possible distances, the only ones that are comparable, because they are available on the KDTree, are the Euclidean and Minkowski. For instance, without the usage of the locks, the overall percentage of errors remain quite good during 50 different

runs. We have tested the quality of the search result for 2'000 data items, used as search input inside the parallel HNSW created starting from a blob data set of size 20'000, and we have obtained a percentage of error of 3.58% on the remote Linux machine.

7.2 MST Computation & HDBSCAN

All the procedures described before are specific to the HNSW creation. If you want to obtain only the HNSW graph, you can follow them. But, since for our case, we need also to perform the remaining part of the FISHDBC algorithm, we have to change a bit the parallel execution of the hnsw_add function, because we should integrate this step with the computation of the MST and later on the HDBSCAN.

MST. In the main process (and in the main script) instead of passing the single hnsw_add function to the multiprocessing pool, we pass the add_and_compute_localMST function.

Such a function does not work with a single data point, but with a range of elements (as demonstrated by the algorithm 8). In fact, we split the array containing all the input data in multiple ranges, as many as the processes that want to use. For example, if we have 20 input items and 4 processes (reminding that the first element is already inserted in a single process), we split the data into 4 ranges: from 1 to 5 (not from 0), from 5 to 10, from 10 to 15 and from 15 to 20.

Algorithm 8 Multi-process Pool for _and_compute_local_mst execution

```
partial_mst ← []
hnsw.hnsw_add(0)
pool ← multiprocessing.Pool(num_processes)
for local_mst in hnsw.add_and_compute_local_mst do
    partial_mst.extend(local_mst)
end for
```

Each process executes the add_and_compute_localMST with its own range of points. Therefore, each execution of this new function, starts the add procedure for only a set of points, and it computes a local version of the MST (explained in the algorithm 9). Basically, a process calls the hnsw_add() for each element of its range. Each single hnsw_add() returns the distance cache associated to the passed item. With this list of caches, we compute a local MST (local because to compute it we consider not all the distance caches, but only the ones associated to the point of the range assigned to the current process).

To compute a partial MST, we loop over the distance caches and the associated points, calculating the list of candidates for being edges inside the MST. At the end of this method, each process returns its own local MST to the parent process. The main script now is

Algorithm 9 add_and_compute_local_mst

```
1: function ADD_AND_COMPUTE_LOCAL_MST(points)
2:     distances ← []
3:     for point in points do
4:         distances.append(hnsw_add(point))
5:     end for
6:     local_mst ← local_mst(distances, points)
7:     return local_mst
8: end function
```

responsible for collecting these local MSTs in a data structure, that will be the input for the global minimum spanning tree. To compute this final MST, we call, from the main process, another function (called indeed global_mst()) that performs the Kruskal algorithm over all the previously computed MSTs. Following this idea we improve also the MST creation, because we avoid calculating the candidates edges for all the input data item, as for the original single process implementation, but we split and parallelize this action. In addition, when we call the global_mst() function we execute Kruskal over an already partially computed solution, saving time, while previously the single process execution was performed to compute the entire MST (re-computing in a single process all the MST candidate edges, the main slowing MST-related part).

An important aspect to consider is the possibility of adding a barrier synchronization object before computing the local MST. A barrier is used to stop the execution of each process until all the others have reached the same position in the concurrent code. In this way, we ensure that each process creates its own MST only when all the others have already inserted all the elements inside the HNSW. Thus, when the father process has to compute the global MST it should not re-compute the mutual reachability distance of the MST's candidates edges, since this distance, computed locally at each process, was already the right one because no more points were added to the HNSW. Actually, this is how this part of the implementation works, except for the usage of the barrier and so without the overhead and the slowdown of this sync object. But why can we not use the barrier and still have a working solution? It works because all the concurrent processes spend almost the same time performing the HNSW computation, hence they start almost at the same time as the local MST creation. It is like an implicit barrier due to the similar execution time of the processes. Following this principle, we can avoid using the barrier object saving overhead, but continue to obtain nice accuracy in the final clustering results as the Table 8.21 will depict.

HDBSCAN. The last operation of the entire procedure, to obtain the final clustering result, is the HDBSCAN. To execute the HDBSCAN clustering, we have used the namesake Python library ([CMS13]). It needs as input an MST (our global MST) formatted in a specific way. It returns the resulting clusters in the form of three arrays:

- **Labels**. HDBSCAN assigns each data point to a cluster label. These labels indicate which cluster a data point belongs to. Typically, in HDBSCAN, noise points (outliers) are labeled as -1, and other data points are assigned positive integer labels for the clusters they belong to. You can access these labels as the clustering result.
- **Probabilities**. The used HDBSCAN library (actually an extended version of the standard HDBSCAN called hdbscan_) can provide probability estimates for each cluster assignment. These probabilities reflect the confidence of the data point belonging to a particular cluster.
- **Stabilities.** The library can also provide stability scores, which indicate how robust the cluster assignments are for each data point. High stability values suggest that the assignment is stable and reliable.

Our final cluster() function also returns the condensed tree. The condensed tree is a representation of how clusters are formed and nested within each other. It provides insights into the hierarchical structure of the data, showing how smaller clusters merge into larger ones and how outliers are treated. The tree is typically organized as a dendrogram, where the vertical lines represent clusters, and the horizontal lines connecting them represent the merging of clusters at different levels of hierarchy. Actually, in our implementation, the condensed tree is a numpy array representing the cluster result, but it can be easily plotted with the MatplotLib visualization tool.

Chapter 8

Experimental Results

Now we know how to create a multiprocessing version of the HNSW with shared memory, but also a working and efficient full parallel FISHDBC procedure, with a fast computation of the minimum spanning tree. Let us see if the parallel HNSW graph structure and the final HDBSCAN clustering result are still correct, and if they outperform the original single-process versions.

In this chapter, in fact, we will analyze some experimental results, especially for what concerns the parallel HNSW creation, focusing on the difference in the execution time between the parallel multiprocessing implementation and the single process approach and also paying attention to the accuracy of answering to the search queries inside the structures.

After, also for the MST creation part, we have compared the two execution times. Always concerning the timing results, we have measured the time of the total original single FISHDBC and of my parallel FISHDBC algorithm, to understand if the entire multiprocessing procedure outperforms the single process execution. Additionally, we have measured the difference between the parallel version executed each time with twice the number of processes, starting from only 1 core to 16 cores, to effectively discover how much the multi-cores speed up the execution. With this kind of test we discover that, between the original single-process FISHDBC version and the parallel FISHDBC using a single core, there is a difference in the total time (the original version, in this case, performs well, since using a multi-process approach with only 1 process causes a lot of overhead to manage the shared memory and the multi-cores when actually you do not need them). Indeed, we have done such tests also for the HNSW and the MST alone.

At the end of the analysis, as for just the HNSW, we measured the quality of the final clustering results, again between our parallel FISHDBC and the original single process FISHDBC algorithm.

8.1 HNSW Results

About the HNSW parallel implementation, firstly we will take a look at the experimental results' execution time, to highlight the differences between the parallel multiprocessing and single process HNSW's creation. After, we will take a look at the accuracy results of our parallel HNSW. The parameters passed to the FISHDBC algorithm to create the parallel HNSW are: efConstructions and ef = 32, M = 5, M0 = 10, K for the search = 5 (only for the accuracy tests) and numbers of centers = 5.

Time Results Regarding the timing results associated with a blob data set, we have used as metric the Euclidean dissimilarity, while for the ones related to the text data set we have considered the Levenshtein distance. In the beginning, we registered the different execution times (over blob data sets) of two types of parallel HNSW's creation: with lock and without lock, as described in Table 8.1.

We can notice that the execution time of the parallel approach with lock, respectively for the different sizes of the blob data set, is quite high compared with the lock-deprived parallel HNSW. To be honest, it is not completely clear why. Yes, the lock introduces overhead, but in this case is as if the many processes go into a lot of conflicting situations for updating the HNSW and they stuck each other even more than we expected.

Of course, when we use locks we know that some slowdown will occur, in fact when you use them to synchronize access to a shared data structure, they can lead to lock contention. This means that processes spend time waiting for the lock to be available, which can significantly slow down the execution, especially when multiple processes are involved. Additionally, they may cause CPU and I/O overhead: inter-process communication (IPC) mechanisms, such as locks and shared memory, can introduce additional CPU and I/O overhead, especially when there's contention for the shared resource.

Size	Mean Parall. LOCK	S.D. Parall. LOCK	Mean Parall. NO LOCK	S.D. Parall. NO LOCK
10,000	18.09s	0.26s	1.57s	0.06s
20,000	36.74s	0.69s	2.82s	0.12s
40,000	74.13s	0.62s	5.47s	0.11s
80,000	150.69s	1.07s	10.62s	0.22s
160,000	304.58s	1.68s	27.92s	0.50s

Table 8.1: Differences of execution time (mean over 10 runs) between Parallel HNSW creation with and without lock, using Blob data sets (Remote Linux machine, 16 physical core). S.D. means standard deviation.

This other Table 8.2, instead, points out that when we have the availability of many cores,

like in our remote Linux machine, and if you don't use locks, the parallelism is effective, and it starts to bring a real speed-up starting from the runs associated to small data set's size. In this experiment, the improvement of the parallel HNSW's execution time is very nice, outperforming the HNSW single process version. In fact, the more you increase the size, the more the difference between the two execution times increases in favor of the multiprocessing version, as visible in the speed-up column.

Size	Mean Single	S.D. Single	Mean Parallel	S.D. Parallel	Speed-up
10,000	6.28s	0.06s	1.57s	0.06s	4
20,000	13.30s	0.15s	2.82s	0.12s	4.6
40,000	27.97s	0.24s	5.47s	0.11s	4.9
80,000	59.03s	0.62s	10.62s	0.22s	5.5
160,0000	122.58s	2.22s	21.43s	0.29s	5.7

Table 8.2: Differences of execution time (mean over 10 runs) between Original Single HNSW and Parallel HNSW creation, using Blob data sets (Remote Linux machine, 16 physical core)

The increase behavior is well depicted by the Figure 8.1. The red line's growth scales very well comparing it with the data size's growth, while the linear increase of the blue line (the one associated with the single process version) follows the increase of the data set's items. The speed-up factor brought by the parallel HNSW starts from 4 and increases until almost 6. If we plotted the time behavior also for higher data set's sizes, we would also reach a more and more high speed-up factor.

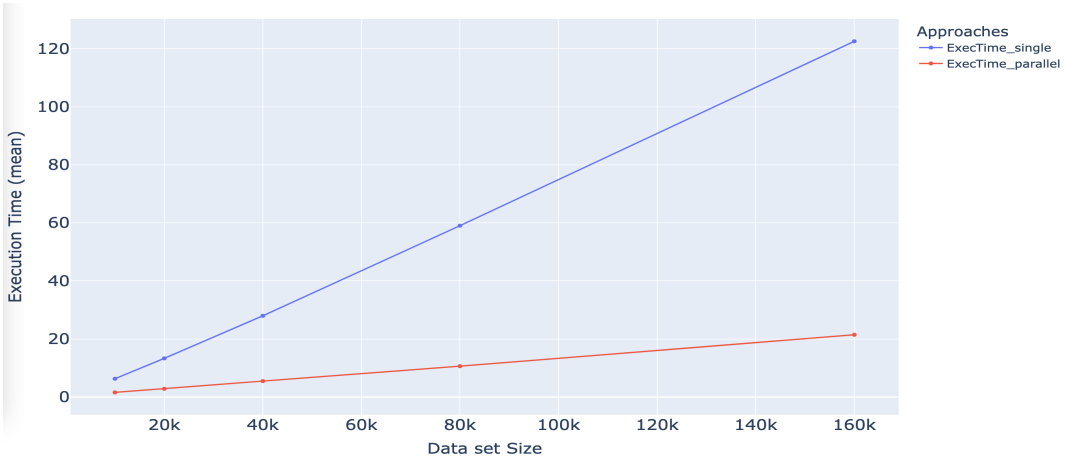

Figure 8.1: Differences of execution time (mean over 10 runs) between Single and Parallel HNSW creation, using blob data sets (Remote Linux machine, 16 physical cores)

Now (in the Table 8.3 and Figure 8.2) we present results about execution of the HNSW construction, but this time doubling the number of used cores starting from 1 process to the parallel version with 16 processes, to really understand what is the speed-up factor when we use many cores. We analyze two scenarios, one with 20,000 items and one with 100,000 to discover if the trends remain the same for different data set sizes.

Size	Mean 1 proc	Mean 2 proc	Mean 4 procs	Mean 8 procs	Mean 16 procs
20,000	66.289s	28.137s	14.530s	8.121s	5.592s
100,000	454.494s	285.75s	151.208s	74.494s	41.05s

Table 8.3: Differences of execution time (mean over 10 runs) between Parallel HNSW creation with 1 process or with 16 processes, using Blob data sets (Remote Linux machine, 16 physical cores)

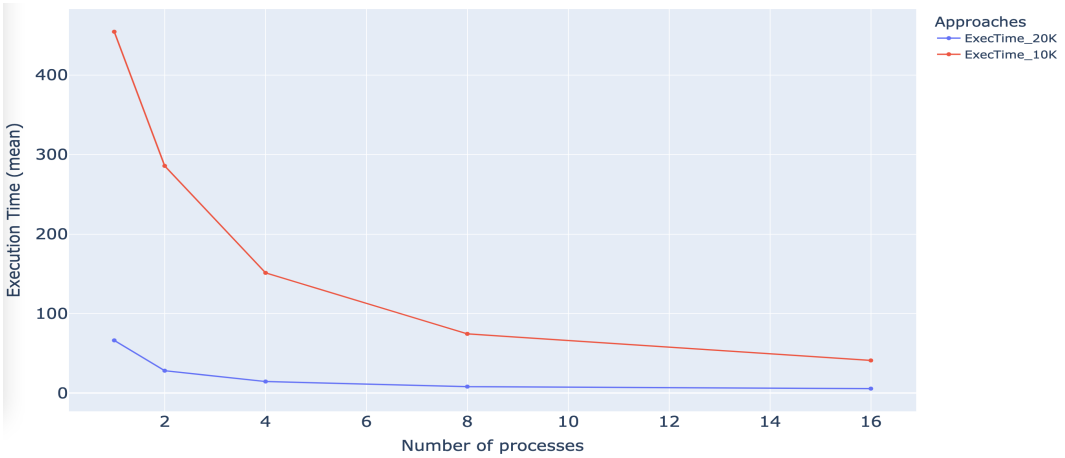

Figure 8.2: Differences of execution time (mean over 10 runs) between Parallel HNSW creation with 1 process or with 16 processes, using blob data sets (Remote Linux machine, 16 physical core)

We have tried also to measure the execution time's differences when the input data set is no more the synth blob data set, but a synth text data set. We recorded results, in this third Table 8.4, for a text data set with different sizes (from 10,000 to 160,000). For this experiment, we used the Levenshtein distance, very adaptable to work with strings data type. The outcome here is that again our multiprocessing HNSW's implementation is faster than the single process one, and the speed-up follows the behavior that we have seen before for the blob data set: the HNSW computational time is decreased. In this test case, the time reduction between my parallel HNSW implementation and the single

process one is less marked than for the blob data set. This happens because, recalling the Snakeviz plot Figure 6.5 in the 6, here we spent also an important amount of time to create the MST, not only for the HNSW add procedure, therefore the total improvement of the HNSW is less obvious.

Size	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.	Speed-up
10,000	6.503s	0.108s	2.98s	0.046s	2.1
20,000	14.345s	0.710s	6.143s	0.117s	2.3
40,000	28.238s	0.375s	12.843s	0.148s	2.2
80,000	60.506s	0.682s	26.772s	0.227s	2.3
160,000	129.952s	1.285s	54.782s	0.611s	2.4

Table 8.4: Differences of execution time (mean over 10 runs) between Original Single HNSW and Parallel HNSW creation, using synth text data set (Remote Linux machine, 16 physical cores)

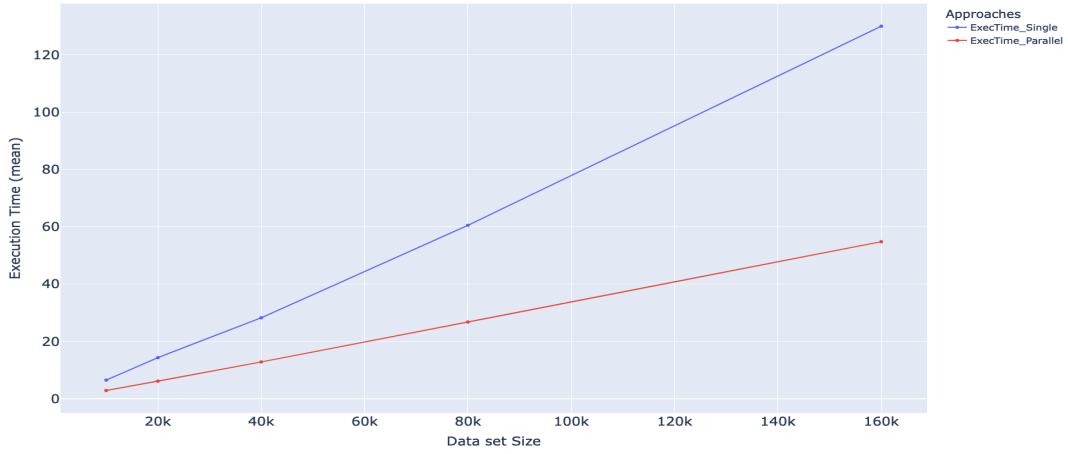

Figure 8.3: Differences of execution time (mean over 10 runs) between Single and Parallel HNSW creation, using synth text data sets (Remote Linux machine, 16 physical cores)

The Table 8.5 depicts timing results, collected after the execution of the state-of-the-art Hnswlib version (executed in single thread mode) and the single process HNSW creation of the original FISHDBC. Again, we have used as input data the numerical data generated by the specific makeBlob() function.

We immediately notice that the library written in C++, one of the fastest programming languages, especially when we talk about parallelism of expensive computation, is not even

remotely comparable to the single process HNSW, since it is a lot faster. So if your need is to perform an approximated KNN search over an HNSW structure (composed of only numerical data) and that's it, you should use the Hnswlib.

Size	Mean Single	S.D. Single	Mean Single Hnswlib	S.D. Single Hnswlib
10,000	6.28s	0.06s	0.973s	0.015s
20,000	13.30s	0.15s	1.502s	0.008s
40,000	27.97s	0.24s	2.578s	0.012s
80,000	59.03s	0.62s	4.871s	0.014s
160,0000	122.58s	2.22s	9.718s	0.043s

Table 8.5: Differences of execution time (mean over 10 runs) between Single and single HNSW creation with the C++ Hnswlib, using Blob data sets (Remote Linux machine, 16 physical core)

The results illustrated in the Table 8.6, instead, tell us that yes, the Hnswlib is also faster than our parallel HNSW implementation, but this is not a surprise, since the Hnswlib is implemented in C++, which provides better performance than Python. However, Hnswlib presents several drawbacks with respect to FISHDBC, that justify the effort in improving it. At first, Hnswlib works only with numerical data, if you want to adapt it to work with string data, you need to represent your text data as numerical vectors, performing an embedding procedure before launching the algorithm. Secondly, even if the Hnswlib is a library that can be used directly in Python, actually it is a Python-wrapped C++ library, thus it obliges us to re-implement, if it is needed, the distance functions in C++. With our parallel HNSW version, we can use an arbitrary distance function without the effort of writing such functions, but simply using the dissimilarities offered by Python's scientific library (like Scipy and Scikit-learn). These reasons could justify the slower final running time of our parallel HNSW implementation with respect to the Hnswlib, also because we can use the resulting HNSW data structure (in the form of two numpy arrays) of our parallel version, but even more importantly, we can track all the computed distances during the creation process. Such collection of distance values will be fundamental to compute the remaining part of the FISHDBC algorithm, which is not possible to do with the library. This last aspect represents the third issue in using Hnswlib with respect to our implementation.

Size	Mean Parall.	S.D. Parall.	Mean Parall. Hnswlib	S.D. Parall. Hnswlib
10,000	1.57s	0.06s	0.518s	0.015s
20,000	2.82s	0.12s	0.565s	0.008s
40,000	5.47s	0.11s	0.647s	0.006s
80,000	10.62s	0.22s	0.807s	0.006s
160,0000	21.43s	0.29s	1.138s	0.008s

Table 8.6: Differences of execution time (mean over 10 runs) between Parallel HNSW and HNSW creation with the C++ parallel Hnswlib, using Blob data sets (Remote Linux machine, 16 physical core)

This additional Table 8.7 refers to the exec times of the HNSW creation (single vs. parallel version) on my local MacBook. It is a MacBook Pro of 2014, with 16 GB of RAM and a 2,2 GHz Quad-Core Intel Core i7 processor, equipped with 4 physical cores (2 hyperthreads per core, 8 logical cores), utilizing the Python's version 3.10.7. I want to show such results because my intention is to point out that when we use a medium-powerful personal PC things could change from a more equipped, in terms of hardware, computer (like the remote machine), since many users could not have the availability of such machine. The data in the table tells us that, even if the parallel HNSW is always faster than the single process implementation, a nice speed-up happens only when we consider a data set's size of 80'000 items. The reason is that on my personal computer, we can use only a few simultaneous processes and we cannot exploit the multiprocessing approach in the best possible way.

Size	Mean Single HNSW	S.D. Single HNSW	Mean Parall. HNSW	S.D. Parall. HNSW	Speed-up
10,000	10.81s	0.37s	9.25s	0.96s	1.1
20,000	23.03s	0.89s	21.40s	3.10s	1.1
40,000	47.75s	0.74s	37.07s	1.64s	1.3
80,000	102.52s	2.75s	77.32s	5.93s	1.3

Table 8.7: Differences of execution time (mean over 10 runs) between Single and Parallel HNSW creation, using Blob data sets (MacBook, 4 physical cores)

Accuracy Results To clarify, all the following HNSW's accuracy results are performed starting from a blob data set, created with different dimensions depending on the specific test case. In addition, as we have mentioned before during the explanation of the process for computing these tests (in the chapter 7), the only distance used to compare accuracy between classical KNN search and our search over the parallel HNSW was the Euclidean distance, so all these quality results are calculated considering such metric. In fact, we cannot use Levenshtein distance and so we cannot take accuracy results for a text data set.

Additionally, I want to point out that for all the charts of section 8.1, the cumulative distribution function (CDF) of the accuracy is displayed. In probability theory and statistics, the cumulative distribution function of a real-valued random variable X, evaluated at x, is the probability that X will take a value less than or equal to x ([Wikc]). For my work, I have used such CDF to show in a useful way the distribution of the error's percentage.

In this first accuracy experiment, shown in the Table 8.8 and Figure 8.4, we have used the lock synchronization system during the modification of the two shared numpy arrays. These are the results of 50 executions over a blob data set of 20,000 items, with 2,000 elements as input for the search, executed over the remote Linux machine, using all the 16 possible physical cores of this machine. The percentage of errors is low, as expected, since in this configuration we are using the lock to modify the shared HNSW structure.

% of Error	Std. Dev	Min	Max
3.13%	41.285%	0.02%	17.28%

Table 8.8: Result's quality of search for 50 runs, with 2000 input items over 20000 elements of Parallel HNSW, WITH lock (remote Linux machine, 16 processes)

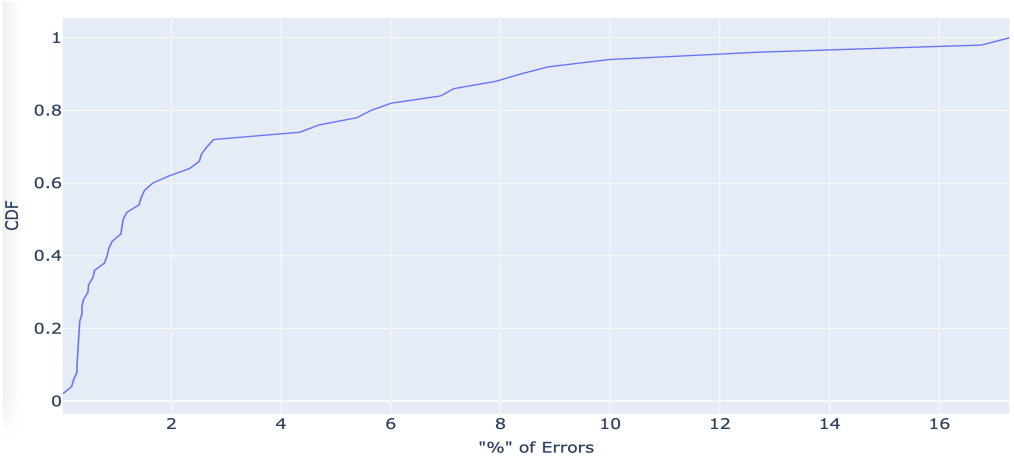

Figure 8.4: CDF of Result's quality of search for 50 runs, 2000 input items over 20000 elements of Parallel HNSW, WITH lock (remote Linux machine, 16 processes)

The Table 8.9 shows the result's accuracy of 50 executions over a blob data set of 20'000 items, with 2000 elements as input for the search, as before, again executed over the remote Linux machine, using all the 16 possible physical cores. This time without using any lock during the updates of the parallel HNSW structure.

The key result here is that the locks are not necessary for our approximated parallel implementation, since the quality of the results remains quite the same, compared with the accuracy of the previous scenario with locks. Before, during the exhibit of the HNSW time results, we noticed that the parallel implementation with lock takes a lot of time to complete with respect to the version without lock, consequently, we have now an additional reason in our favor to implement and use a lock-free parallel version. The Figure 8.5 depicts the variation of the number of errors during all these 50 runs.

% of Error	Std. Dev	Min	Max
3.58%	3.158%	0.46%	10.23%

Table 8.9: Result's quality of search for 50 runs, with 2000 input items over 20000 elements of Parallel HNSW, NO lock (remote Linux machine, 16 processes)

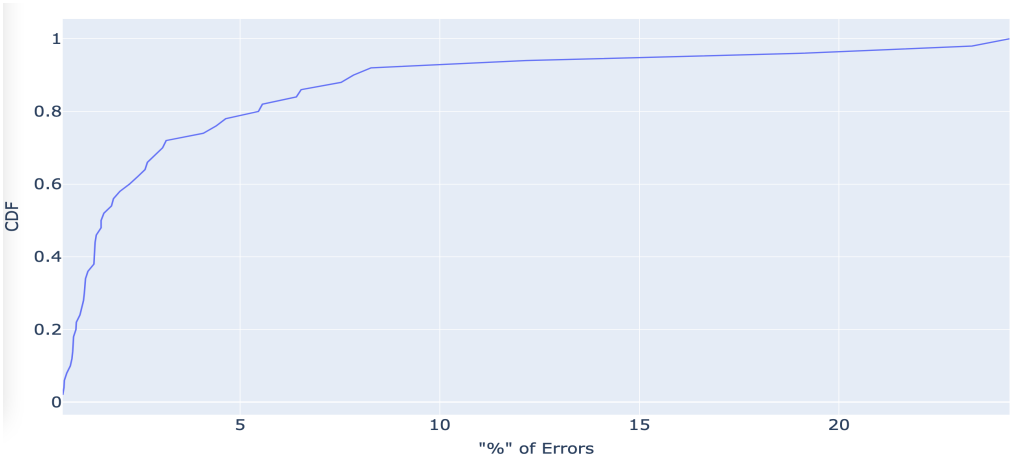

Figure 8.5: CDF of result's quality of search for 50 runs, with 2000 input items over 20000 elements of Parallel HNSW, NO lock (remote Linux machine, 16 processes)

The third experiment, as reported in the Table 8.10 and in the correlated Figure 8.6, explains that increasing the number of search input elements (from 2,000 to 10,000), the percentage of errors of the HNSW creation (without locks) actually decreases during the 50 experimental runs.

% of Error	Std. Dev	Min	Max
2.69%	3.692%	0.09%	17.292%

Table 8.10: Result's quality of search for 50 runs, with 10000 input items over 100000 element of Parallel HNSW, NO lock (remote Linux machine, 16 processes)

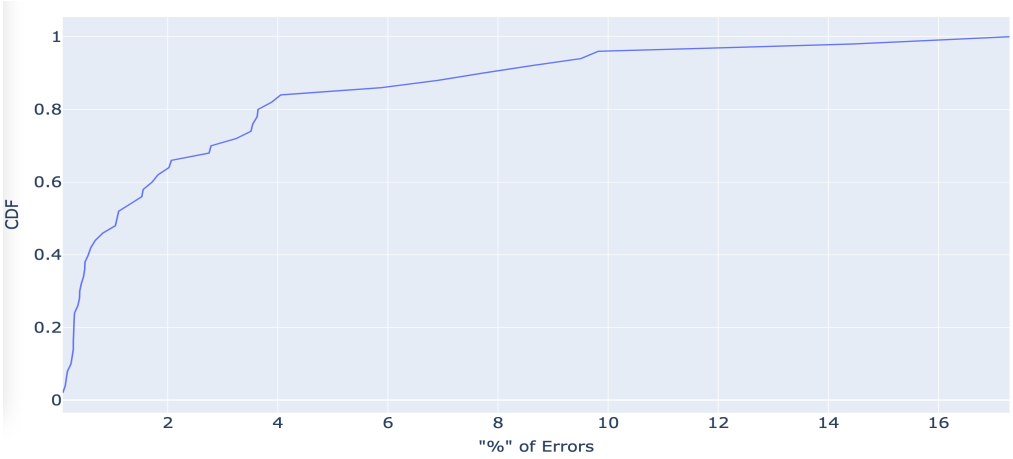

Figure 8.6: CDF of Result's search quality for 50 runs, 10000 input items over 100000 elements of Parallel HNSW, NO lock

This fourth experiment was again performed on the remote machine with the same configuration as for the Table 8.9, but for this case, we use the single process HNSW approach to see if the result's accuracy is better, worse, or similar to the parallel one. As we can see in the Table 8.11 and in the relative Figure 8.7 the variation of the errors during the different runs is very similar to my parallel HNSW implementation, proving that there is a little loss in the parallel HNSW quality result, even if it is implemented in a way in which many processes manipulate at the same time the shared structure.

% of Error	Std. Dev	Min	Max
2.30%	2.869%	0%	28.7%

Table 8.11: Result's quality of search for 50 runs, with 2000 input items over 20000 elements of HNSW, NO lock (remote Linux machine, single process)

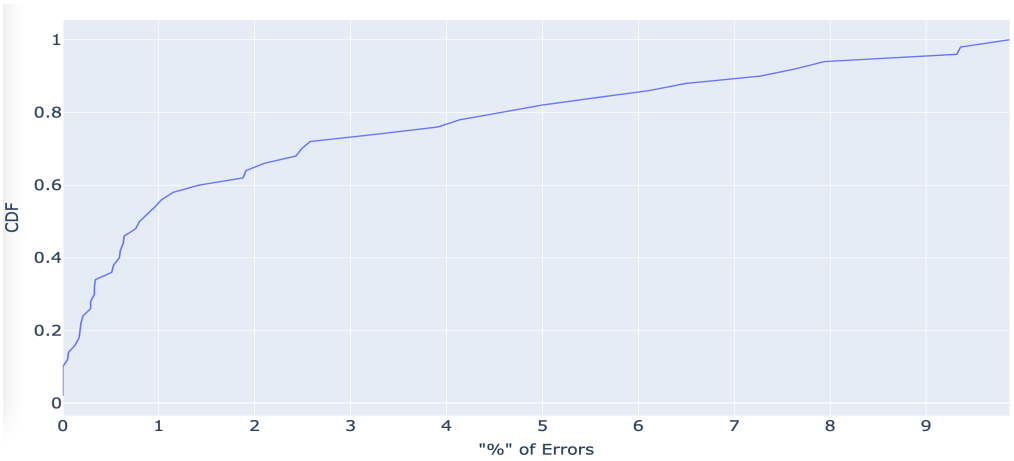

Figure 8.7: CDF of result's quality of search for 50 runs, with 2000 input items over 20000 elements of HNSW, NO lock (remote Linux machine, single process)

Since in the chapter 2 we talked about the state-of-the-art HNSW implementation, the Hnswlib, comparing it with our parallel HNSW version, we should display the Table 8.12 and the Figure 8.8, telling us that yes, the Hnswlib provides an approximated solution to the search too, but it performs better than our parallel HNSW, since the average error is lower.

% of Error	Std. Dev	Min	Max
0.469%	1.45%	0%	6.38%

Table 8.12: Result's quality of search for 50 runs, with 10000 input items over 100000 elements of Parallel Hnswlib(remote Linux machine, 16 processes)

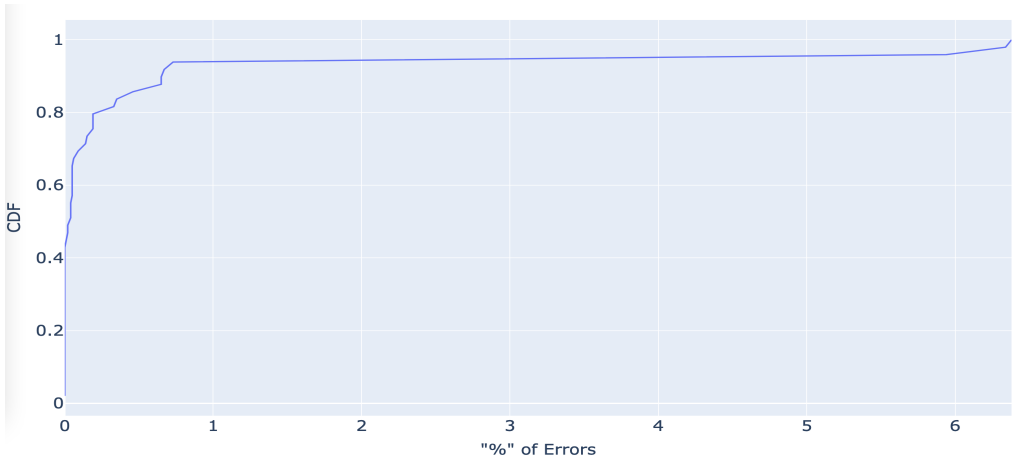

Figure 8.8: CDF of Result's search quality for 50 runs, 10000 input items over 100000 elements of Parallel Hnswlib

8.2 MST Results

In this second experimental results section we will analyze some time results related to another specific part of the entire algorithm, as well as one of the two main hot-spot, the MST creation. I want to point out the fact that the times of the single process MST for the various data set sizes are higher than the scores of the single processes HNSW's results, going against what we said in the hot-spot analysis in chapter 6. The reason is that in these MST-related tables, we kept track of the time spent by the Kruskal algorithm, but also the time spent in the FISHDBC add function to prepare the candidate edges for the MST algorithm. Therefore, if we want to consider only Kruskal's execution time as MST-related timing, it is lower than the HNSW, otherwise, the total MST time is higher. For this portion of the procedure, we will consider only results concerning the running time and not the accuracy, since the quality of the MST is strictly related to the accuracy of the final clustering, for which we have saved the outcomes.

Time Results . Firstly, we recorded outcomes related to executions of the MST over blob data (described in the following Table 8.13 and Figure 8.9). As for the previous hot-spot, the HNSW, also for the current one we obtain a good improvement with the usage of the parallel methodology explained in the chapter 7. The speed-up brought by the parallel MST version is very nice with respect to the original single-core version, and it becomes more significant as the data set size increases, like it is demonstrated by the Figure 8.9. This happens because the workload for the main process (in single process mode) becomes very important when we have to deal with a lot of data and consequently

a lot of candidates for creating the MST.

Size	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.	Speed-up
10,000	6.316s	0.041s	0.880s	0.010s	7.1
20,000	16.658s	0.106s	2.136s	0.042s	7.8
40,000	38.314s	0.438s	5.376s	0.266s	7.1
80,000	93.003s	0.870s	13.281s	0.328s	7
160,000	215.971s	1.518s	33.848s	1.579s	6.3

Table 8.13: Differences of execution time (mean over 10 runs) between Single and Parallel MST creation, using Blob data sets (remote machine, 16 physical core)

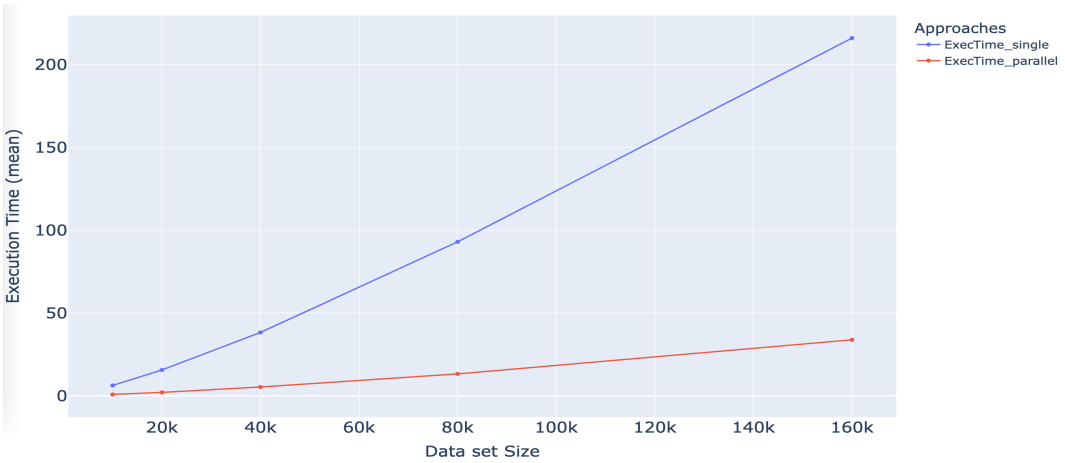

Figure 8.9: Differences of execution time (mean over 10 runs) between Original Single MST and Parallel MST creation, using Blob data sets (remote machine, 16 physical core)

In this second MST-related table (Table 8.14) we can observe that, as expected, the running time of the MST execution when only 1 process is used in the parallel implementation, is very high. It is even a little higher than the original single-process solution. This, as introduced before, is due to the useless amount of resources to be shared, useless because everything is running in a single core. The overhead becomes very significant and it slows down the entire procedure, without obtaining, of course, any kind of parallelism. Additionally, we can understand what is the scalability of the parallel MST construction and what are the speed-up factors when we double the number of used cores for the parallel execution (see the Figure 8.10). Looking at the outcomes, we can assert that for the creation of the MST there is a speed-up factor of almost 7, meaning that from 1 process to 16 processes the time is reduced by 7. For example, for 20,000 items we pass from 17

seconds (1 process) to a little more than 2 seconds. This happens because doubling the number of cores doesn't correspond to an ideal halving of the execution time. The main problem causing this behavior is the overhead introduced for managing the multi-processes and also the shared memory objects, even more so because in our scenario these shared objects could be very big. However, the only creation of the MST scales quite well with respect to the increase of the cores' number, even if not perfectly.

Size	Mean 1 proc	Mean 2 procs	Mean 4 procs	Mean 8 procs	Mean 16 procs
20,000	17.328s	9.156s	5.080s	2.991s	2.101s
100,000	109.758s	68.322s	39.911s	25.053s	17.481s

Table 8.14: Differences of execution time (mean over 10 runs) between Parallel MST creation with 1 process or with 16 processes, using Blob data sets (Remote Linux machine, 16 physical cores)

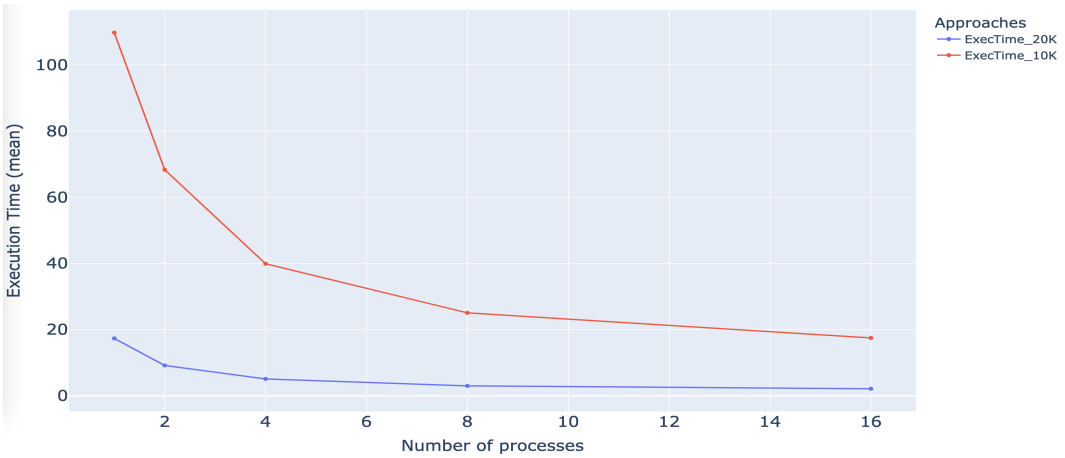

Figure 8.10: Differences of execution time (mean over 10 runs) between Parallel HNSW creation with 1 process or with 16 processes, using blob data sets (Remote Linux machine, 16 physical core)

Also looking at the time results in the Table 8.15 and the Figure 8.11, regarding the MST creation taking as input text data, it is possible to ascertain that the speed-up brought by the parallel version is very similar to the previous one associated with the scenario with blob data. The only difference is that the overall time to complete the MST computation (both parallel and single process) is higher than with blob data, since dealing with string and executing the Levenshtein distance function could be more costly.

Size	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.	Speed-up
10,000	9.707s	0.107s	1.001s	0.024s	9.7
20,000	27.956s	0.239s	2.942s	0.079s	9.5
40,000	63.832s	0.619s	6.203s	0.125s	10.3
80,000	158.977s	1.436s	15.090s	0.309s	10.5
160,000	365.48s	4.398s	37.976s	0.757s	9.6

Table 8.15: Differences of execution time (mean over 10 runs) between Original Single MST and Parallel MST creation, using text data sets (remote machine, 16 physical core)

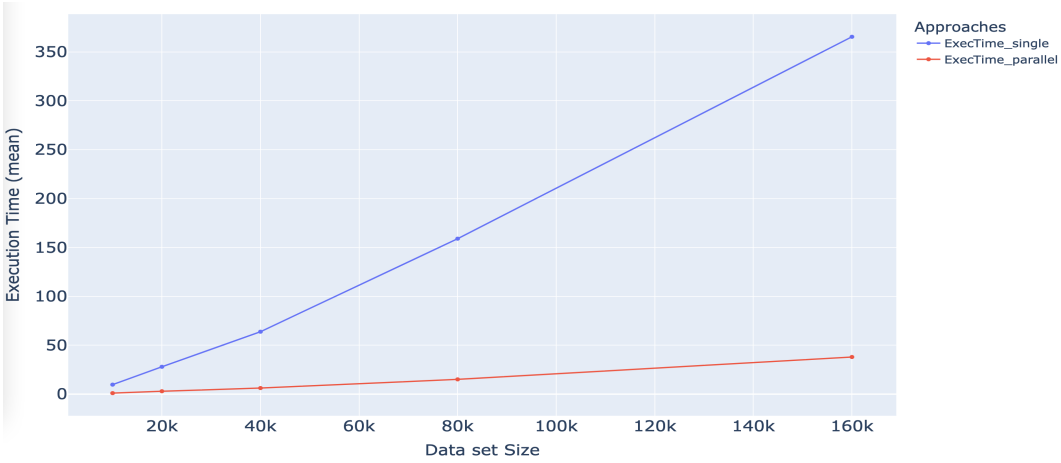

Figure 8.11: Differences of execution time (mean over 10 runs) between Original Single MST and Parallel MST creation, using text data sets (remote machine, 16 physical core)

This last MST-related table (Table 8.16) is the one associated with the timing result of the MST creation, using my local computer (MacBook Pro with 4 physical cores). Again, we want to highlight how is the execution over a local machine equipped with fewer cores than a powerful computer. As we can see, the speed-up is still present with a factor of nearly 2.6.

Size	Mean Single MST	S.D. Single MST	Mean Parall. MST	S.D. Parall. MST	Speed-up
10,000	10.554s	0.790s	3.642s	0.326s	2.8
20,000	24.560s	1.485s	9.434s	0.787s	2.6
40,000	53.853s	0.774s	20.174s	0.660s	2.6
80,000	130.682s	2.638s	49.330s	1.398s	2.6

Table 8.16: Differences of execution time (mean over 10 runs) between Single and Parallel MST creation, using Blob data sets (MacBook, 4 physical cores)

8.3 Discussion

Now we will compare also the final running time of the entire FISHDBC procedure, differentiating again between the parallel multiprocessing and single process implementation of the algorithm, to understand if the total parallel FISHDBC version can really be a faster implementation, maintaining a nice level of accuracy.

Time Results . For what concerns the speed-up obtained by the parallel version of the entire FISHDBC procedure, the outcome is very similar to the one about the MST and HNSW. That is an important improvement in the final running time thanks to the concurrent execution of processes. As always, the related Table 8.17 and Figure 8.12 prove our claim.

Size	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.	Speed-up
10,000	12.799s	0.130s	3.323s	0.075s	3.8
20,000	29.001s	0.244s	6.894s	0.295s	4.2
40,000	66.417s	0.675s	14.804s	0.877s	4.4
80,000	151.879s	1.722s	33.659s	1.700s	4.5
160,000	339.532s	2.888s	76.783s	3.273s	4.4

Table 8.17: Differences of execution time (mean over 10 runs) between Single and Parallel FISHDBC algorithm, using Blob data sets (remote machine, 16 physical cores)

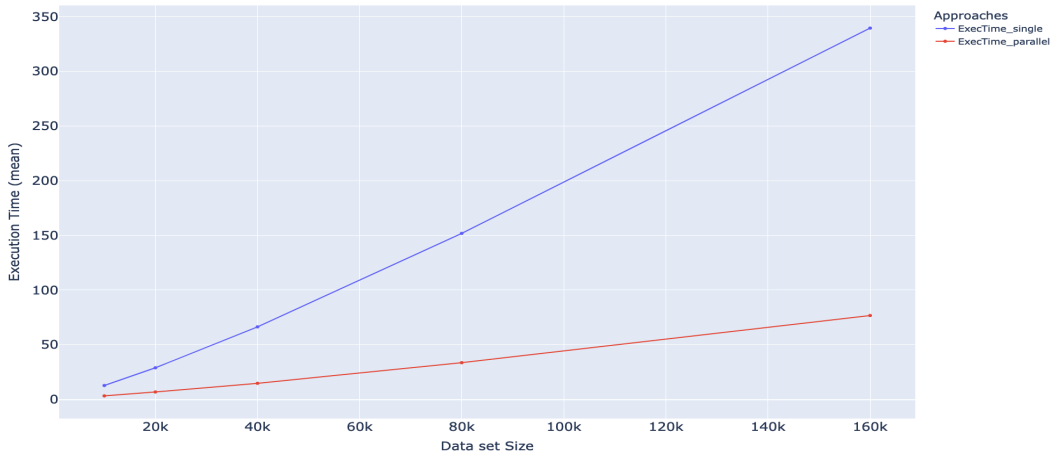

Figure 8.12: Differences of execution time (mean over 10 runs) between Single and Parallel FISHDBC algorithm, using Blob data sets (remote machine, 16 physical cores)

Also for this Table 8.18 below we can see the same behavior as for the previous Table 8.14. The execution time of the parallel FISHDBC with only one core available is significantly higher than the parallel version exploiting all the physical cores (16). Not only that, it is also remarkably higher than the original single-process FISHDBC algorithm. Such a trend is well delineated by the associated Figure 8.13, in which we can also compare the growth of the algorithm's running time when we double the cores until 16 are used. Also for this FISHDBC result the speed-up factor is again almost 10 (and not 16 as someone could expect) when the number of cores is doubled, for the same reasons analyzed before for the Table 8.14 (overhead). We can conclude that the overall FISHDBC scales well, but not perfectly, with respect to the cores' number, even if allows reaching a significant reduction of the final time.

Size	Mean 1 proc	Mean 2 procs	Mean 4 procs	Mean 8 procs	Mean 16 procs
20,000	55.43s	28.037s	14.835s	8.230s	5.300s
100,000	302.671s	166.978s	89.808s	52.133s	33.122s

Table 8.18: Differences of execution time (mean over 10 runs) between Parallel FISHDBC algorithm using different number of processes, blob data sets (remote machine, 16 physical cores)

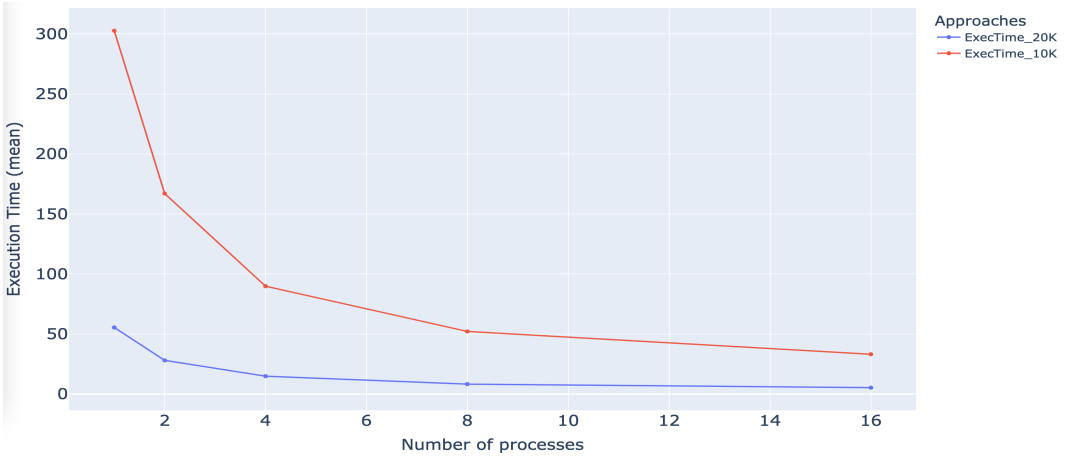

Figure 8.13: Differences of execution time (mean over 10 runs) between Parallel FISHDBC algorithm using different number of processes, blob data sets (remote machine, 16 physical cores)

When the FISHDBC algorithm deals with text data sets, the findings remain the same: an important improvement brought by the multi-process version with respect to the original single-process procedure. In fact, as pointed out by the Table 8.19 and the Figure 8.14,

the growth of the red line (parallel MST) scales very well, especially compared with the blue one (single-process MST), as the data set's size increases.

Size	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.	Speed-up
10,000	15.661s	0.271s	3.004s	0.062s	5.2
20,000	41.757s	0.455s	7.134s	0.079s	5.8
40,000	92.022s	1.498s	14.448s	0.235s	6.3
80,000	218.781s	3.196s	31.842s	0.456s	6.8
160,000	493.937s	8.572s	72.534s	1.214s	6.8

Table 8.19: Differences of execution time (mean over 10 runs) between Single and Parallel FISHDBC algorithm, using text data sets (remote machine, 16 physical cores)

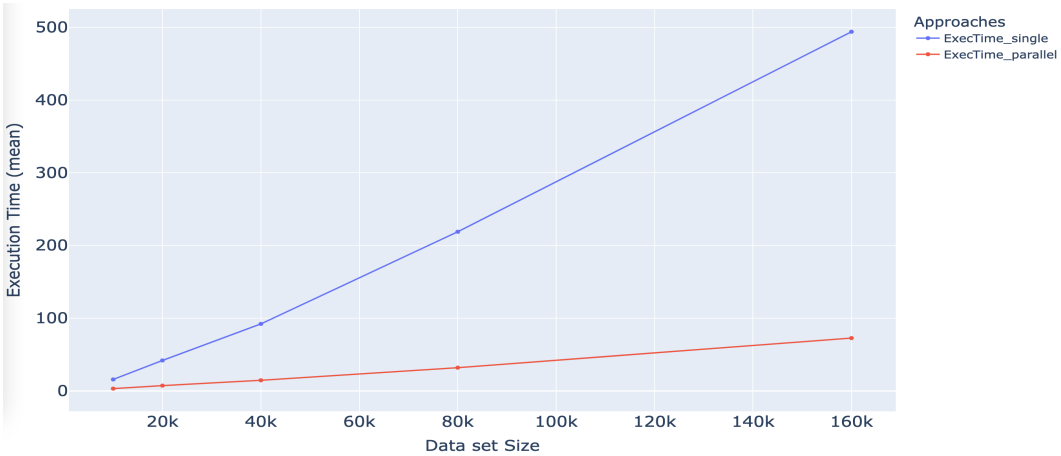

Figure 8.14: Differences of execution time (mean over 10 runs) between Single and Parallel FISHDBC algorithm, using text data sets (remote machine, 16 physical cores)

As for the previous experimental results section, also for what concerns the overall FISHDBC procedure, we have collected timing outcomes associated with some runs over the MacBook. The findings, illustrated by the Table 8.20, are the same as before, namely that the improvement in the final execution time is real also using 4 cores, but of course, is less effective than when are available many cores. Anyway, nearly halving the total time is already a good result.

Size	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.	Speed-up
10,000	21.863s	1.292	13.099s	1.396s	1.6
20,000	47.543s	1.967s	29.638s	2.210s	1.6
40,000	100.472s	1.122s	57.831s	2.293s	1.7
80,000	227.942s	3.922s	126.847s	1.827s	1.8

Table 8.20: Differences of execution time (mean over 10 runs) between Single and Parallel FISHDBC, using Blob data sets (MacBook, 4 physical core)

Accuracy Results To measure the accuracy of our final clustering results, we have used some common evaluation metrics tailored for such kind of problem, as suggested by [Lea]. We compare the resulting labels of our parallel FISHDBC with the original labels generated during the creation of the blob data set and the synthetic text data set. These original labels are used as ground truth for the comparison. It is worth mentioning that when we consider accuracy results starting with blob data set, there are cases in which the generation of such a data set (since it is random) could lead to a distribution of the points for which is almost impossible to perform a clustering operation (when the data of the different classes are almost completely overlapping). The results of such kind of cases, therefore, are not taken into account to compute the accuracy average. The metrics used for the confrontation between clustering results are:

- **The Rand Index (RI)** is a measure used for evaluating the accuracy of clustering in data analysis and machine learning. It quantifies the similarity between two data clusters, such as a clustering produced by a clustering algorithm and a ground truth or reference clustering. The Rand Index calculates the percentage of agreements between pairs of data points based on whether they are in the same cluster (either in both clusters or in neither) or in different clusters (in one clustering but not the other). It returns a value between 0 and 1, where: a Rand Index of 1 indicates a perfect agreement between the two clusters; a Rand Index of 0 means there is no agreement between the two clusters beyond what would be expected by random chance.
- **The Adjusted Rand Index (ARI)** is a variation of the Rand Index (RI) that adjusts for chance or randomness in clustering evaluations. It measures the similarity between two data clusters while taking into account the expected similarity due to random chance. ARI is often used in clustering evaluation, particularly when assessing the quality of clustering results compared to ground truth or reference clustering. The ARI value ranges from -1 to 1, where: ARI = 1 indicates a perfect agreement between the two clusters, beyond what would be expected by random

chance; ARI almost = 0 suggests that the clustering is no better than random chance, while ARI ¡ 0 indicates that the clustering results are worse than random chance, suggesting a significant disagreement between the two clusters.

- **The Adjusted Mutual Information (AMI)** score is another metric used for evaluating the quality of clustering results by measuring the agreement between a clustering and again a reference (ground truth) clustering. The Adjusted Mutual Information is particularly useful when dealing with unsupervised clustering tasks and can be a more informative metric than some other measures like the Rand Index (RI) or Normalized Mutual Information (NMI). AMI calculates the mutual information between two clusters while adjusting for chance. It considers the similarity of cluster assignments and penalizes random clustering agreements that might occur by chance. The AMI score ranges from 0 (no mutual information) to 1 (perfect agreement), and it can also take on negative values if the agreement is worse than random.

- **The Normalized Mutual Information (NMI)** score is a normalized version of the Mutual Information (MI) and provides a measure of how much information two clusters share while considering the size of the clusters and the total number of data points. NMI is a commonly used metric in clustering evaluation because it quantifies the quality of the clustering results, taking into account the degree of agreement beyond what would be expected by random chance. The NMI score ranges from 0 (no mutual information) to 1 (perfect agreement), and it can also take on negative values if the agreement is worse than random, as for the AMI.

- **Homogeneity**, **completeness**, and **V-measure** are three commonly used metrics in clustering evaluation. They provide insights into different aspects of clustering quality:

  - *Homogeneity* measures the extent to which all data points within a given cluster belong to the same class or category. It quantifies the purity of clusters with respect to their true class labels in a ground truth or reference clustering. A higher homogeneity score indicates that clusters contain data points that all belong to the same class.
  - *Completeness* measures the extent to which all data points that belong to the same class are assigned to the same cluster. It quantifies the ability of the clustering to capture all members of the same class. A higher completeness score indicates that all data points of a class are assigned to a single cluster.
  - *V-measure* is a combination of homogeneity and completeness, providing a single score that measures the balance between these two aspects. It is the harmonic mean of homogeneity and completeness and is useful for situations where both aspects of clustering quality are important. A higher V-measure score indicates

a good balance between clustering purity and capturing all members of the same class.

These two following tables (Table 8.21 and Table 8.22) highlight the nice accuracy of the final clustering results, both for textual and blob data sets. We notice that for the text data, the final scores are slightly lower than the ones referred to blob data, but the difference is so low that it may be disregarded, it could depend on some unlucky runs of the algorithm that have lowered a little bit the final average. In addition, the results' quality still remains almost the same for the experiments with 20,000 items and 100,000 items, proving the robustness of the algorithm as the size of input data increases. Looking carefully at the outcomes we find that the highest evaluation metrics are the Rand Index and the Homogeneity, confirming the good quality of individual clusters in terms of class or category consistency, while the worst score is always the one associated with the completeness measure, meaning that the algorithm less effectively captures or groups all data points that belong to the same class or category, but fortunately in a minimal way, since the completeness score still remains nice.

20,000 Items							
Dataset	AMI	NMI	ARI	RI	Homog.	Compl.	V-Measure
Blob	96%	96%	96%	99%	99%	93%	96%
Text	94%	94%	94%	98%	99%	89%	94%

Table 8.21: Accuracy of clustering (means over 50 runs), both text and blob data sets with 20,000 items (remote machine, 16 physical core)

100,000 Items							
Dataset	AMI	NMI	ARI	RI	Homog.	Compl.	V-Measure
Blob	97%	97%	96%	99%	100%	94%	97%
Text	93%	93%	93%	98%	100%	88%	93%

Table 8.22: Accuracy of clustering (means over 50 runs), both text and blob data sets with 100,000 items (remote machine, 16 physical core)

Additional Experimental Result In this other additional experiment, we want to confirm our previous findings (about time) when we deal with a real numerical data set, the California housing data set. The Table 8.23 depicts the timing results of all the three analyzed parts, the HNSW, the MST and the total FISHDBC for what concerns executions

of the algorithm with this real data set. The California housing is a popular data set in machine learning and statistics used for various purposes, such as regression analysis, predictive modeling, and data exploration, but also clustering. It contains information about housing in California and it is composed of 20,640 items.

Part of the alg	Mean Single	S.D. Single	Mean Parall.	S.D. Parall.
HNSW	17.08s	0.355s	3.631s	0.082s
MST	18.894s	0.799s	3.355s	0.029s
FISHDBC	33.185s	1.210s	6.978s	0.141s

Table 8.23: Differences of execution time (mean over 10 runs) between Single and Parallel (HNSW, MST, overall FISHDBC), real numerical data set (remote machine, 16 cores)

Instead, in this other Table 8.24 we describe some accuracy results of the HNSW creation when we deal with the same real numerical data set as before, proving the fact that with our parallel implementation of the HNSW, we can reach a correct creation of such structure, even when we deal with real data (we did not perform final clustering accuracy test because we did not have the original labels available for the confrontation).

% of Error	Std. Dev	Min	Max
0.85%	21.42%	0.39%	1.61%

Table 8.24: Result's quality of search for 50 runs, with 1640 input items over 19,000 elements of Parallel HNSW, real numerical data set(remote Linux machine, 16 processes)

Chapter 9

Conclusion & Future Works

Among all the possibilities to improve the performance of an algorithm in Python (described in the chapter 5) we have chosen the parallel implementation, due to some specific needs and constraints of our case. I have reached this parallel solution with a multi-process approach to execute all the work. Why multi-processes and not multi-threads? Simply because in Python (the language in which both the original and my parallel implementations are written) it is not efficient the usage of a multi-threading approach, due to some specific features of the language, in particular, the GIL or Global Interpreter Lock (a detailed analysis of such aspect is described in chapter 5).

But writing parallel code was not easy, even if at the end of the work we obtained the results we wanted. For the two main bottlenecks (which emerged after the profiling exposed in the chapter 6), we were able to significantly reduce the execution time of a factor of almost 5 times for the blob data set and 2.2 times for the text data set, speaking about the HNSW, while a factor of 7 times for blob data set and almost 10 times for text data set, for what concerns the MST. Consequently, we have reached a speed-up for the overall FISHDBC procedure, succeeding in decreasing the total running time by a factor of 4.2 folds for the blob data and 6.1 times for text data. Collecting the time results we understood also how the parallel FISHDBC scales, based on the number of cores/processes used during its execution: doing the experiment from 1 single process until 16, it turned out that we can reach a speed-up factor of 10 times, meaning that there is not a perfect decrease of time as the increase of the core, but it still good (we have to take into account that when we deal with shared memory and multi-processes an overhead is normally present). Another important aspect is the accuracy of the output, both for the HNSW and the final HDBSCAN clustering. In fact, by measuring the quality of the HNSW structure created with our parallel implementation without locks, we can still maintain a good accuracy of the search results. If we compare such results with a state-of-art search method (like the K-NN) we obtain a percentage of error of almost 3%. Also regarding the quality of the final

clustering, we can assert that the results are satisfying: for almost all the used clustering evaluation metrics, we have achieved a similarity of more than 95% between our resulting labels and the labels associated with the original data. A good feature of all such outcomes is that they remain very similar (both time and accuracy results) whether we consider blob data set (numerical data) or text data set (textual data).

Drawbacks & Future Works As usual for the implementation of a new approach, also our work has some drawbacks.

- Firstly, we have to say that, despite the good accuracy and considering that we are talking about a concurrent and approximated algorithm, there is a loss in the result's quality, both for the HNSW and the final HDBSCAN clustering. If someone needs a precision of 100% for its use case, this implementation could be not tailored for him/her.
- Additionally we have ascertained that there is not a perfect speed-up when we use the total number of possible physical cores (16), both comparing it with the original single process FISHDBC's time and the time associated with the parallel version executed with only 1 process. In this last scenario in fact one might expect to have a 16 times faster version, but this does not happen since the overhead introduced by the multi-processes and shared memory usage is quite remarkable.
- As presented in this document, for creating the Minimum Spanning Tree, the Kruskal algorithm was implemented. Kruskal is a widely used MST algorithm with a quite nice complexity (O(m log n)), but ideally, faster methods exist. For example, in a comparison model, in which the only allowed operations on edge weights are pairwise comparisons, Karger, Klein and Tarjan ([Wike]) found a linear time randomized algorithm based on a combination of Boruvka's algorithm (see the [Wikb]) and the reverse-delete algorithm (for more info read the [Wikg]). Another example is the fastest non-randomized comparison-based algorithm with known complexity, by Bernard Chazelle, that is based on the soft heap, an approximate priority queue. Its running time is O(m α(m,n)), where (α) is the classical functional inverse of the Ackermann function ([Wika]). The function α grows extremely slowly, so that for all practical purposes it may be considered a constant no greater than 4, thus Chazelle's algorithm takes very similarly to linear time. Trying to use one of these two fast MST algorithms in our implementation could be a future improvement.
- A possible future improvement could be a revision of some parts of the code in order to implement them using cython ([Cyt]), or even directly in C language, for example. After, we can wrap this C code to work with the Python environment. For instance, an idea could be to write a C code for the creation of the HNSW, this time also with lock, since in C they work well.

Implications. During the implementation of the thesis, we have discovered more. Working on the parallel version of the HNSW, we studied how the main state-of-the-art approach, the Hnswlib, implements the concurrent creation of such data structure, especially to understand the synchronization mechanism. In fact, the library uses locks to prevent concurrent access to the graph structure when a node should be inserted and connected to others. Trying to recreate this mechanism we ended up with a surprising discovery: the locks synchronization system could be not implemented, since the HNSW is used for an approximated nearest neighbor search and consequently we could create an approximated HNSW structure too. While the avoidance of synchronization system seems blasphemy, actually it can be done, because looking at the accuracy results for the HNSW, we have understood that such quality remains the same as if we have used the lock, whereas the execution time to create the data structure increases a lot when the lock system is in action (recalling the Table 8.1).

Another interesting breakthrough, always related to approximation and to the non-use of synchronization, concerns the barrier sync object. In fact, when each process starts to compute its own local MST it should wait for all the other processes to go ahead, because, otherwise, all the candidate edges' list of each MST should be recomputed. But also for this case, we can avoid using such barrier, because we have discovered that each process reaches the end of the HNSW creation, and starts to perform the MST, almost at the same time, thus the sync system could be not required, allowing the total execution time to decrease.

After these two findings, we can assert that, in general, could be a good idea to implement a concurrent approximated algorithm without synchronization saving a lot of time and resources. It could be a possible approach also because, not wanting 100% accurate results, the quality of the outcomes could remain excellent, very similar to a non-approximated implementation. The suggested approach in such a situation is to start to implement your work with all the possible kinds of sync objects and systems. After, it is plausible to try to remove some of these mechanisms and see if the time improves and if the accuracy remains almost the same. If this happens we can try to remove all the synchronization logic and again observe the result, if they are satisfying or not. This was the procedure that we adopted for our work, where, at first, we executed everything with a complete lock synchronization system, tailored for the HNSW structure modifications, but also a barrier before the MST computation. After, we tried to remove both the locks and the barrier, to find out that the accuracy of the HNSW and the final clustering remained very great, while the running time decreased a lot.

To conclude, we can say that with our thesis we have offered a usable way to perform a good parallel and quite fast clustering algorithm, completely written in Python. Our work also allows a user to pass as input both numerical and text data sets, but not only, it also allows the use of many already defined distance functions, some adapted to work

with numerical data, and one adapted for dealing with textual data. Additionally, if a user wants to use a customized function, he/she should only insert a line of code with a call to an existing distance function thanks to Python libraries. So the easy usage, the flexibility, the only Python dependency, the quite fast execution time, but also the original good properties of the FISHDBC, are the features offered by our parallel implementation. Furthermore, our thesis explains a possible way to write complex concurrent Python code to speed-up a program in an approximated way, therefore it could be used as a reference for such situations. We believe that, in a situation in which there is the need to perform a fast clustering operation in Python, our implementation could be useful, but also when there is the necessity to obtain an HNSW structure in a simple and quick way.

Appendix A

Code Scripts

Listing A.1: Parallel pi computation

```
import time
from multiprocessing import Value
from multiprocessing import Process, cpu_count

PI25DT = 3.14159265358979323846264 3
INTERVALS = 100000000

def compute_pi(intervals, idx, num_proc, total_pi):
    summation = 0.0
    dx = 1.0 / float(intervals)
    for i in range(idx+1, intervals, num_proc):
        x = dx * (float(i - 0.5))
        f = 4.0 / (1.0 + x*x)
        summation = summation + f
    partial_pi = dx*summation
    with total_pi.get_lock():
        total_pi.value = total_pi.value + partial_pi

if __name__ == '__main__':
    intervals = INTERVALS
    val = Value('f', 0.0)

    processes = []
    start_time = time.time()
```

```
    num_proc = cpu_count()
    for i in range(num_proc):
        process = Process(
        target=compute_pi,
        args=(intervals, i, num_proc, val)
        )
        processes.append(process)
        process.start()

    for process in processes:
        process.join()
    pi = val.value
    print("Computed PI", pi)
    print("The true PI:", PI25DT)

    end = time.time()
    print("The time is :", (end-start_time))
```

Listing A.2: Parallel bubble sort with ctype Array

```
import multiprocessing
import numpy as np
import time
def bubble_sort_shared(arr):
    n = len(arr)
    for i in range(n):
      for j in range(len(arr)-1):
        if arr[j] > arr[j+1]:
          arr[j], arr[j+1] = arr[j+1], arr[j]

def main():
    array = np.random.randint(0,1000,
                              size=1000,
                              dtype=np.int32
                              )
    sh_arr = multiprocessing.Array('i', array)
    lock = multiprocessing.Lock()
    num_processes = multiprocessing.cpu_count()
    chks_len = len(array) // num_processes
    chunks = [sh_arr[i:i+chks_len]
```

```
    for i in range(0,len(array),chks_len)]

    pool = multiprocessing.Pool(processes=num_processes)
    pool.map(bubble_sort_shared, chunks)
    with lock:
        for i in range(len(sh_arr)):
          for j in range(len(sh_arr)-1):
            if sh_arr[j] > sh_arr[j+1]:
              sh_arr[j], sh_arr[j+1] = sh_arr[j+1], sh_arr[j]

    sorted_array = list(sh_arr)

if __name__ == '__main__':
    start_time = time.time()
    main()
    end = time.time()
    print("The time is :", (end-start_time))
```

Listing A.3: Parallel bubble sort with shared numpy array

```
import multiprocessing
from multiprocessing import shared_memory
import multiprocessing as mp
import numpy as np
import time
def bubble_sort(name, dtype, shape, idx, num_proc):
    shm = shared_memory.SharedMemory(name=name)
    sh_arr = np.ndarray(shape=shape, dtype=dtype, buffer=shm.buf)
    n = len(sh_arr)
    for i in range(n):
      for j in range(0, n-i-1):
        if sh_arr[j] > sh_arr[j+1]:
          sh_arr[j], sh_arr[j+1] = sh_arr[j+1], sh_arr[j]

def main():
    array = np.random.randint(0, 1000, size=10000, dtype=np.int32)
    shm = shared_memory.SharedMemory(create=True, size=array.nbytes)
    shared_array = np.ndarray(array.shape,
                      dtype=array.dtype,
                      buffer=shm.buf
```

```
                    )
    shared_array[:] = array[:]
    num_processes = multiprocessing.cpu_count()
    processes = []

    for i in range(num_processes):
        process = multiprocessing.Process(
                    target=bubble_sort,
                    args=(
                        shm.name, array.dtype,
                        array.shape, i, num_processes)
                    )
        processes.append(process)
        process.start()

    for process in processes:
        process.join()
    shm.close()
    shm.unlink()

if __name__ == '__main__':
    start_time = time.time()
    main()
    end = time.time()
    print("The time is :", (end-start_time))
```

Appendix B

Installation & Execution

In this appendix section, we will see some basic instructions to install everything you need to execute the FISHDBC algorithm on a Linux machine (you can find the complete code at `https://github.com/edo-pasto/Parallel-Flexible-Clustering.git`) First of all, you have to download or clone the GitHub repository:

```
git clone "url of the repo"
```

Secondly, I really suggest to create a virtual environment, or with conda or with pip as package manager (in the following instructions i will use conda, but with pip the process is very similar and you can check [Gui]):

```
conda create --name myenv python=3.10.12
```

After the creation of the environment (the suggestion is to create the environment with python version 3.10.12 or 3.11.3) we can activate it with:

```
conda activate myenv
```

Now that the preliminary operations are done we can start to install all the dependencies needed by the algorithm to work (always inside the just-created environment). These first packages can be installed directly with one single conda install command:

```
conda install numpy scipy pandas scikit-learn numba matplotlib
```

The package allowing us to use the Leveshtein distance should be installed from a different conda source:

```
conda install -c conda-forge levenshtein
```

Also Cython is installed with a different command:

```
conda install -c anaconda cython
```

The last package to be installed is the one associated with the HDBSCAN, this time installed with pip (is not possible with conda):

```
pip install hdbscan
```

The last installation step is to execute the setup.py file in the following way:

```
python3 setup.py install
```

If you are using a pip env you can simply install everything with a single command:

```
pip install numpy scipy pandas scikit-learn
numba matplotlib cython hdbscan levenshtein
```

Now we will see how to execute experiments of the algorithm launching the fishdbc_example.py file. If you want to execute a classical run of the parallel FISHDBC you should type:

```
python3 parallel_fishdbc_example.py --dataset blob --nitems 10000
--centers 5 --distance euclidean --parallel 16
```

where you can specify the data set that you want to use as input (blob or text), the number of items of the input data set (–nitems), the numbers of the data set's centroids (–centers), depending on the type of data the distance to be used (the available distances are: euclidean, sqeuclidean, Minkowski, cosine and Levenshtein for text data) and the number of processes to use (–parallel 0 means original single process FISHDBC, –parallel ¿ 0 means multi process, you can pass from 1 to 16 processes). If you want to execute the parallel FISHDBC with text data as input you can write:

```
python3 parallel_fishdbc_example.py --dataset text
--distance levenshtein --nitems 1000 --centers 10 --parallel 16
```

If, instead, you desire to execute the original single process FISHDBC you can also write:

```
python3 parallel_fishdbc_example.py --dataset blob
--nitems 1000 --centers 5 --distance euclidean
```

It is possible to take a look at the accuracy of the final parallel (but also the single process) FISHDBC clustering enabling the –test option:

```
python3 parallel_fishdbc_example.py --dataset blob --nitems 10000
--centers 5 --distance euclidean --parallel 16 --test True
```

There is also the opportunity to execute an example of only the parallel HNSW creation (again with the possibility to perform some accuracy tests thanks to the –test True option):

```
python3 parallel_hnsw_example.py --dataset blob
--distance euclidean --parallel 16 --nitems 10000 --centers 5
```

Of course, you can execute the parallel HNSW also for textual data (but in this case, you cannot perform any kind of test)

```
python3 parallel_hnsw_example.py --dataset text
--distance levenshtein --nitems 1000 --centers 10 --parallel 16
```

Bibliography

[AAA+23] L. AlGhamdi, M. Alkharraa, S. AlZahrani, H. Bawazir, W. AlHajri, A. AlHajri, N. Nagy, and M. Gollapalli. Improved parallel k-means clustering algorithm. In *2023 3rd International Conference on Computing and Information Technology (ICCIT)*, pages 416–420, 2023. doi:10.1109/ICCIT58132.2023.10273969.

[AWW+23] X. An, Z. Wang, D. Wang, S. Liu, C. Jin, X. Xu, and J. Cao. Strp-dbscan: A parallel dbscan algorithm based on spatial-temporal random partitioning for clustering trajectory data. *Applied Sciences*, 13(20), 2023. URL: https://www.mdpi.com/2076-3417/13/20/11122, doi:10.3390/app132011122.

[BDG16] M. Baydoun, M. Dawi, and H. Ghaziri. Enhanced parallel implementation of the k-means clustering algorithm. In *2016 3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, pages 7–11, 2016. doi:10.1109/ACTEA.2016.7560102.

[Bel] E. Belval. https://github.com/Belval/hdbscan.git.

[CMS13] RJGB. Campello, D. Moulavi, and J. Sander. Density-based clustering based on hierarchical density estimates. *Advances in Knowledge Discovery and Data Mining (Lecture Notes in Computer Science)*, pages 160 – 172, 2013.

[CuP] CuPy. Cupy: Numpy & scipy for gpu. https://cupy.dev/.

[CVD+20] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The tangled genealogy of iot malware. In *Annual Computer Security Applications Conference*, ACSAC '20, page 1–16, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3427228.3427256.

[Cyt] Cython. Cython: C-extensions for python. https://cython.org/.

[Del19] M. Dell'Amico. Fishdbc: Flexible, incremental, scalable, hierarchical density-based clustering for arbitrary data and distance. *Symantec Research Labs.*, 2019. doi:10.48550/arXiv.1910.07283.

[EKSX96] M. Ester, HP. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery and Data Mining*, 1996.

[FBCM04] C. Fowlkes, S. Belongie, F. Chung, and J. Malik. Spectral grouping using the nystrom method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):214–225, 2004. doi:10.1109/TPAMI.2004.1262185.

[FLC15] JS. Fu, Y. Liu, and HC Chao. Ica: An incremental clustering algorithm based on optics. *Wireless Pers Commun*, page 2151–2170, 2015. doi:10.1007/s11277-015-2517-9.

[fPa] MPI for Python. Mpi for python - mpi for python 3.1.4 documentation. https://mpi4py.readthedocs.io/en/stable/index.html.

[fPb] MPI for Python. Overview - mpi for python 3.1.4 documentation. https://mpi4py.readthedocs.io/en/stable/overview.html.

[fPc] MPI for Python. Tutorial - mpi for python 3.1.4 documentation. https://mpi4py.readthedocs.io/en/stable/tutorial.html.

[FRCC08] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell. A parallel implementation of k-means clustering on gpus. *In Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 340–345, 2008.

[Geea] GeeksForGeeks. Kruskal's minimum spanning tree (mst) algorithm. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.

[Geeb] GeeksForGeeks. Multithreading in python. https://www.geeksforgeeks.org/multithreading-python-set-1/.

[Geec] GeeksForGeeks. Prim's algorithm for minimum spanning tree (mst). https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/.

[Geed] GeeksForGeeks. What is minimum spanning tree (mst). https://www.geeksforgeeks.org/what-is-minimum-spanning-tree-mst/.

[Gor17] A. Goryachev. https://github.com/rust-cv/hnsw.git, 2017.

[Gro] NIH High Performance Computing Group. Python in hpc. https://hpc.nih.gov/training/handouts/171121_python_in_hpc.pdf.

[gro20] Rust Computer Vision group. https://github.com/rust-cv/hnsw.git, 2020.

[Gui] Python Packaging User Guide. Install packages in a virtual environment using pip and venv. https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/.

[HGJ+23] J. Han, X. Guo, R. Jiao, Y. Nan, H. Yang, X. Ni, D. Zhao, S. Wang, X. Ma, C. Yan, C. Ma, and J. Zhao. An automatic method for delimiting deformation area in insar based on hnsw-dbscan clustering algorithm. *Remote Sensing*, 15:4287, 08 2023. doi:10.3390/rs15174287.

[KSG08] P. Kraj, A. Sharma, and N. Garge. Parakmeans: Implementation of a parallelized k-means algorithm suitable for general laboratory use. *BMC Bioinformatics 9*, 200, 2008. URL: https://doi.org/10.1186/1471-2105-9-200.

[LDDR15] A. Lulli, T. Debatty, M. Dell'Amico, and L. Ricci. Scalable k-nn based text clustering. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 958–963, 2015. doi:10.1109/BigData.2015.7363845.

[LDR16] A. Lulli, M. Dell'Amico, and L. Ricci. Ng-dbscan: scalable density-based clustering for arbitrary data. *Proceedings of the VLDB Endowment*, 10:157–168, 2016. doi:10.14778/3021924.3021932.

[Lea] Scikit Learn. 2.3 clustering - scikit-learn 1.3.2 documentation. https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation.

[Lisa] The Code-It List. Python profiling - cprofile and line profiler tools (part 2). https://alexisalulema.com/2022/07/24/python-tips-cprofile-and-line_profiler-tools/.

[Lisb] The Code-It List. Python profiling - time profiling (part 1). https://alexisalulema.com/2022/07/17/python-tips-time-profiling/.

[MB19] R Mohapatra and S Basarkod. Hdbscan-cpp. https://github.com/rohanmohapatra/hdbscan-cpp.git, 2019.

[MH17] L. McInnes and J. Healy. Accelerated hierarchical density based clustering. In *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*, pages 33–42. IEEE, 2017.

[MS] L. Medium Sena. Sharing big numpy arrays across python processes. https://luis-sena.medium.com/sharing-big-numpy-arrays-across-python-processes-abf0dc2a0ab2.

[MY20] Y.A. Malkov and D.A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 42(4), 2020.

[NERa] NERSC. Openmp - nersc documentation. https://docs.nersc.gov/development/programming-models/openmp/.

[NERb] NERSC. Parallel python - nersc documentation. https://docs.nersc.gov/development/languages/python/parallel-python/#pyomp.

[Numa] Numba. Building cython code - cython 3.0.2 documentation. https://numba.readthedocs.io/en/stable/cuda/overview.html.

[Numb] Numba. Numba: A high performance python compiler. https://numba.pydata.org/.

[Numc] NumPy. Numpy: the absolute basics for beginners. https://numpy.org/doc/stable/user/absolute_beginners.html.

[Pan] Pandas. pandas - python data analysis library. https://pandas.pydata.org/.

[Pin] Pinecone. Hierarchical navigable small worlds (hnsw). https://www.pinecone.io/learn/hnsw/.

[pyc] pycuda. pycuda 202.2.2 documentation. https://documen.tician.de/pycuda/index.html.

[pyo] pyopencl. pyopencl 2023.1.2 documentation. https://documen.tician.de/pyopencl/index.html.

[PyP] PyPy. Pypy - features. https://www.pypy.org/.

[Rea] RealPython. An intro to threading in python. https://realpython.com/intro-to-python-threading/.

[RPA] A. Real Python Ajitsaria. What is the python global interpreter (gil)? https://realpython.com/python-gil/.

[SBa] J. SuperFastPython Brownlee. Multiprocessing shared ctypes in python. https://superfastpython.com/multiprocessing-shared-ctypes-in-python/.

[SBb] J. SuperFastPython Brownlee. Python multiprocessing: The complete guide. https://superfastpython.com/multiprocessing-in-python/.

[Scia] Toward Data Science. Understanding hdbscan and density-based clustering. https://towardsdatascience.com/understanding-hdbscan-and-density-based-clustering-121dbee1320e.

[Scib] SciPy. Scipy user guide. https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide.

[SEK98] J. Sander, M. Ester, and HP. Kriegel. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data Mining and Knowledge Discovery*, page 169–194, 1998. doi:10.1023/A:1009745219419.

[Sor18] J. Sorokin. https://github.com/Jorsorokin/HDBSCAN.git, 2018.

[Sym] SymPy. Sympy. https://www.sympy.org/en/index.html.

[um] ubuntu manuals. Ubuntu manpage: multitime. https://manpages.ubuntu.com/manpages/impish/man1/multitime.1.html.

[Wika] Wikipedia. Ackermann function. https://en.wikipedia.org/wiki/Ackermann_function.

[Wikb] Wikipedia. Borůvka's algorithm. https://en.wikipedia.org/wiki/Bor%C5%AFvka's_algorithm.

[Wikc] Wikipedia. Cumulative distribution function. https://en.wikipedia.org/wiki/Cumulative_distribution_function.

[Wikd] Wikipedia. just-in-time compilation. https://en.wikipedia.org/wiki/Just-in-time_compilation.

[Wike] Wikipedia. Minimum spanning tree. https://en.wikipedia.org/wiki/Minimum_spanning_tree.

[Wikf] Wikipedia. pandas (software). https://en.wikipedia.org/wiki/Pandas_(software).

[Wikg] Wikipedia. Reverse-delete algorithm. https://en.wikipedia.org/wiki/Reverse-delete_algorithm.

[XJK02] X. Xu, J. Jäger, and H.P. Kriegel. *A Fast Parallel Clustering Algorithm for Large Spatial Databases*, pages 263–290. Springer US, Boston, MA, 2002. doi:10.1007/0-306-47011-X_3.

[ZNFVW22] X. Zhang, X. Niu, P. Fournier-Viger, and B. Wang. Two-stage traffic clustering based on hnsw. In H. Fujita, P. Fournier-Viger, M. Ali, and Y. Wang, editors, *Advances and Trends in Artificial Intelligence. Theory and Practices in Artificial Intelligence*, pages 609–620, Cham, 2022. Springer International Publishing.

[ZWS21] W. Zheng, D. Wang, and F. Song. Designing a parallel feel-the-way clustering algorithm on hpc systems. *The International Journal of High Performance Computing Applications*, 35:154–169, 2021. doi:10.1177/1094342020975194.