POLITECNICO
MILANO 1863

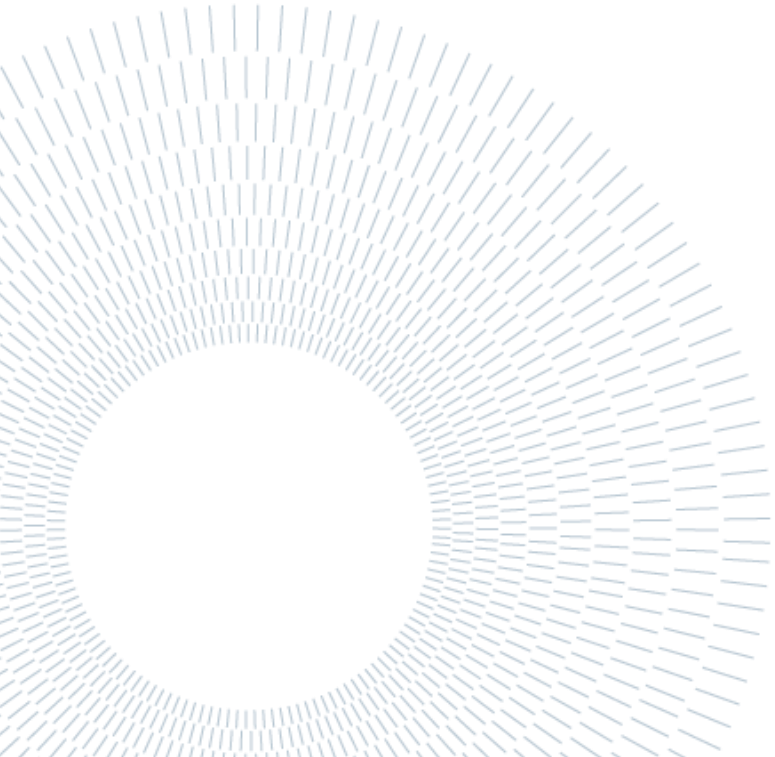
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Computation of Flexible Skylines in a distributed environment

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING
INGEGNERIA INFORMATICA

Author: **Emilio De Lorenzis**

Student ID: 10612395
Advisor: Davide Martinenghi
Academic Year: 2021-22



Abstract

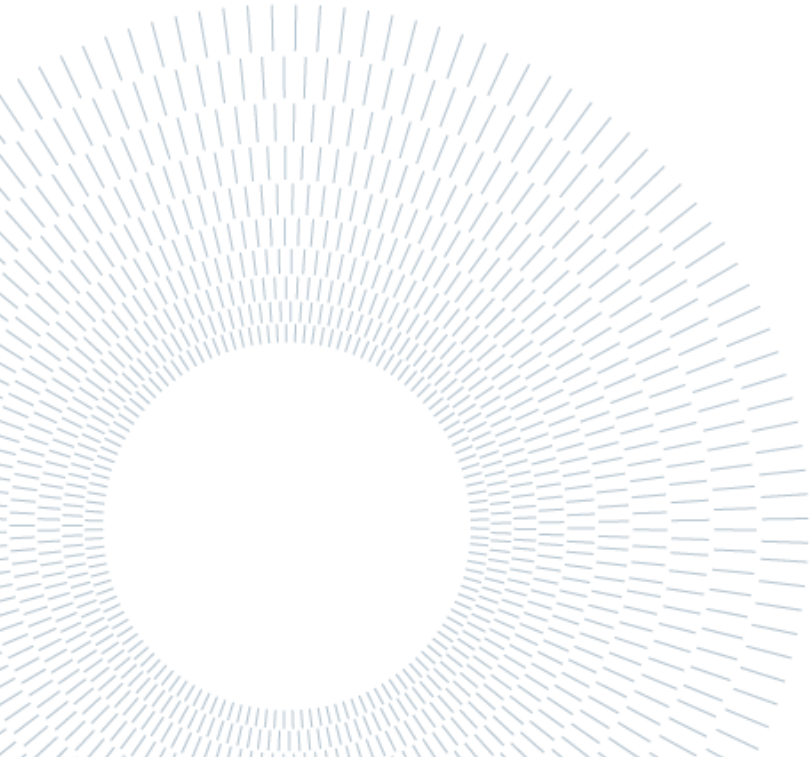
Skyline queries are a way of finding interesting data within a large dataset by considering attributes all at the same level. Flexible skylines, on the other hand, are methods of finding interesting data by applying constraints on the attributes, thereby also reducing the amount of data returned. The computation of these skylines can become very time-consuming in the case of very large datasets. In this thesis, we will implement parallel computation algorithms for these datasets, trying to reduce the execution time as much as possible. We will introduce several algorithms for parallel computation taken from the literature applied to skyline computation and also adapt them to the computation of the Flexible Skyline ND and PO operators. We will try to introduce improvements to these algorithms with an initial filtering phase aimed at decreasing the dataset size before performing the parallel phase, and once it is established that the sequential part is the most expensive, we will propose an all parallel algorithm in which we will eliminate it totally. To carry out the parallelization of these algorithms, we will use the PySpark framework and see in detail how these algorithms behave by changing the size of the dataset and the dimensions.

Key-words: Skyline, Flexible Skyline, PySpark, Parallel Algorithms.

Abstract in italiano

Le Skyline query sono un modo per trovare dati interessanti all'interno di un dataset di grandi dimensioni che considerano gli attributi tutti allo stesso livello. Le Flexible Skyline invece, sono dei metodi per trovare dati interessanti applicando dei vincoli sugli attributi, riducendone cosi anche la quantità di dati restituiti. Il calcolo di queste Skyline può diventare molto dispendioso nel caso di dataset di grandi dimensioni. In questa tesi andremo ad implementare algoritmi paralleli per il calcolo di questi set di dati cercando di diminuirne il tempo di esecuzione quanto più possibile. Introdurremo diversi algoritmi per il calcolo parallelo presi dalla letteratura applicati al calcolo delle Skyline e li adatteremo anche al calcolo degli operatori Flexible Skyline ND e PO. Cercheremo di introdurre miglioramenti a questi algoritmi con una fase di filtraggio iniziale volta a diminuire la grandezza del dataset prima di effettuare la fase parallela e una volta assodato che la parte sequenziale è la più dispendiosa, proporremo un algoritmo tutto in parallelo in cui la elimineremo del tutto. Per effettuare la parallelizzazione di questi algoritmi utilizzeremo il framework PySpark e vedremo nel dettaglio come si comportano questi algoritmi cambiandone la grandezza del dataset e le dimensioni.

Parole chiave: Skyline, Flexible Skyline, PySpark, Algoritmi Paralleli.



Contents

1 Introduction

One of the main tasks when working with big data is to find all the most interesting data in a dataset, whether working in the field of Database Systems or in the field of Data Mining or Machine learning. Multi-criteria analysis [1] deals with decision-making applications, selecting the best alternatives in different contexts, such as databases, by finding the best tuples that are not dominated by any other.

1.1. Motivations and objectives of the work

In recent years, with the growth of big data, attempts are being made to find a way to search within larger and larger datasets, the data that might be more interesting and to pay special attention to. Skyline queries are an efficient and fast way to select a subset of these huge datasets that might be more interesting, returning the set of tuples not dominated by any other tuple. We have that a tuple t dominates another tuple s if and only if t is not worse in any attribute than s and strictly better in at least one.

The top-k or ranking queries approach [2], instead, involves reducing the original multi-objective problem into a single-objective problem using a scoring function. This function incorporates parameters, such as weights, to adjust scales and reflect the user's preference for different attributes. Skylines provide a global view of data that might be of interest but, unlike top-k queries, do not take user preferences into account because they consider attributes all at the same level, in addition they might contain too many tuples and thus give too little useful information that cannot help the user make decisions. The greater the dimensionality of the attributes, the larger the skyline set will be.

To try to overcome these problems in this thesis, we will also discuss Flexible skylines [3], which are a hybrid between Skyline queries and top-k queries where with constraints on attributes we try to give different importance to different attributes. The concept of F-dominance was introduced for this purpose: A tuple t F-dominates another tuple s if and only if t is always better than or equal to s according to all scoring functions in F. Flexible skylines thus return a subset of the skylines and are divided into 2 operators: ND which returns the subset of non F-dominated tuples and PO

which returns a subset of ND representing all tuples that are potentially optimal with respect to a scoring function in F. A benefit of using F-skylines is that the cardinality of the interesting tuples decreases compared to that of skylines since we apply constraints on the attributes, so the tighter the constraints, the narrower the set of results. Like skylines, these also suffer from dimensionality, and the more we increase the dimensions, the larger these subsets will be.

In this thesis, several algorithms will be presented for computing Skylines and Flexible Skylines. We will see how the centralized version of these algorithms is difficult to use as the cardinality of the dataset increases, as the execution time of these algorithms will become unmanageable. Therefore, it is important for us to introduce parallel frameworks to try to divide the computation of these subsets into several partitions that will be executed in parallel. Each partition will have a part of the dataset and will return a local skyline that will be equal to the skyline set of that subset of points and not all the dataset. Once all the local skylines have been computed, they will merge and there will be a sequential phase in which we will find the global skyline from the local skylines found from each partition.

The important goal is to find the best possible partitioning in order to have local skylines as small as possible and thus reduce the final sequential phase, which will be the slowest. Later in this thesis we will present improvements to these types of partitioning by attempting to perform an initial filtering phase, thus decreasing the load on the parallel part, and then we will study a method to eliminate the sequential phase totally.

One aim of this thesis will be to implement the algorithms in both the centralized and parallel versions with all the types of partitioning widely used for skyline queries and apply them to our specific case of the Flexible Skyline computation using the PySpark framework [4], and to verify which algorithm performs best and what conditions makes it perform the best.

Another goal will therefore be to find new algorithms that allow for greater speed of execution, in the next chapters we will look at initial point filtering techniques and also introduce a new algorithm for computing the most representative points for filtering, which in this thesis we have adapted to make them work for both skyline computation and Flexible Skyline operators.

After that, algorithms will be proposed in which the final sequential part will be eliminated for the computation of the Flexible Skyline operators ND and PO, and as we will see, these algorithms will be the ones that will give us the best results compared to the other techniques because we will eliminate the sequential part which is the longest to perform.

We will try to increase the cardinality and dimensionality of the datasets as much as possible, and will use both a local machine with 4 cores and a virtual machine granted by the Polimi Datacloud with 30 cores to run the algorithms.

1.2. Structure of the work

The first chapter is the introduction, after that, the second chapter called preliminaries, focuses on giving a theoretical definition of Skylines and Flexible skylines, explaining their properties and the theorems used to compute them. We will also see the distinction of the computation of the two operators ND and PO in different strategies.

In the third chapter we will introduce the sequential algorithms for skyline and Flexible skyline computation, in particular we will see Block Nested Loop (BNL), the Sort-Filter-Skyline Algorithm (SFS) and the Sort and Limit Skyline algorithm (SaLSa) for Skyline computation, while for ND computation we will see sorted and unsorted versions of two main algorithms, 2-phase LP and 1- and 2-phase VE. Finally, for the computation of PO we will see two main algorithms, one applying a Primal PO Test and the other performing the same version but with the Dual version.

In chapter four, we will introduce parallel algorithms with different types of partitioning, and in particular we will look at random partitioning, grid partitioning, angular-based partitioning and One-Dimensional Slice Partitioning.

After that, in the fifth chapter we will try to improve these algorithms by applying several enhancements. First of all, by applying an initial filtering phase so as to decrease the initial dataset using certain techniques, thus eliminating all those points that cannot definitely be Skyline points. And after that we will see an algorithm that eliminates the final sequential phase, and performs everything in parallel.

In Chapter 6, we will introduce the Spark framework and the environment where the code will be executed. We will also present the packages utilized for computing the various algorithms.

In chapter seven we will verify the results of these algorithms performed both with a local machine with 4 cores and with a virtual machine with 30 cores. We will analyse the results using synthetically created datasets, paying particular attention to the anticorrelated ones, which take the longest to finish the computation. We will change the cardinality of these datasets and dimensionalities to see how long the algorithms take to finish. We will also look at how changing the number of partitions affects the duration of the parallel algorithms and how changing the number of cores affects the execution time.

Finally, in Chapters eight and nine we will discuss the related works and the conclusions.

1.3. Thesis contributions

The contributions of this thesis are several. First, pre-existing algorithms for skyline computation and Flexible Skyline operators are introduced and tested in using Python as a programming environment. After that, the Spark framework is introduced with which we are going to parallelize the computation of these algorithms.

One of the contributions of this thesis is to take some partitioning algorithms from the literature applied so far only for Skyline computation, and readapt them for the computation of the Flexible Skyline ND and PO operators.

We will see some improvements to these algorithms, some taken from the literature, others introduced for the first time in this thesis. The one proposed in this thesis is to do filtering with representatives points taking the first n sorted points of each partition after the angular partitioning technique. Another contribution will be to take this technique, and readapt it to the computation of the ND set with some improvements.

Another contribution will be to create a parallel algorithm without a sequential part that is good for the computation of the two Flexible Skyline operators ND and PO.

Having taken all these algorithms, we will implement them using the Pyspark framework and derive an experimental analysis in which we will test the efficiency of the various algorithms and find the optimal setting to make them work best.

2 Preliminaries

In this chapter, we will give a theoretical definition to the concept of Skyline and Flexible Skyline, introducing their definitions and discussing their properties. All definitions, theorems and results have been taken from the literature and explained in this chapter using examples to facilitate the understanding. We will look in detail at two methods for the computation of ND points using two different theorems and two methods for the computation of PO to solve a LP, one with primal and the other with dual.

2.1. Skyline query

The skyline query was first introduced in [5] to find all the best tuples in a database, and after that it was adapted to more fields. It is based on the concept of dominance, in fact a skyline query returns all tuples (points) that are not dominated within a dataset. In this thesis we consider smaller values as better, but it is only a convention we use here, the opposite could be used. Let us first introduce the definition of dominance between tuples.

Definition 2.1: Given two tuples t, u $\in \mathbb{R}^d$ belonging to the same dataset S, t dominates u, written t $\prec$ u, if and only if t is not worse than u in all dimensions and better in at least one. Equivalently:

$$t \prec u \leftrightarrow \begin{cases} \forall i \in [1,d] \rightarrow t[i] \leq u[i] \\ \exists j \in [1,d] \rightarrow t[j] < u[j] \end{cases} \tag{2.1.1}$$

Thus we have that the skyline set of a Dataset S, denoted by SKY(s) is equal to:

$$SKY(s) = \{t \in s | \nexists u \in s.\ u \prec t\} \tag{2.1.2}$$

The first formula (2.1.1) gives us the definition of dominance between tuples, telling us when one tuple dominates another and is therefore "better". The second (2.1.2) gives us the definition of the Skyline set which is equal to all non-dominated tuples

within a dataset, hence those tuples for which there is no "better" tuple within the same dataset.

Consequently, we can also give a definition to the set of dominated tuples that will be simply equal to the set of tuple of given dataset that are not Skyline.

$$Dominated(s) = S - SKY(s) \tag{2.1.3}$$

We can also give another definition of Skyline by introducing the concept of monotone scoring functions. Let us first introduce the definition of a monotone scoring function:

Definition 2.2: A scoring function f is a function that takes a tuple with non-negative real values for attributes and returns a non-negative real value representing the score. For a tuple t belonging to a dataset S, the value f(t) represents the score. We have that a scoring function F is monotone if, for any tuple t, u on S, we have that:

$$\forall\, i \ \in [1, d] \mid t[i] \leq u[i] \rightarrow f(t) \leq f(u) \tag{2.2.1}$$

We could see the scoring functions, in our case where we consider the lowest values to be the better ones, as the measure of distance from the origin of the point, preferring those closer to it. The new definition of Skyline can be specified in this way as shown in [3]:

$$SKY(s) = \{t \in s | \exists f \in MF.\ \forall\, u \ \in s.\ u \neq t \ \rightarrow f(t) < f(u)\} \tag{2.2.2}$$

Where *MF* is the infinite set of all monotone scoring functions.

Example: An example of a Skyline computation could concern the choice of a restaurant. We take into account two characteristics of each restaurant, the average cost (in euros) per person and the distance from the centre (Km). We want to find all restaurants that have the best compromise between these two characteristics, to have both as low as possible. Suppose we have 5 restaurants with the following values for the attributes A={cost, distance}, r_1={30, 2}, r_2={20, 4}, r_3={35, 2.5}, r_4= {50, 1}, r_5 = {40, 3}.

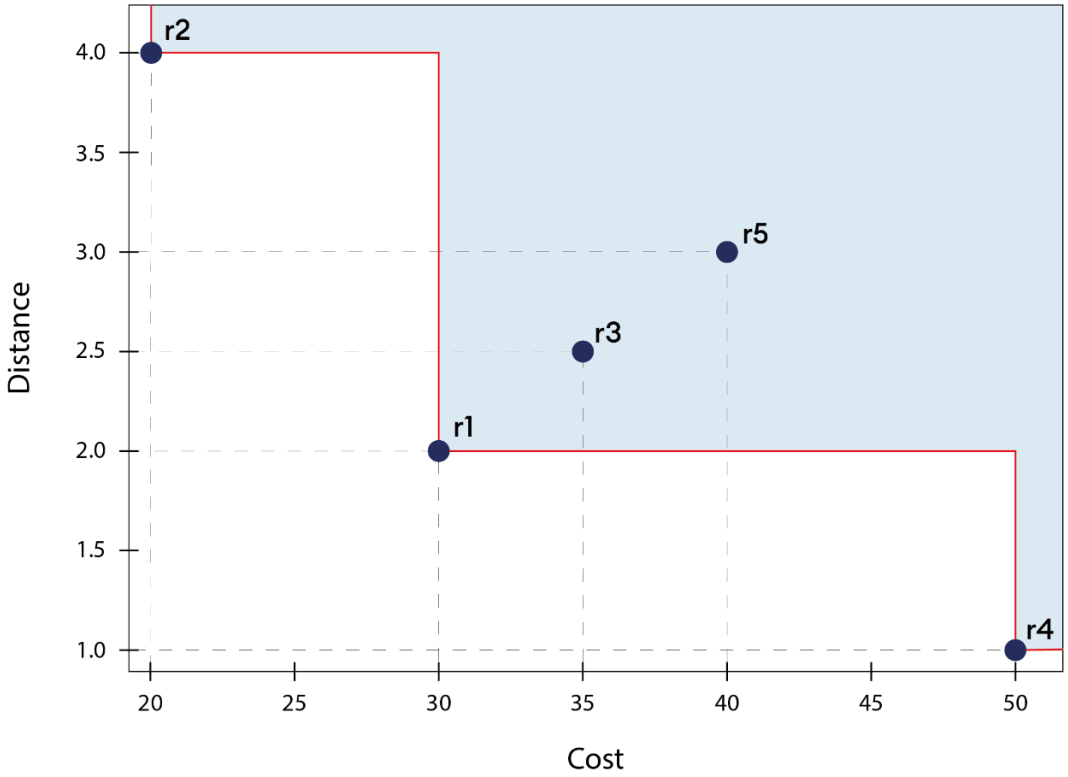


Figure 2.1: A Skyline example with 5 points in 2 dimensions

As we can see in the figure 2.1 we have that the skyline points are r_1, r_2, r_4, i.e. all the points that have no other points that have both lower cost and distance values. We can also see the set of dominated points that is the one in the upper right area and is composed of points r_3 and r_5 which are both dominated by r_1.

2.2. Flexible Skyline

2.2.1 F-dominance

With the flexible skylines we try to overcome the problem of skyline queries which do not take into account user preferences by applying constraints on attributes. To do this we present two operators introduced in [3] that behave like skyline queries but are applied to a non-empty set of monotone scoring function *F*. From now on, we will introduce another type of dominance between tuples that takes scoring functions into account.

Definition 2.3: (F-dominance) Let *F* be a non-empty set of monotone scoring functions and t, u tuples belonging to a dataset S, with t ≠ u. We have that t *F-dominates* u, written as $t \prec_F u$, if:

$$\forall f \in F \rightarrow f(t) \leq f(u) \tag{2.3.1}$$

It can be deduced from the definition that F-dominance is transitive. So if $t \prec_F u$, and $u \prec_F r$, then $t \prec_F r$.

Definition 2.4: (Dominance Region) Given a tuple t and a set of monotone scoring functions F, the F-dominance region *DR(t; F)* of tuple t in a dataset S under the set F is the set of all tuples that are *F-dominated* by t:

$$DR(t;F) = \{ u \in s \mid t \prec_F u \} \tag{2.4.1}$$

Example: Given the tuple t = (1, 0) and u = (0.5, 1), and the monotone scoring functions $f_1(x,y) = x + y$ and $f_2(x,y) = 2x + y$. We have that $t \prec_F u$, since the *definition 2.3* holds: $f_1(t) = 1 < f_1(u) = 1.5$ and $f_2(t) = f_2(u) = 2$.

2.2.2 Non-dominated Flexible Skyline (ND)

We now introduce the first Flexible Skyline operator, called non-dominated Flexible Skyline (ND), which is the set of all tuples that are not F-dominated within a dataset.

Definition 2.5: The set of non-dominated Flexible Skyline (ND) of a dataset s with respect to a set of monotone scoring functions $F \subseteq MF$, is defined as:

$$ND(s;F) = \{t \in s \mid \nexists\, u \in s.\ u \prec_F t\} \qquad (2.5.1)$$

This formula 2.5.1 is similar to the Skyline formula except that instead of simple dominance $\prec$ we have F-dominance $\prec_F$.

We now introduce two main strategies for calculating ND when each function F is a monotonically transformed, linear-in-the-weights (MLW) function, i.e. a function with the following form:

$$f(t) = h\left(\sum_{i=1}^{d} w_i g_i(t[i])\right) \qquad (2.5.2)$$

Where $W = (w_1, \dots, w_d) \in W(C)$, and $W(C)$ is the subset of all normalized weight vectors that satisfies C, C is a set of linear constraints, and h and g_i are continuous and monotone transforms, such that are all either non-decreasing or non-increasing. We refer to $g_i(t[i])$ as the marginal score of tuple t.

Theorem 1: (*F-dominance test)* Let F be a set of *MLW* functions subject to a set $C = \{C_1, \dots, C_c\}$ of linear constraints on weights, where $C_j = \sum_{i=1}^{d} a_{ji} w_i \leq k_j$ (for $j \in [1,c]$) . Then, $t \prec_F u$ if and only if the following linear programming (LP) problem in the variables $W = (w_1, \dots, w_d)$ has a non-negative solution:

$$\begin{aligned} &\textit{minimize} && \textstyle\sum_{i=1}^{d} w_i(g_i(u[i]) - g_i(t[i])), \\ &\textit{subject to} && w_i \in [0,1] \quad i \in [1,d], \\ & && \textstyle\sum_{i=1}^{d} w_i = 1, \\ & && \textstyle\sum_{i=1}^{d} a_{ji} w_i \leq k_j \quad j \in [1,c]. \end{aligned} \qquad (2.5.3)$$

Example: Given 2 tuples t = (0.3, 0.5) and u = (0.6, 0.35) and the following constraints on the weights $w_2 \leq w_1$, according to *Theorem 1,* we have that t $\prec_F$ u if and only if this LP has a non-negative solution:

$$\begin{aligned} &\textit{minimize} && w_1(0.6-0.3)+w_2(0.35-0.5) \\ &\textit{subject to} && w_1, w_2 \in [0,1], \\ & && w_1+w_2=1, \\ & && w_2 \leq w_1. \end{aligned}$$

In this case we have that t $\prec_F$ u when $w_1 = 1$ and $w_2 = 0$.

This strategy of computation of ND is heavy and time-consuming because we have to solve a different LP problem for each of the *F-dominance* Tests, that are $O(N^2)$, where N is the cardinality of the dataset. The second strategy involves computing the dominance region of each point and eliminating all points that are part of at least one of these regions. By doing so we avoid solving linear problems that are time-consuming, and we only calculate the dominance region of each point and check whether the others are part of it or not, and if so we can deduce if a point is not part of the ND set. To compute the dominance region of t (*DR(t; F)*), we have to compute the vertices of the convex polytope *W(C)* . For this purpose, we introduce a new theorem:

Definition 2.6: (F-dominance Region) Let F be a set of MLW functions subject to a set C = $\{C_1,...,C_c\}$ of linear constraints on weights, where $C_j = \sum_{i=1}^{d} a_{ji} w_i \leq k_j$ (for j ∈ [1,c]) and let $W^{(1)}, \dots, W^{(q)}$ be the vertices of the convex polytope *W(C).* The dominance region of a tuple t under *F* is the locus of the points u defined by the q inequalities:

$$\sum_{i=1}^{d} w_i^{(l)} g_i(u[i]) \geq \sum_{i=1}^{d} w_i^{(l)} g_i(t[i]), \qquad l \in [1,q] \tag{2.6.1}$$

Theorem 2: A tuple t F-dominates another tuple u if and only if u belongs to the F-dominance region of t, i.e. if u satisfies the equations of Definition 2.6:

$$t \prec_F u \leftrightarrow u \in DR(t;F) \tag{2.6.2}$$

Example: Let us take the same example as above. We have the constraint $C = \{w_2 \leq w_1\}$ and we have to consider also $w_1 + w_2 = 1$ and $w_1, w_2 \in [0,1]$. The vertices of *W(C)* are

$W^{(1)} = (1,0)$ and $W^{(2)} = \left(\frac{1}{2}, \frac{1}{2}\right)$. According to *Theorem 2* we can compute the dominance region *DR(t; F)* of t = (0.3, 0.5):

$$\begin{cases} u[1] \geq 0.3 \\ \frac{1}{2}u[1] + \frac{1}{2}u[2] \geq \frac{1}{2}0.3 + \frac{1}{2}0.5 \end{cases}$$

We have that the tuple u = (0.6, 0.35) satisfies this equations, so we have that $t \prec_F u$.

The advantage of this strategy w.r.t. the first one is that the computation of the dominance region *DR(t; F)* of a tuple t is computed only once, and so the computation of the vertex enumeration of the polytope which introduce the most significant overhead is performed only once.

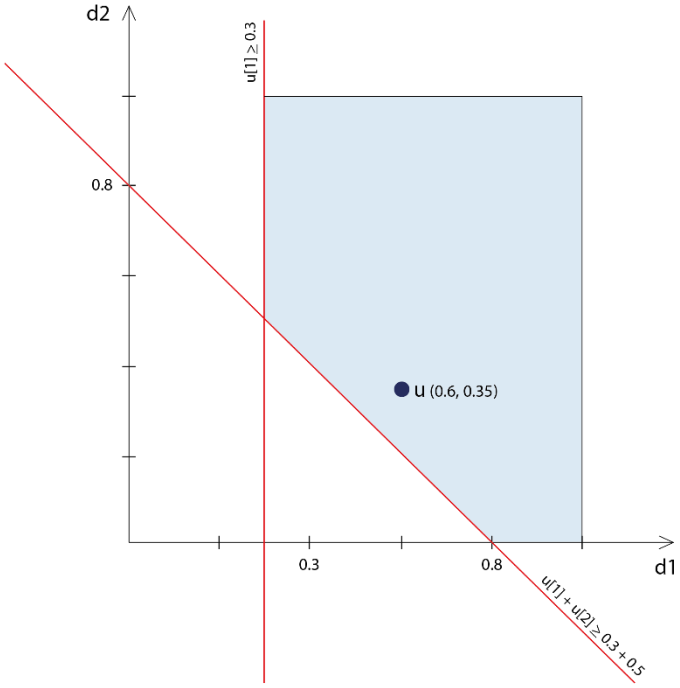


Figure 2.2 : u is in the F-dominance region of t, so $t \prec_F u$.

2.2.3 Potentially Optimal Flexible Skyline (PO)

We now introduce the second operator, called Potentially Optimal Flexible Skyline (PO), which returns a set of tuples considered optimal (the best) according to some monotone scoring functions *F*.

Definition 2.7: The set of Potentially Optimal Flexible Skyline (PO) of a dataset s with respect to a set of monotone scoring functions $F \subseteq MF$, is defined as:

$$PO(s;F) = \{t \in s \mid \exists f \in F . \forall u \in s.\ u \neq t \rightarrow f(t) < f(s)\} \qquad (2.7.1)$$

We have that a tuple t that is potentially optimal is certainly also ND, but the fact that it is not F-dominated is only a necessary condition for t to be potentially optimal and not sufficient. There are two ways to find the PO set: start from the whole dataset and find the set in one step or find the ND set first and then compute PO from that. In both cases, we have to solve some LP and there are two strategies: solve the Primal or the Dual PO test.

Theorem 3: (Primal PO Test) Let F be a set of MLW functions subject to a set $C = \{C_1,\ldots,C_c\}$ of linear constraints on weights, where $C_j = \sum_{i=1}^{d} a_{ji}\, w_i \leq k_j$ (for $j \in [1,c]$) and $ND(s;\ F) = \{t_1,\ t_2, \ldots, t_\sigma, t\}$. Then $t \in PO(s;F)$ if and only if the following LP problem in the variables $W = (w_1, \ldots, w_d)$ and ϕ has a strictly positive optimal solution:

$$\begin{aligned} &\textit{maximize} && \phi, \qquad (2.7.2)\\ &\textit{subject to} && \textstyle\sum_{i=1}^{d} w_i(g_i(u[i]) - g_i(t_j[i])) + \phi \leq 0 \quad j \in [1,\sigma],\\ & && w_i \in [0,1] \quad i \in [1,d],\\ & && \textstyle\sum_{i=1}^{d} w_i = 1,\\ & && \textstyle\sum_{i=1}^{d} a_{ji} w_i \leq k_j \quad j \in [1,c]. \end{aligned}$$

Theorem 4: (Dual PO Test) Let F be a set of MLW functions subject to a set C = {C_1,...,C_c} of linear constraints on weights, $W^{(1)}, \dots, W^{(q)}$ be the vertices of the convex polytope $W(C)$ and $ND(s; F) = \{t_1, t_2, \dots, t_\sigma, t\}$. Then $t \in PO(s; F)$ if and only if the following linear system in the variables $\alpha = (\alpha_1, \dots, \alpha_\sigma)$ is unsatisfiable:

$$\sum_{i=1}^{d} w_i^{(l)} \left(\sum_{j=1}^{\sigma} \alpha_j \, g_j(u_j[i])\right) \leq \sum_{i=1}^{d} w_i^{(l)} g_i(t[i]), \quad l \in [1, q] \tag{2.7.3}$$

$$\alpha_j \in [0, 1] \quad j \in [1, \sigma],$$

$$\sum_{j=1}^{\sigma} \alpha_j = 1.$$

2.2.4 Properties

The first property of the Flexible Skyline operators regards the relationship with the Skyline operator SKY, in fact we have that when the monotone scoring function set coincides with the infinite set of all monotone scoring functions MF, we have that $\prec_{MF}$ coincides with $\prec$ and thus:

$$PO(s; MF) = ND(s; MF) = SKY(s) \tag{2.8.1}$$

From this property we can deduce that:

Property 1: PO is a subset of ND and ND is a subset of SKY:

$$PO(s; F) \subseteq ND(s; F) \subseteq SKY(s) \tag{2.8.2}$$

Another property of ND and PO is that they are monotone operators with respect to the set of scoring functions.

Property 2: Given two sets F_1 and F_2 of monotone scoring functions, if $F_1 \subseteq F2$ then:

$$ND(s; F_1) \subseteq ND(s; F_2) \tag{2.8.3}$$
$$PO(s; F_1) \subseteq PO(s; F_2)$$

From this property we can see that both the ND and PO sets become smaller as the constraints become more restrictive.

Property 3: Let F be a set of MLW functions subject to a set C of linear constraints on weights. For any dataset S, we have that $PO(s;F) = PO(ND(s;F);F)$

This property tells us that we can calculate the PO set either from the original dataset S or from the ND set.

3 Sequential Algorithms

This chapter will present some of the sequential algorithms for both the computation of the Skyline and the ND and PO sets. All the algorithms we will see in this chapter are taken from the literature.

Symbol	Meaning
S	Input dataset
t, u	Tuples
C	Constraints
F	Family of MLW functions.
W	Vertices of W(C)
SKY	Skyline set
nd	ND set
po	PO set
f	Sorting Function
n	Number of partitions
P	Partitions
d	Number of dimensions
r	Number of representatives
LS	Local Set
GS	Global Set

Table 5.1: Table of Notation

3.1. Skyline Algorithms

3.1.1. Block Nested Loops (BNL)

The first algorithm we will see for sequential Skyline computation was introduced by Borzsony et al. [5] and is called *Block Nested Loop (BNL)*. The basic idea is to use a memory called window (SKY), initially empty, in the main memory. We scan the original dataset S without performing any sorting and for each new tuple t we check whether or not it is dominated by the tuples in SKY. If there is a tuple u in SKY that dominates t, i.e. $u \prec t$, then t is discarded, otherwise added to SKY. If the tuple is added to SKY then we must check whether there are tuples in SKY previously added that are dominated by the new addition. In this case, we sequentially scan SKY and discard all tuples dominated by t from it. At the end of the scan of the entire dataset, only the tuples belonging to the Skyline will be left.

Algorithm 1 Block Nested Loops (BNL)

```
     Input: S: dataset
     Output: SKY: Skyline Set
1:   SKY ← [ ]
2:   for t in S do
3:       for u in SKY do
4:           if u ≺ t then continue to line 2
5:       for u in SKY do
6:           if t ≺ u then SKY ← SKY \ { u }
7:       SKY ← SKY ∪ { t }
8:   return SKY
```

The advantage of this algorithm is that it is very simple and can be applied to a dataset without doing any kind of data pre-processing, but is not very efficient when increasing the cardinality of the dataset.

3.1.2. Sort Filter Skyline (SFS)

The *SFS algorithm* is an improvement of the BNL algorithm introduced by Chomicki et al [7]. The algorithm is very similar to BNL, except that there is a pre-processing of the data in which the tuples are sorted using a monotone function f. By performing this sorting, we are sure that no tuple coming after a certain tuple t dominates it. So the functioning of SFS is very simple, first the dataset is ordered using a monotone function f, after which BNL is applied without, however, having to perform a check on the set SKY as in BNL, because we are sure that once a tuple t has been added to the set SKY, there can be no tuple within SKY that can be dominated by t.

Algorithm 2 Sort Filter Skyline (SFS)

```
      Input: S: dataset, f: sort function
      Output: SKY: Skyline Set
1:    SKY ← [ ]
2:    S_sorted ← f(S)
3:    for t in S_sorted do
4:        for u in SKY do
5:            if u ≺ t then continue to line 3
6:        SKY = SKY ∪ { t }
7:    return SKY
```

Compared to BNL, we have greatly reduced the number of comparisons, however the pre-processing phase could be heavy for very large datasets.

3.1.3. Sort and Limit Skyline algorithm (SaLSa)

This algorithm [8] was born with the idea of improving SFS by trying to avoid reading the entire dataset S. The basic idea is that if the dataset is sorted using a monotone function f, there is no need to check the entire dataset S but we can stop earlier. In general, the number of comparisons may decrease a lot depending on the type of function we use to sort the data.

The algorithm involves the initialization of a set of unread tuples U equal to the dataset S and a stop point p_{stop}, which will be used to stop the algorithm in advance. First we need to perform the sorting using a monotone function f. After that we scan the set U sequentially and compare it with the set SKY which will be initially empty. If no point in SKY dominates the point under evaluation then we add it in SKY, and if necessary update the value of the stop point. At the end of each iteration, the algorithm verifies whether it can terminate by making a check. This check consists of seeing if the stop point dominates all points in the set U that have not been read yet. If yes, the algorithm terminates, otherwise it continues.

Algorithm 3 Sort and Limit Skyline algorithm (SaLSa)

```
     Input: S: dataset, f: sort function
     Output: SKY: Skyline Set
1:   SKY ← [ ], U ← S, stop ← false, p_stop ← undefined
2:   U ← f(U)
3:   while not stop ∧ U ≠ ∅ do
4:       p ← get next point from U, U ← U \ {p}
5:       if not SKY ≺ p then SKY ← SKY ∪ {p}, update p_stop
6:       if p_stop ≺ U then stop ← true
7:   return SKY
```

The key factors that influences the performance of SaLSa are the choice of the sorting function and the strategy for updating the stop point.

3.2. Flexible Skyline Algorithms

3.2.1. ND computation

There are two strategies for computing ND [3], the first is to start from the skyline set and then calculate the ND set (2-phase), the second is to start from the entire input dataset (1-phase). In the first case, no restriction is placed on the algorithm for computing the skyline set, we will see two variants, one unsorted which uses BNL and one sorted which uses SFS instead. It is also valid for the one-step version to choose whether or not to order the input dataset. The main difference between these algorithms is the way of checking the F-dominance, whether to verify it with *Theorem 1* by solving LP, or to use *Theorem 2* by solving a system of inequalities.

The main steps can be summarized by first choosing the number of phases: 1-phase if we want to compute ND directly from the input dataset, 2-phase if we want to compute ND after having computed the Skyline. After that, we choose whether or not to apply sorting ('U' for the unsorted version, 'S' for the sorted version) to produce a topological sort with respect to the F-dominance relation, i.e. if a tuple t precedes a tuple u in the sorted input dataset, then $u \not\prec_F t$. To get the topological sort, we will use as sort function a weighted sum in which the weights are the coordinates of the centroid of the polytope *W(C).* Next, we choose the way in which to verify F-dominance, and to do so we have seen two alternatives: the first by solving an LP problem as done in Theorem 1 (denoted 'LP' in the name of the algorithm), and the second by verifying whether the tuple is part of the dominance region of another tuple as in Theorem 2, using the vertex enumeration of the polytope *W(C)* (denoted 'VE' in the name of the algorithm).

We'll see in chapter 7 that the Sorted algorithms are faster than the Unsorted, and the second strategy to check the F-dominance with Vertex Enumeration is much faster than the LP strategy. For this reason, we'll see only the sorted 1-phase VE strategy.

3.2.1.1. ULP2 and SLP2

The first two algorithms we will see are ULP2 and SLP2, the principle of both is the same, the only difference is the algorithm with which the skyline is computed from, since they are 2-phase algorithms. ULP2 uses BNL to compute the skyline, while SLP2 uses SFS with a sorting function that is equal to the weighted sum with the coordinates of the centroid of *W(C)* as weights.

Algorithm 4 ULP2

```
     Input: S: dataset, C: constraints, F: family of MLW functions.
     Output: nd: ND Set
1:   skyline ← BNL(S)
2:   nd ← skyline
3:   for s in skyline do
4:       for t in nd in reverse order do
5:           if s == t then continue to line 4
6:           if t ≺_F s then nd ← nd \ {s} and continue to line 3
7:   return nd
```

Algorithm 5 SLP2

```
     Input: S: dataset, C: constraints, W: vertices of W(C), F: family of MLW functions.
     Output: nd: ND Set
1:   S ← sort(S, W)
2:   skyline ← SFS(S)
3:   nd ← [ ]
4:   for s in skyline do
5:       for t in nd do
6:           if t ≺_F s then continue to line 4
7:       nd ← nd ∪ {s}
8:   return nd
```

ULP2 after computing the skyline with BNL, initializes the ND set equal to the skyline, and sequentially scans the entire skyline set comparing it with the ND set. Using Theorem 1, it removes from the ND set all points F-dominated by the point under consideration. SLP2, on the other hand, exploits initial sorting, so no tuple will ever be removed from the ND set for the sorting property, and the vertices of the polytope *W(C)* are computed only once at the beginning. So if a tuple is not F-dominated by all the tuples in the ND set, it can be added to it and will never be removed from it.

3.2.1.2. UVE2 and SVE2

These two algorithms are the unsorted and sorted versions of the strategy that performs the F-dominance check using Vertex Enumeration and thus *Theorem 2*. Being two 2-phase algorithms, as always there is the first phase where there is the skyline computation using BNL for the Unsorted version and SFS for the Sorted version.

Algorithm 6 UVE2

```
    Input: S: dataset, C: constraints, W: vertices of W(C), F: family of MLW functions.
    Output: nd: ND Set
1:  skyline ← BNL(S)
2:  nd ← skyline
3:  for s in skyline do
4:      compute left-hand side of Inequalities
5:      for t in nd do
6:          if s == t then continue to line 4
7:          if t ≺_F s then nd ← nd \ {s} and continue to line 3
8:  return nd
```

Algorithm 7 SVE2

```
    Input: S: dataset, C: constraints, W: vertices of W(C), F: family of MLW functions.
    Output: nd: ND Set
1:  S ← sort(S, W)
2:  skyline ← SFS(S)
3:  nd ← [ ]
4:  for s in skyline do
5:      Compute left-hand side of inequalities
6:      for t in nd do
7:          if t ≺_F s then continue to line 4
8:      nd ← nd ∪ {s}
9:  return nd
```

UVE2 after computing the skyline with BNL, initializes the ND set equal to the skyline, and sequentially scans the entire skyline set, computing the left-hand side of the inequalities one time for each point, and then compare it with the ND set. UVE2 scans the unordered skyline set to verify that the point into consideration is part of the ND set, compares it with all the different points in ND, and for each computes the right-hand side of the inequalities.

If there is at least one point in the ND that by *Theorem 2* F-dominates the point into consideration, then it will be removed from the ND set and the iteration will continue, otherwise if no point F-dominates it then it will be kept in the ND set. SVE2 on the

other hand, taking advantage of initial sorting and its property that a tuple that comes later in the sorted dataset cannot F-dominate one that comes earlier, initializes the nd set empty, and adds, by scanning the skyline set, the points that are not F-dominated by any point within the ND set, according to *Theorem 2*.

3.2.1.3. SVE1 and SVE1F

Now we move on to the one-phase algorithms, but we will only consider the sorted and VE ones because they are the fastest compared to the unsorted and LP versions. SVE1 and SVE1F are both two algorithms that start from the original input dataset and not from the skyline set like the others we have seen, and their peculiarity is that they check the points one by one simultaneously verifying if they are skylines and if so if they are part of the ND set or not.

Algorithm 8 SVE1

```
    Input: S: dataset, C: constraints, W: vertices of W(C), F: family of MLW functions.
    Output: nd: ND Set
1:  S ← sort(S, W)
2:  nd ← [ ]
3:  for s in S do
4:      for t in nd do
5:          if t ≺ s then continue to line 3
6:      compute left-hand side of Inequalities
7:      for t in nd do
8:          if t ≺_F s then continue to line 3
9:      nd ← nd ∪ {s}
10: return nd
```

Algorithm 9 SVE1F

```
    Input: S: dataset, C: constraints, W: vertices of W(C), F: family of MLW functions.
    Output: nd: ND Set
1:  S ← sort(S, W)
2:  nd ← [ ]
3:  for s in S do
4:      compute left-hand side of Inequalities
5:      for t in nd do
6:          if t ≺ s ∨ t ≺_F s then continue to line 3
7:      nd ← nd ∪ {s}
8:  return nd
```

The difference between the two algorithms lies in the dominance check, SVE1 first checks if the point is dominated by the ND set using simple skyline dominance and then does the same with F-dominance. SVE1F on the other hand, combines both

dominance tests by doing a single scan of the ND set unlike SVE1 which in the worst case will have to do it 2 times. The advantage of SVE1F is that if F-dominance has more effect in making selection than simple dominance, we could decrease the number of dominance tests a lot compared to SVE1.

3.2.2. PO computation

For the computation of PO [3], as for ND, several strategies can be implemented, such as starting from the original input dataset or given property 3 for which $PO(s;F) = PO(ND(s;F);F)$, starting from the ND set. Since checking for optimality is very time consuming, we will see in this thesis two algorithms that both start from the ND set that can be computed with any algorithm seen so far. The difference between the two algorithms is in the Optimality test: the first algorithm use the Primal PO Test (*Theorem 3),* the second instead, the Dual PO Test (*Theorem 4).*

In both algorithms, a heuristic optimization technique can be used to try to reduce the number of optimality tests, which will attempt to discard non-optimal tuples using an incremental strategy. This strategy attempts to limit the number of σ tuples with which to test whether a tuple t is potentially optimal or not. The test is done by solving an LP problem that takes into account the marginal scores of all the σ + 1 tuples in the set, but when σ is large this procedure may be too costly in terms of time. And here the heuristic optimization technique comes into play: we solve an LP problem using only a subset of σ tuples: if the solution to *Theorem 3 or 4* tells us that t is not potentially optimal, then we can safely discard t. This allows us to discard some tuples that are not potentially optimal with small LP problems, so that the check with all σ tuples is performed only for those tuples that cannot be discarded with smaller LP problems.

Algorithm 10 PO non-incremental(Primal and Dual)

```
      Input: nd: nd set, C: constraints, W: vertices of W(C), F: family of MLW functions
      Output: po: PO set
1:    po ← nd
2:    for t in po in reverse order do
3:        if isNonPO(t, po \ {t}) then po ← po \ {t}
4:    return po
```

Algorithm 11 PO (Primal and Dual)

```
      Input: nd: nd set, C: constraints, W: vertices of W(C), F: family of MLW functions
      Output: po: PO set
1:    po ← nd
2:    σ ← 2; lastRound ← false
3:    while not lastRound do
4:        if σ ≥ |po| - 1 then lastRound ← true
5:        for t in po in reverse order do
6:            T ← first min(σ, |po| - 1) tuples in po \ {t}
7:            if isNonPO(t, T) then po ← po \ {t}
8:        σ ← σ * 2
9:    return po
```

In the non-incremental method we perform the Optimality Test of each tuple against all the tuples in po except t. In the incremental method, instead, we initialize $\sigma = 2$, which will give us small LP problems initially, this condition is necessary for pruning but not sufficient, so if the non-optimality check of a tuple gives a negative answer it does not necessarily mean that tuple is potentially optimal. For each t in PO we check if it is not potentially optimal compared to the first σ tuples in PO, since according to the order they are the best. In the last round where all remaining tuples are considered, they are compared with all the other tuples still in PO, which now becomes a necessary and sufficient condition for pruning. To check whether a tuple is not potentially optimal in line 7, we can use either the Primal PO Test as in *Theorem 3* or the Dual PO Test as in *Theorem 4*.

4 Parallel Algorithms

This chapter presents algorithms that enable us to parallelize the computation of the Skyline and Flexible skyline operator, resulting in decreased computation time. These algorithms operate by decomposing the dataset into multiple partitions, where each partition is responsible to compute a skyline without having to consider all the points in the dataset but only its assigned points, the resulting set of points is known as the local set of a specific partition. Subsequently, a sequential phase is performed, where the final skyline set, called global set since it collects all skyline points in the entire dataset, is determined by merging the previously found local sets.

Algorithm 12 Parallel Algorithm

```
    Input: S: dataset, n: number of partitions
    Output: GS: global set
1:  LS ← [ ], P← [ ]
2:  P ← PartitionAlgorithm(S, n)
3:  for p in P do
4:      LS ← LS ∪ ComputeLocalSet(p)
5:  GS ← SequentialPhase(LS)
6:  return GS
```

Let us take the Skyline computation as an example, but the same applies to the Flexible Skyline operators: we take a dataset S and we decompose it into n partitions S_0, S_1, ..., S_n. We will have that $SKY(S) = SKY(SKY(S_0) \cup SKY(S_1) \cup ... \cup SKY(S_n))$. The parallel algorithm is divided into two phases, the first in which each partition i is assigned to a thread that computes its local skyline set SKY(S_i), and the second which involves merging these local skyline sets and computing their global set SKY(S).

By applying specific partitioning to the initial dataset, we aim to reduce the time it takes to complete the algorithm. The effectiveness of the partitioning directly impacts the workload, with smaller local sets being preferred to decrease the workload in the final sequential part, which is the slowest.

In this chapter we are going to introduce 4 types of partitioning taken from the literature with the goal of dividing the workload and improving the algorithm's

efficiency. Our task in will be to present them and explain how the algorithms assign points to a given partition.

Our final conclusions will be that all algorithms speed up execution time compared to the centralized version because they are able to parallelize work on several partitions simultaneously, but as we will see in chapter 7, some types of partitioning are more effective than others depending on how they partition the space.

From now on, we will see these algorithms applied to 3 different types of datasets, independent, correlated and anticorrelated. The difference between these is that an independent dataset does not exhibit any proportional relationship between dimensions, while the other two do. An anticorrelated dataset is characterized by points in one dimension that have an inverse proportionality relationship with all the points in another dimension. In contrast, a correlated dataset is one where the points are directly proportional to one dimension.

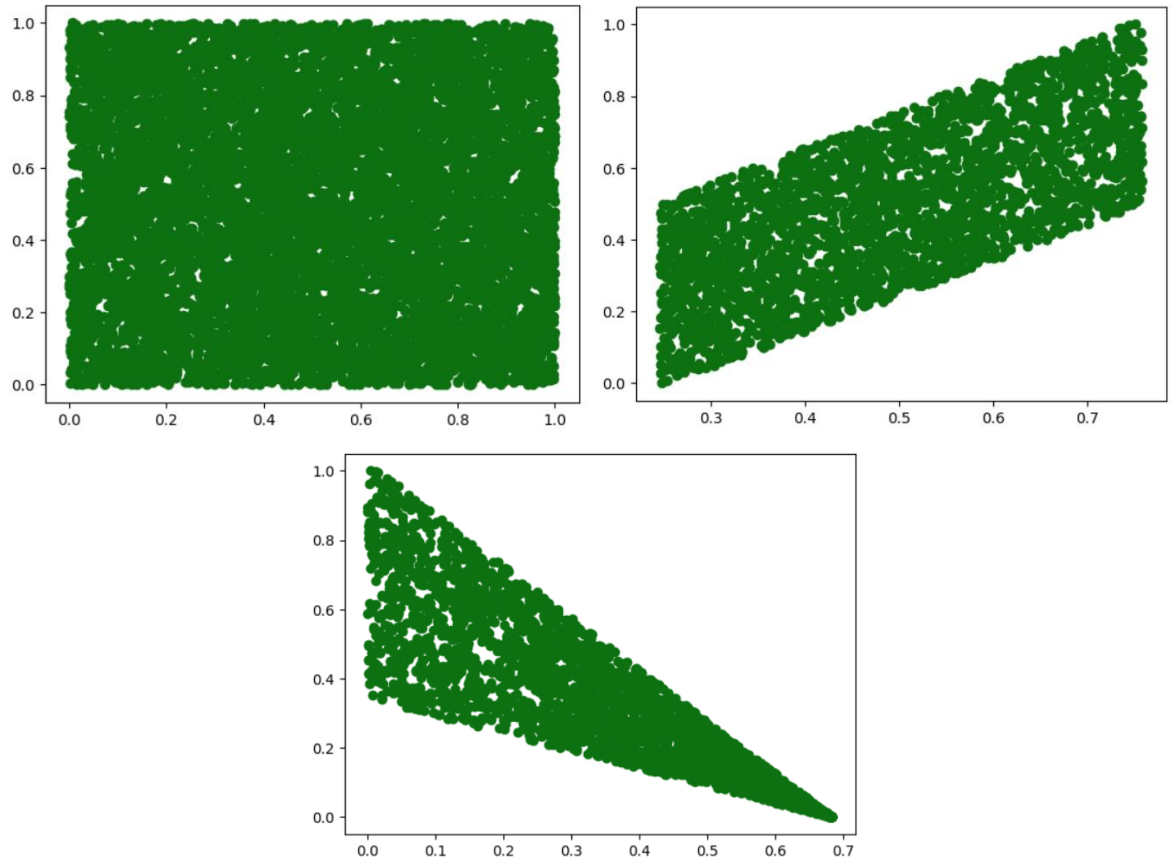


Figure 4.1: The upper-left image displays an independent dataset, while the upper-right a correlated one. In the bottom we have an anticorrelated dataset. All are 2d datasets.

4.1. Random Partitioning

The initial partitioning method we will examine is *Random Partitioning*. This technique was initially proposed in [9] with the goal of ensuring that each set S_i represents a sample of S that has similar structural characteristics. To achieve this, points are randomly distributed among the various partitions, with the number of points evenly allocated to the n partitions.

This partitioning method's key benefit is its simplicity since the random assignment of points eliminates the need for pre-processing the data, as is required by other techniques such as index computation or sorting. On the other hand, this method's drawback is that it produces a comparatively large local skyline set in comparison to other partitioning techniques. As a result, the sequential phase will require more effort, and the algorithm's overall runtime will be impacted.

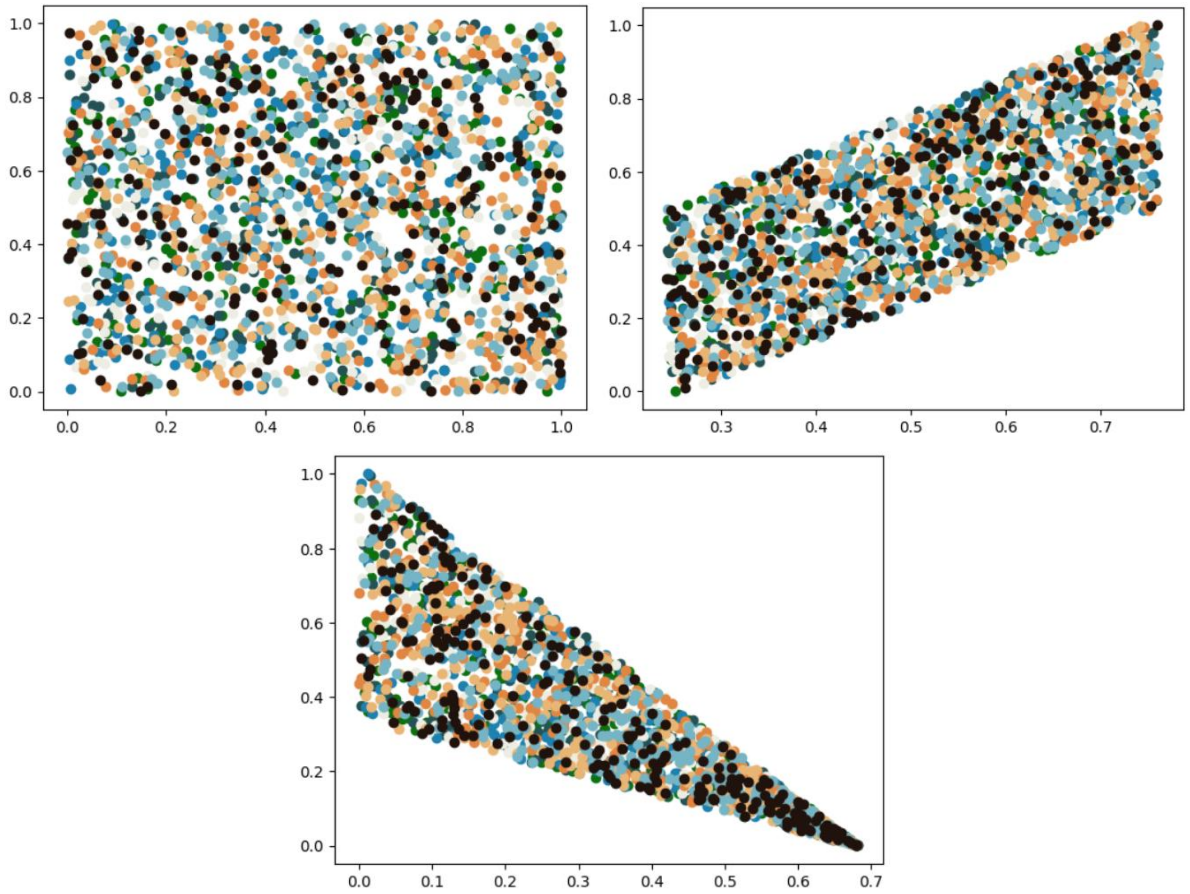


Figure 4.2: Application of Random Partitioning to three distinct datasets, each containing 2000 points and 8 partitions where a different colour corresponds to a different partition. The upper-left image displays an independent dataset, while the upper-right a correlated one. In the bottom we have an anticorrelated dataset.

4.2. Grid Space Partitioning

The next partitioning technique we will examine is *Grid Partitioning*, which was first introduced in [10]. This method entails dividing the space into an nxn grid to enable parallel computation on different partitions. In this approach, each dimension is divided into n parts, resulting in a total of n^d partitions, where d denotes the total number of dimensions. The simplest approach is to compute the local sets for all n^d partitions, combine them and sequentially compute the global set.

Despite this, this partitioning method is efficient because it enables us to eliminate partitions that cannot contain points that belong to the global set. To accomplish this, we employ a principle that is comparable to dominance between tuples, although it involves a dominance relationship between partitions, with the corner points serving as the basis for comparison. We use p.max and p.min to respectively represent the maximum and minimum corners of a given partition p.

A partition's maximum corner p.max is the corner that has the highest (worst) values on all dimensions, while its minimum corner p.min is the corner that has the lowest (best) values on all dimensions.

Therefore, we can group every tuple between the best and worst representative point together, enabling us to compute the local skyline of each square in the grid. An index is computed for each specific point, and then all points with the same index are grouped together within a specific partition:

$$index(x_i) = \sum_{i=1}^{d} \lfloor x_i * N \rfloor * N^{i-1}$$

where x_i is the point for which we are computing the index and N is the number of partitions per dimension.

Example: Let us consider the following independent dataset of 40 points in two dimensions, in which we want to apply grid partitioning with N=4 partitions per dimension. We will then have a total of 4^2 partitions (grids).

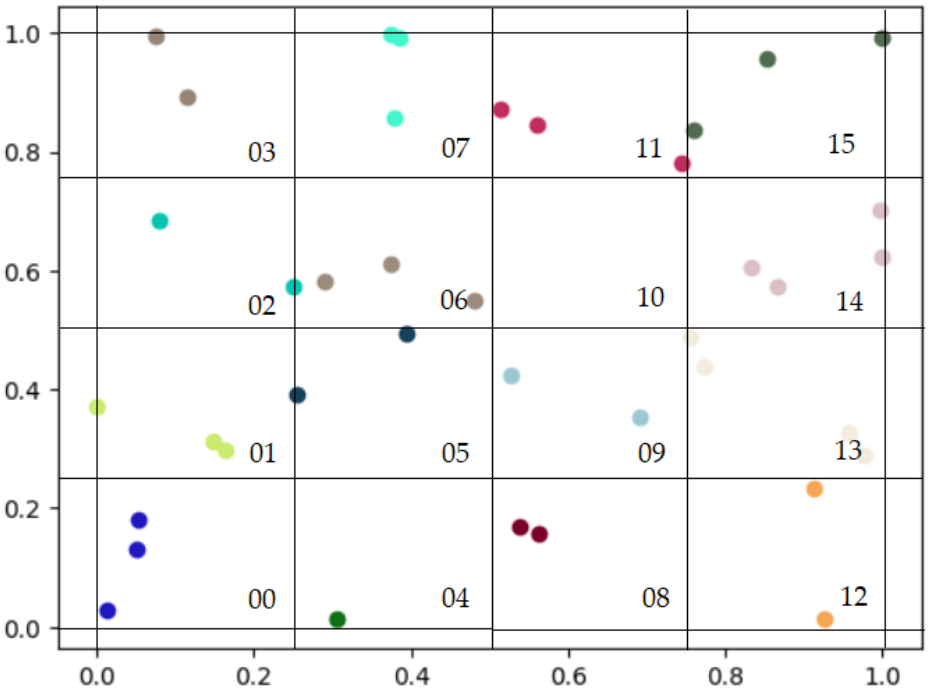


Figure 4.3: An independent dataset with grid partition where a different colour corresponds to a different partition.

This figure show us how grid partitioning works and how the points are distributed between the partitions.

Using this type of partitioning, we will have many points in the local set that will be dominated later in the computation of the global set, and this leads us to lose a lot of time. In the next chapter, we'll see how to use grid filtering to prune certain partitions before the parallel phase, thereby saving some time.

With Grid Partitioning, we have no control on how many points the different partitions will have, and we have limited control over the number of partitions since they will depend on the number of dimensions and the number of slices per dimension assigned to them, unlike Random Partitioning which manages to divide the work equally between the various partitions and you can select as many partitions as you want. However, it does manage to group the points within partitions better, managing to eliminate many points in the parallel phase compared to random partitioning, thus greatly reducing the local set.

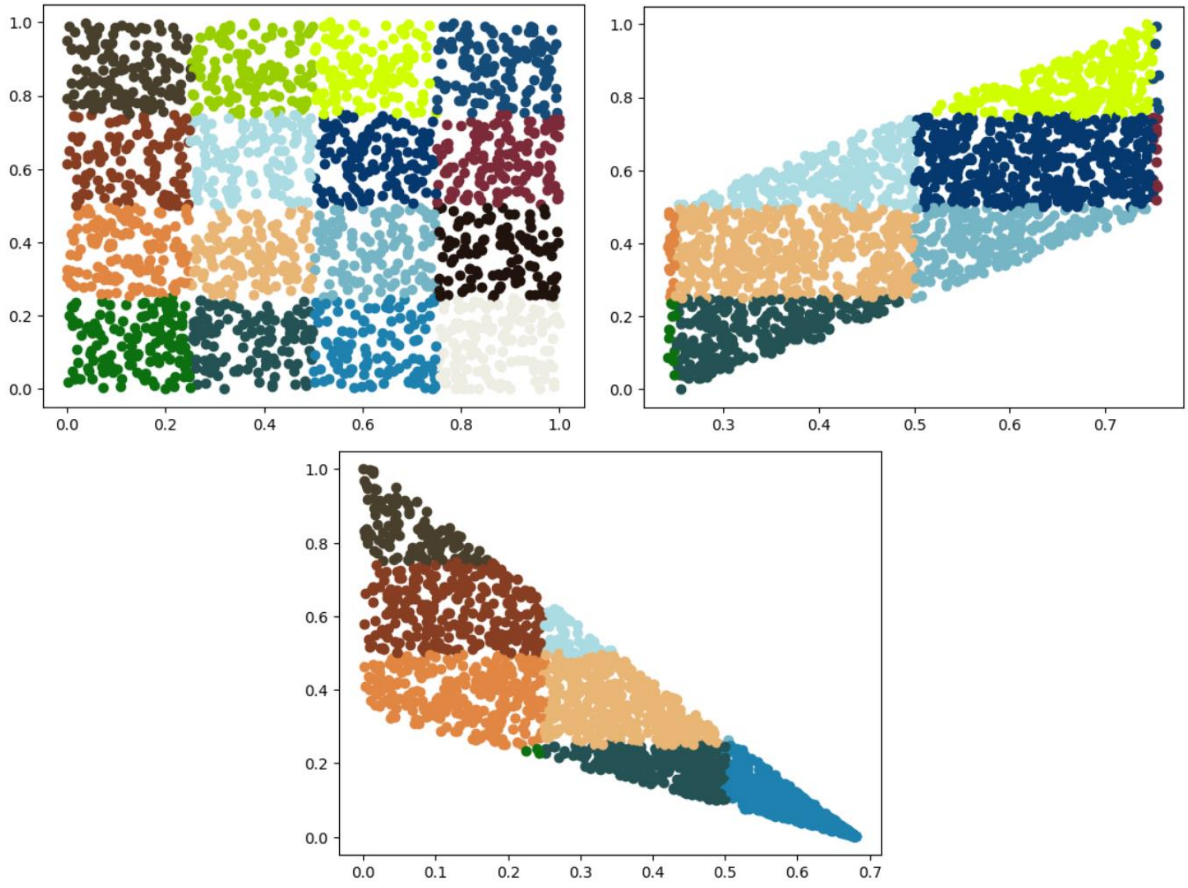


Figure 4.4: An example of how data are partitioned using Grid Partitioning in a 2d independent (upper-left), correlated(upper-right), anticorrelated (bottom) datasets with 4^2 partitions where a different colour corresponds to a different partition.

As we can see from Figure 4.4, Grid Partitioning succeeds in dividing the dataset equally in the case of independent datasets, but in the case of anticorrelated datasets, which will be the ones to which we will give our attention in the experimental results, it divides the load into some partitions, leaving some of them empty or almost empty. This as we shall see in Chapter 7 will be a disadvantage because the parallel phase will last longer than the others, since some partitions will be heavier.

4.3. Angle-based Space Partitioning

Introduced in [11], the *Angle-based Partitioning* technique aims to address the issue of local sets containing an excessive number of successively dominated points and the problem of load sharing between partitions, which was observed in previous partitioning techniques. The technique involves mapping the Cartesian coordinate space to a hyperspherical space and partitioning the resulting data space into N partitions based on the angular coordinates.

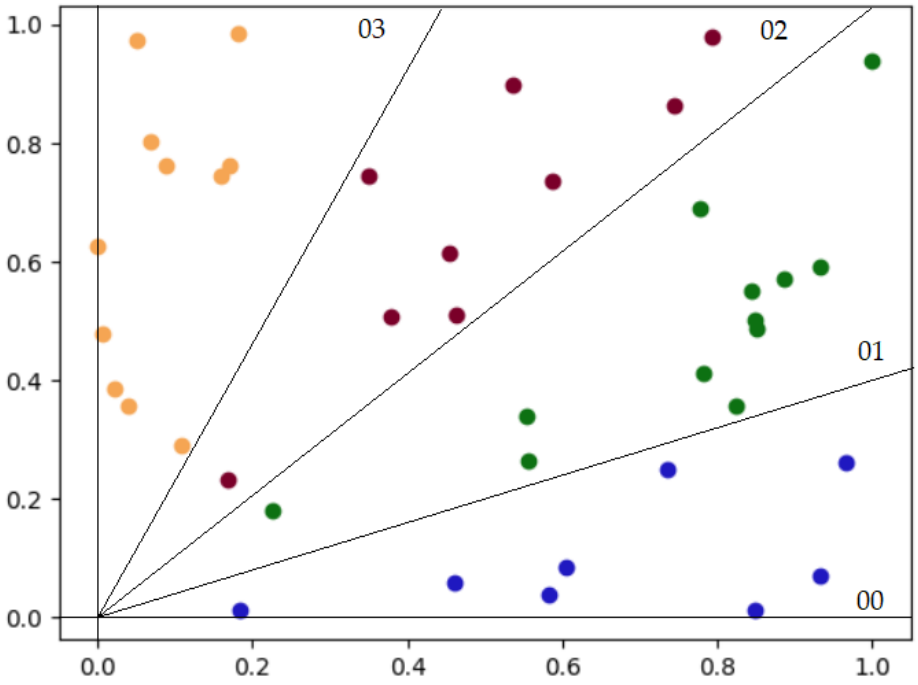


Figure 4.5: An example of angular partitioning on a independent 2d dataset with 4 partitions where a different colour corresponds to a different partition.

The angle-based approach can achieve a better balance of workload, as each partition includes both good and bad points in the data space. This results in a larger number of points being dominated in the local set computation phase compared to grid partitioning and Random Partitioning, leading to a smaller set of local sets and reducing the workload for the sequential phase. As a result, the execution time is reduced. Unlike grid partitioning, angular-based partitioning does not allow for pruning. However, in the next chapter, we will explore methods for performing filtering before partitioning in order to address this limitation.

Like the grid technique, every point in the dataset needs to be assigned an index and allocated to a corresponding partition based on that index. The index for each point is computed by first transforming its Cartesian coordinates into Hyperspherical Coordinates, which includes a radial coordinate r and d-1 angular coordinates φ_1, φ_2, …, φ_{d-1}. This transformation can be accomplished using set of equations:

$$r = \sqrt{x_n^2 + x_{n-1}^2 + \cdots + x_1^2}$$

$$\tan(\varphi_1) = \frac{\sqrt{x_n^2 + x_{n-1}^2 + \cdots + x_2^2}}{x_1}$$

……..

$$\tan(\varphi_{d-2}) = \frac{\sqrt{x_n^2 + x_{n-1}^2}}{x_{n-2}}$$

$$\tan(\varphi_{d-1}) = \frac{x_n}{x_{n-1}}$$

Then , we can compute the index of each point in the dataset, for a given number of partitions N, using this formula:

$$index(x) = \sum_{i=1}^{d-1} \left\lfloor \varphi_i \times N^i \times \frac{2}{\pi} \right\rfloor$$

The difficulty with this type of partitioning lies in finding the best number of partitions so as not to have too much parallel work and at the same time to have partitions as large as possible so as to be able to obtain local sets as small as possible. It suffers from the same problem as Grid Partitioning where we are limited in choosing the number of partitions because it will depend on the number of sizes and the number of slices per size we choose.

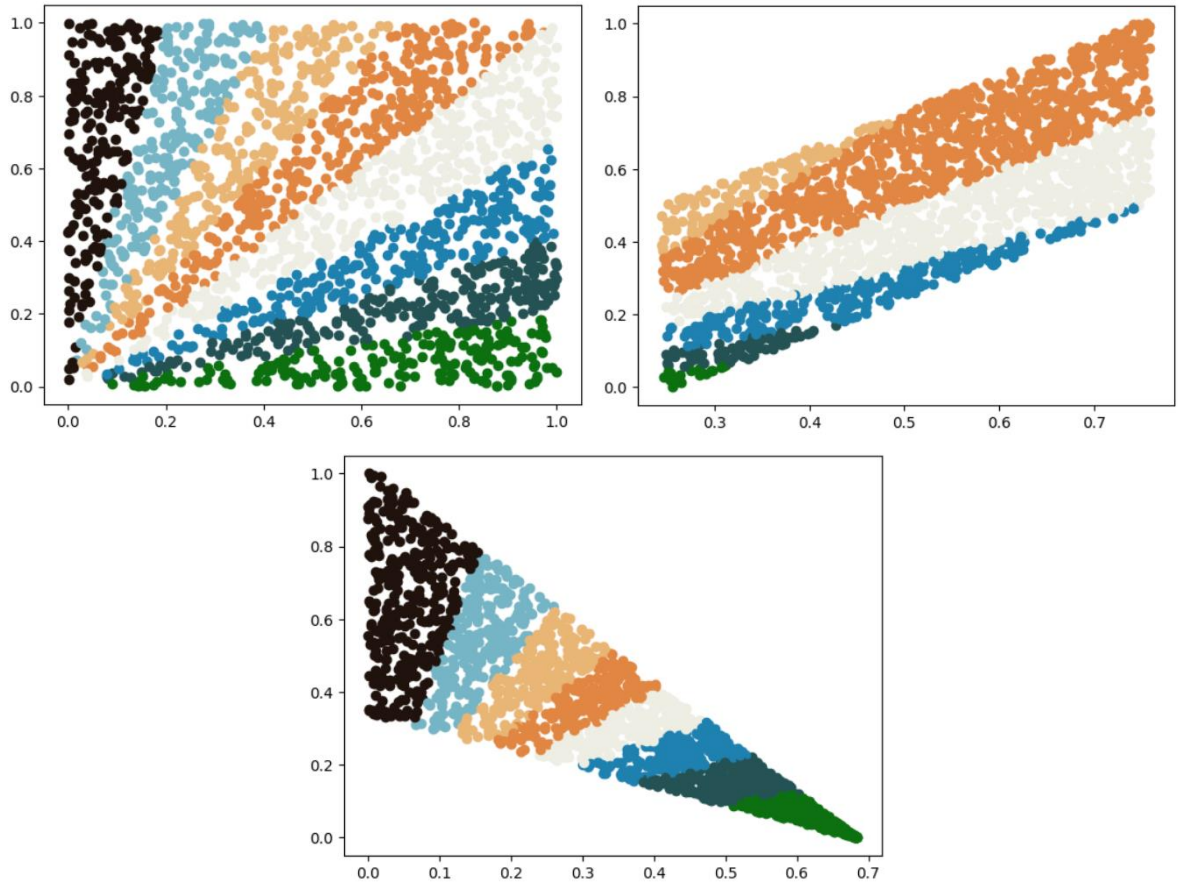


Figure 4.6: An example of how data are partitioned using Angular Partitioning in a 2d independent (upper-left), correlated(upper-right), anticorrelated (bottom) datasets with 8 partitions where a different colour corresponds to a different partition.

As we can see from the figure, Angular Partitioning manages the division of the workload better than Grid Partitioning, being able to divide the load between all partitions more or less equally, but never as well as Random Partitioning, even for anticorrelated datasets without leaving empty partitions.

4.4. One-dimensional Slicing (Sliced Partitioning)

The last partitioning method we will see is called *Sliced Partitioning*. It is introduced in [12] and the basic idea is to overcome one of the limitations of grid and angular partitioning methods which is that we have no control over how many partitions we can use and how many points there will be in each partition. The idea of this partitioning algorithm is to sort the dataset on one dimension and divide it between the partitions equally, so that each partition will have the same number of points and we decide how many partitions to have.

We must keep in mind, however, that sorting on one dimension is not a topological sort and that therefore the property for which a tuple that comes after another cannot dominate the one that comes before is not verified in this case. This method does indeed return false positives in the case of algorithms that use sorting to terminate execution first such as SFS. These false positives, however, will be dominated in the following sequential phase during the computation of the global set.

Algorithm 13 Sliced Partitioning

```
      Input: S: dataset, n: number of partitions
      Output: LS: Local Set
1:    Sorted_dataset ← sort(S[0])     //sort on the first dimension
2:    P ← [ ] , LS ← [ ]
3:    section ← S.length / N
4:    for id in [0, N-1] do
5:        P.add (id)
6:    for id in P do
7:        data ← Sorted_dataset[id ∗ section, (id + 1) ∗ section]
8:        LS ← LS ∪ ComputeLocalSet(data)
9:    return LS
```

The functioning of the algorithm 13 is very simple, first we sort the dataset on the basis of the first dimension in ascending order, and after that we assign each partition a unique id, which will be assigned a number of points from the sorted dataset within a range. Each partition will have points ranging from id * section to (id+1) * section. After that the local set for each partition is computed and finally the set of these local sets is returned.

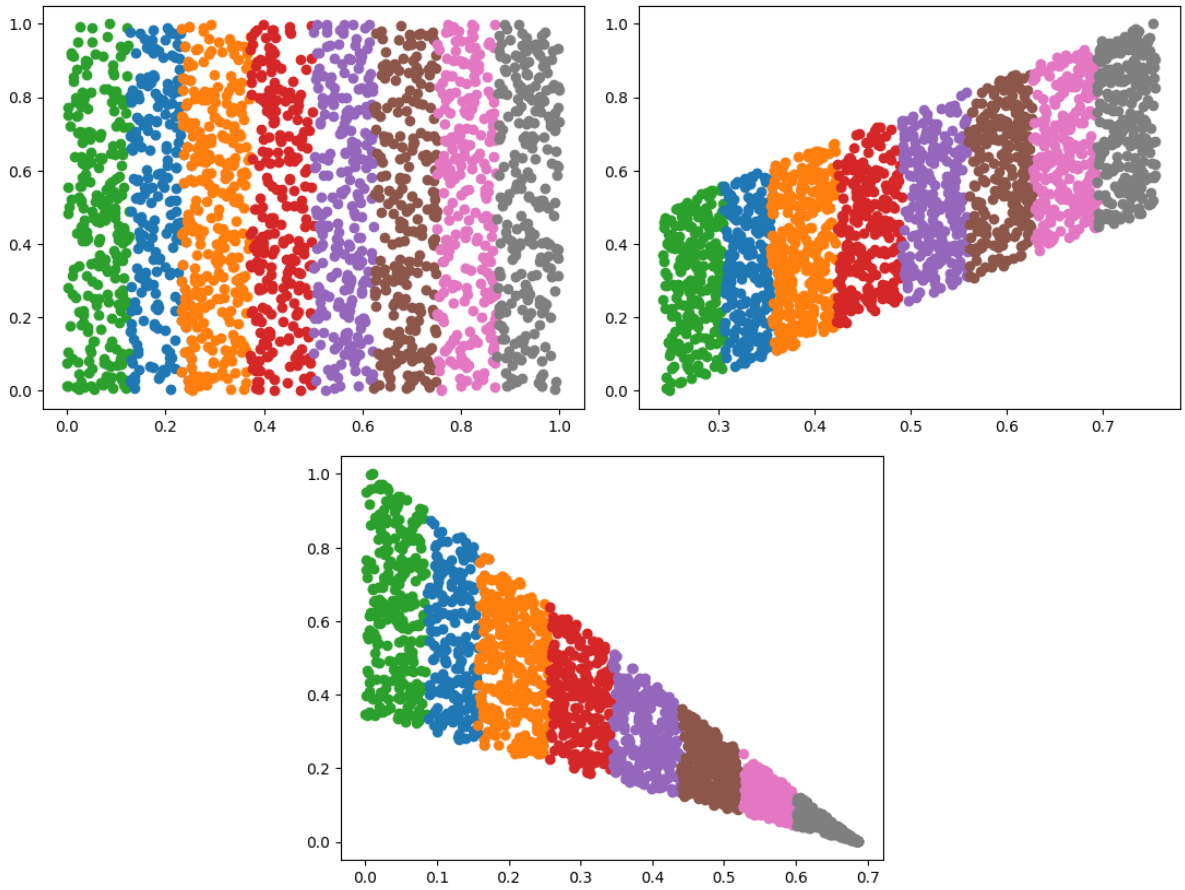


Figure 4.7: An example of how data are partitioned using Sliced Partitioning in a 2d independent (upper-left), correlated(upper-right), anticorrelated (bottom) datasets with 8 partitions where a different colour corresponds to a different partition.

With this type of partitioning we manage to divide the workload equally among all the partitions, and as we will see in chapter 7 it will be the one that will be able to eliminate more points in the parallel phase by returning a smaller local set.

5 Improvement of Parallel Algorithms

This chapter introduces techniques to improve parallel algorithms by reducing their execution time. As we have seen in the previous chapter, the type of partitioning we apply is very important in affecting the execution time, because it will be important in putting together within the same partition as many dominatable points as possible with points of the global set, so as to have at the end of the parallel phase a local set as small as possible, since the sequential part will be the one that takes the longest.

The first technique called *Grid Filtering* introduced in [11] for the Skyline computation involves eliminating as many partitions as possible using the Grid Partitioning before computing the local sets, but as we shall see in the case of anticorrelated datasets it does not work well because it eliminates few points.

The second technique called *Representative Filtering* introduced in [12] for the skyline computation set, involves selecting a few points that will act as representatives (better points) and will be used to remove as many points as possible. We will see several techniques for choosing these representatives, the one proposed in this thesis is to take the first n sorted points of each partition after the angular partitioning technique. As we shall see, however, for independent and correlated datasets, the proposed method is better than the one proposed in [12], while with anticorrelated datasets, the latter succeeds in filtering more, so that we prefer it since we are going to use anticorrelated datasets for our experiments.

The last technique we will see called *All Parallel* is used to avoid performing the sequential part, by performing two parallel phases, passing in the last one to each partition the union of the local sets found during the first parallel phase. We will see how this technique has a positive impact on the total duration of the algorithm, since the part that takes the longest is the final sequential phase in parallel algorithms. Our original contribution here will be to create an algorithm without a sequential part that is good for the computation of the two Flexible Skyline operators ND and PO.

These techniques are applied for both the computation of skylines and the computation of Flexible Skylines operators. However, the filtering technique is only applicable for computing the Skyline and ND, while for PO, since we start with the ND set, we can use these techniques for the initial computation of ND but not for the computation of PO. Regarding the All Parallel technique, it can also be applied to PO.

5.1. Grid Filtering

This technique introduced in [11] addresses the issue of simple Grid Partitioning, where the number of local sets can become unmanageable. When employing the standard grid partitioning approach, local sets are computed for all partitions, including those containing points that will surely be dominated in the next phase. To overcome this challenge, the grid filtering technique is implemented, which removes points in dominated partitions before starting the parallel phase.

The technique operates by first determining the best and worst points of each partition and then assigning all the points belonging to each partition. The next step involves comparing two partitions, and if one partition dominates another, all the points within the dominated partition can be safely excluded from the global set. We use p.max and p.min to respectively represent the maximum and minimum corners of a given partition p.

A partition's maximum corner p.max is the corner that has the highest (worst) values on all dimensions, while its minimum corner p.min is the corner that has the lowest (best) values on all dimensions.

Definition 5.1: (Partition dominance) A partition p_i dominates partition p_j ($p_i \prec p_j$) only when p_i.max dominates p_j.min. This means that $p_i \prec p_j$ is equivalent to $p_i.max \prec p_j.min$.

Thanks to Definition 5.1, we have that if $p_i \prec p_j$ for every t in p_i and u in p_j, $t \prec u$.

As result of this dominance, we can carry out a filtering step that involves eliminating all points within dominated partitions, so as to decrease the number of points in the initial dataset efficiently. Once the value N representing the number of partitions per size has been set, we will have a total of N^d partitions (grids). Each partition p will thus have a worst point p.max and a best point p.min equal to:

$$p_i.max = \frac{\left\lfloor \frac{1}{N^{i-1}} \right\rfloor mod\ N}{N} + \frac{1}{N}$$

$$p_i.min = \frac{\left\lfloor \frac{1}{N^{i-1}} \right\rfloor mod\ N}{N}$$

We can observe that the best and worst tuple's difference is an identity vector multiplied by the constant value 1/N.

By executing this step, as many dominated points as possible are eliminated before proceeding with partitioning and running the parallel algorithm.

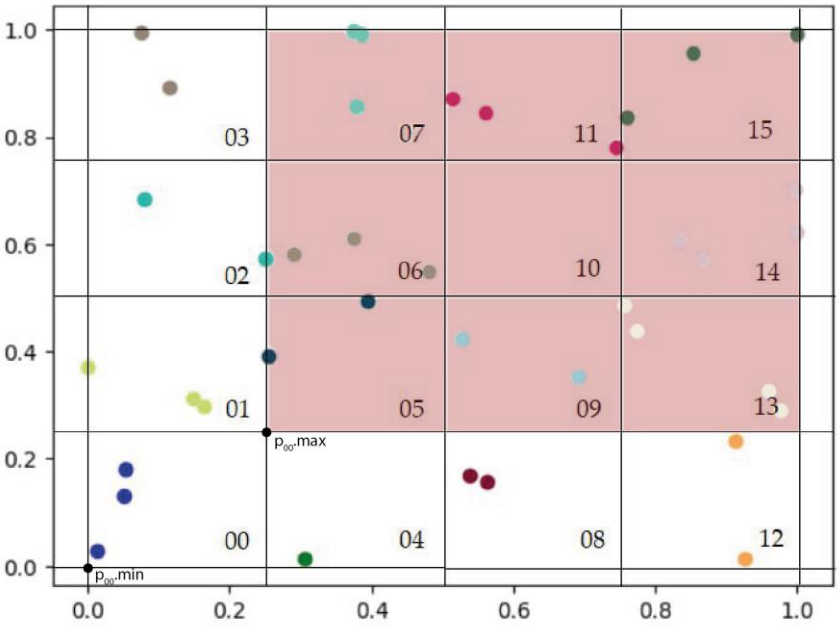


Figure 5.1: Example of partition dominance in an independent dataset using grid filtering.

As depicted in the figure above, using grid partitioning, we can remove points that belong to dominated partitions. The red-colored partitions located in the upper right-hand corner are dominated, with partition 00 dominating all the red partitions. Similarly, partition 04 dominates 09 , 10, 11, 13, 14 and 15 and so on.

Algorithm 14 Grid Filtering

```
     Input: S: dataset, n: number of partitions per dimension
     Output: T: filtered dataset
1:   T ← [ ], ContainerList ← [ ]
2:   d ← len(S[0])                    //number of dimensions
3:   for I in [0, n^d] do
4:       for j in [0, d] do
5:           worst[j] ← (⌊i/n^j⌋ mod n)/n + 1/n
6:           best[j] ← (⌊i/n^j⌋ mod n)/n
7:       ContainerList[i] ← (worst, best, [ ])
8:   for s in S do
9:       index ← ∑_{i=0}^{d-1} ⌊s[i] * n⌋ * n^i
10:      ContainerList[index] ← ContainerList[index] ∪ s
11:  ContainerList.sort(min(best))
12:  for c in ContainerList do
13:      for t in T do
14:          if t.worst ≺ c.best then continue to line 11
15:      T ← T ∪ c
16:  return T.lists
```

This algorithm 14 involves an initial step where the best and worst points are computed for each container (partition), which is then initialized as empty. Next, an index is computed for each point in the dataset and is added to the container with that index. After that, the containers are sorted based on the value of the best tuple of each container. Finally, we sequentially scan the containers and compare them with the non-dominated set T, which is initially empty. During each comparison, we verify if the container being scanned is dominated by any of those in set T. If not, it is added to set T. Once finished, we return the list of points of non-dominated containers.

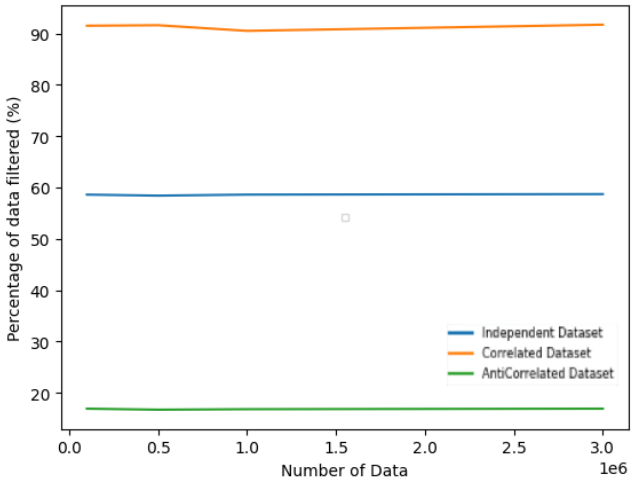


Figure 5.2: Percentage of filtered data using different types of 4d datasets and 8^4 partitions.

As depicted in Figure 5.2, the grid filtering approach exhibits distinct behaviour based on the dataset type. Regarding the independent datasets, 58% of the points being filtered for various dataset sizes, indicating effective filtering. On the other hand, for correlated datasets, the filtering performance is superior, with over 90% of the points being filtered. In contrast, the grid filtering approach does not offer much advantage for anticorrelated datasets, where only 16% of the points are filtered, resulting in a time loss. In the following section, we will explore a filtering method that works effectively for all three dataset types.

5.2. Representative Filtering

Another method of filtering could be to choose a few "best" points and pass them to each partition to try to delete as many dominated points as possible in a short time. The choice of these representative points can be done in several ways, and the effectiveness of the algorithm depends precisely on the choice of these points, which the better they are the more points they will dominate, thus drastically decreasing the size of the input dataset.

There are various methods to select representative points, one of which is to randomly choose them. However, this approach is unreliable as it may include weak points that are not part of the global set. A more effective method would be to select the first points in the sorted dataset. This approach is promising as we know that, by virtue of the sorting property, a tuple that comes later in the sorted dataset cannot dominate the tuple that comes earlier. Especially in the case of computing the ND set, we could use as sort function a weighted sum in which the weights are the coordinates of the centroid of the polytope *W(C)*.

In this thesis, we will present an algorithm that exploits angular partitioning to search for the best points to make as representatives. Its operation is simple, it simply partitions the points using angle partitioning and for each partition returns the best points based on a sorting function.

Another method would be to take points that have a larger dominance region, as seen in [12]. In [12] the objective was to apply filtering before computing the skyline set, our objective in this thesis will be to apply the same type of filtering but adapting it to the computation of ND. To compute the dominance region of a point we need to compute the area between the point and the limit point, which in the case of normalized datasets between the interval [0,1] is equal to $[1]^d$. This filtering technique therefore only works if we have a limit to the maximum value a tuple can take, and thus the dataset must first be normalized to the interval [0,1]. There are other methods for computing representatives, again in [12] there is a method that takes all points within a radius with the origin as centre.

Algorithm 15 Representative Filtering

```
    Input: S: dataset, r: number of representatives, f: sorting function, n: number
    of partitions
    Output: T: filtered dataset
1:  T ← [ ], representatives ← [ ], P ← [ ]
2:  representatives ← getRepresentatives(S)
3:  P ← PartitioningAlgorithm(S,n)          //partitioning the dataset
4:  for p in P do
5:      for data in p do
6:          for rep in representatives do
7:              if rep dominates data then continue to line 5
8:      T ← T ∪ data
9:  return T
```

Now we present a method to choose the representatives taking advantage of angular partitioning that partitions in the best possible way, and our task will be to take the first r points of each partition, since they will be the 'best' according to a sorting function.

Algorithm 16 Representative Filtering SKY(Sorted method)

```
    Input: S: dataset, r: number of representatives, f: sorting function, n: number
    of partitions
    Output: T: filtered dataset
1:  T ← [ ], representatives ← [ ], P1 ← [ ], P2 ← [ ]
2:  P1 ← AngularPartitioning(S,n)
3:  for p in P1 do
4:      p.sort(f)
5:      representatives ← first r tuples in p
6:  representatives ← SFS(representatives)   //discard dominated t
7:  P ← PartitioningAlgorithm(S,n)          //partitioning the dataset
8:  for p in P do
9:      for data in p do
10:         for rep in representatives do
11:             if rep ≺ data then continue to line 9
12:     T ← T ∪ data
13: return T
```

This, instead, is the dominance Region method seen in [12] that selects the best points according to the area they cover.

Algorithm 17 Representative Filtering SKY (Dominance Region method)

```
    Input: S: sorted dataset, r: number of representatives, d: dimensions
    Output: T: filtered dataset
1:  T ← [ ], representatives ← [ ], P ← [ ]
2:  P ← Partitioning(S,d)                    // part. the dataset in d partitions
3:  for index in [0, len(P)] do
4:      R ← [ ]
5:      for data in S do
6:          area ←∏_{i=0}^{d-1}(1 − t[i])
7:      if area > R[⌊t[index] ∗ r⌋].area then R[⌊t[index] ∗ r⌋] ← (area, data)
8:      for rep in R do
9:          representatives ← representatives ∪ rep.data
10: representatives ← SFS(representatives)    //discard dominated t
11: P ← PartitioningAlgorithm(S, n)           //partitioning the dataset
12: for p in P do
13:     for data in p do
14:         for rep in representatives do
15:             if rep ≺ data then continue to line 13
16:     T ← T ∪ data
17: return T
```

The algorithms 16 and 17 exhibit differences in how they choose representative points. In the first algorithm (16),after we have partitioned using angular partitioning, we take the best points from each partition by simply taking the first r points in the set of points sorted. The partitions are sorted using a sorting function, which may be based on the Euclidean distance from the origin or the weighted sum of the points with the centroids of the polytope W(C) as weights. Next, the dominated points in the set of representatives are discarded, if any. After obtaining the representative points, the computation is split into multiple partitions by dividing the original dataset. Each partition is designed to contain both good and bad points, ensuring that each partition has similar computation time. Within each partition, a sequential scan of the points is conducted, and each point is compared to the set of representatives. If a point is not dominated by any representative, it is added to the set of non-dominated points.

The second algorithm (17) introduced by [12] has the same working principle, only the representatives are chosen according to their dominance region, the larger this is, the stronger the points are and thus the more representative. The computation of the dominance region is equal to the area between the points and the maximum point they

can reach. This is why it is important to normalize the data in the interval [0,1] so that the limit point equals $[1]^d$ otherwise this algorithm does not work.

The advantage of the first method is that the computation of the representatives is very simple and there is no need to normalize the dataset, unlike the second method which requires a step of computing the representatives and does not work with non-normalized datasets.

We will now see the behaviour of the two filtering algorithms in different types of datasets with respect to execution time and the percentage of filtered points aimed at computing the skyline set.

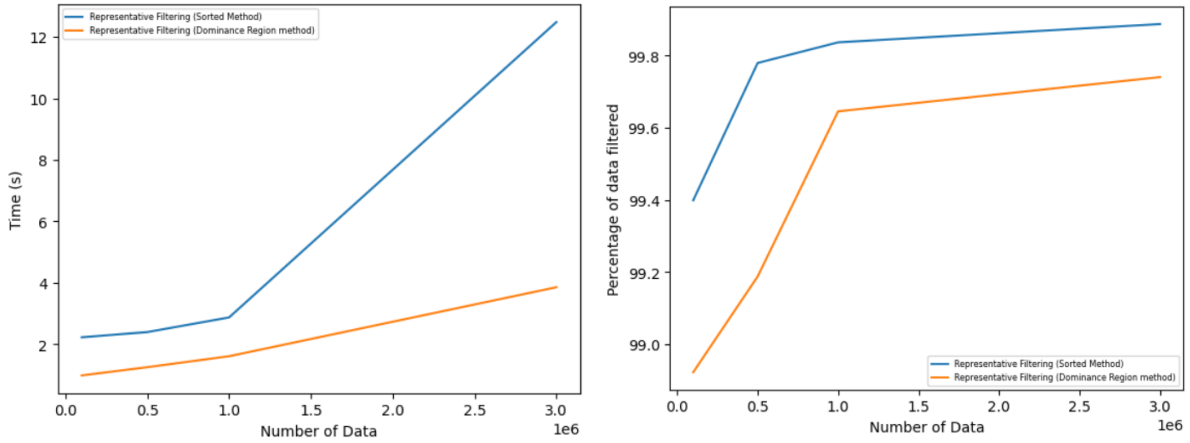


Figure 5.3: Comparison of the two algorithms in terms of time taken to filter the data and percentage of filtered data using 4d independent datasets of different sizes

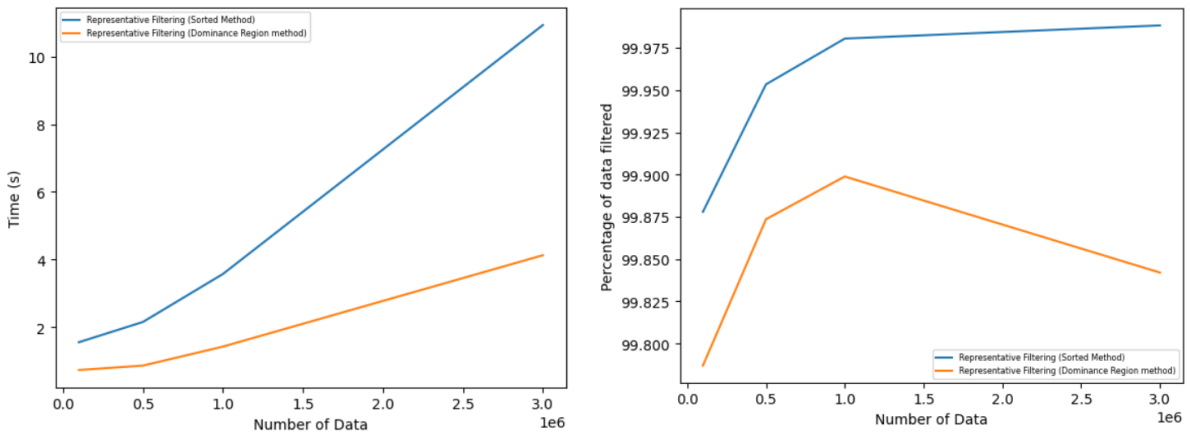


Figure 5.4: Comparison of the two algorithms in terms of time taken to filter the data and percentage of filtered data using 4d correlated datasets of different sizes

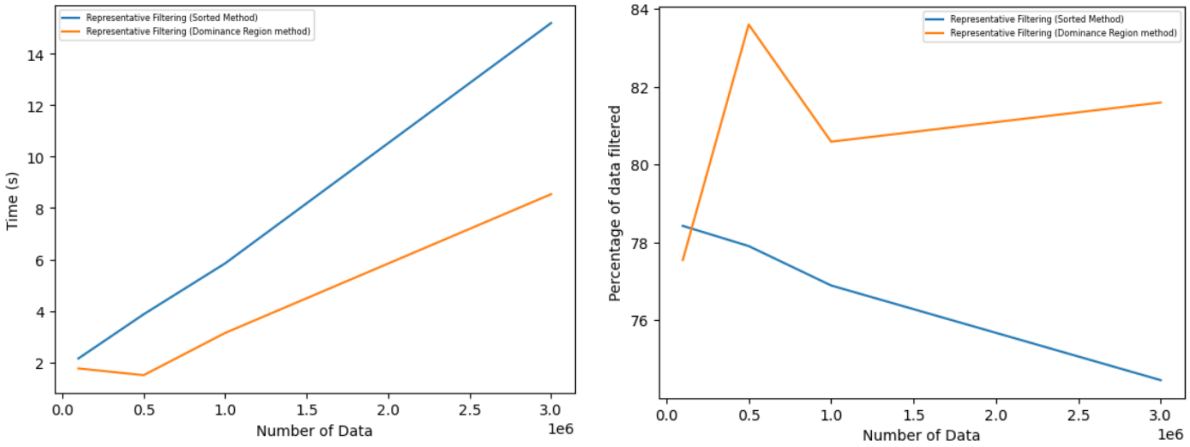


Figure 5.5: Comparison of the two algorithms in terms of time taken to filter the data and percentage of filtered data using 4d anticorrelated datasets of different sizes

Based on the characteristics of the dataset, the optimal approach for selecting representatives may vary. For independent or correlated datasets, the method that filters the most is the Sorted Method even if it takes a little longer. However, in the case of an anticorrelated dataset, the same method cannot be used since this is not the best measure of the goodness of a point, as we can see from Figure 5.5, and therefore the best method is to compute the dominance region for each point and select representatives according to the size of their respective dominance regions.

In all cases, the Sorted Method is a little slower in finding representatives due to the sorting function that takes some time, whereas with the dominance region method no sorting is needed.

Since our experiments will be carried out on anticorrelated datasets, representative filtering with the dominance region method will be used from now on.

So far we have seen these algorithms being applied for skyline computation and in particular using the SFS algorithm. This algorithm works for both Skyline and Flexible Skyline computations, in the sense that if points are filtered that are not Skyline for sure, then they can also be discarded because they are not ND since ND ⊆ SKY.

However, this type of filtering can be better adapted for the computation of ND using a modified version that we will introduce in this thesis. The difference lies in the number of filtered points, which with the second method will filter many more points than the first since it will also check F-dominance.

Algorithm 18 Representative Filtering ND (Dominance Region method)

```
    Input: S: sorted dataset, r: number of representatives, d: dimensions
    Output: T: filtered dataset
1:  T ← [ ], representatives ← [ ], P ← [ ]
2:  P ← Partitioning(S,d)                    // part. the dataset in d partitions
3:  for index in [0, len(P)] do
4:      R ← [ ]
5:      for data in S do
6:          area ←∏_{i=0}^{d-1}(1 − t[i])
7:      if area > R[⌊t[index] ∗ r⌋].area then R[⌊t[index] ∗ r⌋] ← (area, data)
8:      for rep in R do
9:          representatives ← representatives ∪ rep.data
10: representatives ← SVE1F(representatives)   //discard dominated t
11: P ← PartitioningAlgorithm (S, n)           //partitioning the dataset
12: for p in P do
13:     for data in p do
14:         for rep in representatives do
15:             if rep ≺ data ∨ rep ≺_F data then continue to line 13
16:     T ← T ∪ data
17: return T
```

The algorithms 17 and 18 have the same step of computing the representative points, which may be done by the sorted method or by the computation of the dominance region, in this case we have used the dominance Region method. The different lines of code between the two algorithms are highlighted in red in Algorithm 18. Unlike the filtering algorithm aimed at computing the skyline set, the one aimed at computing ND when checking between partition points and representative points, not only checks for simple dominance but also for F-dominance. If the point is neither dominated nor F-dominated then it is added to the set of points not dominated by the representatives.

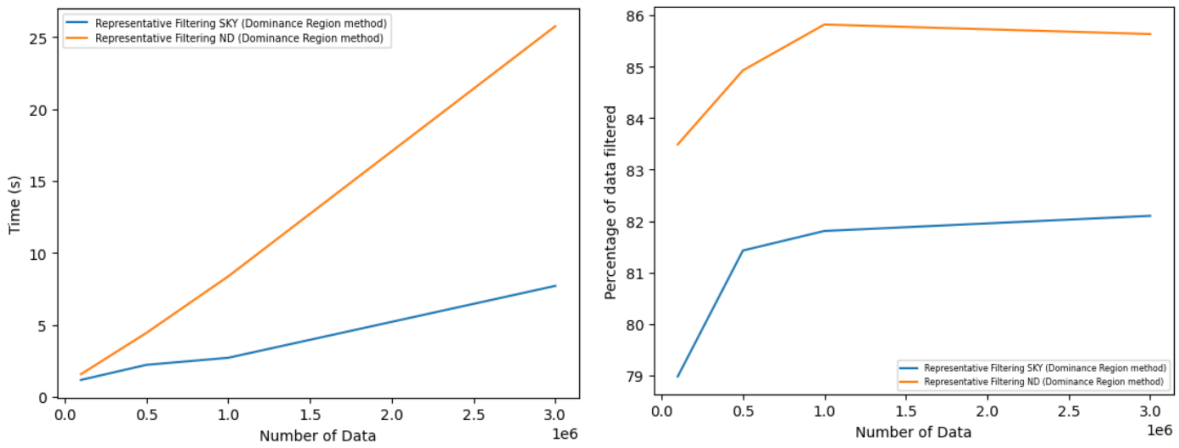


Figure 5.6: Comparison of the two algorithm aimed at computing SKY and ND in terms of time taken to filter the data and percentage of filtered data using 4d anticorrelated datasets of different sizes

In Figure 5.6, we compare two filtering methods: one based only on simple dominance, and the other based on both simple dominance and F-dominance. The latter approach filters a greater number of points but incurs slightly longer computation time. We will see that this extra time it takes to filter the dataset will be recovered by running it faster when we perform the final sequential phase that takes the longest to compute. Therefore, having fewer points in the local set, the final sequential part will take much less time.

5.3. All parallel Algorithm without sequential phase

So far, we have explored methods for speed up the parallel phase of local set computation and partitioning techniques that minimize the size of local sets to prevent overloading the final sequential phase, which is typically the most time-consuming aspect of the execution. Our objective now is to present an algorithm that eliminates the need for the final sequential step and enables parallel computation of the global set.

The algorithm originally introduced in [12] consists of two distinct phases. During the first phase, the local sets are computed as usual for each partition. However, instead of computing the global set sequentially from the local sets, we divide the set of local sets into multiple partition and pass to it the entire set of local sets. This enables each partition to have its own set of points to extrapolate non dominated points and compare against the entire set of local sets. This method guarantees that every point that returns every partition in the last parallel phase will be part of the global set. So this method allows us to bypass the final sequential step and perform parallel computation of the global set.

In this thesis we present an improvement that leverages the sorting property, which dictates that a point that comes after another in the sorted dataset cannot dominate it. As a result, the improvement only works with sorted algorithms such as SFS for Skyline computation and 'S' algorithms for ND operator computation.

Algorithm 19 All Parallel (Improved version)

```
     Input: S: dataset, n:number of partitions
     Output: GS: global set
1:   GS ← [ ], P1← [ ], P2← [ ]
2:   P1← PartitioningAlgorithm(S,n)
3:   for p in P1 do
4:       LocalSet ← LocalSet ∪ ComputeLocalSet(p)
5:   P2← PartitioningAlgorithm(LocalSet, n)
6:   for p in P2 do
7:       p.sort()
8:       for data in p do
9:           for s in LocalSet do
10:              if s == data then GS ← GS ∪ data and continue to line 8
11:              if s dominates data then continue to line 8
10:  return GS
```

As with all algorithms, we can carry out an initial filtering phase, thus making our algorithm three phases. The first phase involves a filtering phase, in our case as seen

earlier, the fastest and most selective is representative filtering using as representatives the points with the greatest dominance region in the case of anticorrelated datasets.

The second phase is to perform a parallel step by taking the unfiltered points and dividing them into several partitions to find the local sets. Once the local sets have been found, there is a third phase which is to compute the final global set, which in this algorithm involves dividing the points of the local sets into several partitions and passing the entire set of local sets to each one so that the dominated points can be deleted.

We also propose an improvement to this algorithm that involves unifying the first and second phases. This is done by computing the local sets together with the filtering phase, thus saving some time.

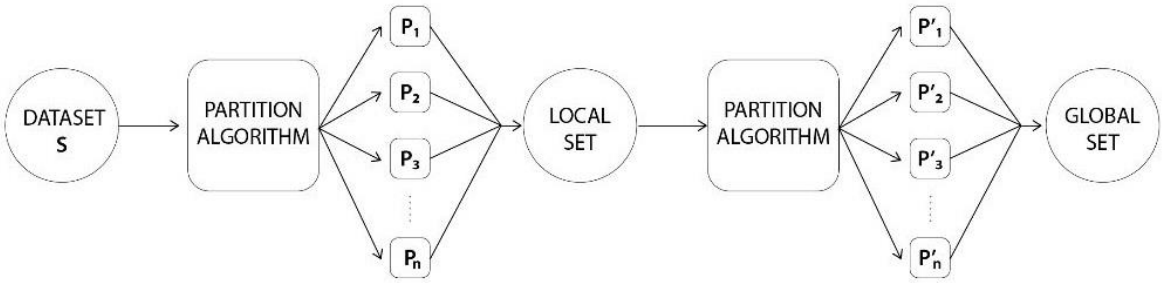


Figure 5.7: Block diagram of the All Parallel algorithm

Figure 5.7 shows a block diagram with all the steps of the All Parallel algorithm. From the dataset S, we perform a partitioning algorithm where we compute the local sets in each partition. After that we take the union of the local sets and partition again by passing the local global set to each partition. At the end, each partition will return the points that will be part of the final global set.

Algorithm 20 All Parallel SKY

```
        Input: S: dataset, n: number of partitions
        Output: SKY: global set SKY
1:      SKY ← [ ], P1← [ ], P2← [ ], P3 ← [ ]
2:      LocalSet ← [ ], representatives ← [ ]
3:      P1 ← Partitioning (S,d)              // part. the dataset in d partitions
4:      for index in [0, len(P1)] do
5:          R ← [ ]
6:          for data in S do
7:              area ←∏_{i=0}^{d-1}(1 − t[i])
8:          if area > R[⌊t[index] ∗ n⌋].area then R[⌊t[index] ∗ n⌋] ← (area, data)
9:          for r in R do
10:             representatives ← representatives ∪ r.data
11:     representatives ← computeLocalSet(representatives)   //discard dominated t
12:     P2 ← PartitioningAlgorithm(S, n)          //partitioning the dataset
13:     for p in P2 do
14:         R ← []
15:         for data in p do
16:             for rep in representatives do
17:                 if rep ≺ data then continue to line 15
18:             R ← R ∪ data
19:         LocalSet ← LocalSet ∪ SFS (R)
20:     LocalSet.sort()
21:     P3← PartitioningAlgorithm(LocalSet, n)
22:     for p in P3 do
23:         p.sort()
24:         for data in p do
25:             for s in LocalSet do
26:                 if s == data then SKY ← SKY ∪ data and continue to line 24
27:                 if s ≺ data then continue to line 24
28:     return SKY
```

This algorithm (20) provides an initial phase in which the representatives are found on the basis of their dominance region and once found, the dataset is divided into n partitions and the representatives are passed to each one. After that each partition is responsible to delete all the points dominated by these representatives and when finished perform the SFS algorithm with the points remaining in the partition not dominated by the representatives so as to find the local set of each partition. Once all the local sets have been found, the points present in the union of the found local sets are repartitioned, passing the global local set to each partition. Each partition will have to sort its points and scan them sequentially comparing them with the points in the

local set which are also sorted, exploiting the topological sort property according to which a point that comes after another cannot dominate it, if the point we are analysing is compared with itself in the global local set then we add it to the global set since all the points that come afterwards will certainly not be able to dominate it, otherwise, if is dominated by a point that comes before it in the sorting then it is discarded. In the end, each partition will return a set of points that will be part of the final global set.

Algorithm 21 All Parallel ND

```
        Input: S: dataset, n: number of partitions, W: vertices of W(C)
        Output: ND: global set ND
1:      ND ← [ ], P1← [ ], P2← [ ], LocalSet ← [ ],
2:      representatives ← computeRepresentatives (S)    //same as SKY
3:      P1 ← PartitioningAlgorithm(S, n)          // partitioning the dataset
4:      for p in P1 do
5:          R ← []
6:          for data in p do
7:              compute left-hand side of Inequalities
8:              for rep in representatives do
9:                  if rep == data then continue to line 11
10:                 if rep ≺ data or rep ≺F data then continue to line 6
11:             R ← R ∪ data
12:         LocalSet ← LocalSet ∪ SVE1F (R)
13:     LocalSet.sort(W)
14:     P2← PartitioningAlgorithm(LocalSet, n)
15:     for p in P2 do
16:         p.sort(W)
17:         for data in p do
18:             compute left-hand side of Inequalities
19:             for s in LocalSet do
20:                 if s == data then ND ← ND ∪ data and continue to line 17
21:                 if s ≺ data or s ≺F data then continue to line 17
22:     return ND
```

The difference between the algorithm 20 and 21 are highlighted in red in the algorithm 21. The all parallel ND algorithm is similar to the SKY one in the computation of the representatives, with the difference that when we go to compute the local sets by dividing into partitions, in addition to testing the simple dominance we have to also test the F-dominance, and once the dominated points have been eliminated, we run an ND algorithm like SVE1F in each partition to find the local sets. Finally, the last parallel

part is similar to that of SKY with the difference that we also test here both simple dominance and F-dominance.

Algorithm 22 All Parallel PO

```
    Input: ND: nd set, n: number of partitions, W: vertices of W(C)
    Output: PO: global set PO
1:  PO ← [ ], P1← [ ], P2← [ ], LocalSet ← [ ],
2:  P1← RandomPartitioning(ND, n)
3:  for p in P1 do
4:      LocalSet ← LocalSet ∪ PO_incremetal_computation(p)
5:  P2← PartitioningAlgorithm(LocalSet, n)
6:  for p in P2 do
7:      for data in p reversed do
8:          if isNonPO(data, LocalSet \ {data}) then p ← p \ {data}
9:      PO ← PO ∪ p
10: return PO
```

The all parallel PO algorithm differs from the others because starting from the ND set it does not perform any initial filtering step, except for the computation of ND. Once the ND set is obtained we partition it and for each partition we compute the local set by performing the incremental or non-incremental PO computation with the points in each partition. Once all local sets are obtained, in the same way as SKY and ND we partition the local set and pass the global local set to each partition. Another time we can use both methods to compute PO, and each point that the partitions return will be part of the global set.

6 Experimental Settings

In this chapter, we will explain the framework used for parallelization and the environment used to execute the algorithms. After that, we will list all the packages used and their specific tasks for computing the various algorithms and the dataset generator used.

6.1. Pyspark framework

Apache Spark [13] is an open source framework for distributed computing. Unlike the MapReduce [14] paradigm which processes data on disk, Spark processes them in memory providing much better performance for the same applications. Spark applications run as a set of independent processes in a cluster and are coordinated by the SparkContext object in the main program, called the driver program. The SparkContext object tells Spark how to access a cluster and to create a SparkContext we first need to build a SparkConf object that contains information about our application.

Every Spark application consists of a driver program that executes the main user function and performs various parallel operations on a cluster. Spark provides a resilient distributed data set (RDD) that is an abstraction, i.e. a collection of partitioned elements between cluster nodes on which parallel operations can be performed.

Spark supports multiple widely-used programming languages like Java, Python, R, and Scala. In this thesis we will use Pyspark that is an interface for Apache Spark in Python.

The parallel computation using Pyspark takes place by making use of the parallelize API, which allows to divide the dataset into multiple partitions, with the number of these that can be input by the user.

6.2. Computational environment and packages utilized

In this thesis, we use Python as a programming language, which is a high-level language very suitable for parallel computing. It is a language that has developed a lot in recent years, and thanks to the numerous packages it contains, it is very useful for the work we will do in this thesis. In particular, here are some of the main packages used for the computation of the algorithms seen in this thesis:

Pycddlib: Is used for generating all vertices (i.e. extreme points) and extreme rays of a general convex polyhedron given by a system of linear inequalities.

Pulp: Is an LP modeler in python used to solve linear problems.

Gurobipy: Gurobi Optimizer is a mathematical optimization software library for solving linear and mixed integer quadratic optimization problems.

FindSpark: package that contains findspark.init() that is used to make pyspark importable as a regular library.

The code implemented for the parallel computation of the algorithms can be reached using the link [18].

For the experiments, we will use synthetic datasets created using a dynamic points generator from code.

All the experiments will be performed on two different machines, the first being a computer running on Windows 10 with 8 GB of RAM and a processor Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz with 4 cores. The second is a virtual machine granted by the Polimi Datacloud running on Ubuntu 22.04 with 8 GB of RAM and a processor with 30 cores.

7 Experimental Results

In this chapter, we will run the various algorithms for the skyline computation set and the Flexible Skyline ND and PO operators. Our goal will be to evaluate the various parallel algorithms seen in this thesis and try to understand which algorithms perform best and in what setting. The various algorithms will be run on two different machines, a local machine with 4 cores and a virtual machine with 30 cores. The substantial difference between the two machines will be seen in the much shorter duration of the parallel algorithms, because since we have more cores, we will be able to take much more advantage of parallelization by having several workers running tasks in parallel.

We will evaluate how long the various algorithms we have seen take to compute the global Skyline set and the ND and PO global sets in the parallel versions. We will only look at the centralized version for SFS, because since some are very time-consuming, the centralized version would take too long to return the final global set. We will evaluate the performance of these algorithms by changing the cardinality of the dataset, the dimensionality of the data, the number of partitions and the number of cores used for the parallelization.

7.1. Summary of findings

We will see that for both the skyline computation and the computation of ND, the best algorithm is Sliced Partitioning which manages to return the smallest local set in the shortest possible time in both cases, thus impacting less on the final sequential part. Compared to the Angular Partitioning algorithm for Skyline computation, we see that in the case where an anticorrelated dataset with 4 dimensions and 3 million points is taken into account, the execution time goes from 244.97 seconds of Angular to 184.55 seconds of Sliced Partitioning. This increased speed is due to the size of the returned local set which goes from the 66251 points of Angular to the 40251 of Sliced, thus having a lighter and consequently faster sequential phase. As far as the computation of the PO set is concerned, on the other hand, the best algorithm is not Sliced Partitioning, which will perform worse than both Random and Angular Partitioning since for the computation of the PO set sorting on one dimension is not important for the computation of the global set, but it is Random Partitioning that succeeds in dividing the load equally between the partitions.

We will see how the improvements in Chapter 5 will affect the standard algorithms, in particular Grid Filtering will not bring any advantage by eliminating a few points in the filtering phase, instead, Representative Filtering manages to eliminate many more points bringing an advantage over the standard version by returning a smaller local set. Thanks to representative filtering, we are able to go from 184.55 seconds for the version of Sliced Partitioning without filtering to 164.1 seconds for the version with filtering in the case of the skyline computation with a 4-dimensional anticorrelated dataset with 3 million points.

The best algorithm for all 3 types of skyline computation is the All Parallel Algorithm. This algorithm is the only one which, by eliminating the final sequential phase and computing the global set in parallel, manages to making the most efficient use of the parallelization and, as we shall see, will take much less time than the version of the same algorithm with the final sequential part. In this case for the skyline computation set we have a duration of 38.17 seconds, while the same algorithm with the final sequential part has a duration of 164.1 seconds which is 4.29 times longer. For the computation of the ND set instead, using a 2-million point dataset manages to finish in 99.66 seconds, about 6.5 times faster than the same version with the final sequential part.

We are going to evaluate how the algorithms behave when changing the number of partitions, size and cores. By changing the number of partitions, we will see how the duration of the algorithms is affected, because if it is true that increasing the number of partitions increases the speed of the parallel part, on the other hand, having more partitions the size of the local set returned will be greater, affecting the duration of the final sequential part.

By changing the number of dimensions, on the other hand, we will see how the duration of the algorithms follows an exponential trend because by increasing the dimensionality of the dataset we will greatly increase the size of the global set, resulting in very high execution times.

Finally, we will see how changing the number of cores affects the execution speed of parallel algorithms. All the algorithms, by increasing the number of cores, will decrease the execution time, with the duration of the parallel part becoming smaller and smaller as the number of cores increases, but since they always have to perform a final sequential part, this advantage is not so great. It is a different matter for PO computation, which, since it has partitions that have to compute many LPs, will have a parallel part with a considerable duration, thus improving the execution time by a significant amount. On the other hand, for the all parallel algorithm, which manages to make the most of parallelization, the change in the number of cores helps reduce the duration by a factor of 2.37 for the skyline computation, 2.28 for the computation of the ND set and 3.53 for the PO set.

7.2. Execution Time of the Serial Algorithms

In this section, we will look at the durations of the various sequential algorithms to see which of them are the best in terms of execution time so as to apply parallelization to these only. We start with the algorithms for skyline computation, as we can see from figure 7.2.1, the slowest algorithm is the BNL algorithm because it has to do many more comparisons than SFS and SaLSa. After that SFS is faster because it uses topological sort to make fewer comparisons than BNL, but it needs a data pre-processing step where we sort the dataset. SaLSa on the other hand manages to go faster than SFS because we can not check the entire dataset but can stop earlier.

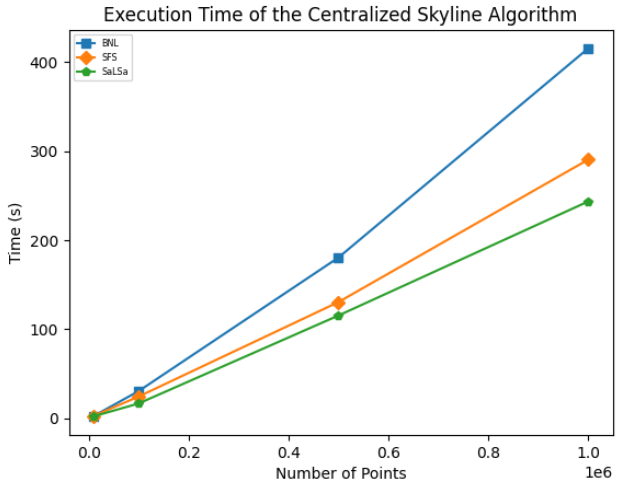


Figure 7.2.1: Execution time of the Centralized Skyline Algorithms using 4d anticorrelated datasets.

However, SFS is highly dependent on the sorting function we use, the better this is, the faster the algorithm will terminate. In figure 7.2.2 we use a sorting function that is equal to the weighted sum with the coordinates of the centroid of W(C) as weights, i.e. the sorting function used to compute ND. As we can see, the execution time drops dramatically, so we are going to evaluate the behaviour of SFS using parallel algorithms. We cannot use the same sort function for SaLSa because in order to stop the execution of the algorithm earlier, we need a symmetric sort function. We have that a function F on d variables is symmetric if it is not invariant under any permutation of the variables, i.e. the function does not privilege any attribute over the others, so all attributes play the same role, therefore this type of sorting function cannot be used since it imposes weights on the attributes by giving them different importance.

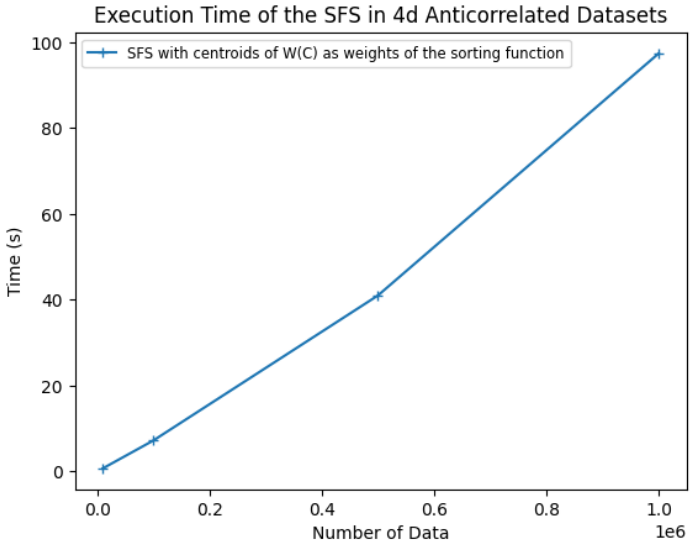


Figure 7.2.2: Execution time of the Centralized SFS with centroids as weights of the sorting function using 4d anticorrelated datasets

In the same way, we can verify which algorithm for computing ND and PO is the best in terms of execution time. For this check, only the algorithms in the sorted ('S') version will be used, since as we have also seen for the skyline computation, topological sorting helps us to decrease the duration of the algorithms compared to the unsorted version. Small datasets will be used for the results, as the running time of the LP algorithms is very expensive.

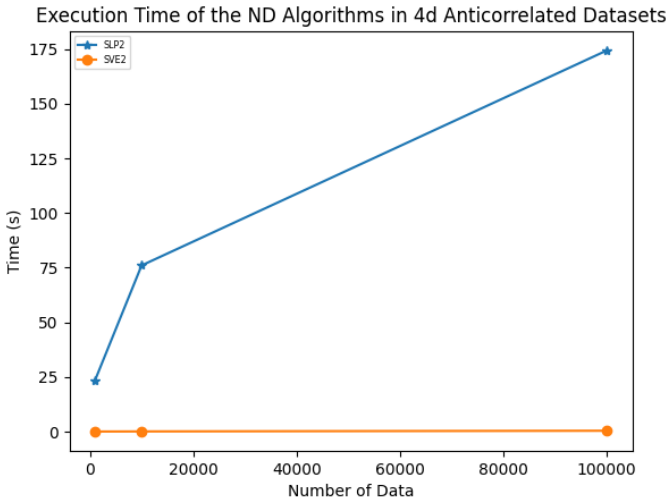


Figure 7.2.3: Execution time of the Centralized ND Algorithms using 4d anticorrelated datasets.

As we can see from figure 7.2.3 in which we compare the two algorithms in the two-phase sorted version in which the former uses the LP technique and the latter the VE

technique, the VE version is much faster than the LP version which has to solve an LP problem with each comparison, as opposed to the VE technique where the computation of the vertex enumeration of the polytope which introduces the most significant overhead is performed only once.

So now we are going to compare the algorithms using the VE technique in the one-phase and two-phase versions to see which are the fastest.

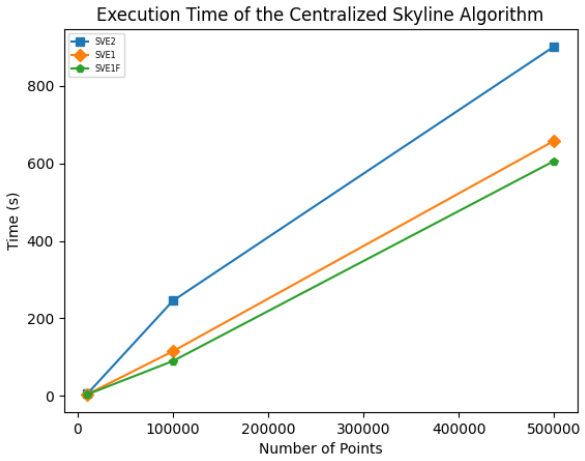


Figure 7.2.4: Execution time of the VE centralized ND Algorithms using 4d anticorrelated datasets

From the figure 7.2.4 we can see that the one-step algorithms are faster, and in particular we note that the SVE1F algorithm is the one that takes less time to finish than its variant SVE1 where the only difference is the way they verify simple dominance and F-dominance.

In this last figure 7.2.5, we see the difference between the version with PO computation using the Dual PO Test and the Primal, both in the incremental version. We can see that the duration of the two algorithms does not differ much, with a small advantage in terms of execution time for the Primal.

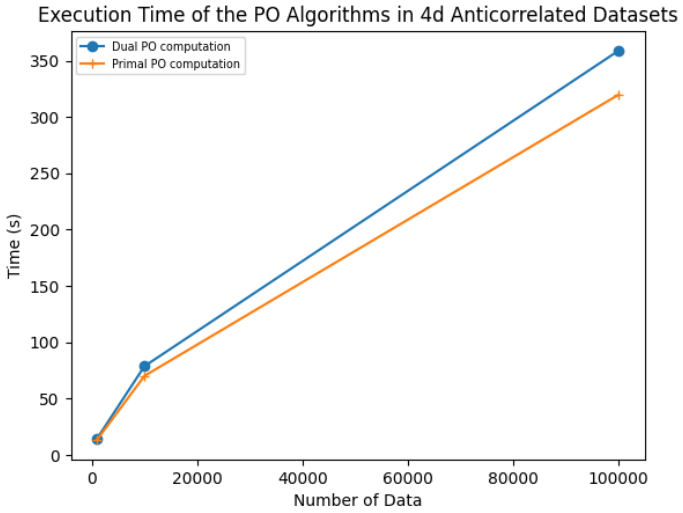


Figure 7.2.5: Execution time of the centralized PO Algorithms using 4d anticorrelated datasets

Having this preliminary information on which algorithms have the shortest execution time for skyline (SFS), ND (SVE1F) and PO (PO Primal Test) computation, we will go on to apply the parallel algorithms only to these which will be the most promising.

7.3. Execution Time of the Parallel Algorithms

In this section we will look at the duration of the parallel algorithms applied using the SFS algorithm for skyline set computation, the SVE1F algorithm for ND computation and finally the incremental algorithm using the PO Primal Test for PO computation.

In detail, we will see how the algorithms using random partitioning, grid partitioning, angular partitioning and finally one-slice partitioning perform. As far as SFS is concerned, the parallel algorithms, together with the centralized version, will run on both the local machine with 4 cores and the virtual machine with 30 cores.

On the other hand, for SVE1F and PO incremental Primal, only the parallel algorithms will be executed without a centralized version on the virtual machine, as this will be the fastest one.

For the experiments we will use anticorrelated synthetic datasets generated, ranging from 100k points up to 3 million points, with the number of dimensions fixed at 4. We will see the execution time on two different machines, one with 4 cores and the other with 30, where we will see that having more cores using Spark's parallelization will drastically decrease the execution time.

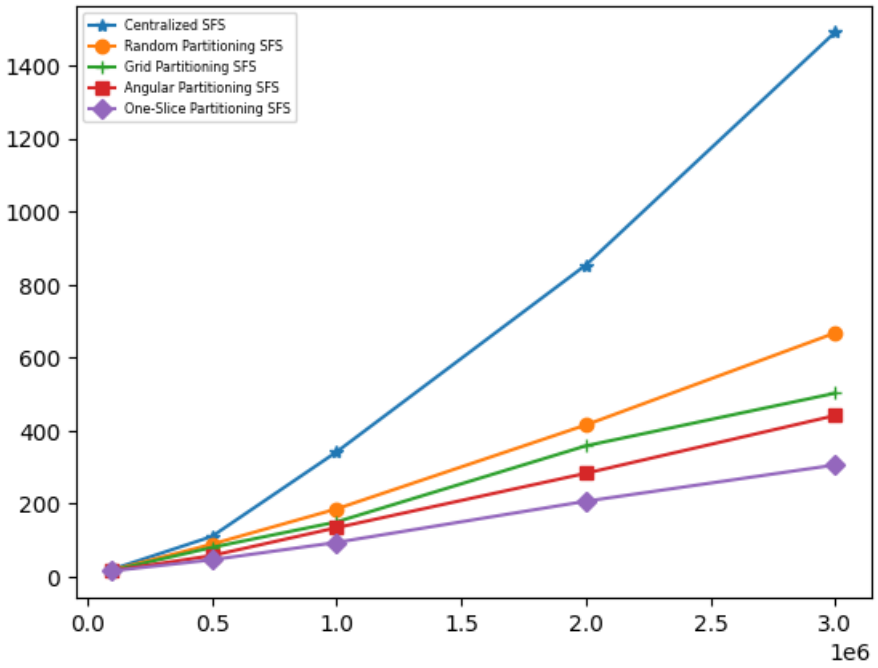


Figure 7.3.1: Execution Time of the SFS algorithm in the centralized and parallel version using 4d anticorrelated datasets and the local machine with 4 cores.

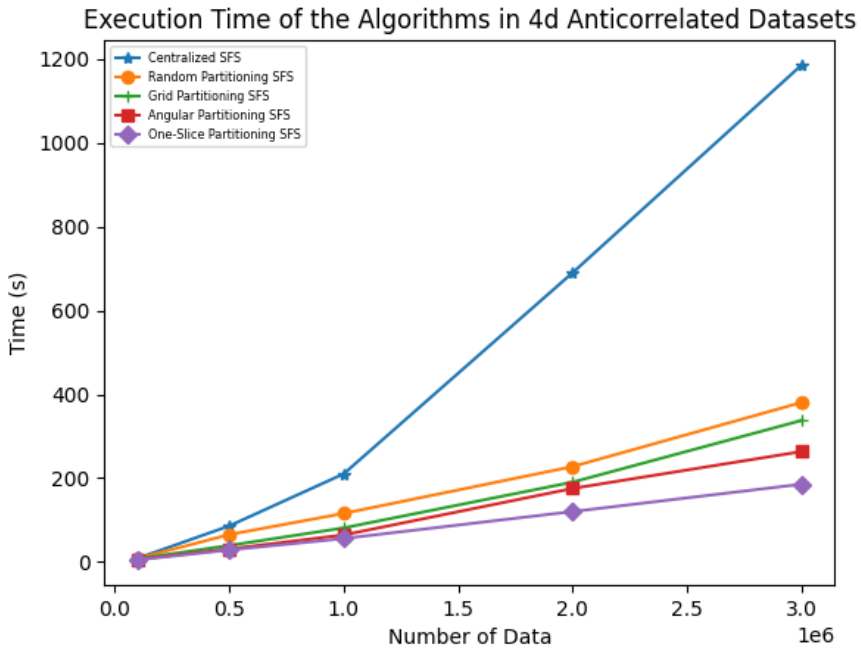


Figure 7.3.2: Execution Time of the SFS algorithm in the centralized and parallel version using 4d anticorrelated datasets and the virtual machine with 30 cores.

From the figures 7.3.1 and 7.3.2, we can immediately see how the execution time drops dramatically between the centralized version and the distributed version on both machines. As we can see between the local machine and the virtual machine, there is a difference in the duration of the algorithms themselves, especially the parallel ones in which it is able to utilise the full power of the 30 cores to best parallelize the computation. In the following figures we see the power of all the algorithms to compute the local Skyline sets with the amount of time it takes to finish each one.

We immediately notice that the parallel algorithm that applies random partitioning is the one that takes the longest to finish. We expected this, however, because as discussed in the previous chapters, it is true that it is the simplest type of partitioning of all and the one that divides the work best between all the partitions, but it is also true that we have no control over what type of points will occur in the partitions, for example, only "good" points may occur in some partitions and therefore the local set of these partitions will be very large, thus having the risk of local sets that are too large.

Next we see how we can improve speed with grid partitioning and angular partitioning, both of which partition better than random partitioning leading to an improvement in execution time.

The best partitioning algorithm as also demonstrated in the thesis [12], however, is sliced partitioning, which succeeds in returning the smallest local sets in the shortest time so as to have the least heavy sequential phase.

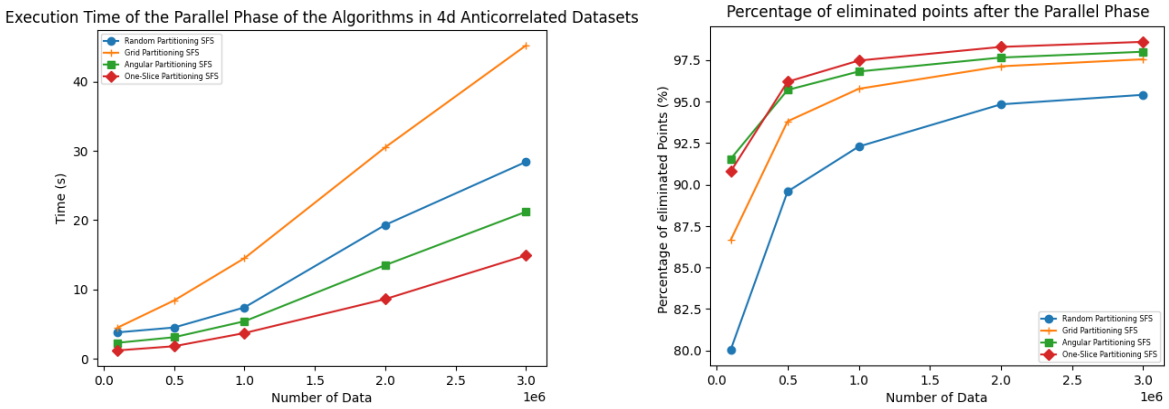


Figure 7.3.3: Execution time of the parallel phase of the SFS Algorithms and percentage of the dominated points in 4d anticorrelated dataset using the virtual machine.

Algorithm's Name	Number of Local Skyline points	Execution time Parallel phase	Execution time Sequential Phase
Random Partitioning	134690	27.5	348.97
Grid Partitioning	79361	46.3	289.01
Angular Partitioning	66251	23.6	244.97
One-Slice Partitioning	40251	14.9	184.55

Table 7.1: Time of parallel and sequential phases of the SFS Algorithms using a 4d anticorrelated dataset with 3 million points and 30071 SKY points on the virtual machine.

As we can see from figure 7.3.3 and table 7.1, the algorithm that applies the best partitioning is One-Sliced Partitioning. Taking table 7.1 as a reference, we see how the One-Sliced algorithm manages to eliminate almost 98.66% of the data in the parallel phase, in a time of about 14.9 seconds. We can see that grid partitioning manages to eliminate several points in the parallel phase, almost as good as One-Slice and Angular, but it takes too long to perform the parallel phase. This is because since it is a technique that uses grids to partition neighbouring points, and since we are evaluating anticorrelated datasets that have a high concentration of points in certain areas, we will have some partitions being very heavy, as opposed to others that will have very few points if any.

From now on, all algorithms will be executed only on the virtual machine, which as we have seen is the fastest one.

We will now look at the same algorithms without the centralized version but applied to the computation of the ND set, specifically using the SVE1F algorithm. For the experiments we will use anticorrelated synthetic datasets generated, ranging from 100k points up to 2 million points, with the number of dimensions fixed at 4. We use only one constraint which is: $w_1 - w_2 \geq 0$.

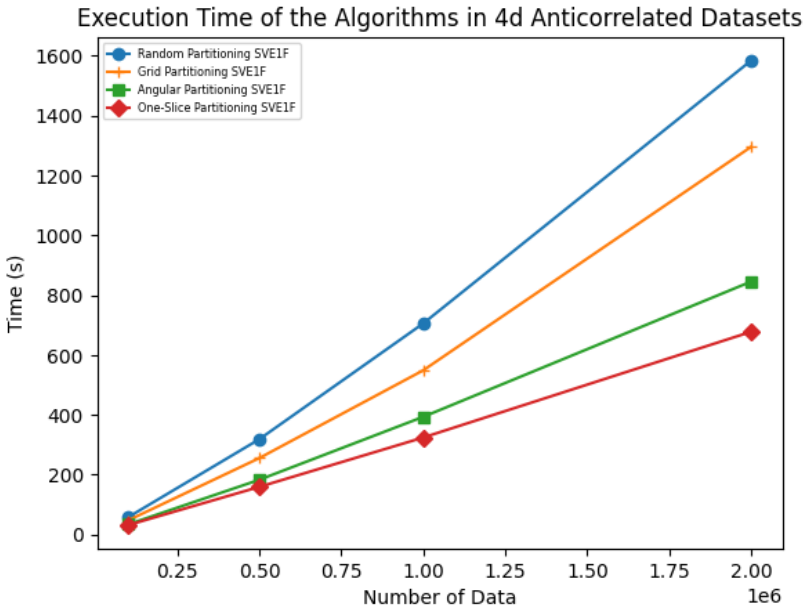


Figure 7.3.4: Execution Time of the SVE1F algorithm in the parallel versions using 4d anticorrelated datasets and the virtual machine.

As we can see from the graph in figure 7.3.4, even with the computation of ND using SVE1F we have the same behaviour as with SFS. Once again, the worst performer is Random Partitioning, while the best performer is One-Sliced Partitioning. This is due to the fact that once again the partitioning types return local sets, which in the case of One-Slice Partitioning are much smaller than the other partitioning techniques. In the following figure 7.2.5, we will look in detail at the amount of data dominated in the parallel phase by all partitioning techniques and the time it takes.

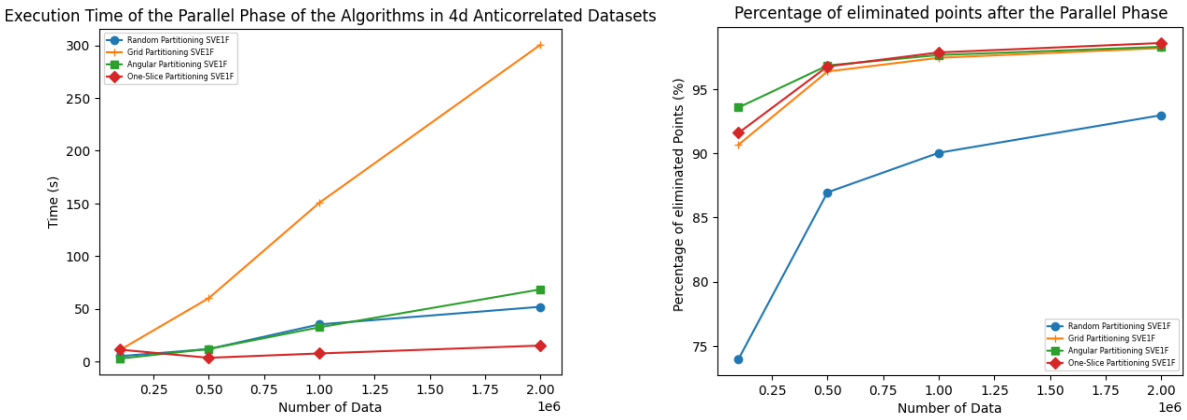


Figure 7.3.5: Execution time of the parallel phase of the SVE1F Algorithms and percentage of the dominated points in 4d anticorrelated dataset using the virtual machine.

Algorithm's Name	Number of Local ND points	Execution time Parallel phase	Execution time Sequential Phase
Random Partitioning	140512	51.9	1480.16
Grid Partitioning	39946	325.6	970.05
Angular Partitioning	33971	68.33	776.98
One-Slice Partitioning	28051	14.8	679.3

Table 7.2: Time of parallel and sequential phases of the SVE1F Algorithms using a 4d anticorrelated dataset with 2 million points and 16781 ND points.

As we can also see from figure 7.3.5 and table 7.2, once again the best algorithm is One-Slice Partitioning, which in this case manages to delete almost 98.6% in a very small amount of time compared to the other partitioning algorithms. Again, grid partitioning has a low number of local skyline points, much lower than random partitioning and similar to angular, but as we can see compared to angular it is 4.76 times slower for the parallel phase, while compared to one-sliced partitioning it is more than 20 times slower.

Finally, we evaluate these algorithms for the computation of the PO set using the incremental version that makes use of the Primal PO Test. For the experiment, we will only consider the time it takes the algorithm to find the PO set from the ND set, thus not taking into account the time for the computation of the ND set.

This time since we will have to solve LP we would like the partitions to have more or less the same number of points. For this reason we will not use Grid Partitioning because with anticorrelated datasets we will have some partitions with many points and others almost empty, incurring in a high overhead in the parallel phase.

For the experiments we will use anticorrelated synthetic datasets generated, ranging from 100k points up to 2 million points, with the number of dimensions fixed at 4. We use only one constraint which is: $w_1 - w_2 \geq 0$.

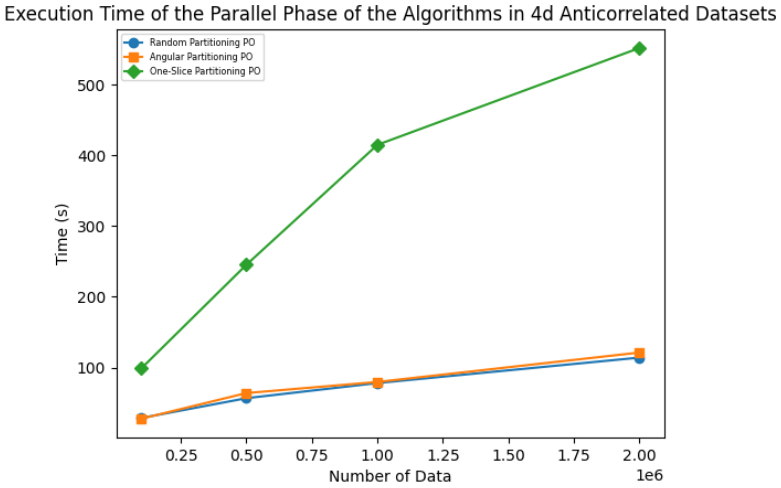


Figure 7.3.6: Execution Time of the PO algorithm starting from the ND set in the parallel versions using 4d anticorrelated datasets and the virtual machine.

Looking at figure 7.3.6, we notice a completely different behaviour of the partitioning algorithms compared to the Skyline and ND computation. As we can see, random partitioning together with angular partitioning are the algorithms that finish in the shortest time, which is almost indistinguishable, while this time the worst is One-Slice partitioning.

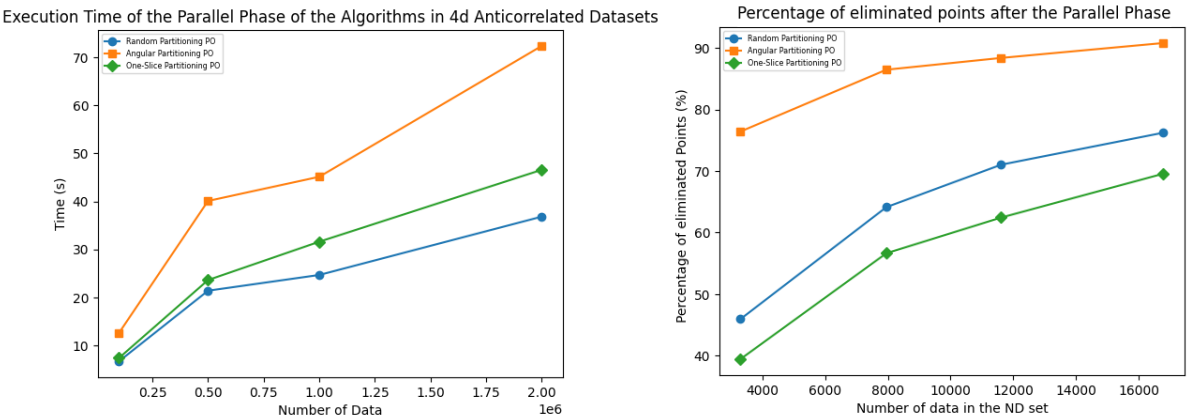


Figure 7.3.7: Execution time of the parallel phase of the PO Algorithms and percentage of the dominated points starting from the ND set in 4d anticorrelated dataset using the virtual machine.

Algorithm's Name	Number of Local PO points	Execution time Parallel phase	Execution time Sequential Phase
Random Partitioning	3990	36.82	77.42
Angular Partitioning	1544	72.4	48.97
One-Slice Partitioning	5108	46.58	504.52

Table 7.3: Time of parallel and sequential phases of the PO Algorithms using a 4d anticorrelated dataset with 16781 ND points and 150 PO points.

In figure 7.3.7 and table 7.3 we see in detail the size of the local sets returned by the parallel partitioning of the various algorithms and the duration of the final sequential phase. The difference lies precisely in the number of local points returned by the various algorithms. If for SFS and SVE1F the One-Slice returned the smallest local set, this is not the case for the PO computation. This is due to the fact that the PO set is a very small set of points compared to ND, e.g. in the case of 2 mil from 16781 ND it goes to 150 PO, so sorting the dataset by one dimension does not help in the computation of PO, while the random factor helps speed up the process. Angular partitioning compared to random partitioning manages to return a smaller local set but takes much longer to compute because we will have some partitions with more points than others and since we have to solve LP to compute the PO set, having many points in some partitions does not help. This time that we lose in the parallel computation is recovered in the sequential part leading to having the same duration for both algorithms.

7.4. Execution Time of the Improved Parallel Algorithms

In this section, we are going to test the execution time of the improved parallel algorithms seen in Chapter 5. We will apply these improvements to the Angular and One-Slice partitioning algorithms, which, as seen in the previous section, are the fastest. In detail we will see how the Angular algorithm behaves with an initial grid filtering phase, Angular and One-Slice with representatives filtering using the Dominance Region method to find representatives, which as seen in Chapter 5 is the best at filtering data.

Finally we will look at the All Parallel algorithm in which the initial parallel phase is performed using One-Slice partitioning with representatives. As we will see, the best results are given by One-Slice partitioning with representatives which manages to speed up the version without initial filtering, but the best of all is the All-Parallel algorithm which eliminates the final sequential part and uses a final parallel part instead. By exploiting parallelism and the 30 cores of the virtual machine with this algorithm we manage to finish the computation, taking SFS as an example, 4.71 times sooner than the same version of One-Slice partitioning in which the final sequential part is used.

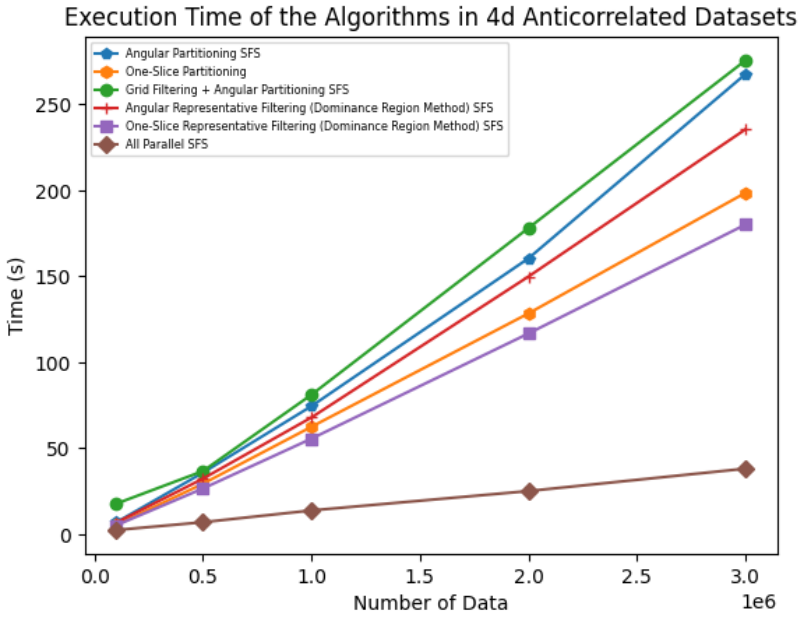


Figure 7.4.1: Execution Time of the improved parallel algorithms using SFS with 4d anticorrelated datasets and the virtual machine.

Looking at the execution time of the algorithms we see that grid filtering gives no advantage over the same version without initial filtering of angular partitioning, in fact it performs worse because the initial filtering phase has a duration and the number of filtered points is not enough to bring an advantage. This is always due to the problem of grid filtering, which with anticorrelated datasets does not partition the dataset well. The advantage over the standard versions is given by the representative filtering, which by performing the initial filtering phase manages to eliminate many points before performing the parallel computation leading to an advantage. Finally, the last algorithm is the All Parallel that manages to exploit the parallel computation to take advantage over the versions with the final sequential part, arriving for 3 million points and 30071 skyline points to have a duration of 38.17 seconds, while the same algorithm with the final sequential part has a duration of 164.1 seconds which is 4.29 times longer.

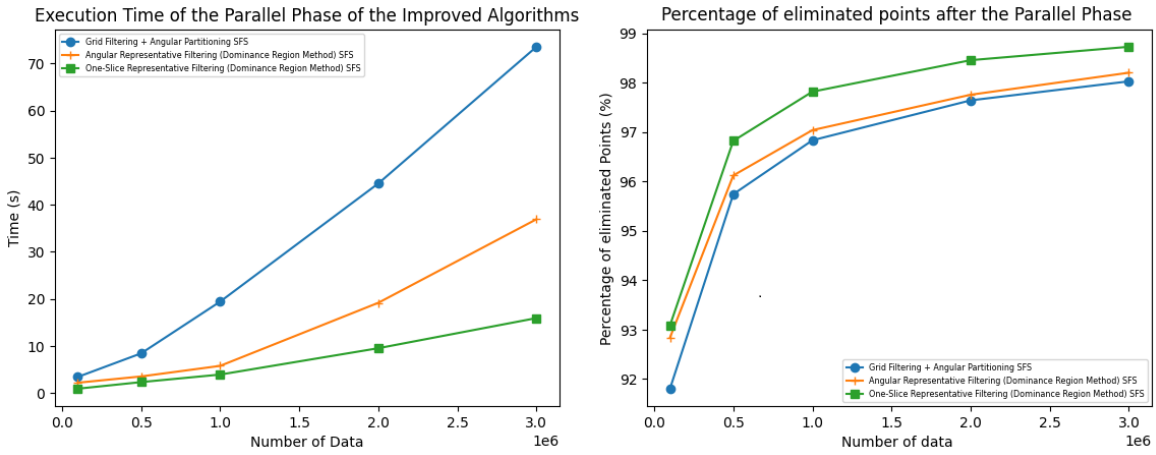


Figure 7.4.2: Execution time of the parallel phase of the Improved parallel Algorithms using SFS and percentage of eliminated points in 4d anticorrelated dataset using the virtual machine.

Algorithm's Name	Number of Local Skyline points	Execution time Parallel phase	Execution time Sequential Phase
Angular Partitioning	66251	23.6	244.97
One-Slice Partitioning	40251	14.9	184.55
Grid Filtering + Angular Partitioning	59253	73.15	201.29
Angular Representative Filtering	55512	39.9	194.82
One-Slice Representative filtering	36268	15.9	164.1

Table 7.4: Time of parallel and sequential phases of improved Parallel Algorithms using SFS with a 4d anticorrelated dataset with 3 million points and 30071 SKY points.

Let us look in detail at how long these algorithms take to perform the parallel phase and how many points they manage to eliminate compared to the basic version. As we can see from table 7.4, the algorithms with initial filtering manage to eliminate many more points in the parallel phase than the standard versions, but they take longer to execute. As far as Grid Filtering is concerned, it is very time-consuming, so even if we manage to reduce the size of the Local Skyline compared to the standard version, the extra time we put in for the parallel phase does not give us any advantage in terms of final execution time, leading to a slowdown of the algorithm. As far as Representative Filtering is concerned, as we can see for both Angular Partitioning and One-Slice Partitioning, even though it takes longer to compute the parallel phase (not as long as Grid Filtering), it returns a Local Skyline set that is much smaller than the basic version, and we are therefore able to speed up the final sequential part.

We are now going to evaluate how these algorithms perform in the case of ND set computation.

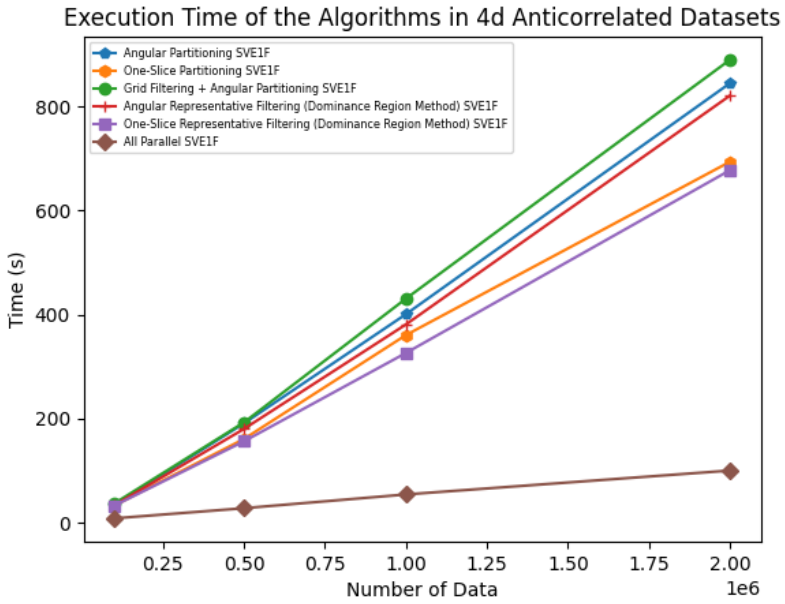


Figure 7.4.3: Execution time of the parallel phase of the Improved parallel Algorithms using SVE1F and percentage of the local set size in 4d anticorrelated dataset using the virtual machine.

The behaviour of these algorithms for the computation of ND set using SVE1F is very similar to that of the skyline set computation using SFS. Also in this case we have that Grid Filtering in the case of anticorrelated datasets does not bring any advantage in terms of final execution time compared to the basic version, while the Representative Filtering manages to speed up the duration of the algorithm compared to the version without. The Representative Filtering used is the one aimed at the computation of ND set computation as opposed to the previous one used for the computation of SFS, which as we have seen in Chapter 5 speeds up the total duration of the algorithm by eliminating many more points in the parallel part during the computation of local sets. Once again, as with SFS, the best algorithm is the All Parallel algorithm with One-Slice Representative Filtering, which in the case of a 2-million point dataset with 16774 ND points manages to finish in 99.66 seconds, about 6.5 times faster than the same version with the final sequential part.

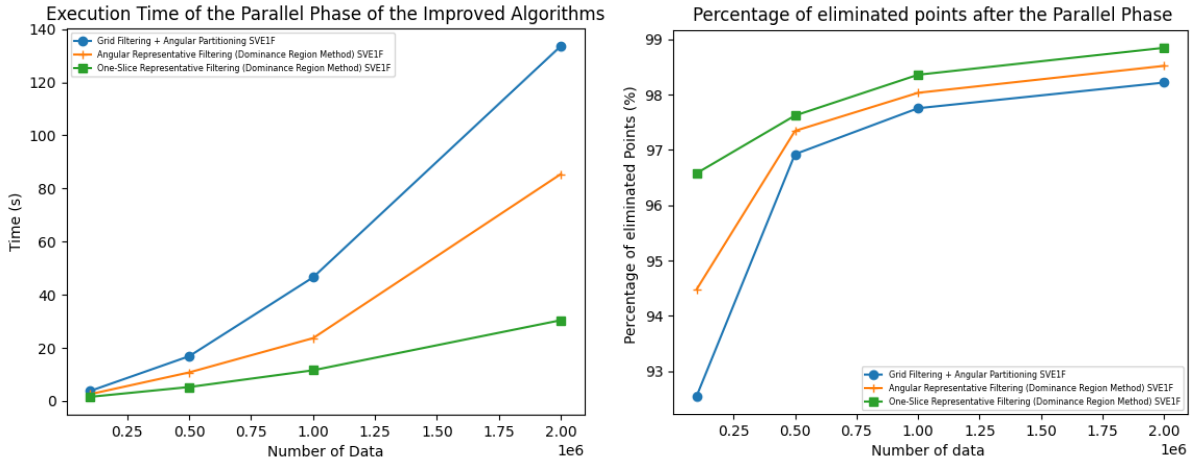


Figure 7.4.4: Execution time of the parallel phase of the Improved parallel Algorithms using SVE1F and percentage of the local set size in 4d anticorrelated datasets using the virtual machine.

Algorithm's Name	Number of Local ND points	Execution time Parallel phase	Execution time Sequential Phase
Angular Partitioning	33971	68.33	776.98
One-Slice Partitioning	28051	14.8	679.3
Grid Filtering + Angular Partitioning	31765	133.5	749.4
Angular Representative Filtering	29640	85.4	731.99
One-Slice Representative filtering	23104	30.36	647.49

Table 7.5: Time of parallel and sequential phases of improved Parallel Algorithms using SVE1F with a 4d anticorrelated dataset with 2 million points and 16781 ND points.

As we can see from table 7.5, the algorithms applying initial filtering succeed in reducing the size of the Local Set computed during the parallel phase, but as far as Representative Filtering is concerned, it brings advantages over the standard version of the same algorithm, whereas Grid Filtering takes too long to return the Local Set, thus bringing a disadvantage in terms of duration compared to the basic version.

Finally, we will evaluate the behaviour of the All Parallel algorithm in the case of PO computation. Since we start from the global set ND to compute PO, we cannot apply

the Representative and Grid Filtering algorithms to the computation of PO, but we can apply the All Parallel algorithm to try to decrease the duration of the algorithm by eliminating the final sequential part, which for this type of algorithm where many LPs have to be solved, brings a huge advantage to be able to parallelize the execution.

Since, as seen above, One-Slice partitioning does not bring any advantage for PO computation, in fact in figure 7.3.6 we can see that it reduces the speed by a lot compared to random and angular partitioning, we are going to use the all parallel algorithm using for the initial parallel part where we are going to compute the local sets, both angular and random partitioning.

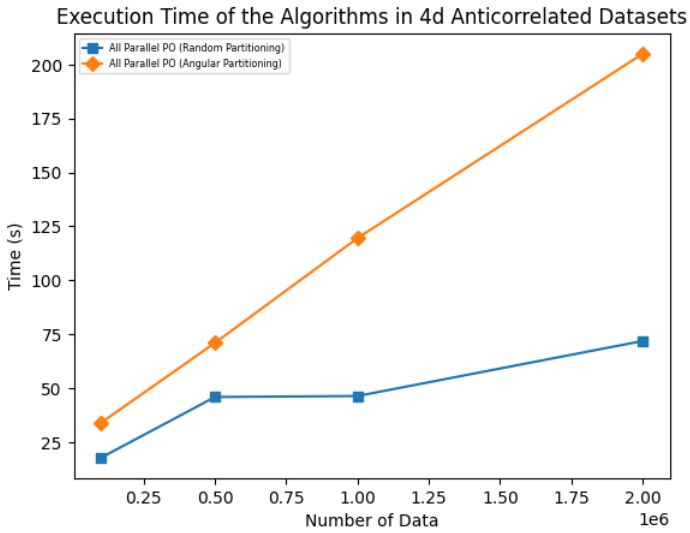


Figure 7.4.5: Execution time of All Parallel algorithms to compute PO in 4d anticorrelated datasets using the virtual machine.

Looking at figure 7.4.5, we can see that for the computation of the PO set, partitioning the data according to the position of the points in space using Angular or One-Slice partitioning does not bring any advantage. In fact, the best algorithm for computing PO as we can see from the same figure is Random Partitioning, which manages to finish executing the All Parallel algorithm in the shortest possible time. This is because partitioning with angular we have some partitions that will be heavier than others leading to a longer duration of the first parallel phase where we go to compute the local set as we can see from figure 7.3.7. Random on the other hand manages to divide the load on all partitions in the best possible way.

In this section we have seen algorithms that have allowed us to speed up the standard parallel algorithms, but as we have seen, the best of all for the computation of both the skyline and the F-Skyline operators ND and PO is the All Parallel Algorithm. In Figure 7.4.6 we will evaluate how this algorithm behaves when we increase the number of points in the 4-dimensional anticorrelated dataset to as many as 10 million points.

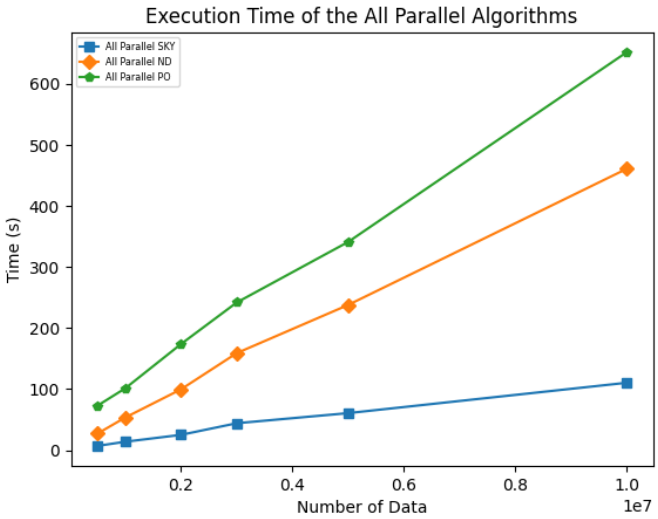


Figure 7.4.6: Execution Time of the All Parallel Algorithm for computing the Skyline, ND and PO sets using a 4d anticorrelated dataset.

# Data	500 K	1 M	2 M	3 M	5 M	10 M
# Skyline Points	4480	11619	23354	29974	39685	73245
# ND points	3307	7954	16781	20623	26996	53142
#PO points	83	117	150	196	215	285

Table 7.6: Number of Skyline, ND and PO points per cardinality of the 4d dataset.

7.5. Change the cardinality

In this section we are going to evaluate the behaviour of the parallel algorithms as the cardinality of the anticorrelated dataset used changes. The dimensionality of the dataset will remain fixed at 4 and we will only change its cardinality.

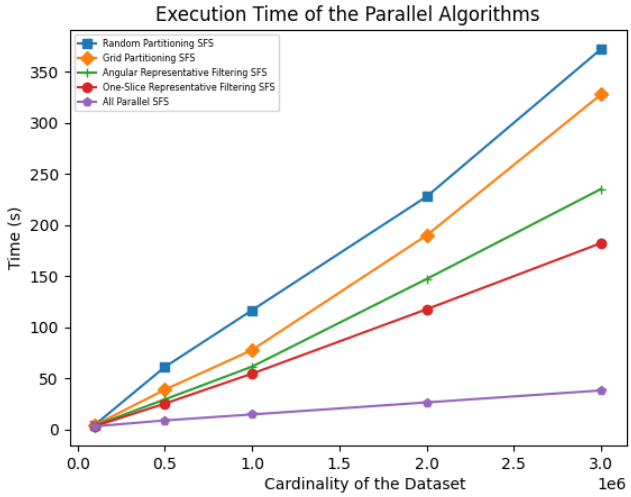


Figure 7.5.1: Execution time of the Parallel Algorithms to compute the Skyline using SFS changing the cardinality of the 4d anticorrelated datasets.

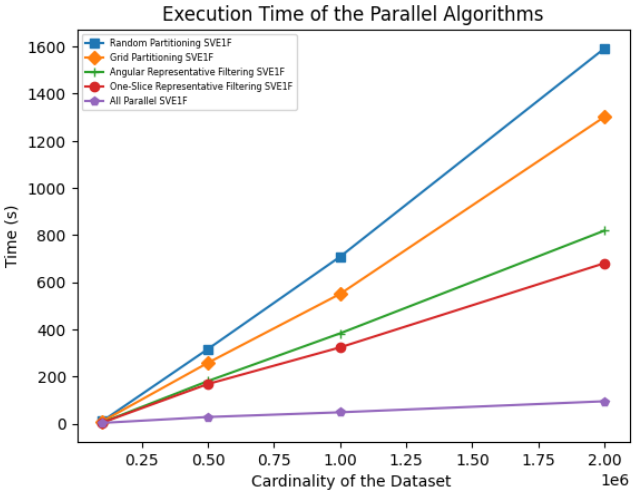


Figure 7.5.2: Execution time of the Parallel Algorithms to compute ND using SVE1F changing the cardinality of the 4d anticorrelated datasets.

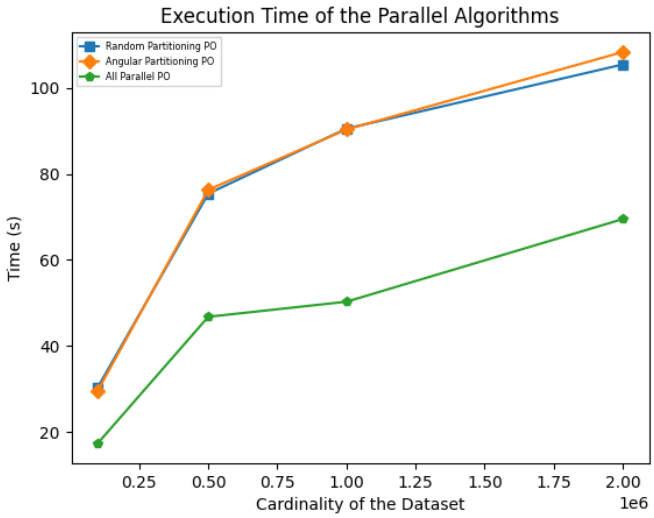


Figure 7.5.3: Execution time of the Parallel Algorithms to compute PO changing the cardinality of the 4d anticorrelated datasets.

As we can see from the figures 7.5.1, 7.5.2 and 7.5.3, changing the cardinality of the dataset increases the execution time for the skyline set computation and for ND and PO and we have more or less linear trend for the computation of all three sets. As the cardinality of the dataset increases, so does the size of the skyline, ND and PO set.

Once again, we can see that the best algorithm is the All Parallel for the computation of all three sets, because it makes the best use of parallelization. Instead, among the algorithms with sequential final part, we have that One-slice partitioning performs best for the computation of Skyline and ND sets, while for the computation of PO both random partitioning and angular partitioning perform almost equally well.

7.6. Change the number of dimensions

In this section we are going to change the number of dimensions of the dataset to see how the parallel algorithms for skyline, ND and PO set computation perform. We will only look at the algorithms that have been the best so far which are Angular Representative filtering, One-Slice Representative Filtering and All Parallel Algorithm. For the experiment we are going to use generated datasets anticorrelated by 1 million points to which we are going to change the number of dimensions.

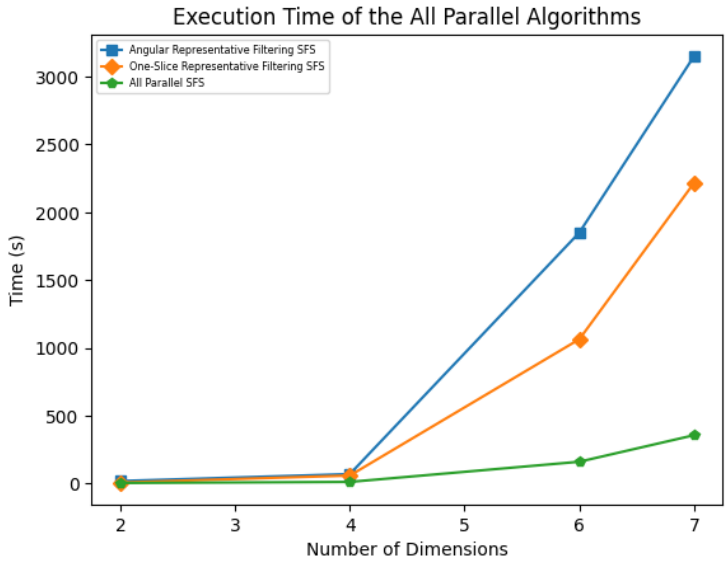


Figure 7.6.1: Execution Time of the Improved Parallel Algorithms using SFS in an anticorrelated dataset of 1 million points.

# dimensions	2	4	6	7
# Skyline Points	1830	16711	70182	120769
# ND points	6	11621	58123	106366
#PO points	5	169	942	2152

Table 7.7: Number of Skyline, ND and PO Points per dimension.

As we can see from graph 7.6.1, increasing the number of dimensions considerably increases the execution time of all the algorithms, because increasing the dimensionality of the dataset consequently increases the number of points that will be

part of the Skyline set (see table 7.7) and therefore the duration of the algorithms to find them since they will have to make many more comparisons. All three algorithms show an exponential trend as the dimensions increase. As always, the best algorithm is the All Parallel algorithm because as the most expensive part is the final sequential part, which with this algorithm we have eliminated and we are able to exploit all the power of the virtual machine with its 30 cores.

Since angular and one-slice algorithms for large dimensions take too long due to the final sequential phase, for the computation of ND and PO we will only evaluate how the All Parallel algorithm performs.

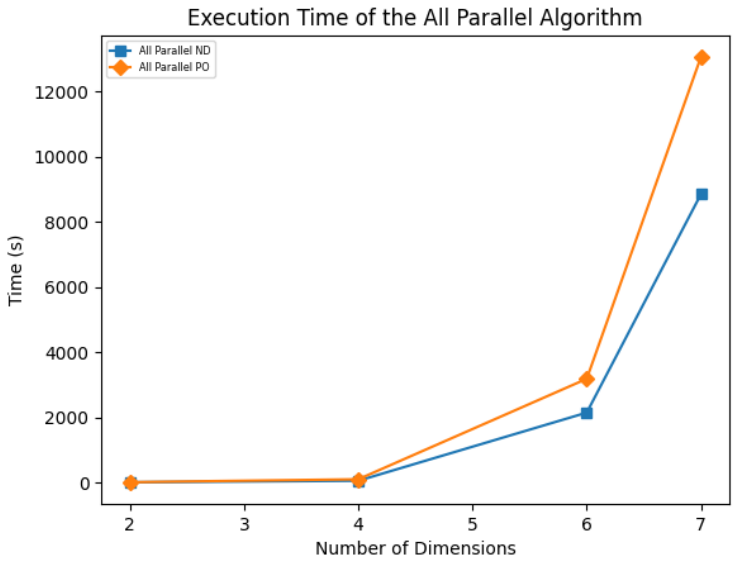


Figure 7.6.2: Execution time of the All Parallel Algorithm for the computation of ND and PO using an anticorrelated dataset of 1 million points.

7.7. Change the number of partitions

In this section, we will see how changing the number of partitions impacts the parallel algorithms for the computation of the skyline and ND sets. For the experiments, we will use an anti-correlated 4d dataset generated with 1 million points.

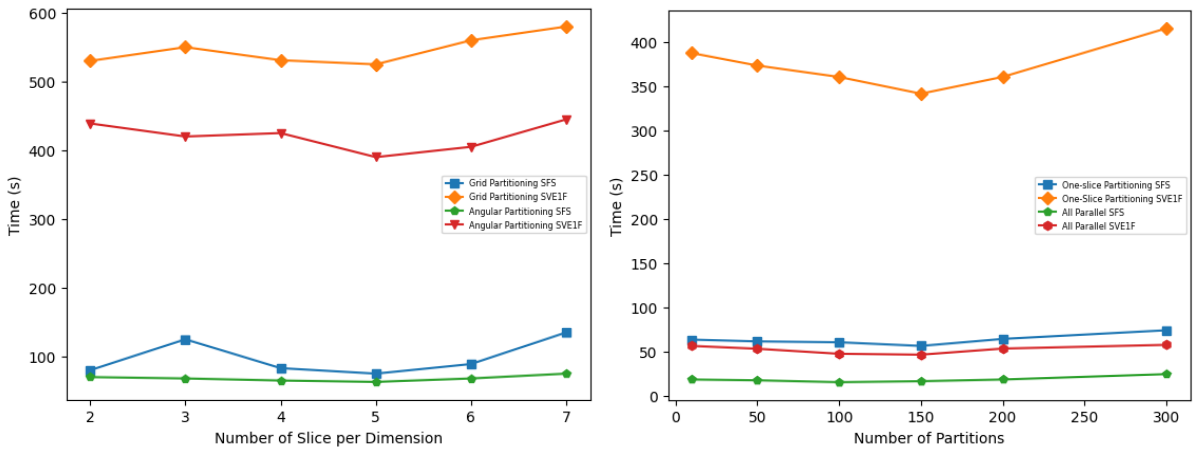


Figure 7.7.1: Execution Time of the Parallel Algorithms using SFS and SVE1F with an anticorrelated 4d dataset of 1 million points changing the number of partitions.

From Figure 7.7.1 we can see how changing the number of partitions affects the duration of the algorithms. The first graph shows how the grid and angular algorithms vary as the number of slices per dimension changes. In the case of Grid Partitioning, the number of partitions will be equal to n^d whereas for Angular Partitioning it will be equal to n^{d-1}. The first thing we notice is that for both the skyline computation with SFS and the computation of ND set with SVE1F, when it comes to algorithms with final sequential part, it is never advisable to choose a number of partitions that is too high, because it is true that the parallel part will be faster, but the number of points in the returned local set will be greater, affecting the duration of the final sequential part. Different reasoning with the All Parallel algorithm, which by increasing the number of partitions is able to better parallelize the computation.

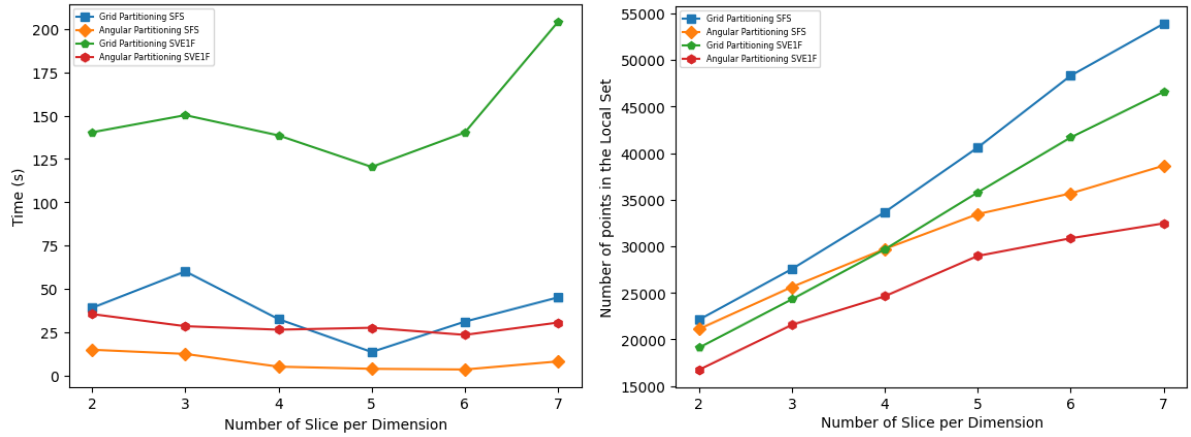


Figure 7.7.2: Execution time and cardinality of the local set after the parallel phase using a 4d anticorrelated dataset of 1 million points changing the slice per dimension

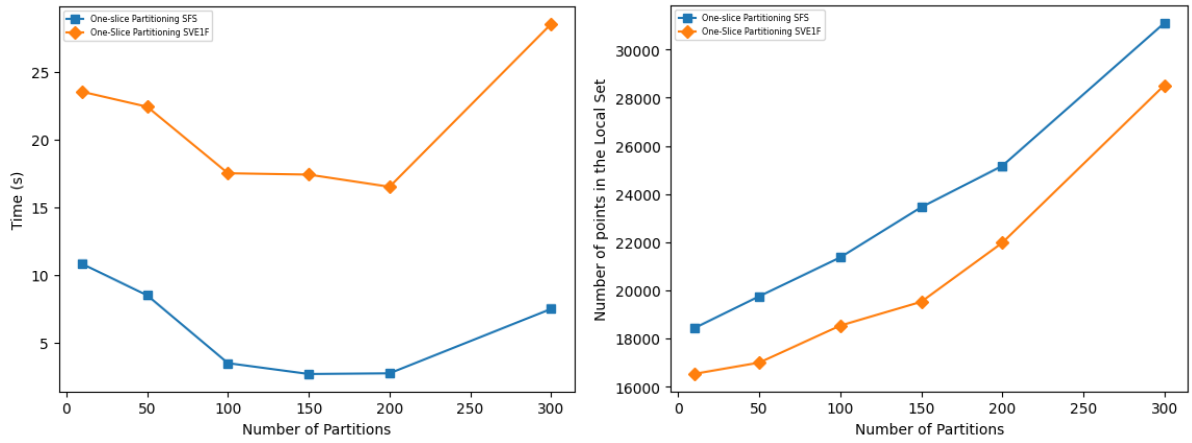


Figure 7.7.3: Execution time and cardinality of the local set after the parallel phase using a 4d anticorrelated dataset of 1 million points changing the number of partitions.

In Figures 7.7.2 and 7.7.3, we can see how changing the number of partitions reduces the duration of the algorithms in their parallel phase, but as a result, the cardinality of the local set is increased, resulting in a longer time for the computation of the global set in the final sequential phase. We need to find the right compromise, which as we can see from figures 7.7.1 in this specific case, seems to be 5 slices per dimension for Grid (5^4 partitions) and Angular (5^3 partitions) partitioning while 100 partitions for One-Slice partitioning and consequently All Parallel for bot Skyline and ND computations.

7.8. Change the number of cores

In this section, we will see how changing the number of cores used by Pyspark for parallelization impacts the parallel algorithms for the computation of the skyline, ND and PO sets. For the experiments, we will use an anti-correlated 4d dataset generated with 1 million points and we will test the behaviour of the algorithms as the cores vary from 5 to 30.

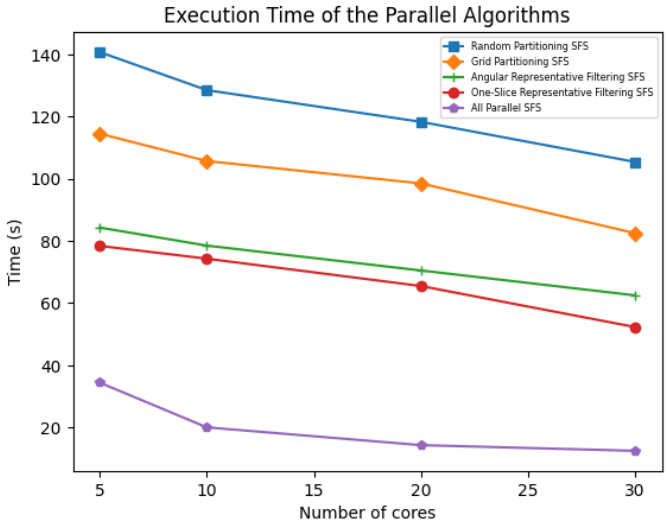


Figure 7.8.1: Execution time of the Parallel Algorithms to compute the Skyline using SFS changing the number of cores used.

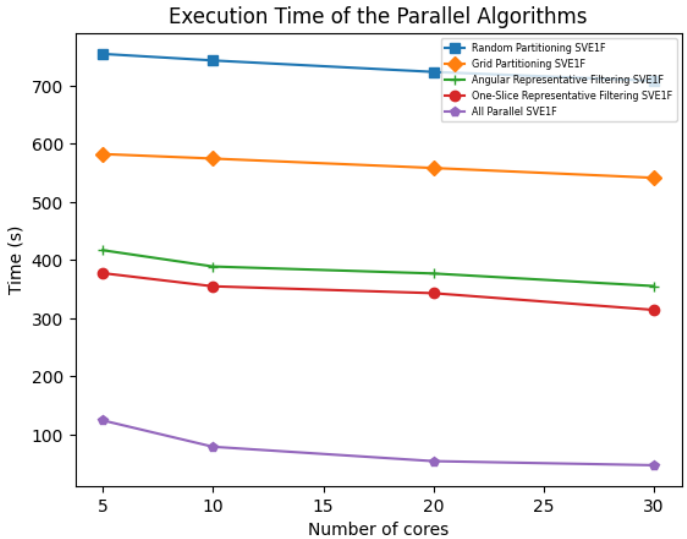


Figure 7.8.2: Execution time of the Parallel Algorithms to compute the ND set using SVE1F changing the number of cores used.

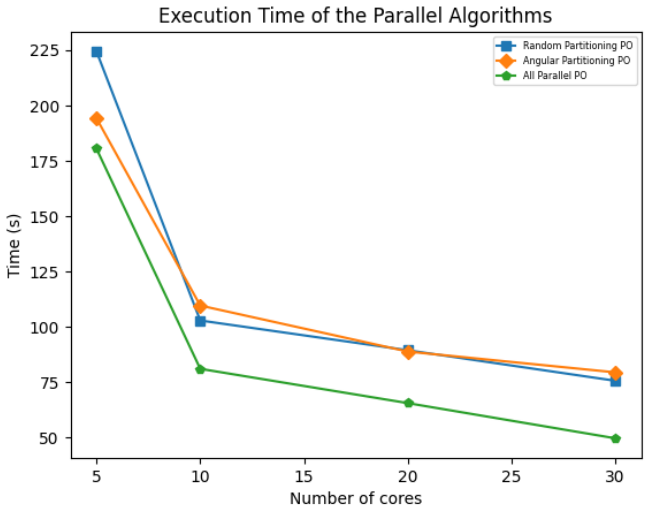


Figure 7.8.3: Execution time of the Parallel Algorithms to compute the PO set changing the number of cores used.

From the figures 7.8.1, 7.8.2 and 7.8.3, we can see that by increasing the number of cores, the execution time decreases. As for parallel algorithms with a sequential final part, we can see that the execution time decreases but not as much between the 30 cores version and the 5 cores version. This is due to the fact that the parallel part in these algorithms covers a much smaller part than the final sequential part and using more cores leads to a speeding up of the parallel part but does not affect the sequential part which will still be very long. Having said that, using more cores for these types of algorithms will speed up the parallel part a little, but the sequential part will not be affected and will not bring a big advantage in execution time.

Taking One-Slice Partitioning with representative filtering as an example, we go from 79.3 seconds of the 5-core version to 53.5 seconds of the 30-core version for SFS, thus speeding up the algorithm by a factor of 1.48. The same applies to the other algorithms and the computation of ND, where the same algorithm is speeded up by a factor of 1.3. A different case is made for PO computation, as each partition has to compute so many LPs, that the execution time of the parallel partitioning has a significant duration, thus speeding up Random Partitioning using the 30 cores compared to the 5 cores by a factor of 2.96.

Finally, the All Parallel algorithms are those that benefit most from parallelization and we can see how the execution speed decreases much more than the other algorithms as the number of cores increases. This is because being an algorithm that does everything in parallel, having more cores at work speeds up the execution time a lot. For the skyline computation we have that the execution time goes from 34.44 seconds of the 5 cores version to 14.5 of the 30 cores version, speeding up the algorithm by a factor of 2.37. For the computation of ND, on the other hand, we go from 124.56 seconds in the 5-core version to 54.4 in the 30-core version, speeding up the algorithm

by a factor of 2.28. Finally, the All Parallel for PO computation goes from 181.3 seconds for the 5-core version to 51.3 seconds for the 30-core version, speeding up the algorithm by a factor of 3.53.

8 Related work

We have seen several algorithms for skyline computation, but there are many more. One such algorithm is the Divide and Conquer (D&C) approach [5], which partitions the data space into multiple regions, computes the skyline in each region, and combines the regional skylines to obtain the final skyline. Papadias et al. [15] proposed a branch and bound algorithm for progressively outputting skyline points from a dataset indexed by an R-Tree, with a guaranteed minimum I/O cost.

There are various methods, such as F-Skyline, that extend the concept of Skyline by taking user preferences into account. For example, Prioritized Skyline (P-skyline) [reference] queries allow users to specify which attributes are more important in skyline queries. As we have seen with F-Skyline, P-Skyline also returns a smaller set compared to Skyline. The key difference with F-Skyline is that P-Skyline assumes a strict priority between attributes.

One of the earliest parallel frameworks for computing the skyline query was Apache Hadoop, which utilizes the MapReduce computational paradigm [14]. This framework breaks down every job into two tasks: the mapping task and the reducing task. In the mapping task, the framework takes the input dataset and divides it into independent portions. Then, it maps each point in the dataset into key-value pairs. The utilization of these two fundamental functionalities has led to the creation of numerous efficient distributed algorithms for skyline query computation like [16][17]. Given that MapReduce operates on disk-based data processing, as opposed to Spark's in-memory approach, we can anticipate that the same applications implemented on Spark will likely exhibit improved performance, particularly for small datasets that can be fit in memory.

In this thesis, we have applied parallel algorithms for computing Skyline and F-Skyline operators. As shown in [12], we have found that One-Dimensional Slice partitioning is the most effective method for computing the Skyline set. Moreover, in this thesis, we have demonstrated that this approach is also the best for computing the ND set, as it returns the smallest local set in the shortest amount of time. The situation is different for the PO set, where One-Dimensional Slice partitioning is not the most effective method. Instead, we have found that random partitioning performs significantly better.

An alternative method for parallel skyline computation is introduced in [6] in which the authors employ a multi-disk design with a single processor, and they leverage the parallel R-Tree. The central aim of this paper is to enhance the efficiency of eliminating non-qualifying points by accessing multiple entries from various disks concurrently. While this approach focuses on optimizing the distribution of nodes in the parallel R-Tree, the methods we have seen so far addresses the problem of data space partitioning in a parallel share-nothing architecture.

9 Conclusion and future developments

In this thesis, we introduced several parallel algorithms and applied them to the computation of Skylines and the Flexible Skyline ND and PO operators using the PySpark framework.

We saw that for both skyline computation and ND, the parallel algorithms behave more or less the same in the case of anticorrelated datasets. Random Partitioning does not perform very well despite the fact that unlike Grid and Angular Partitioning it succeeds in dividing the dataset into partitions of equal size, but the fact of joining random points does not bring any advantage, in fact the computed local set is very large, thus introducing a high overhead compared to the others in the final sequential part. Grid Partitioning, on the other hand, manages to eliminate enough points in the parallel phase by returning a smaller local set than Random Partitioning, but in anti-correlated datasets, as we have seen, it has some partitions that are much heavier than others, thus incurring a fairly large computation time for the parallel phase. Angular partitioning succeeds in overcoming the problem of Grid Partitioning by being able to divide the dataset more evenly while returning small local sets. Finally, One-Slice partitioning succeeds in overcoming the problems of Grid and Angular Partitioning by managing to partition the dataset equally between partitions and still return the smallest local set of all the parallel algorithms.

Improvements have been made to these algorithms. Grid filtering as we have seen brings no advantage in terms of execution time, because once again, the dataset being anticorrelated, grid partitioning fails to divide the space well, leaving many more points in some partitions than in others, leading to the elimination of many partitions with few points. The real improvements are found with Representative Filtering, where the algorithms manage to return a smaller local set than the standard version, thus reducing the duration of the global set computation in the final sequential part by using a few "better" points to filter the dataset before performing parallel computation.

We have seen that the results obtained are in line with the thesis [12] in which One-Slice Partitioning turns out to be the best partitioning algorithm among those proposed for skyline set computation. In this thesis, we were able to establish that it is also the best algorithm for ND set computation, being able to return the smallest local set in the shortest time. On the other hand, different considerations apply to the computation of PO, which is much more efficient when starting from the ND set and partitioning the space randomly or using Angular partitioning, preferring the former as it succeeds

in dividing the set more evenly between the partitions without overloading them too much since the computation of LP is very onerous.

Finally, we have seen that the best algorithm is the All Parallel since it manages to eliminate the final sequential part, which is the longest. The algorithm performs best when using the virtual machine with 30 cores, managing to finish execution 4.29 times earlier than the same version with the final sequential part for the skyline computation, 6.5 times earlier for ND and 2.1 times earlier for PO.

Bibliography

[1] M. Ehrgott. Multicriteria Optimization. Springer, 2nd edition, 2005.

[2] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. 40, 4 (2008).

[3] Paolo Ciaccia and Davide Martinenghi. 2020. Flexible Skylines: Dominance for Arbitrary Sets of Monotone Functions. ACM Trans. Database Syst. 45, 4, Article 18 (December 2020), 45 pages.

[4] https://spark.apache.org/docs/latest/api/python/

[5] S. Börzsönyi, D. Kossmann and K. Stocker, "The skyline operator," Proceedings of International Conference on Data Engineering (ICDE), p. pp. 421–430, 2001.

[6] Y. Gao, G. Chen, L. Chen, and C. Chen. Parallelizing progressive computation for skyline queries in multi-disk environment. In Proc. Int. Conf. of Database and Expert Systems Applications (DEXA), pages 697–706, 2006.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In IEEE Computer Society, pages 717–719. IEEE Computer Society, 2003.

[8] I. Bartolini, P. Ciaccia, and M. Patella. Efficient Sort-Based Skyline Evaluation. ACM Transactions on Database Systems, 33(4), 2008.

[9] A. Cosgaya-Lozano, A. Rau-Chaplin and N. Zeh, Parallel computation of skyline queries. In Proc. of Int. Symp. on High Performance Computing Systems.

[10] K. Mullesgaard, J. L. Pederseny, H. Lu and Y. Zhou, "Efficient Skyline Computation in MapReduce," Advances in Database Technology - EDBT 2014, 17th International Conference on Extending Database Technology, pp. pp. 37-48, 2014.

[11] A. Vlachou, C. Doulkeridis and Y. Kotidis, "Angle-based Space Partitioning for Efficient Parallel Skyline Computation", SIGMOD, 08, June 9–12, 2008, Vancouver, BC, Canada

[12] Etion Pinari, "Parallel Implementations of the Skyline Query using PySpark", 2022.

[13] Apache Spark: "Spark overview" https://spark.apache.org/docs/latest/

[14] Map Reduce: “Map Reduce Tutorial” https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[15] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. ACM Transactions on Database Systems (TODS), 30(1):41–82, 2005.

[16] L. Chen, K. Hwang and J. Wu, "MapReduce Skyline Query Processing with a New Angular Partitioning Approach," IPDPS Workshops, pp. 2262-2270, 2012.

[17] J.-L. Koh, C.-C. Chen, C.-Y. Chan and A. L. P. Chen, "MapReduce skyline query processing with partitioning and distributed dominance tests," 2017.

[18] Github repository with the implemented code:

“https://github.com/emilio99-del/Master-Thesis”

List of Figures

List of Tables

Acknowledgments

Prima di tutto ringrazio il mio relatore, il professor Davide Martinenghi il quale mi ha seguito per tutti questi mesi in cui ho svolto il lavoro di tesi, è riuscito a guidarmi con i suoi consigli ed è sempre stato disponibile nel momento del bisogno.

Ringrazio i miei genitori Damiano e Miranda, che non mi hanno mai fatto mancare il loro appoggio e il loro amore anche a chilometri di distanza in tutti questi anni. Mi hanno reso l'uomo che sono oggi e insegnato cosa vuol dire essere supportato, ed è grazie a loro che non ho mai dubitato di me stesso e continuerò a non farlo mai.

Ringrazio i miei fratelli Giuseppe e Raffaele, i migliori fratelli che si possano desiderare. Da fratello minore non mi hanno mai fatto mancare l'affetto e sono per me da sempre un' ispirazione e non smetterò mai di essergli grato per tutto l'amore che mi hanno dato in questi anni.

Infine ringrazio tutti i miei amici con il quale ho condiviso momenti bellissimi e mi hanno aiutato a mantenere il sorriso sulle labbra anche nei momenti più difficili.