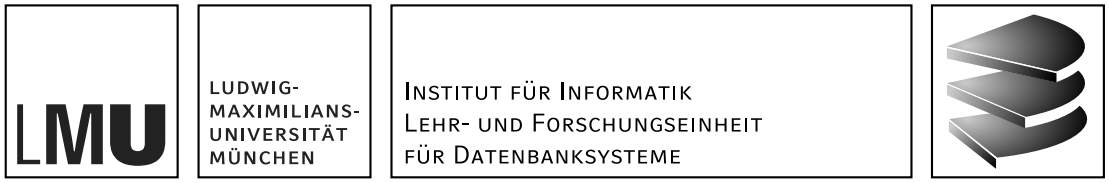

Masterarbeit
in Informatik

Distributed linear correlation cluster detection using MapReduce

Aeneas Orest Friedrich Soter Rekkas

Aufgabensteller:	Prof. Dr. Peer Kröger
Betreuer:	Daniyal Kazempour
Abgabedatum:	20. Februar 2018

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 20. Februar 2018

.....
Aeneas Orest Friedrich Soter Rekkas

Abstract

The problem of finding arbitrarily oriented linear correlated clusters in high dimensional data is solved by algorithms such as ORCLUS, 4C, and CASH. However, these algorithms encounter high execution times as the detection of clusters is computationally expensive in high dimensional data. Further in the context of BigData, with regards to one of its aspects, the volume of data which needs to be analyzed may exceed the capacities of single machines, rendering it impossible to examine the whole data. With today's broad availability of cloud computing, it is possible to deploy parallelized computer programs easily and efficiently. Thus, finding parallelization strategies for the aforementioned algorithms is a promising approach for reducing execution time and making it possible to analyze vast amounts of data. In this paper, we describe and implement two parallelized adaptations of CASH called CASH-MR and BRELMAR-CASH. Our experiments show that both algorithms achieve significant parallel scalability with robust speedup ratio and parallel efficiency. We conclude that distributed linear correlation cluster detection is solvable with MapReduce, and hence applicable in the context of BigData.

Contents

Chapter 1

Introduction

Clustering is a data mining technique in the area of Knowledge Discovery in Databases (KDD), which is the "process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data" [Zim08]. Clustering algorithms partition data points into groups based on their similarity [Bec05]. Widely adopted clustering algorithms such as k-means and k-median partition data given a predetermined number of clusters. Determining said number is non-trivial and requires multiple executions. Correlation Clustering [DEFI06], introduced by Bansal, Blum, and Chawla (2006), partitions data points into the optimal number of clusters without prior specification of cluster count.

Global Correlation Clustering Based on the Hough Transform (CASH) belongs to correlation clustering algorithms and specifically to arbitrarily oriented subspace clustering algorithms. Subspace clustering has gained considerable popularity over the past decade, as full-space (conventional) clustering algorithms are unable to find meaningful patterns in datasets of higher dimensionality. Features of high dimensional datasets are frequently noisy, have correlations amongst one another, and only a few of them attribute to the cluster structure. Subspace clustering avoids these issues and is capable of detecting similarities in high dimensional data. Results can be used for pattern recognition, data compression, similarity search and other important tasks. [ABD+08]

Subspace clustering algorithms are grouped into two categories, those for axis-parallel subspaces, and those for arbitrarily oriented subspaces. CASH belongs to the latter. A general issue with subspace clustering is the exponentially expensive exploration of high dimensional spaces with regards to run-time. Applying these algorithms to large databases with many features is therefore infeasible. Due to a lack of parallelization strategies, it is currently not possible to leverage increased availability and reduced costs of cloud and high-performance computing. [ABD+08]

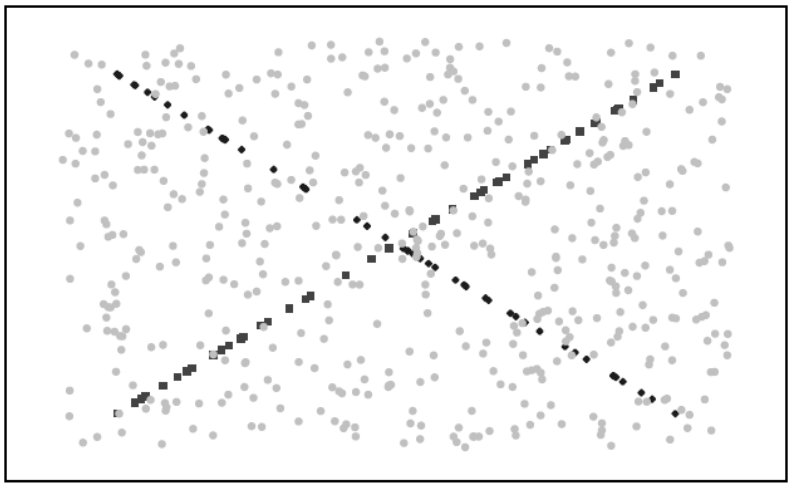

Figure 1.1: Dataset with two non-dense general subspace clusters in a noisy environment. [ABD+08]

In this paper, we explored the hypothesis that linear correlation cluster detection is parallelizable using the MapReduce framework. To prove this hypothesis, we developed a MapReduce Distributed Global Correlation Clustering Based on the Hough Transform (CASH-MR), which is a close adaptation of the CASH algorithm. We further explored the applicability of Bresenham's line algorithm in relation to distributed systems and correlation cluster detection and defined a Map-Reduce Distributed Global Correlation Clustering Based on the Hough Transform and Bresenham's Line Drawing Algorithm (BRELMAR-CASH). We implemented both algorithms using the Apache Flink framework and evaluated execution times with up to eight worker nodes. We proved that both algorithms achieve robust speedup ratio and significant parallel efficiency when executed on two or more workers. Specifically, CASH-MR achieves an efficiency of up to 1.71 and BRELMAR-CASH of up to 0.88. Additionally, we observed that BRELMAR-CASH is capable of solving specific clustering tasks in $O(n)$, for which CASH and CASH-MR require $O(n^2)$.

In section one we discuss related work, specifically the CASH algorithm. In section two we describe our methodology and preliminaries such as MapReduce, Apache Flink, the Hough Transform and Bresenham's line algorithm. In section three we explore and define possibilities for applying the MapReduce framework to CASH. In section four we define an adaptation of CASH-MR using Bresenham's line algorithm (BRELMAR-CASH). In section five we evaluate efficiency and speedup of both algorithms, explore the run-time complexity of various parametrizations and finally compare our algorithms with one another. In section six we draw a conclusion and outline possibilities for future work.

Chapter 2

Related Work

2.1 CASH: Global Correlation Clustering Based on the Hough Transform

Global Correlation Clustering Based on the Hough Transform [ABD+08], introduced by Atchtert, Böm et al. (2008), is capable of finding the optimal number of arbitrarily oriented subspace clusters in high dimensional data, identifying cluster hierarchies within the data, and reducing high dimensional data to lower dimensional principal components.

CASH is inspired by the Hough Transform, explained in detail in Section 3.1.3. In summary, the Hough Transform maps each point of a two-dimensional dataset, called picture space, onto a function in the parameter space. Each point $p = (x_p, y_p)$ of the picture space is mapped as slope $-x_p$ and intercept y_p - in the parameter space - to the linear function f_p as represented by $t = -m \times xp + yp$. Thus, f_p represents all linear segments passing through p. Function f_p is referred to as parameterization function. Figure 3.4 shows three points p_1, p_2, p_3 in the picture space that are mapped to their parameterization functions $f_{p_1}, f_{p_2}, f_{p_3}$ in parameter space P.

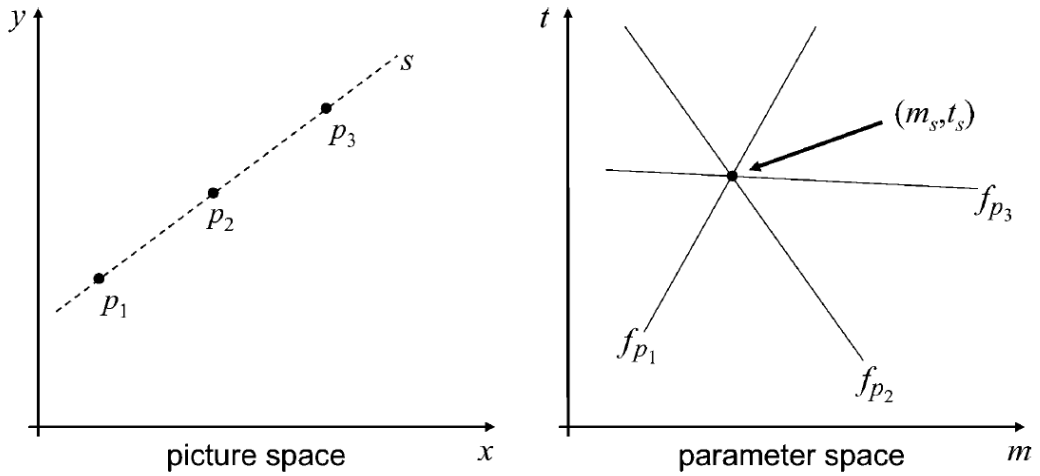

Figure 2.1: Hough transform from picture space to parameter space using slope and intercept parameters. [ABD+08]

Common line s from Figure 3.4 is defined by intersection (m_s, t_s) with slope m_s and intercept t_s . Intersection (m_s, t_s) is independent from the distance amongst points p_1, p_2, p_3, which greatly increases the algorithm's effectiveness in high dimensional data.

That intersection represents a common $d-1$-dimensional hyperplane, on which the points in the picture space are located. In most applications however, these hyperplanes are identified through dense regions (Figure 2.3), rather than intersections. CASH discretizes the parameter space by some grid and searches for dense grid cells. A grid cell is considered dense if at least *min_points* parameterization functions intersect with it. As both slope and intercept may assume infinite values in the Cartesian coordinate system, both variables are considered unbounded. Finding dense grid cells in an infinite search space is not scalable, and predefining boundaries unpractical. To resolve this issue, the authors propose using spherical (polar) coordinates in the parameter space. The linear segment s is thus given by the angle α_s of its normal and its radius δ_s from the origin (Figure 2.2). Angle α_s is naturally bound by $[0; \pi)$ and thus global extrema, which constitute the boundaries of radius δ_s, exist. CASH computes these extrema which define the search space.

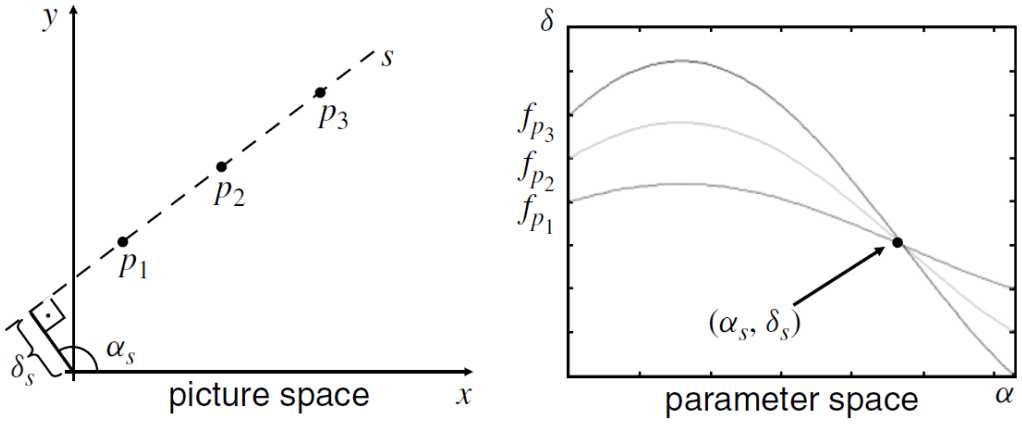

Figure 2.2: Hough transform from picture space to parameter space using angle and radius parameters. [ABD+08]

The boundaries of a grid cell must be sufficiently small to define clusters free from noise and without interference from functions belonging to other clusters. However, finding dense grid cells with small intervals requires exponential time and space complex in high-dimensional data.

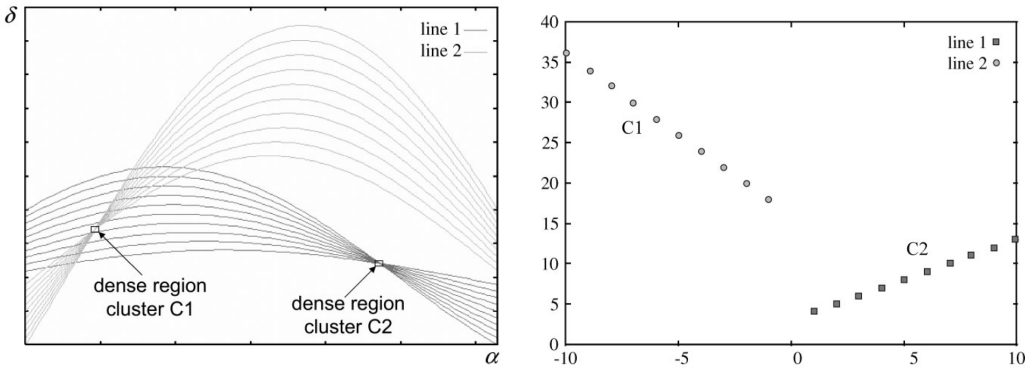

Figure 2.3: Dense regions in parameter space capturing two lines in picture space. [ABD+08]

CASH avoids this issue with a recursive search strategy - in [KMKS17] referred to as spatial descent - which identifies regions of interest. A dense grid cell is divided along one of the grid's axis "successively in a static order given by $\delta, \alpha_1, ..., \alpha_d - 1$" [ABD+08]. The number of parameterized functions crossing the two resulting cells are counted, and this process is repeated for those cells where the *min_points* is exceeded. If none of the resulting cells satisfy this condition, the search path is discontinued. After a predefined recursion depth

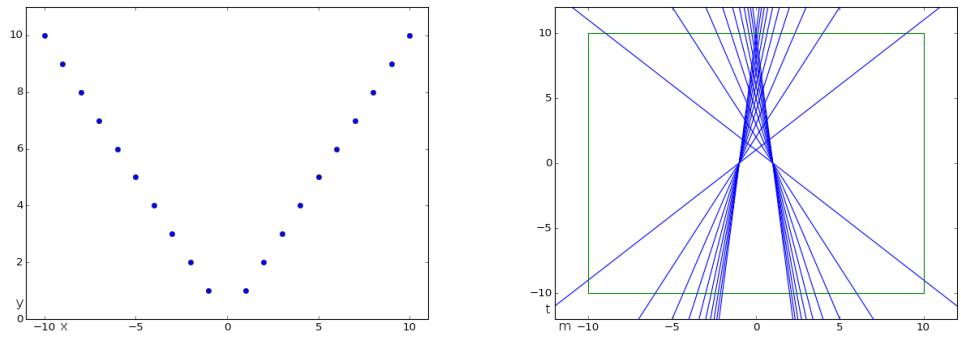

Figure 2.4: Exemplary spatial descent (right picture, first iteration) on a linear dataset (left) in the Cartesian coordinate system.

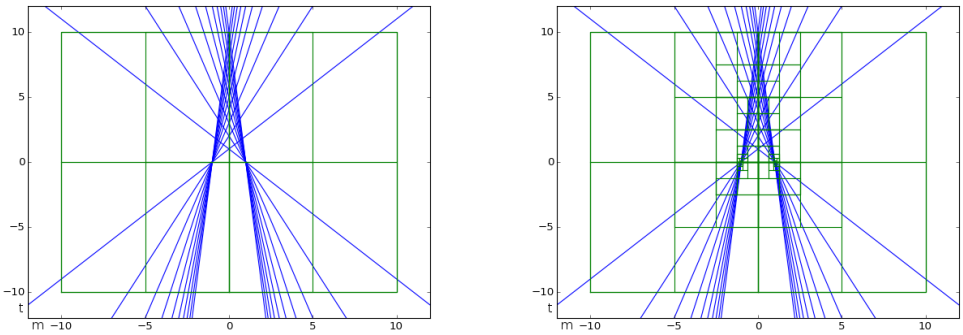

Figure 2.5: Iterations 2 (left) and 15 (right) of the search strategy in the Cartesian coordinate system.

max_recursion - also referred to as the number of successive splits - a dense grid cell defines a sufficiently small $(d-1)$-dimensional hyperplane which is considered a subspace cluster.

If the algorithm is applied to a d-dimensional dataset with $d > 3$, *max_recursion* is reached, dense grid cells are used as a new data space and the full CASH algorithm is applied recursively, until no subspace cluster is found or $d > 2$. This process is called recursive descent.

The algorithm's output, without any additional processing, is considered a set of arbitrarily oriented subspace clusters and their hierarchies.

2.2 4C: Computing Correlation Connected Clusters

4C (Computing Correlation Connected Clusters) [BKKZ04] combines Principal Component Analysis (PCA) with density-based clustering (DBSCAN) [EKSX96] to identify clusters of correlation connected objects. PCA extracts similarity patterns from data tables [AX10] by computing a set of new principal components (orthogonal variables). PCA finds correlations in global, linear, uniform, noiseless vectors [AX10] - but has difficulties finding correlations that are not global, a task common in various disciplines such as medical diagnosis and molecular biology [BKKZ04].For each point of a dataset, DBSCAN evaluates the number of points (density) within a given radius of the said point in the Euclidean space. If the density in the neighborhood exceeds a certain threshold, those points are considered to be part of a cluster, and noise otherwise [EKSX96]. By Combining DBSCAN and PCA, 4C uses the Euclidean neighborhood of a point to find the subspace features that cluster said point best. As such, 4C is is robust against noise, but also assumes a dense clustering structure in the entire feature space and will fail to produce meaningful results otherwise.

2.3 ORCLUS: Arbitrarily Oriented Projected Cluster Generation

ORCLUS uses a combination of PCA and k-means, a partitional clustering algorithm using seeds to partition data points [JD88]. ORCLUS iteratively eliminates each clusters' most sparse subspaces and projects data points into subspaces with the greatest similarity for these points. However, ORCLUS will fail to detect meaningful patterns when the local neighborhood of initial clusters is noisy or the clustering structure is too sparse. [AY00]

Chapter 3

Methodology

To test the hypothesis that the MapReduce framework is applicable to linear correlation cluster detection and specifically CASH, an adaptation - called CASH-MR - of the algorithm is formalized and implemented in Python and Apache Flink (Java). Additionally, a novel approach called BRELMAR-CASH is defined, implemented and evaluated. For brevity, we implement both algorithms for 2-dimensional data only and use the Cartesian coordinate system instead of the polar coordinate system. We show that our results are applicable to the original CASH algorithm, and point to further possibilities for parallelization of the CASH algorithm when being applied to datasets of higher dimensionality.

First, we formalize possibilities for applying the MapReduce Framework to CASH and introduce a CASH adaptation using Bresenham's line algorithm. Second, we implement a proof of concept for CASH-MR and BRELMAR-CASH using the Python programming language. Third, we port the Python implementations to the Apache Flink Java framework. Finally, we evaluate both algorithms using a dockerized Apache Flink cluster, a range of different input parameters and synthetic datasets of various shapes (linear, scattered-linear, and random).

3.1 Preliminaries

3.1.1 MapReduce

MapReduce is a programming model developed by Alphabet Inc. designed to process computation on very large datasets. It uses map and reduce/fold primitives available in most functional programming languages such as SML, Lisp, and widely adopted languages such as JavaScript. [DG08] The MapReduce environment consists of one "master" node and several "worker" nodes. The

MapReduce system takes care of partitioning input data, scheduling, shuffling, managing inter-machine communication and handling slow and failing worker nodes. A master node orchestrates the work of worker nodes. Worker nodes apply user-defined map and reduce functions onto a partition of the input data.

The MapReduce workflow has six steps: [SRB+18]

1. The MapReduce environment is set up on a cluster of physical or virtual machines.
2. The master node splits input data into C chunks (smaller groups of data) and assigns these chunks to a free worker node.
3. Each worker node applies the map function to the assigned chunk. The resulting key-value pairs are written to local disk and their location is passed to the master node.
4. The master node notifies free worker nodes. The nodes use remote procedure calls to read key-value pairs belonging to the same key.
5. The worker nodes apply the reduce function on the sorted intermediate key-value pairs.
6. The results are collected by the master node and returned to the user.

The map function takes a dataset and a user-defined function that iterates over each key-value pair of the dataset and transforms the dataset in accordance with the MapReduce algorithm: [SRB+18]

```
map (in_key, in_value) -> list (out_key, intermediate_value)
```

The reduce function ”combines all intermediate values for a particular key” and ”produces a set of merged output values”: [SRB+18]

```
reduce (out_key, list(intermediate_value)) -> list (out_value)
```

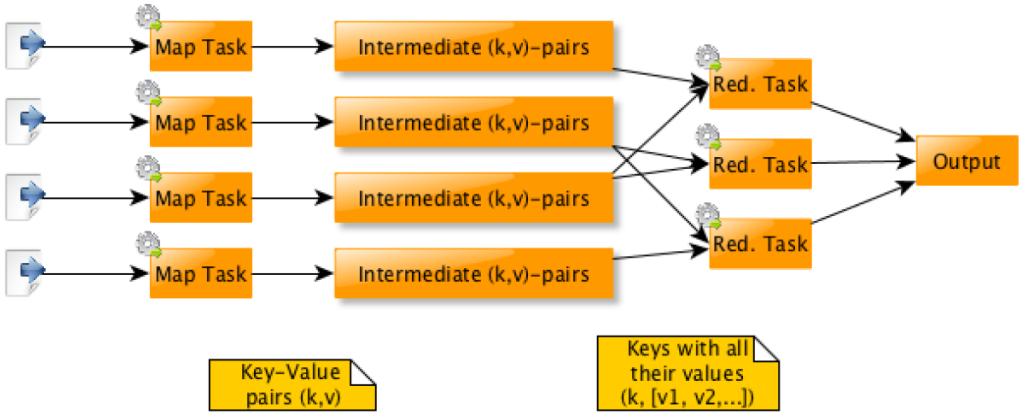

Figure 3.1: High-level MapReduce diagram. [SRB+18]

3.1.2 Apache Flink

Apache Flink "is an open-source system for processing streaming and batch data." [CKE+15]

The software has two types of execution models, streaming and batch mode [Fou18]. This paper describes and uses the batch mode scenario only.

Flink programs are regular programs that implement transformations on distributed collections, for example filtering, mapping, joining, grouping, aggregating. Collections are initially created by reading from files, or from local, in-memory collections. Results are returned via sinks, which may, for example, write the data to (distributed) files, or to standard output. Flink programs run in a variety of contexts, standalone, or embedded in other programs. The execution can happen in a local Java Virtual Machine, or on clusters of many machines. [Fou18]

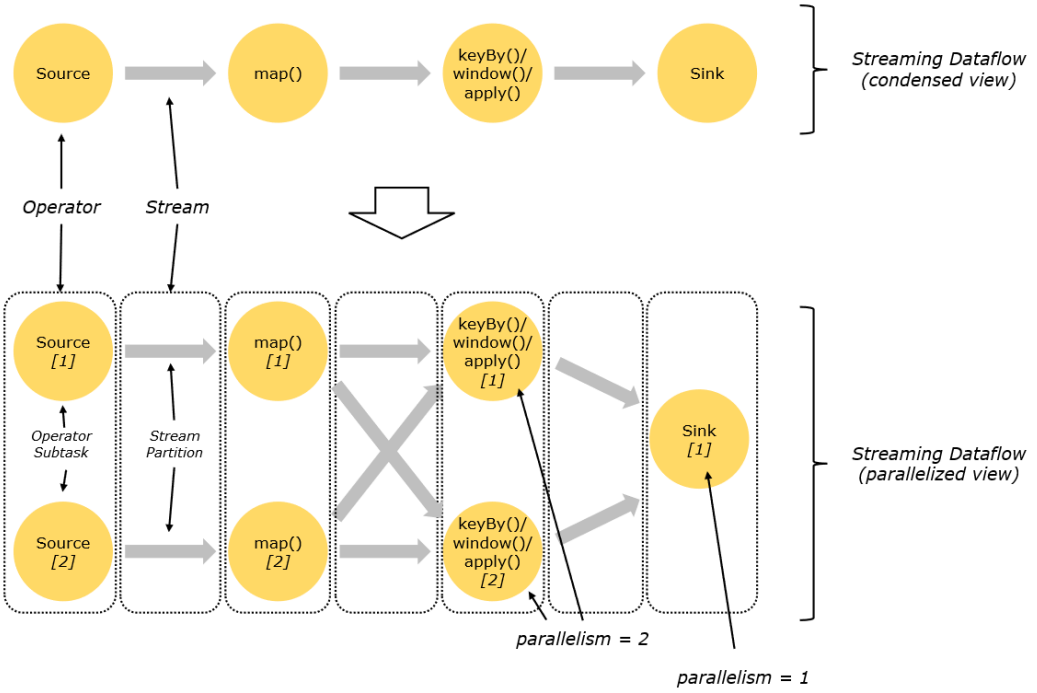

Figure 3.2: Parallel Dataflows for Batch and Stream Programs. [Fou18]

In Java, initializing a Flink program starts with setting up an *ExecutionEnvironment*:

```
final ExecutionEnvironment env = ExecutionEnvironment
  .getExecutionEnvironment();
```

Setting the parallelism of the Flink program is done with *setParallelism*:

```
env.setParallelism(parallelism);
```

Creating a data source requires a reference of type *ExecutionEnvironment*:

```
DataSet<String> data = env.fromElements(
  "Who's there?",
  "I think I hear them. Stand, ho! Who's there?"
);
```

In this work *map*, *reduce*, *groupBy*, and *flatMap* are used primarily. The following sections explain their usage with example in the context of Apache Flink.

3.1.2.1 Map Transformation

A user-defined map function is applied on each element of a *DataSet*. For each element, exactly one element must be returned by the function. [Fou18] The following example duplicates each string in *DataSet* data.

```
DataSet<String> mappedData = data
  .map(word -> data + data);
```

This transformation implements the Map step from MapReduce.

3.1.2.2 FlatMap Transformation

FlatMap applies a user-defined flat-map function on each element of a *DataSet*. Contrary to the Map transformation, the FlatMap function can return an arbitrary number of result elements for each input element. [Fou18]

```
public class Tokenizer implements FlatMapFunction<String, String
    ↪ > {
 @Override
 public void flatMap(String value, Collector<String> out) {
  for (String token : value.split("\\W")) {
   out.collect(token);
  }
 }
}

// [...]
DataSet<String> textLines = // [...]
DataSet<String> words = textLines.flatMap(new Tokenizer());
```

3.1.2.3 GroupBy Transformation

Groups a dataset based on a given key, which can be ”specified as position keys, expression keys, and key selector functions.” [Fou18]

```
DataSet<WC> wordCounts = words
  // DataSet grouping on field "word"
  .groupBy("word")
```

This transformation implements the *GroupBy* step from MapReduce.

3.1.2.4 Reduce Transformation

The Reduce transformation combines a group of elements into a single element by repeatedly combining two elements into one. Reduce may be applied on a full dataset or on a grouped dataset. [Fou18]

```
data.reduce(new ReduceFunction<Integer> {
 public Integer reduce(Integer a, Integer b) { return a + b; }
});
```

This transformation implements the *Reduce* step from MapReduce.

3.1.2.5 Data Serialization

Passing variables to user-defined transformation functions is possible as long as they are of a simple type, or implement *Serializable*. Complex types, such as *DataSet* and *ExecutionEnvironment*, do not implement *Serializable*. It is thus not possible pass either of the two to a Flink function.

```
// Not possible
ExecutionEnvironment env = //...
DataSet<String> mappedData = data.map(
  word -> env.FromCollection(data + data).map( /* ... */ )
);
```

This limitation will impact the applicability of Flink's parallelization strategies to the spatial descent as explained in Section 4.1.2 and others.

3.1.3 Hough-Transform

The Hough Transform is widely used in computer graphics to extract features from digital images. The classical Hough Transform identifies line segments in images and has later been extended for identification of arbitrary shapes

such as circles and ellipses. [DH72] The key idea of the Hough Transform is to "map each point of a two-dimensional picture (or data space D) such as a pixel onto a set of points (e.g. a line) in a parameter space P" [ABD+08]. In a Cartesian coordinate system, a linear segment s is generally represented by the line equation

$$y = m_s \times x + t_s$$

where m_s is the slope of s, and t_s its intercept. By reformulating the line equation to

$$t_s = -m_s \times x + y$$

and using m and t as the axes of the parameter space, each two-dimensional picture point $p = (x_p, y_p) \in D$ is mapped on a line f_p with slope $-x_p$ and intercept $-y_p$ in the parameter space as stated in [ABD+08].

Intersections $(m_i, t_i) \in P$ from several linear segments $f_{p1}, f_{p2}, ..., f_{pk}$ in the parameter space P indicate that points $p_1, p_2, ..., p_k \in D$ have a common line passing through each of them. The line's formula in the picture space is:

$$y = m_i \times x + t_i$$

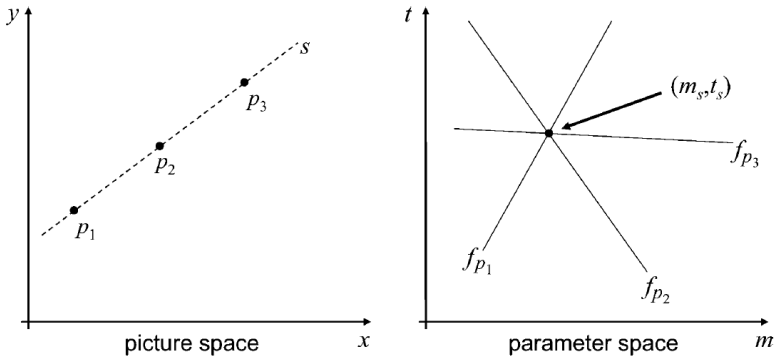

Figure 3.3: Hough transform from picture space to parameter space using slope and intercept parameters. [ABD+08]

3.1.4 Bresenham's Line Algorithm

Bresenham's line algorithm [Bre65] is a basic line drawing algorithm originally developed to draw lines on digital plotters and has since been adopted in the computer graphics community [Joy99]. The algorithm's goal is plotting line functions on computer displays and digital plotters by mapping those functions onto a set of pixels.

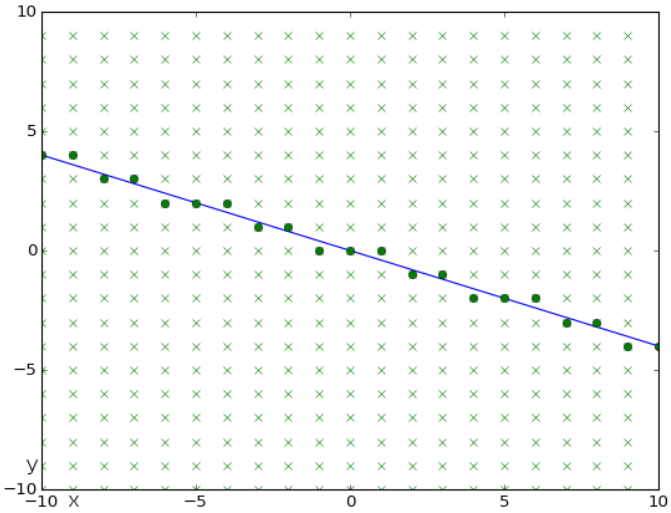

Figure 3.4: Using Bresenham's line algorithm to plot a line from start point $(-10, 4)$ to end point $(10, -4)$. The blue dots represent the plotted pixels, the blue line the original line segment.

Bresenham's algorithm defines driving axis as the "axis of control", and "the axis of maximum movement". The main loop of the algorithm uses the driving axis to increment its corresponding coordinate by one unit. Given a line with initial point (x_1, y_1) and terminal point (x_2, y_2), and their differences $\Delta y = y_2 - y_1$ and $\Delta x = x_2 - x_1$, the driving axis da is: [Joy99]

$$da = \begin{cases} x, & \text{if } |\Delta x| \geq |\Delta y| \\ y, & \text{otherwise} \end{cases}$$

Bresenham's algorithm can be described algorithmically as follows, assuming that the driving axis is the x-axis and $x_1, y_1, x_2, y_2 \in \mathbb{Z} : x_1 \neq x_2 \wedge y_1 \neq y_2 \wedge x_2 > x_1 \wedge y_2 > y_1$.

```
procedure bresenham(x1, y1, x2, y2)
  let x = x2 − x1
  let y = y2 − y1
  let m = Δy/Δx
  let j = y1
  let ϵ = m − 1

  for i = x1 to x2 − 1
```

```
    illuminate(i, j)
    if (ε ≥ 0)
      j = j + 1
      ε = ε − 1.0
    endif
    i = i + 1
    ε = ε + m
  end
end
```

The driving axis defines which axis is increased in every loop. The other axis is increased occasionally. If the driving axis is x, then x is increased by one in every loop while y is increased only if $\epsilon \geq 0$. If the driving axis is y, then y is increased by one in every loop and x only occasionally.

The example depicted in Figure 3.5 applies Bresenham's algorithm to a line from pixels $(0, 0)$ to $(5, 3)$.

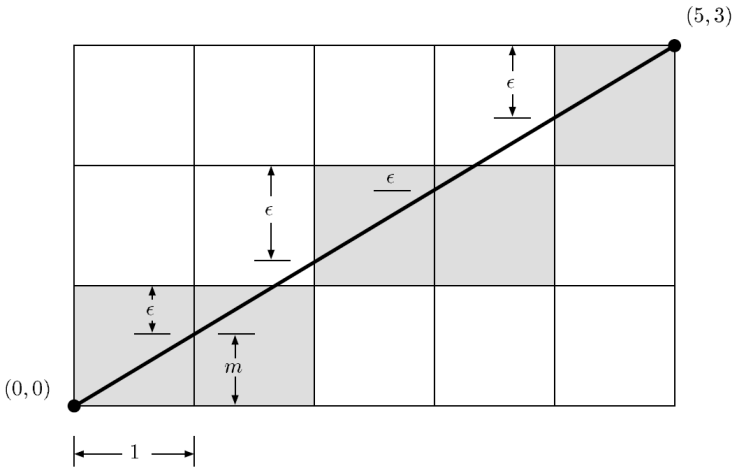

Figure 3.5: Bresenham's line drawing algorithm from $(0, 0)$ to $(5, 3)$ with error ϵ and slope m. [Joy99]

With $x_1 = 0, y_1 = 0, x_2 = 5, y_2 = 3$, slope $m = \frac{3}{5} = 0.6$ and $\epsilon = m - 1 = -0.4$ are set. The first iteration begins with point $(0, 0)$ and illuminates that pixel. With the driving axis being x, the x axis is increased by one and ϵ is set to the new value of $\epsilon = -0.4 + m = -0.4 + 0.6 = 0.2$. The second iteration illuminates pixel $(1, 0)$, evaluates $\epsilon \geq 0$ to true, and increases the y axis by one while decreasing ϵ by one. Then, the x axis is increased by one and ϵ is

decreased by slope m. This continues until point $(5, 3)$ is reached. The table below shows the algorithm's progression for all steps. [Joy99]

(x, y)	$\bar{\epsilon}$	description
$(0, 0)$	-2	illuminate pixel $(0, 0)$
	1	increment ϵ by Δy
$(1, 0)$		increment x by 1
$(1, 0)$	1	illuminate pixel $(1, 0)$
		since $\bar{\epsilon} > 0$
$(1, 1)$		increment y by 1
	-4	decrement $\bar{\epsilon}$ by 5
	-1	increment ϵ by Δy
$(2, 1)$		increment x by 1
$(2, 1)$	-1	illuminate pixel $(2, 1)$
	2	increment ϵ by Δy
$(3, 1)$		increment x by 1
$(3, 1)$	2	illuminate pixel $(3, 1)$
		since $\bar{\epsilon} > 0$
$(3, 2)$		increment y by 1
	-3	decrement $\bar{\epsilon}$ by 5
	0	increment ϵ by Δy
$(4, 2)$		increment x by 1
$(4, 2)$	0	illuminate pixel $(4, 2)$

Figure 3.6: Each iteration of Bresenham's algorithm and its determining variables. [Joy99]

Finding an implementation of Bresenham's line algorithm which works with any driving axis, line segments with slope $m = 0$, and line segments with slope $m = \infty$ was challenging. We settled with an implementation authored by Wojciech Mula[1] for our work.

[1] `http://wm.ite.pl/articles/bresenham-demo-line.html` (accessed 2018/2/13)

Chapter 4

CASH-MR: A Map-Reduce Distributed Global Correlation Clustering Based on the Hough Transform

Parallelization of knowledge discovery algorithms is an important part in processing large datasets. Broad availability of cloud computing and frameworks, such as MapReduce, Apache Spark, and Apache Flink allow researchers to run parallelized computer programs with less effort than ever before.

This section explores possibilities for applying the MapReduce framework to CASH. We define a parallelizable variation of the CASH algorithm called CASH-MR and show that CASH-MR achieves significant speedup and efficiency.

For brevity the following parts of CASH have been simplified:

1. CASH is applicable to d-dimensional data with $d > 1$. Our implementation of CASH-MR works with 2-dimensional data only. Because the recursive descent is applied to regions of interest with $d > 3$ only, we do not implement the recursive descent operation. We hypothesize that the recursive descent is parallelizable as each recursive descent iteration works autonomously and is processed using a queue [ABD+08]. If that hypothesis is wrong, but CASH is generally parallelizable with MapReduce, the recursive descent would still benefit from CASH-MR's speedup and efficiency.

2. CASH uses a grid-based search strategy to identify regions of interest. Parametrized functions with slope m and intercept t are unbounded,

resulting in a potentially infinite search space. By using spherical coordinates, global maxima and minima can be computed. Parameterized function f_p is given by angle α_s and distance δ_s from the origin, resulting in representation $x \times \cos_s + y \times \sin_s = \delta_s$. Angle α_s is restricted with $[0; \pi)$. Distance δ_s has global maxima and minima which are computed in CASH. Our implementation uses Cartesian coordinates and adds a user-defined boundary parameter which restricts slope m and intercept t. Choosing a correct boundary parameter is crucial for receiving meaningful results when using CASH-MR.

3. CASH adds a split's resulting tasks to a prioritized queue. Prioritization order is given by number of points within the hypercuboid and number of previous splits. The CASH-MR implementation does not prioritize and relies on Apache Flink for queuing and scheduling.

4. CASH keeps track of cluster hierarchy, which is a major benefit compared to other clustering algorithms. Because this features is not relevant in the context of parallelization, our implementation ignores cluster hierarchy.

The aforementioned limitations are applied to this whole chapter as well as Chapter 5.

4.1 Implementing CASH using MapReduce

A better understanding of CASH is required to identify which parts of CASH can be solved using MapReduce. The following list represents the algorithm's core steps:

1. Maximum recursion $mr = max\_recursion$, min points $mp = min\_points$, d-dimensional dataset D (in picture space) with points $p \in D$ and $d = 2$, recursion index $r \in \mathbb{Z} : r = 0$ and split index $s \in \mathbb{Z} : s = 0$, and the initial cell with boundary $\{(x_{start}, y_{start}), (x_{end}, y_{end})\} = B_{r=0s=0}$ are set by the user.

2. Each point p of dataset D (in picture space) is mapped to its parameterization function f_p in parameter space P using an adapted version of the Hough Transform algorithm. Intersections for each f_p with grid cell boundary B_{rs} are computed. Points p whose parameterization functions f_p are intersecting with B_{rs} are appended to list $G \wedge G \subseteq D$.

3. If intersection count $count(G) \geq mp$, the grid cell is considered dense. A cell is not considered dense if $count(G) < mp$. If a grid cell is not dense, the search in the current path is stopped by returning $\emptyset$.

4. The algorithm checks $r > mr$ and if true, returns B_{rs} and points p.

5. The next spatial descent iteration is prepared. Using index $di = r \pmod d$, an axis, which will be divided in two, is chosen. Assuming $di = 0$, the x axis is chosen resulting in two new boundaries $\{(x_{start}, y_{start}), (x_{end} - \frac{\Delta x}{2}, y_{end})\}$ and $\{(x_{start} + \frac{\Delta x}{2}, y_{start}), (x_{end}, y_{end})\}$. Given that $d = 2$ (2-dimensional dataset) and if $di = 1$, the y axis is split in two instead.

6. The new grid cells are passed in an iterative fashion[1] as input to step two with $r = r + 1$, split index $s \in \{1, 2\}$, and G. Each iteration returns an empty set $\emptyset$ if mp was not satisfied, and a region of interest (roi) otherwise.

Given $d = 2$, the algorithmic description of the core iteration is as follows:

```
procedure spatialDescent(D, B_rs, r, mp)
  let G = findIntersections(D, B_rs)
  if count(G) < mp
    return ∅
  endif

  if r ≥ mr
    return {B_rs, G}
  endif

  let B_r+1 = splitGridCell(B_rs)

  let rois = {}
  for s = 0 to (*count(B_r+1) − 1
    rois = rois ∪ spatialDescent(G, B_r+1s, r + 1, mp)
  endfor

  return rois
end
```

This high level description contains three potential candidates for applying MapReduce which are covered in Sections 4.1.1, 4.1.2, and 4.1.3.

We use Python (3.x) to implement a proof of concept for CASH-MR and to be able to quickly iterate and visualize each step in the algorithm. Once

[1]Iterative is here not to be confused with stateful loop function as used in the context of MapReduce

the proof of concept is implemented, the algorithm is ported to the Apache Flink Java SDK.

Various obstacles appeared during the proof of concept implementation in Python. These obstacles limited what input data and parameters could be used, but did not negatively impact the proof of concept:

1. The programming language Python has well-known limitations with regard to floating point arithmetic.[2] For example, $6.89 + 0.1$ results in 6.989999999999999 instead of 6.90 in Python 3.x.

2. Mapping a point from the picture space to the parameter space using $\frac{p_y - CB_{upper_y}}{p_x}$ results in a division by zero if $p_x = 0$. To mitigate this case, we apply the following:

$$x = \begin{cases} 1 \times 10^{-200}, & \text{if } x = 0 \\ x, & \text{otherwise} \end{cases}$$

3. Python does not have a native implementation of *flatMap* which is why *reduce* was used to flatten nested arrays where needed.

We compare the algorithm's outputs with results from an implementation of CASH provided by ELKI[3]. The results show that our implementation returns similar results to the original algorithm. Next, the Python code was ported to Java (Apache Flink SDK), which caused other implementation hurdles covered in the next sections.

[2] `https://docs.python.org/3/tutorial/floatingpoint.html` (accessed 2018/2/13)
[3] `https://elki-project.github.io/` (accessed 2018/2/13)

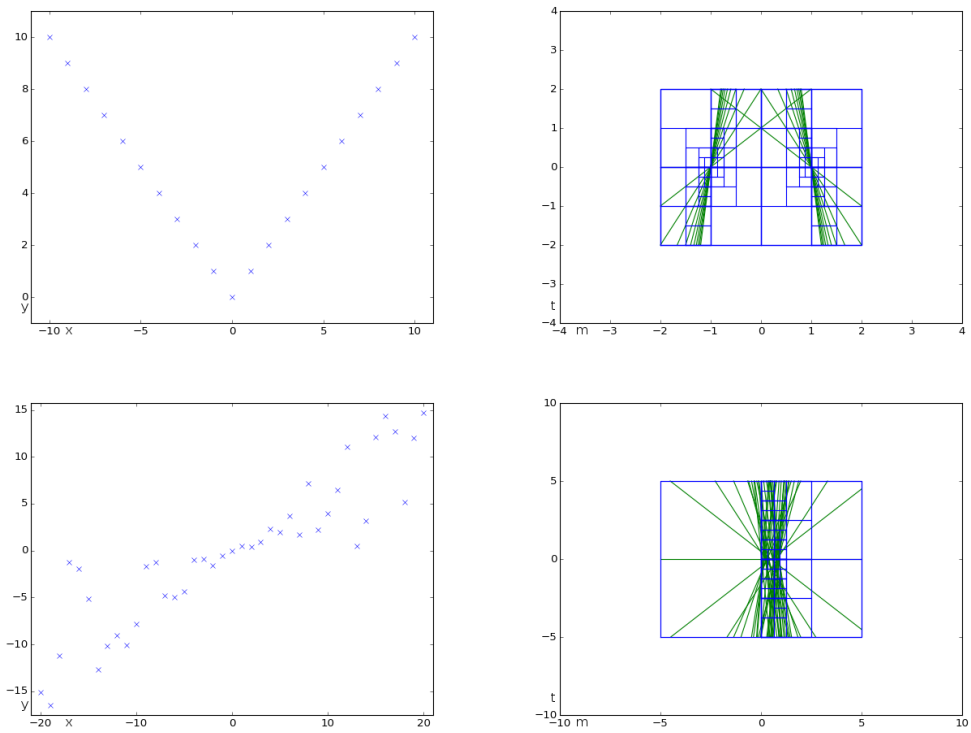

Figure 4.1: Results from the CASH-MR proof of concept implementation in Python.

4.1.1 Counting Intersections of Parameterization Functions with Grid Cell Boundaries using Map and Reduce

Determining whether a cell is dense or not is done in three steps:

1. Mapping $\forall p \in D$ to their parameterization functions f_p using an adapted version of the Hough Transform.
2. Computing intersections G of f_p with cell boundary B_{rs}.
3. Elements in G are counted. If $count(G) \geq mp$, the cell is considered dense.

Instead of transforming each point p to f_p, we compute intersection points of f_p with $CB = B_{rs}$ directly:

$$intersection_{upper_t} = p_y - CB_{upper_x} \times p_x \tag{4.1}$$

$$intersection_{lower_t} = p_y - CB_{lower_x} \times p_x \tag{4.2}$$

$$intersection_{upper_m} = \frac{p_y - CB_{upper_y}}{p_x} \tag{4.3}$$

$$intersection_{lower_m} = \frac{p_y - CB_{lower_y}}{p_x} \tag{4.4}$$

Due to division by p_x, it is important that $\forall p \in D : p_x \neq 0$. The grid cell is a (hyper-)cube with $2^{d-1} \times d$ edges. All edges must be tested for intersections with f_p. A 2-dimensional grid cell has $2^{2-1} \times 2 = 4$ edges and thus four possible intersections (top, right, bottom, and left edge) with f_p.

The points computed above do not necessarily lie within the grid cell. They describe where f_p intersects with the hyperplane (here an axis-parallel line) defined by one of the cell's edges (see Figure 4.2).

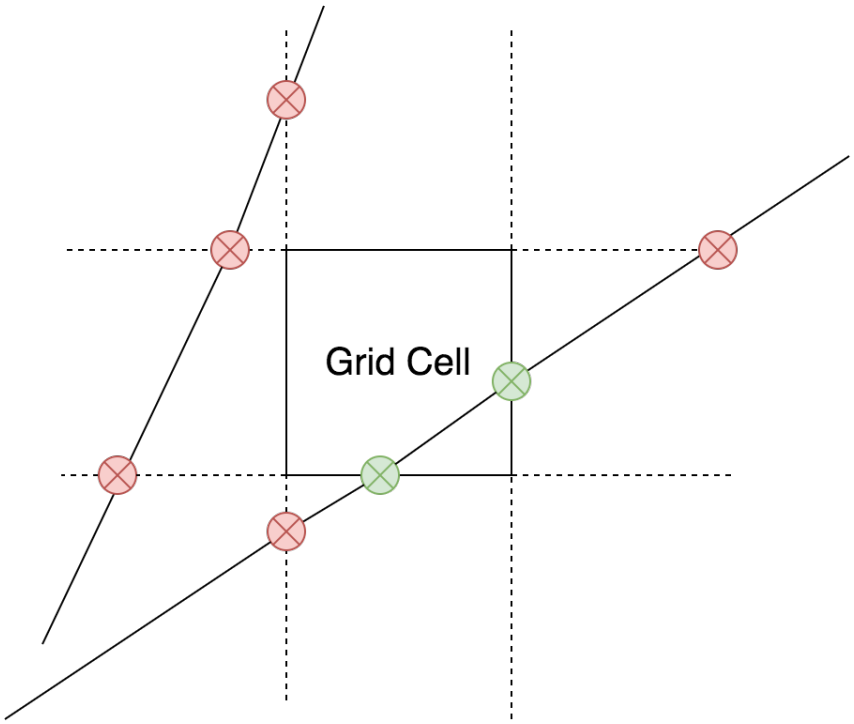

Figure 4.2: Not all computed intersections lie within the grid cell (red). An additional boundary check is required to identify those points that are within the cell's boundary (green).

We assemble a list of all four intersection points and store it in V:

$$i_1 = (CB_{upper_x}, intersection_{upper_t}) \tag{4.5}$$

$$i_2 = (intersection_{upper_m}, CB_{upper_y}) \tag{4.6}$$

$$i_3 = (CB_{lower_x}, intersection_{lower_t}) \tag{4.7}$$

$$i_4 = (intersection_{lower_m}, CB_{lower_y}) \tag{4.8}$$

$$V = \{i_1, i_2, i_3, i_4\} \tag{4.9}$$

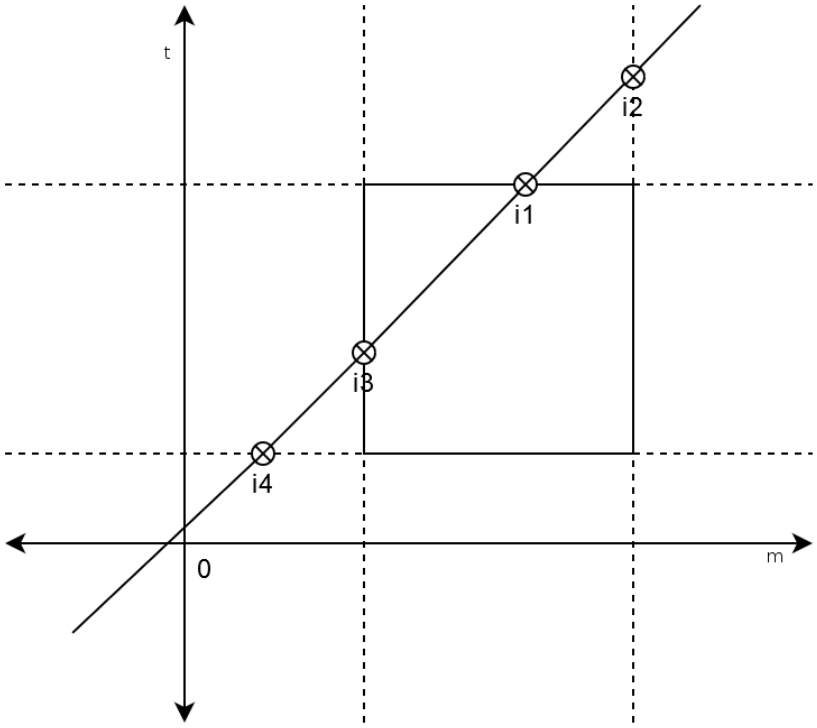

Figure 4.3: Intersection points $V = \{i_1, ..., i_4\}$ of parameterized function f_p.

By checking if point $i_p \in V$ lies within the boundaries of a cell's edge, we are able to tell whether that function intersects with the cell or not:

```
for ip ∈ V
  if CB_lower_x ≤ ip_x ≤ CB_upper_x ∧ CB_lower_y ≤ ip_y ≤ CB_upper_y
```

```
    // Intersection confirmed
  endif
endfor
```

In summary, the function for finding intersections of each point's parameterization function with a grid cell is as followed:

```
procedure findIntersections(D, CB)
  for p ∈ D
    let intersection_{upper_t} = p_y − CB_{upper_x} × p_x
    let intersection_{lower_t} = p_y − CB_{lower_x} × p_x
    let intersection_{upper_m} = (p_y − CB_{upper_y}) / p_x
    let intersection_{lower_m} = (p_y − CB_{lower_y}) / p_x

    let i_1 = (CB_{upper_x}, intersection_{upper_t})
    let i_2 = (intersection_{upper_m}, CB_{upper_y})
    let i_3 = (CB_{lower_x}, intersection_{lower_t})
    let i_4 = (intersection_{lower_m}, CB_{lower_y})

    let V = {i_1, i_2, i_3, i_4}

    for ip ∈ V
      if CB_{lower_x} ≤ ip_x ≤ CB_{upper_x} ∧ CB_{lower_y} ≤ ip_y ≤ CB_{upper_y}
        yield (CB, 1, p)
      endif
    endfor
  endfor
end
```

In essence, *findIntersections* transforms point p from the picture space to the parameter space and computes intersections with with the cell's boundary. If p intersects with one of the edges in parameter space, intersection vector $iv_p = (CB, 1, p)$ containing the cell's boundary CB, intersection count 1, and data point p is collected and added to the function's output.

Instead of iterating over each $p \in D$ and applying the aforementioned steps to it, we use *map* to achieve the same.

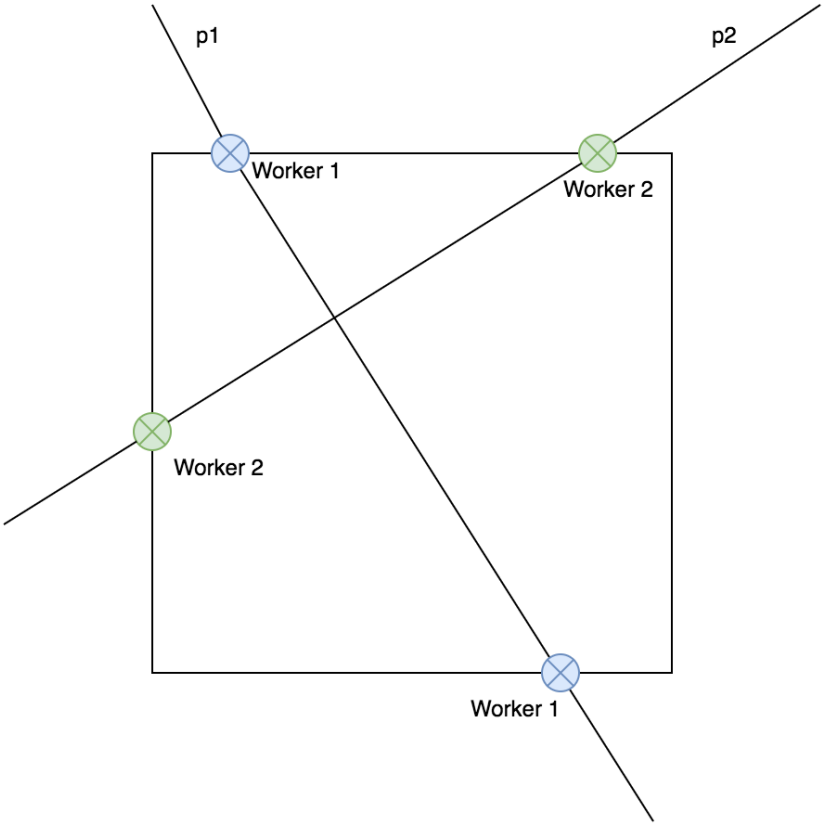

Figure 4.4: Using two workers to compute intersections for points p_1 and p_2 in parallel.

By changing the method signature of $findIntersections(D, CB)$ to $findIntersections(p \in D, CB)$, function map can be applied as followed.

```
let G = map(p → findIntersections(p, CB), D)

procedure findIntersections(p, CB)
  let intersection_upper_t = p_y − CB_upper_x × p_x
  let intersection_lowert = p_y − CB_lowerx × p_x
  let intersection_upper_m = (p_y − CB_upper_y) / p_x
  let intersection_lower_m = (p_y − CB_lower_y) / p_x

  let i_1 = (CB_upper_x, intersection_upper_t)
  let i_2 = (intersection_upper_m, CB_upper_y)
  let i_3 = (CB_lowerx, intersection_lowert)
  let i_4 = (intersection_lower_m, CB_lower_y)
```

```
  let V = {i_1, i_2, i_3, i_4}

  for ip ∈ V
    if CB_lower_x ≤ ip_x ≤ CB_upper_x ∧ CB_lower_y ≤ ip_y ≤ CB_upper_y
      yield (CB, 1, p)
    endif
  endfor
end
```

The next step is counting the number of items in G. This is possible with $reduce(reducer, data, defaultValue)$:

```
let intersectingPoints = reduce((prev, next) → prev + next_1, G, 0)

if intersectingPoints < mp
  return ∅
endif
```

Alternatively, *count*, if available, can be used as well:

```
if count(G) < mp
  return {}
endif
```

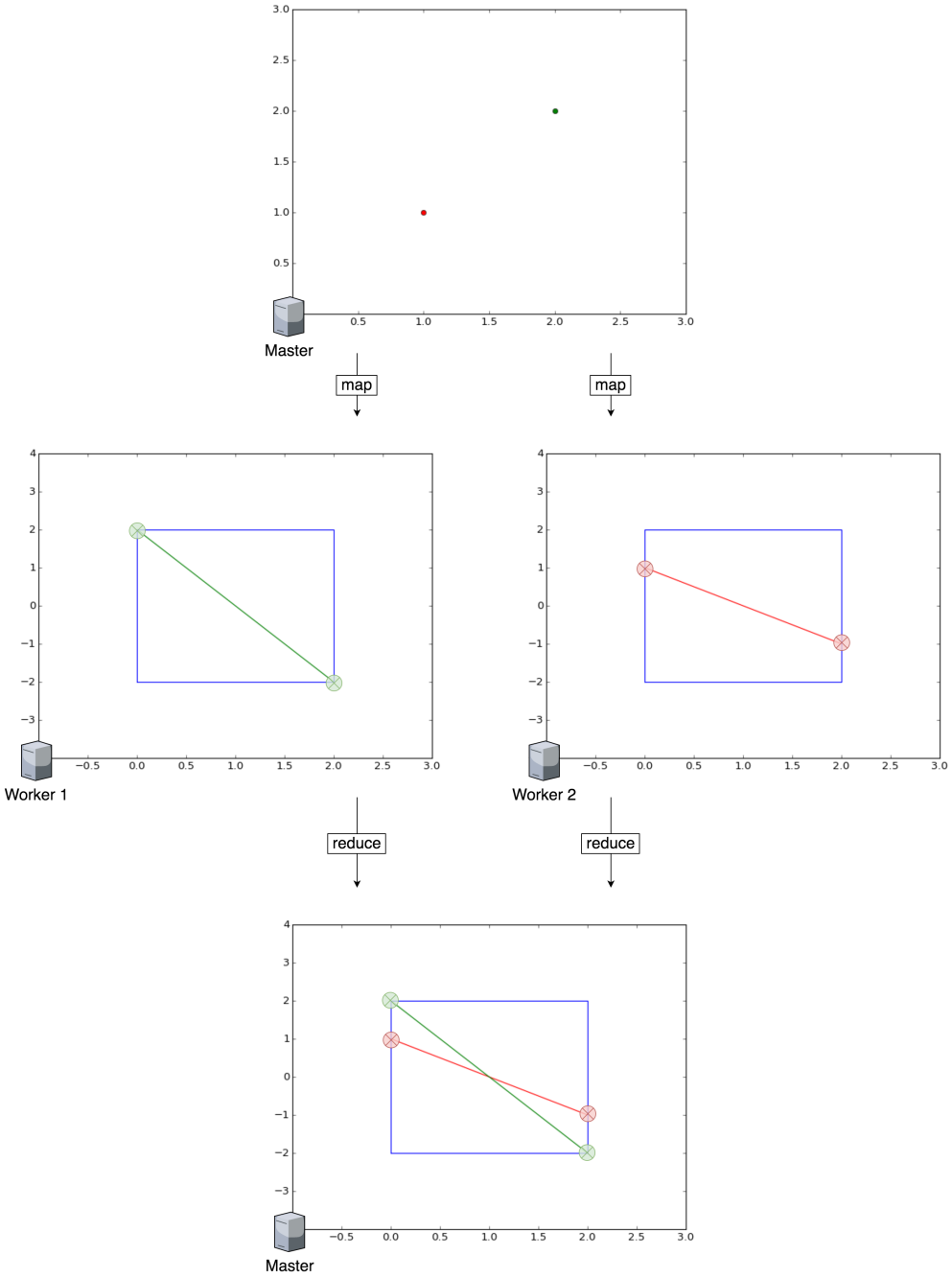

Figure 4.5: Using map to compute intersections of parameterization functions with a cell's boundary in parallel, and reduce to combine the results.

4.1.1.1 Implementation using the Apache Flink SDK

A primary challenge of implementing this approach with Java is Apache Flink's serialization limitation (Section 3.1.2.5). While we were able to overcome this obstacle, it required engineering effort to implement user defined *map* and *reduce* functions which work with Apache Flink's data model. For the implementation of these functions, we relied heavily on Java 8's lambda functions. Several data structures had to be defined to express grid cell boundaries, 2D points, and other vectors which could then be used as inputs and outputs for these functions.

Apart from these challenges, counting intersections of parameterization functions with grid cell boundaries using parallelized map and reduce functions is achievable with Apache Flink.

4.1.2 Running Spatial Descent using FlatMap

In principle, the spatial descent splits dense grid cells into two - with equal volumes each - along one of its axis. CASH then checks if those resulting grid cells are dense and if so, splits them again, until a certain threshold is reached or a cell is no longer dense.

CASH adds each split operation to a priority queue which is executed iteratively. As discussed previously, we simplified this approach by removing the queue:

```
for s = 0 to count(B_{r+1}) − 1
  rois = rois ∪ spatialDescent(G, B_{r+1_s}, r + 1)
endfor
```

This step can be parallelized regardless of whether a queue is used or not. Using *flatMap*, CASH-MR iterates over each spatial descent task autonomously and collects the result in a list:

$$rois = flatMap((b) \rightarrow spatialDescent(G, b, r + 1), B_{r+1})$$

A graphical representation of the algorithm's parallelization is depicted in Figure 4.6.

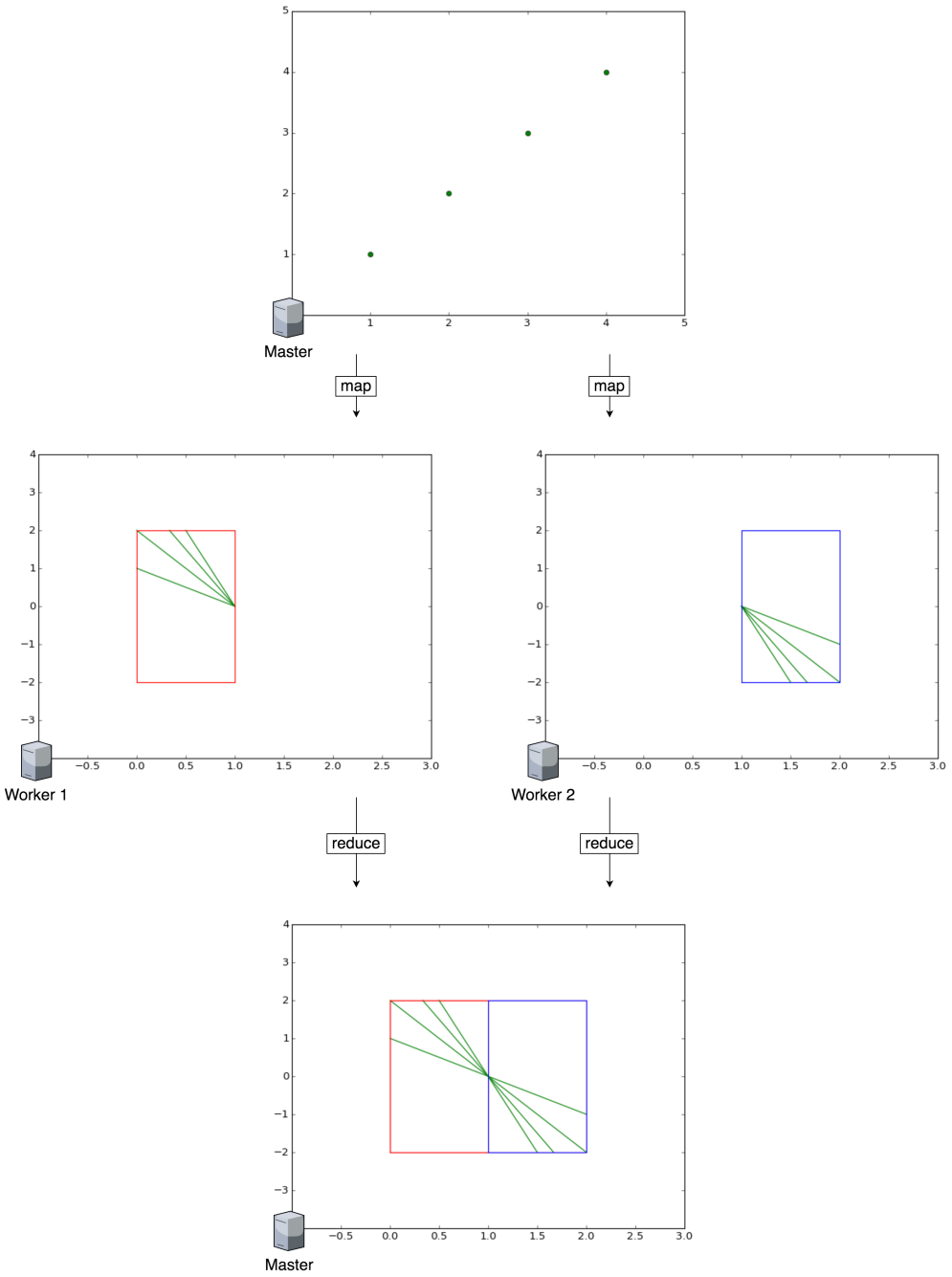

Figure 4.6: Using MapReduce to parallelize spatial descent on two worker nodes.

4.1.2.1 Implementation using the Apache Flink SDK

Section 3.1.2 explains that instantiation of Apache Flink's distributable *DataSet* type requires either access to a *ExecutionEnvironment* reference or a transformation on a *DataSet* reference. Both types are not serializable. We found no possibility of passing these types to user-defined function bodies of *map*, *reduce*, *flatMap* nor any other *DataSet* transformation available in Apache Flink. This section explains why this approach failed in detail.

The spatial descent splits a grid cell into two:

```
let B_{r+1} = splitGridCell(B_{rs})
```

Then, spatialDescent is applied to the new grid cells.

```
let rois = flatMap((b) → spatialDescent(G, b, r + 1), B_{r+1})
```

There are two obstacles that prevent parallelization of this operation in Flink.

First, if G is of type *DataSet*, it becomes impossible to pass that variable to the inner function of *flatMap* because it is not serializable. If G is not of type *DataSet*, it is not possible to implement the approach covered in Section 4.1.1.1.

Second, the set of grid cell boundaries B_{r+1} must be of type *DataSet* because *flatMap* would not be applicable to that variable otherwise. If B_{r+1} is of type *DataSet*, then grid cell boundary b must be of type *DataSet* as well, because b is used as input to $splitGridCell(b)$ which must return variable B_{r+1} of type *DataSet*. It is not possible to use the *ExecutionEnvironment* to generate a variable of type *DataSet* within the function body of *flatMap*. Thus b must be of type *DataSet*.

Even if these limitations would not exist, some worker nodes would be underutilized when the split index is low. For example, Figure 4.6 shows that only two workers are required when split index $sp = 1$. If four nodes were available, two of them would be idle.

For these reasons, this approach has not been implemented in this work but may be explored in future research.

4.1.3 Intersection Boundary Checks using Map and Reduce or Filter

One parallelization strategy for function *findIntersections* has already been discussed in Section 4.1.1. Another candidate is checking whether or not intersection points lie within the boundaries of a grid cell (see Figure 4.6):

```
for ip ∈ V
  if CB_lower_x ≤ ip_x ≤ CB_upper_x ∧ CB_lower_y ≤ ip_y ≤ CB_upper_y
    yield (CB, 1, p)
  endif
endfor
```

The same can be achieved using using *map* and *reduce* (see Figure 4.7):

```
let items = map(ip → (CB_lower_x ≤ ip_x ≤ CB_upper_x ∧ CB_lower_y ≤ ip_y ≤
    CB_upper_y), V)
let hasIntersectionWithGridCell = reduce((prev, next) →
    prev ∨ next), V, false)

if hasIntersectionWithGridCell
  return (CB, 1, p)
endif
```

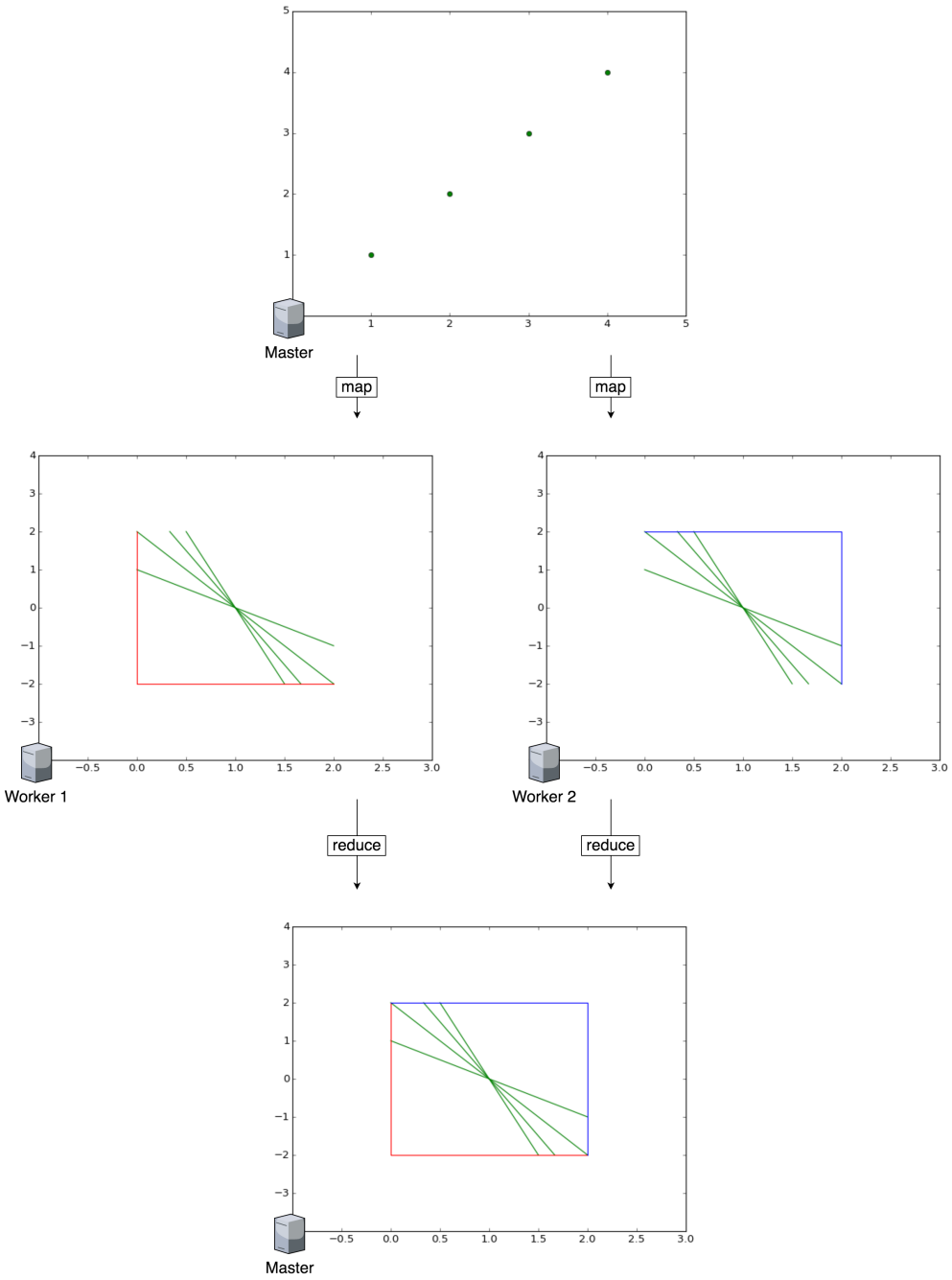

Figure 4.7: Using MapReduce to check if parameterization functions intersect with a grid cell's boundary.

Alternatively, $filter(condition, elements)$ and $count$ can be used to achieve the same:

```
let items = filter(ip → (CB_lowerx ≤ ip_x ≤ CB_upper_x ∧ CB_lowery ≤ ip_y ≤
    CB_upper_y), V)

if count(items) = 0)
  return ∅
else
  return (CB, 1, p)
endif
```

4.1.3.1 Implementation using the Apache Flink SDK

To parallelize this operation in Apache Flink V must be of type *DataSet*. V is initialized by:

$$V = \{i_1, i_2, i_3, i_4\} \tag{4.10}$$

Thus, access to the *ExecutionEnvironment* is necessary in order to convert V to the *DataSet* type for further processing:

```
let VD = env.fromCollection(V)
let items = map(ip → (CB_lowerx ≤ ip_x ≤ CB_upper_x ∧ CB_lowery ≤ ip_y ≤
    CB_upper_y), VD)
```

Therefore the approach covered in Section 4.1.1 would not be possible. Additionally, we hypothesize that creating a new *DataSet*, each time this check is executed, involves a lot of overhead compared to the potential speedup of performing these checks on a total of four edges (in 2-dimensional data).

Therefore we did not implement this approach using Apache Flink, but it could be explored for high-dimensional data in future research.

Chapter 5

BRELMAR-CASH: A Map-Reduce Distributed Global Correlation Clustering Based on the Hough Transform and Bresenham's Line Drawing Algorithm

CASH uses a grid search strategy called spatial descent. This strategy has a worst-case run-time complexity of $O(n^2)$ in, for example, very noisy datasets. We explore, define and validate a MapReduce Distributed Global Correlation Clustering Based on the Hough Transform and Bresenham's Line Drawing Algorithm (BRELMAR-CASH). The algorithm is a novel adaptation of CASH and aims at reducing run-time complexity. For brevity, the same limitations are applied as in Chapter 4.

In summary, BRELMAR-CASH is a close adaptation of CASH. Concepts, algorithms, and implementation are equal in both algorithms. The major difference between the two is how spatial partitioning is approached. CASH converges with each iteration to the desired cell resolution by splitting each cell into two along alternating axis with each recursion. BRELMAR-CASH uses grids of arbitrary resolution and reduces the number of required splits compared to CASH. We hypothesize that BRELMAR-CASH has a lower run-time complexity than CASH under certain conditions, such as noisy data.

This chapter introduces BRELMAR-CASH by firstly explaining the algorithm's motivation, second giving an overview of the algorithm, thirdly defining an algorithmic description, and finally exploring possibilities for parallelizing

the algorithm using MapReduce.

5.1 Motivation

A key operation of CASH is the spatial descent. Given a relatively small *min_points* parameter, for example, $mp = min\_points = 50\%$, and a noisy dataset, each grid cell generated by a spatial descent iteration is likely to satisfy mp and thus is considered dense. The spatial descent is applied recursively to dense grid cells as visualized in Figure 5.1. Thus, a worst case run-time complexity of $O(2^m ax\_recursion)$ applies to the spatial descent concept. [ABD+08]

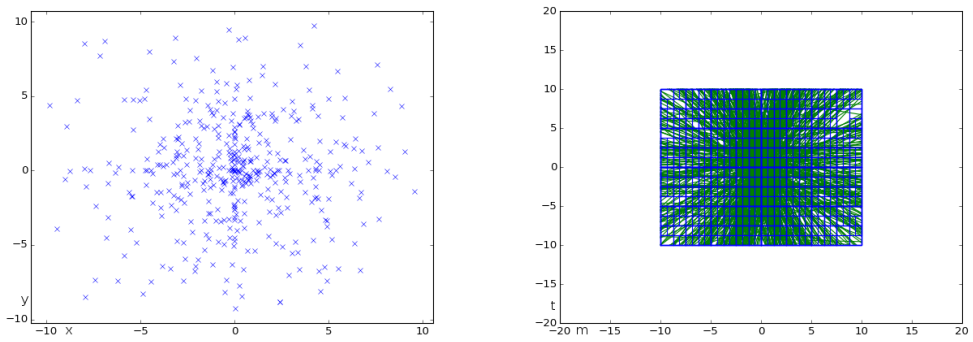

Figure 5.1: Noisy dataset D_1 (left image) applied to CASH with $min\_points = 50\%$ finds dense grid cells almost everywhere, resulting in high computational costs (right image).

In this chapter, we investigate using Bresenham's line algorithm as the grid search strategy instead of CASH's spatial descent. We hypothesize that fewer splits are required compared to CASH, which might reduce run-time complexity. Bresenham's line algorithm is a promising candidate because it approximately projects line segments onto a predefined grid. By combining and counting those projections we identify dense grid cells.

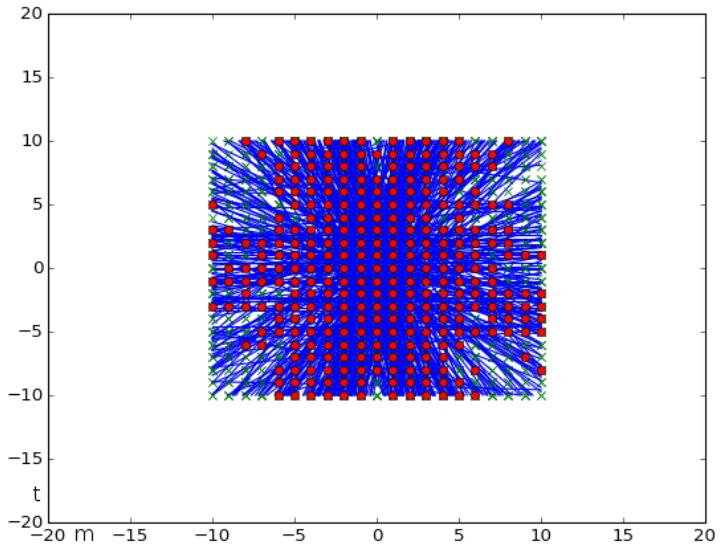

Figure 5.2: Sparse grid cells (green X's) and dense grid cells (red dots) found using Bresenham's line algorithm for dataset D_1 and $min\_points = 50\%$.

5.2 Finding Dense Grid Cells using Bresenham's Line Algorithm

Bresenham's line algorithm, explained in Section 3.1.4, projects line segments onto a grid. Figure 5.3 shows two line segments plotted with Bresenham's line algorithm.

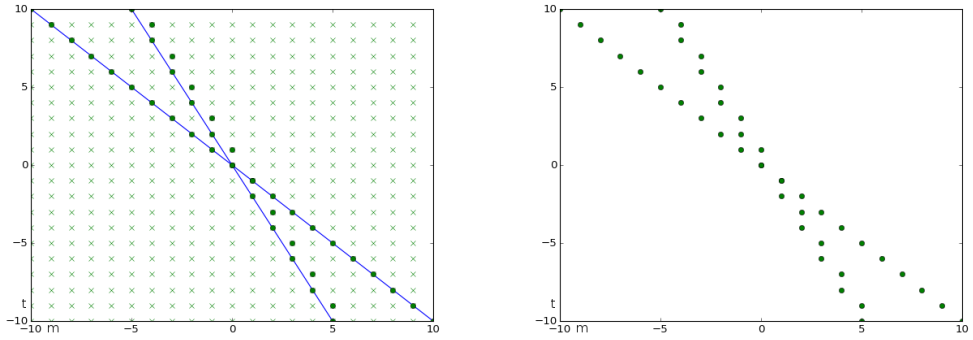

Figure 5.3: Two line segments plotted using Bresenham's line algorithm. The left image shows the original line, its projected data points, and the grid for orientation. The right image shows the projected data points only.

To decide whether a grid cell is dense or not, BRELMAR-CASH maps data points $p \in D$ (here $D = D_1$ from Figure 5.2) to their parameterization functions f_p in parameter space P. Then, intersections with f_p and the grid cell's boundaries are computed, resulting in one line segment per parameterization function. Each line segment is used as input to Bresenham's line algorithm. The algorithm's output is a list of new grid cells where that line segment approximately passes through. Multiple line segments may have some grid cells in common. If a grid cell has at least min_points line segments in common, it is considered dense.

While CASH converges, with each split, to the desired grid resolution, BRELMAR-CASH uses a predefined grid resolution, thus requiring fewer or no splits at all. BRELMAR-CASH's search strategy is efficient because approximation is used to decide if a line segment intersects with a grid cell or not. In d-dimensional data this search strategy has a run-time complexity of roughly $O(n^d)$, which might become impractical for high-dimensional datasets.

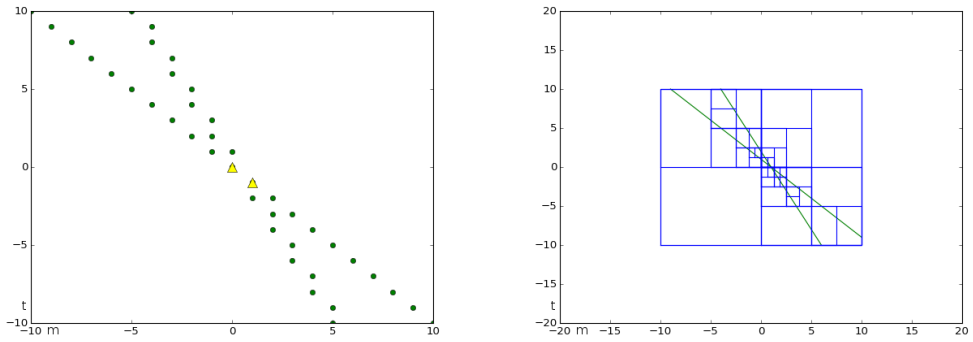

Figure 5.4: Side by side comparison of finding areas of interest using Bresenham's line algorithm (left image) with one recursion, and finding areas of interest using CASH (right image) and recursion depth of 10.

Based on the aforementioned findings, it is apparent how Bresenham's line drawing algorithm can be used as grid search strategy. By solving multiple spatial descent iterations in one step with approximation, superior performance might be observed compared to CASH in lower-dimensional datasets.

5.2.1 Grid Resolution

Computer graphics work with pixel-based monitors. Illuminating half a pixel is impossible, hence the atomic size of 1.0 is used for iterations in Bresenham's line algorithm. Grids with higher or lower resolution are not described by Bresenham's line algorithm or related work. Subspace clusters however must be found in any dataset D, for example $min(D) < max(D) < 1.0$. Allowing Bresenham's line algorithm to work with arbitrary grid resolutions is therefore imperative for applicability as grid search strategy.

We explore two possible solutions. First, scaling the dataset, and second reducing the step size of Bresenham's line algorithm. Reduction of the algorithm's step size is a promising solution, as finding global minima and maxima and resizing the complete dataset, potentially consisting of billions of data points, is computationally expensive. The downside of reducing the algorithm's step size is, that it is no longer possible to rely on integer operations. The impact of this limitation should be neglectable as modern CPUs are capable of efficiently executing simple floating point operators such as addition[1]. We develop a method for setting the algorithm's step size in the following paragraphs.

[1] `https://stackoverflow.com/questions/2550281/floating-point-vs-integer-calculations-on-moder` (accessed 2018/2/13)

Bresenham's line algorithm increases axis x and y, depending on the driving axis, in each step by 1.0. In the following example, we assume that the driving axis is x. Variables i, j are increased by step size $s = 1$ in the original algorithm:

```
procedure bresenham(x_1, y_1, x_2, y_2)
  let s = 1
  let x = x_2 − x_1
  let y = y_2 − y_1
  let m = Δy/Δx
  let j = y_1
  let ϵ = m − 1

  for i = x_1 to x_2 − 1
    illuminate(i, j)
    if (ϵ ≥ 0)
      j = j + s
      ϵ = ϵ − 1.0
    endif
    i = i + s
    ϵ = ϵ + m
  end
end
```

Introducing scale factor $sf > 0$, we define step size $s = \frac{1}{sf}$, reducing the step size of variables i, j and increasing the number of iterations:

```
procedure bresenhamWithResolution(x_1, y_1, x_2, y_2, sf)
  let s = 1/sf
  // ...

  for i = x_1 to x_2 − 1
    illuminate(i, j)
    if (ϵ ≥ 0)
      j = j + s
      ϵ = ϵ − 1.0
    endif
    i = i + s
    ϵ = ϵ + m
  end
end
```

Figure 5.5 shows the same two line segments used in Figure 5.3 and 5.7 plotted on a grid with $sf = 10$. With $s = \frac{1}{sf} = \frac{1}{10}$, each grid cell has height and width of 0.1.

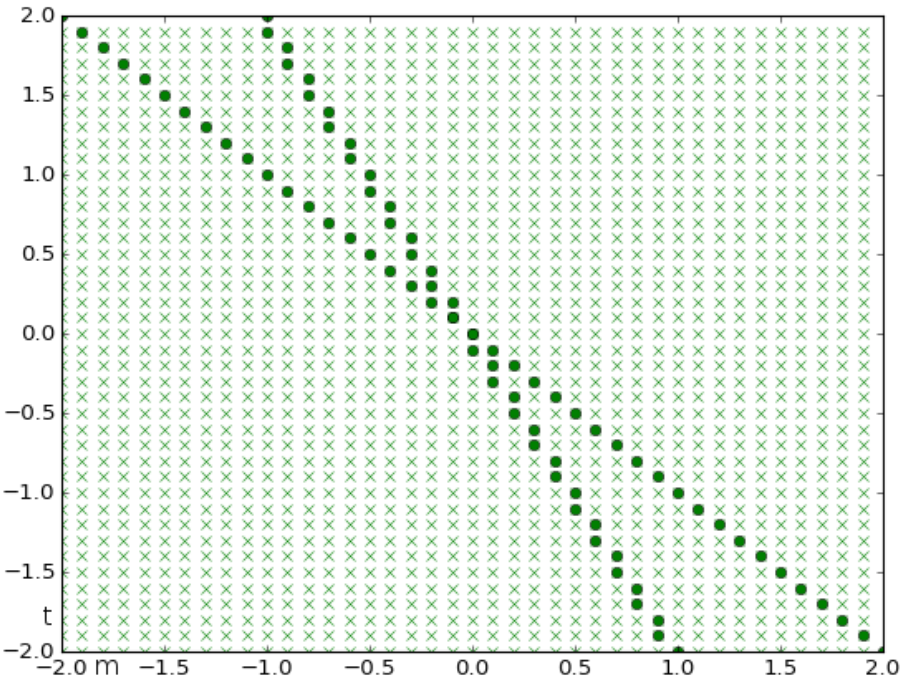

Figure 5.5: Plotting two line segments onto a grid with scale factor $sf = 10$ and step size $s = \frac{1}{sf} = 0.1$.

We derive two applications for scale factor sf. First, setting sf during the program's initialization and thus the ability to define the algorithm's initial grid resolution. Second, the adaptation of CASH's spatial descent where sf increases with each recursion as explained in the following section.

5.2.2 Spatial Descent

CASH converges with each spatial descent recursion to the desired grid resolution. By increasing scale factor sf in each recursion, it is possible to mimic this behavior, if desired. For example, results form Figure 5.3 can be used for further iterations. Spatial partitions generated by Bresenham's line algorithm

$b = bresenham(f_p)$, in Figure 5.3 $b_1 = (0, 0)$ and $b_2 = (1, -1)$ (marked yellow) are used as boundaries $\{(b_x - 1, b_y - 1), (b_x + 1, b_y + 1)\}$ for the next recursive descent iteration.

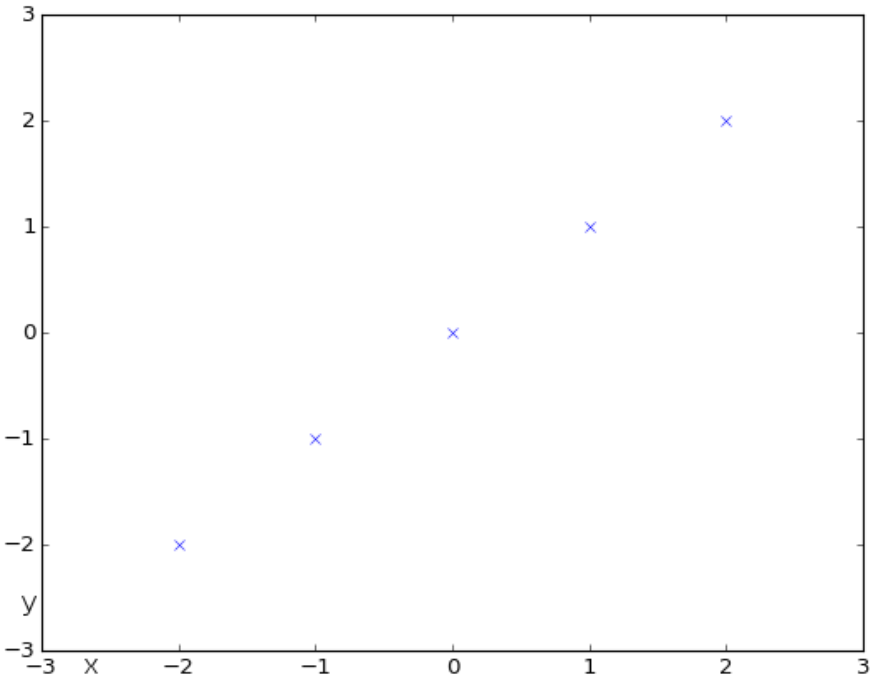

Figure 5.6: Dataset D_3.

Given linear dataset $D_3 = \{(-2, -2), (-1, -1), ...(2, 2)\}$ with points $p \in D_3$ (Figure 5.6) and using Bresenham's line algorithm for detecting dense grid cells, the spatial descent takes each cell satisfying *min_points* and passes it to the next recursion while increasing the grid scale by one as visualized in Figure 5.7.

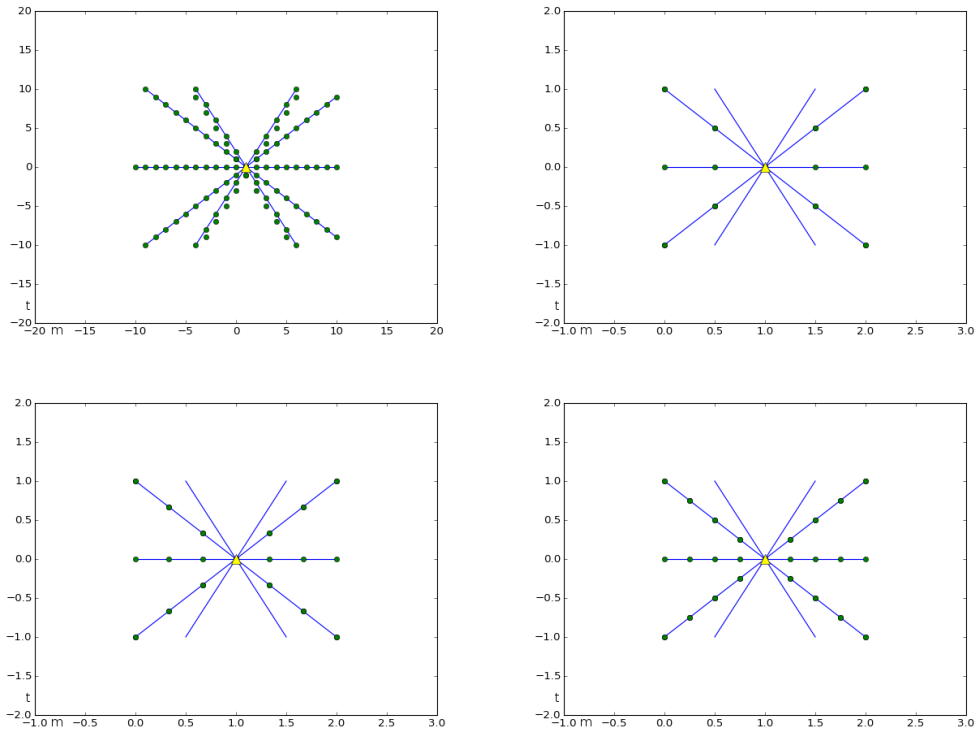

Figure 5.7: Spatial descent iterations one (top left) to four (bottom right) for $p \in D_3$, each increasing the grid scale factor sf by one with dense grid cells marked yellow.

Bresenham's line algorithm defines a pixel as cell bounded from $\{(b_x, b_y), (b_x + 1, b_y + 1)\}$. Using this narrow cell boundary for spatial descent may prune potential dense grid cells and return inaccurate results. To reduce this effect, we chose to define cell boundaries of spatial partitions generated by Bresenham's line algorithm as $\{(b_x - 1, b_y - 1), (b_x + 1, b_y + 1)\}$ (see Figure 5.8).

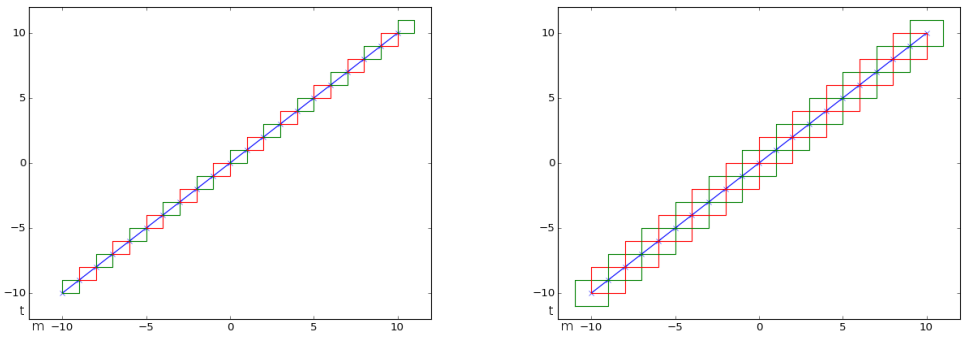

Figure 5.8: Computed cell boundaries in BRELMAR-CASH (left) and Bresenham's line algorithm (right).

On the downside, duplicate dense grid cells may potentially be found in separate spatial descent iterations. To reduce the impact of this issue, we decide to merge all results and prune duplicates based on equality checks before running further recursions.

An important factor in the algorithms initialization is choosing the initial scale factor sf and how step size s is increased in each recursion r. We define three models for increasing the step size in each recursion. Specifically:

- Linear step size model: $s = \frac{1}{sf+r}$
- Polynomial step size model: $s = \frac{1}{(sf+r)^2}$
- Exponential step size model: $s = \frac{1}{2^{(sf+r)}}$

In this work, we use the linear step size model exclusively. Evaluation of other models could be the topic of future work.

5.2.3 Challenges with Approximation and Arbitrary Endpoints

Because BRELMAR-CASH uses approximation, the number of clusters found may vary when using different initial scale factors and recursion limits. Bresenham's line algorithm accepts integer inputs for endpoints only. Due to this, small input value changes may cause large output value changes. For example, input value $(0.49, 0.49)$ is rounded to $(0, 0)$ and input value $(0.51, 0.51)$ is rounded to $(1, 1)$, resulting in varying outputs as seen in Figure 5.9.

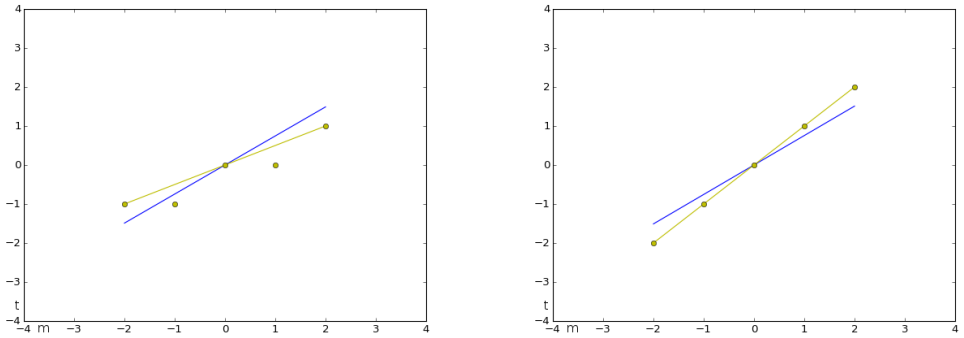

Figure 5.9: Blue line segments $\{(-2.0, -1.499), (2.0, 1.499)\}$ (left) $\{(-2.0, -1.501), (2.0, 1.501)\}$ (right) yield diverging outputs values (yellow).

As input values to Bresenham's line algorithm are directly correlated to computing intersections between f_p and boundary b, BRELMAR-CASH may find a varying number of clusters depending on parameters maximum recursion mr and initial scale factor sf.

We suspect this effect, in combination with our approach to computing boundaries (see Figure 5.8) to be the primary reason why higher recursion limits find more cluster candidates than lower ones.

An adaptation of Bresenham's line algorithm accepting floating point inputs is described in [Joy99]. Its impact on BRELMAR-CASH's accuracy should be explored in future work.

5.3 Algorithmic Formalization

We formalize the algorithm's principal steps as follows.

- Maximum recursion $mr > 0$, min points $mp > 0$, initial scale factor $sf \neq 0$ and step model $s = \frac{1}{sf}$, d-dimensional dataset D (in picture space) with points $p \in D$ and $d = 2$, recursion index $r \in \mathbb{Z} : r = 0$ are defined by the user. List A_r containing cell boundaries b is initialized and user defined initial cell boundary $b = \{(x_{start}, y_{start}), (x_{end}, y_{end})\}$ is added to it: $A_{r=0} = \{b\}$.

- Each item $b \in A_r$ is passed to a spatial descent subroutine with arguments r, b, mp, sf, D:

  - Each point $p \in D$ is mapped to parameterization function f_p in parameter space P using an adapted version of the Hough Transform.

Intersections ix_{lower}, ix_{upper} are computed for each line function f_p with cell boundary b. If no intersections with b are found, the next point is processed. In 2-dimensional datasets, a line function has either zero or two distinct intersections with a rectangle. This step is equal to step 2 from Section 4.1.

- Using intersection points ix_{lower}, ix_{upper} as start and end point arguments to Bresenham's line algorithm, a set of spatial partitions $H = \{h_1, h_2, ..., h_{n-1}, h_n\}$ is generated. For each $h \in H$, vector $k = (h, \{p\})$ is appended to list K. If spatial partition h is yielded by multiple parameterization functions f_p, all the corresponding data points p are added to k, so that $k = (h, \{p_1, p_2, ..., p_{n-1}, p_n\})$ when

$$i, j \in [0, count(D)] \wedge i \neq j \wedge \exists h_1 \in Bresenham(f_{p_i}), h_2 \in Bresenham(fpj) \wedge h_1 = h_2. \quad (5.1)$$

- For each vector $k \in K$ the number of points the vector contains is counted and if $count(v_1) < mp$, the vector is removed from list K. Remaining elements in K are considered dense.
- For each $h = k_0 \in K$, cell boundary $\{(k_{x-1}, k_{y-1}), (k_{x+1}, k_{y+1})\}$ is computed and appended to the subroutine's return value.

- The subroutine's result is unioned with A_{r+1}.
- Duplicates are removed from A_{r+1}.
- If $r \geq mr$, A_{r+1} is returned.
- A_{r+1}, $r_{r+1} = r_r + 1, sf_{r+1} = sf_r + 1$ are passed as input to step two.

Further, the algorithm's operations are defined by the following pseudo code:

```
procedure spatialDescent(D, A_r, r, mp, sf)
  let A_{r+1} = {}
  for b ∈ A_r
    A_{r+1} = A_{r+1} ∪ findDenseGridCells(D, b, r, mp, sf)
  end

  A_{r+1} = set(A_{r+1})
```

```
  return spatialDescent(D, A_{r+1}, r, mp, sf + 1)
end

procedure findDenseGridCells(D, b, r, mp, sf)
  if r > mp
    return b
  endif

  let K = {}
  for p ∈ D
    ix = findIntersectionsForPoint(p, b)
    if count(ix) < 2
      continue
    endif

    let H =
        bresenhamWithResolution(ix_{lower_x}, ix_{lower_y}, ix_{upper_x}, ix_{upper_x}, sf)
    for h ∈ H
      if ∃z_0 ∈ K : z_0 = h
        K = K ∪ (z_0, z_1 ∪ {g0})
      else
        K = K ∪ (h, g_0)
      endif
    end
  end

  for k ∈ K
    if count(k_1) ≥ mp
      let h = k_0
      yield {(h_x − 1, h_y − 1), (h_x + 1, h_y + 1)}
    end
  end
end


procedure findIntersections(p, b)
  let intersection_{upper_t} = p_y − b_{upper_x} × p_x
  let intersection_{lower_t} = p_y − b_{lower_x} × p_x
  let intersection_{upper_m} = (p_y − b_{upper_y}) / p_x
  let intersection_{lower_m} = (p_y − b_{lower_y}) / p_x
```

```
  let i_1 = (b_upper_x, intersection_upper_t)
  let i_2 = (intersection_upper_m, b_upper_y)
  let i_3 = (b_lower_x, intersection_lower_t)
  let i_4 = (intersection_lower_m, b_lower_y)

  let V = {i_1, i_2, i_3, i_4}

  for ip ∈ V
    if b_lower_x ≤ ip_x ≤ b_upper_x ∧ b_lower_y ≤ ip_y ≤ b_upper_y
      yield ip
    endif
  endfor
end
```

5.4 Applying MapReduce

Some of the aforementioned operations are potential candidates for applying MapReduce. The following sections introduce different parallelization strategies and evaluate their applicability to Apache Flink.

5.4.1 Intersection Boundary Checks using Map and Reduce or Filter

Method *findIntersectionsForPoint* is different from *findIntersections* (defined in Section 4.1.1) as it accepts a single point as argument only, and returns the coordinates of the intersection points instead of a vector containing various values. The computation of intersection points with the boundaries is algorithmically equal. Thus all concepts, limitations and implementation details covered in Section 4.1.1 are applicable here as well.

5.4.2 Spatial Descent using FlatMap

BRELMAR-CASH uses Bresenham's line algorithm to find dense grid cells, and passes these as input to the next spatial descent recursion, which uses each dense grid cell as input to method *findDenseGridCells*:

```
for b ∈ A_r
```

$A_{r+1} = A_{r+1} \cup findDenseGridCells(D, b, r, mp, sf)$
end

It is apparent that the same output can be achieved in a parallelized fashion using Apache Flink's *flatMap*:

$A_{r+1} = flatMap((b) -> findDenseGridCells(D, b, r, mp, sf), A_r)$

In frameworks that do not support *flatMap*, *map* is used to map each dense grid cell to a list of smaller dense grid cells, and *reduce* to flatten the array (see Figure 5.10).

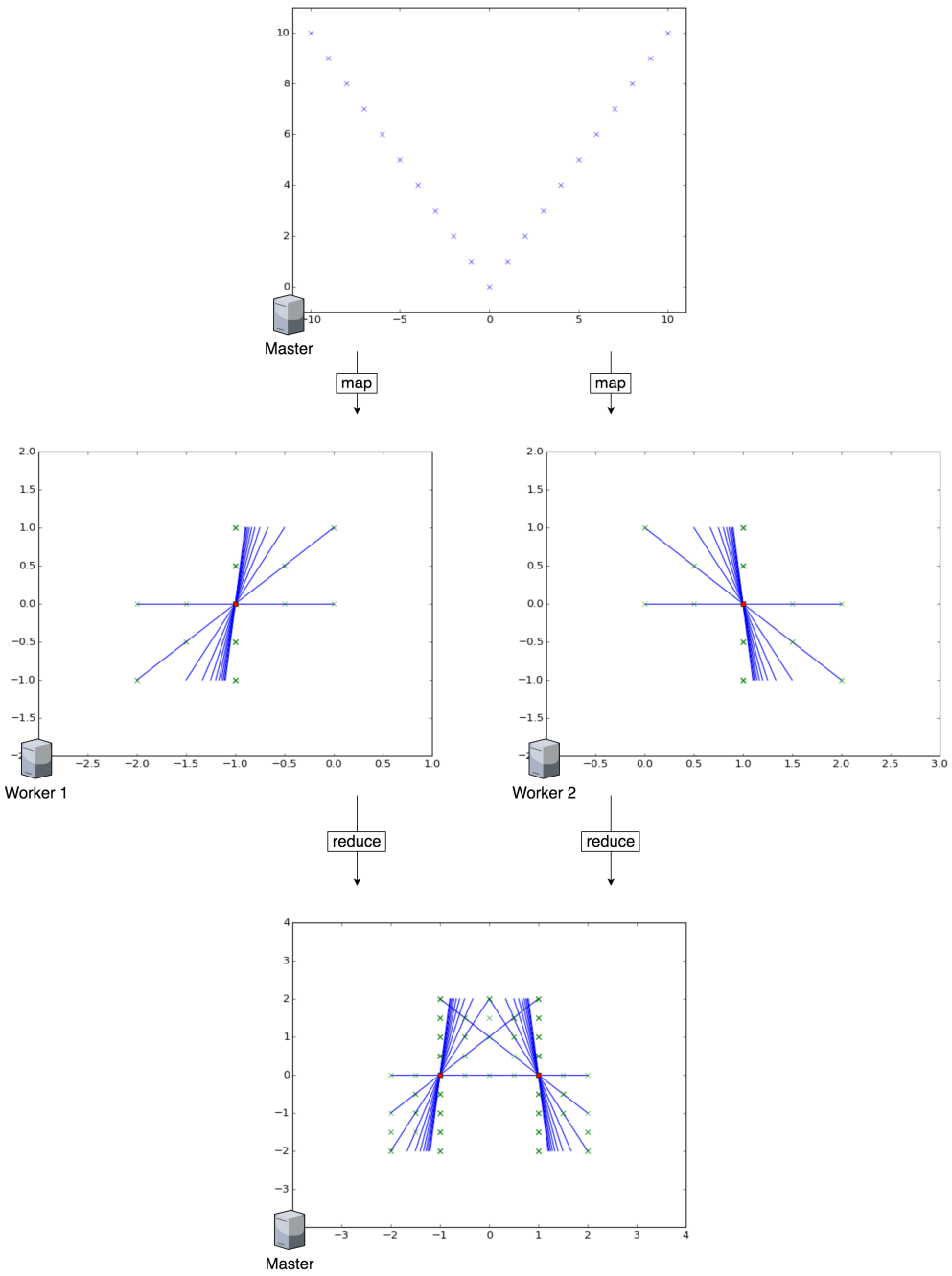

Figure 5.10: Using map and reduce to find dense grid cells in a parallelized fashion.

5.4.2.1 Implementation using the Apache Flink SDK

We were unable to identify an approach for solving *findDenseGridCells* using *flatMap* in Apache Flink for similar reasons as mentioned in Section 4.1.2.1. Passing variables of type *DataSet* or *ExecutionEnvironment* to *findDenseGridCells* is not possible with Apache Flink. While it is possible to type A_r as *DataSet*, the inner method of *findDenseGridCells* would not benefit from any parallelization:

$$A_{r+1} = flatMap((b) -> findDenseGridCells(D, b, r, mp, sf), A_r)$$

We hypothesize that worker nodes have higher saturated when processing each data point in a parallel fashion, instead of running spatial descent on dense grid cells - as fewer dense grid cells are found than points exist in the dataset. When $r = 0$, then $count(A_0) = 1$ which implies that for $r = 0$ one worker is saturated. We did not use *flatMap* in our implementation of *findDenseGridCells* for this reason.

5.4.3 Using Map, GroupBy, Reduce and Filter to Identify Dense Grid Cells

To decide whether a grid cell is dense or not, all intersections of parameterization function f_p with the cell's boundaries are computed. In a 2-dimensional dataset, zero or two distinct intersections are found which are used as start and end point for Bresenham's line algorithm. The algorithm returns a list of spatial partitions which the line segment crosses. These partitions are merged, and if multiple line segments yield the same spatial partitions, a counter denoting the line segments that pass through this partition is increased by one:

```
let K = {}
for p ∈ D
  ix = findIntersectionsForPoint(p, b)
  if count(ix) < 2
    continue
  endif

  let H = bresenhamWithResolution(ix_lowerx, ix_lowery, ix_upperx, ix_upperx, sf)
  for h ∈ H
    if ∃z0 ∈ K : z0 = h
      K = K ∪ (z0, z1 ∪ {g0})
```

```
    else
      K = K ∪ (h, g_0)
    endif
  end
end
```

We hypothesize that MapReduce can be used to parallelize these steps. We identify the possibility of using *map* / *flatMap*, *groupBy*, *reduce*, and *filter* to count the number of line segments crossing spatial partitions generated by Bresenham's line algorithm. An overview of this approach is described as followed and visualized in Figure 5.11.

- Using *map*, points $p \in D$ are mapped their parameterization functions f_p. Intersections with f_p and boundary b are computed and passed to Bresenham's line algorithm, resulting in spatial partitions H. The map function returns vector $v = (p, H)$. The result of map is stored as list V.
- Results V from the first step are flattened using $flatMap$. For each $h \in v_1$, a new vector $w = (h, 1, v_0)$ is added to the resulting list W.
- List W is grouped by the first element in vector $w \in W : groupBy(w \rightarrow w_0)$.
- Each grouped set of K is reduced with $reduce((prev, next) \rightarrow (prev_0, prev_1 + 1, prev_2 \cup next_2)$. The results are stored in list K.
- All items of K not satisfying min_points are filtered out, so that: $\forall k \in K : k_1 \geq min\_points$.

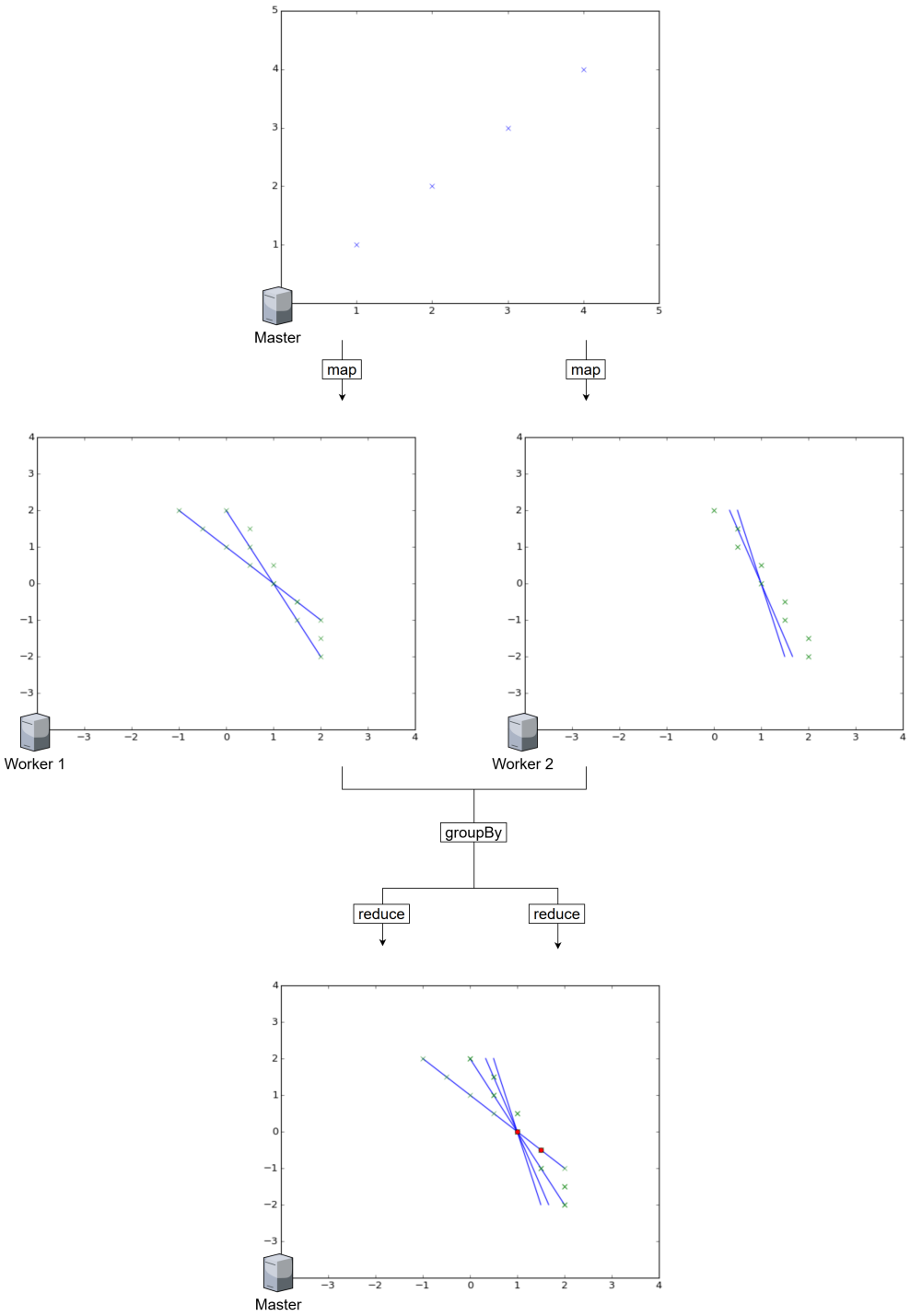

Figure 5.11: Mapping points $p_1, p_2, ..., p_n$ in a distributed fashion to spatial partitions, and using *reduce* to count the number of common line segments each for each spatial partition H.

The algorithmic description is as follows:

```
let V = map(p → bresenhamTransform(p, b, sf), D)

procedure bresenhamTransform(p, b, sf)
  let ix = findIntersections(p, b)
  if count(ix) < 2
    return ∅
  endif

  let H = bresenhamWithResolution(ix_lowerx, ix_lowery, ix_upperx, ix_upperx, sf)
  return (p, 1, H)
end

let W = flatMap(v → map(h → (h, 1, {v0}), v1), V)

W = groupBy(w− > w0, W)

let K = reduce((prev, next) → (prev0, prev1 + 1, prev2 ∪ next2), W)

K = filter(k1 → k1 ≥ min_points, K)
```

5.4.3.1 Implementation using the Apache Flink SDK

The algorithmic description in the previous section uses a combination of *flatMap* and *map* to flatten spatial partitions generated by Bresenham's line algorithm:

```
let W = flatMap(v → map(h → (h, 1, {v0}), v1), V)
```

This assumes that the user-defined *flatMap* function is expected to return a list which is merged with all other lists returned from each flatMap iteration. However, the Apache Flink SDK uses a collector instead:

```
let W = flatMap((v, collector) → forEach(h →
    collector.collect((h, 1, {v0})), v1), V)
```

Thus, *map* can not be used to flatten these spatial partitions. Apart from

this limitation, we were able to parallelize this the described pieces of the algorithm using Apache Flink.

5.4.4 Combining Duplicate Areas of Interest

Section 5.4.2 describes that separate spatial descent iterations may find duplicate dense grid cells (see Figure 5.8). This causes the search strategy to be executed on the same grid cell multiple times, which has a serious negative impact on run-time performance and output quality. This effect can be limited by identifying these duplicates and combining them before continuing the search for dense grid cells.

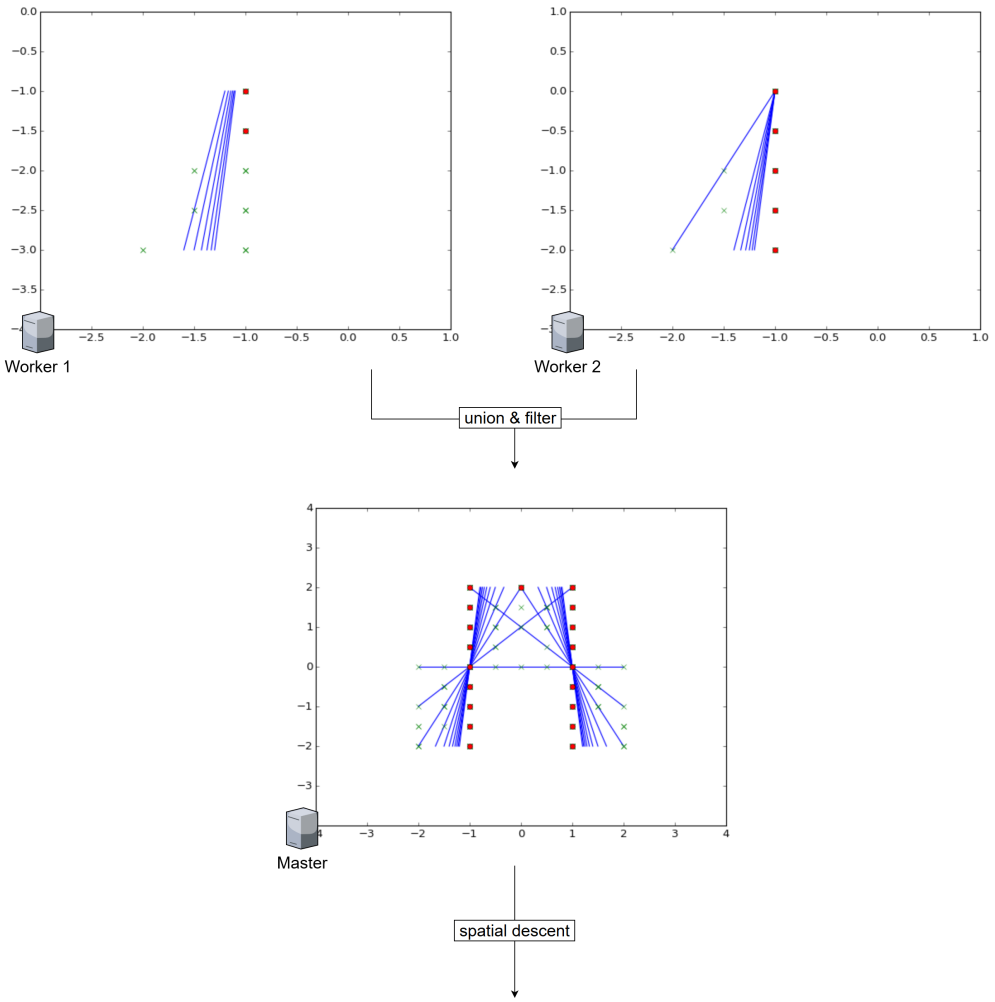

Figure 5.12: Two spatial descent iterations with different boundaries find duplicate areas of interest $(-1.0, -1.0)$ and $(-1.0, -1.5)$. Using union and filter, the duplicates are merged before being passed to further spatial descent iterations.

5.4.4.1 Implementation using the Apache Flink SDK

We identified the issue of duplicate areas of interest during benchmark evaluation of BRELMAR-CASH's Apache Flink implementation. To reduce the effect, we tried using *union* to merge results from recursive descent iterations on the worker nodes, without having to *collect* them in the master process. Unfortunately, Apache Flink is capable of applying union to at most 64 datasets,

which made this approach impractical.[2] We used Java 8's Stream API to merge duplicates by collecting the results from Apache Flink in the master node and merging duplicates in a Java Array. We hypothesize that this effect greatly increases overhead and run-time when many candidates are found. Finding a better merge strategy should be a focus of future work.

[2] `http://mail-archives.apache.org/mod_mbox/flink-user/201708.mbox/\%3CCAAdrtT2EANv7MHtG=-XpZWhU=j6L4r9v3TAu9j5o9V9CND2=bA@mail.gmail.com\%3E` (accessed 2018/2/13)

Chapter 6

Results and Discussion

In this chapter, we evaluate speedup and efficiency of CASH-MR and BRELMAR-CASH and compare both algorithms with one another. All benchmarks are executed on a Windows 10 PC with an Intel Core i7-4790K CPU at 4.00GHz and 4 Cores (8 Logical Processors) and using an Apache Flink 1.3.2 job manager, an Apache Flink 1.3.2 task manager with 8 assigned CPUs, and an Apache Flink 1.3.2 job submission image.

To reliably evaluate how both algorithms perform under various conditions, we generate three artificial datasets using a *random()* function with a fixed seed:

- Linear-origin scattered dataset (Figure 6.1): $f_0(x) = (x \times random(), x \times random())$
- Linear dataset (Figure 6.2): $f_1(x) = (x, x)$
- Random dataset (Figure 6.3): $f_2(x) = (20 \times random(), 20 \times random())$

By setting the random number generator's seed to 10, we ensure that each benchmark uses the same dataset. To avoid out-of-order issues in the data generator, no parallelization strategy is used here.

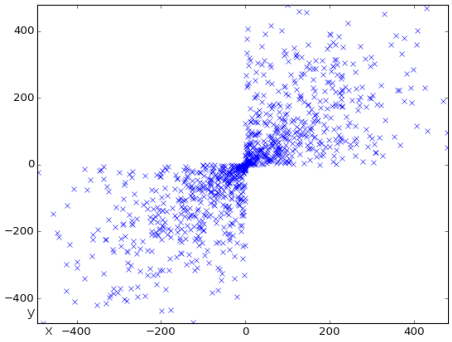

Figure 6.1: Linear-origin scattered dataset generated by $f_0(x) = (x \times random(), x \times random())$.

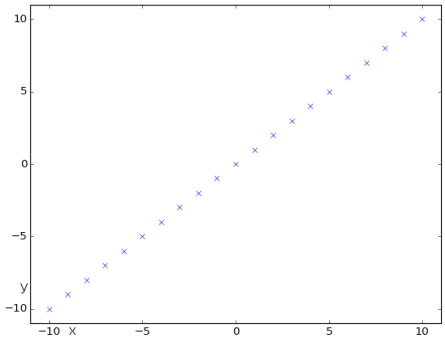

Figure 6.2: Linear dataset generated by $f_1(x) = (x, x)$.

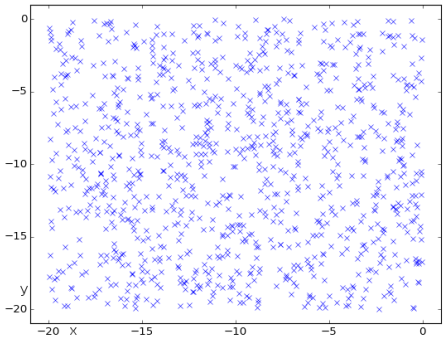

Figure 6.3: Random generated dataset using $f_2(x) = (20 \times random(), 20 \times random())$.

This chapter is structured as follows. First, we evaluate and interpret benchmarks with regards to CASH-MR's and BRELMAR-CASH's parallelization performance (speedup and efficiency). Second, we apply different parametrizations to both algorithms and observe run-time behavior. Third, we compare both algorithms with one another using parametrizations defined in the second step and make meaningful observations with regards to their performance differences

6.1 CASH-MR Parallelization Performance

We apply the CASH-MR algorithm to all three datasets listed above with 20001 data points each. CASH-MR is parameterized with a maximum recursion depth of 15, the minimum points required for a cluster set to 10000, which is 50% of the total available data points, and the initial boundary to $\{(-128, -128, 128, 128)\}$. To receive robust results, each execution is repeated three times with parallel tasks $pt \in \{1, 2, 3, ..., 7, 8\}$.

6.1.1 Parallelization Performance on Linear-Origin Scattered Dataset

We observe significant speedup with two to four parallel tasks. Speedup converges at ≈ 3.20 for six to eight parallel tasks. Although only six subspace clusters are found, CASH-MR finds many regions of interest when the spatial descent split index is low. Figure 6.4 shows that many cluster candidates are found for $t = 0$ (t is the horizontal axis), but only a few of them are actual subspace clusters.

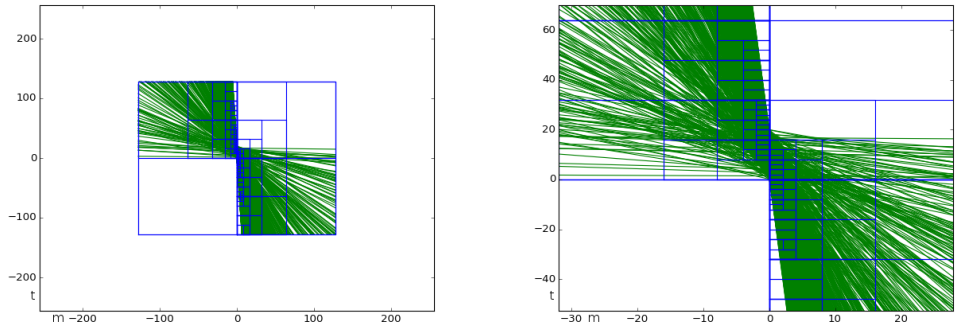

Figure 6.4: Dense grid cells found by CASH-MR. The left image shows the complete search space, the right one is zoomed in.

We assume that all four physical CPUs are saturated with finding dense grid cells, which implies that speedup does not increase significantly with more parallel tasks.

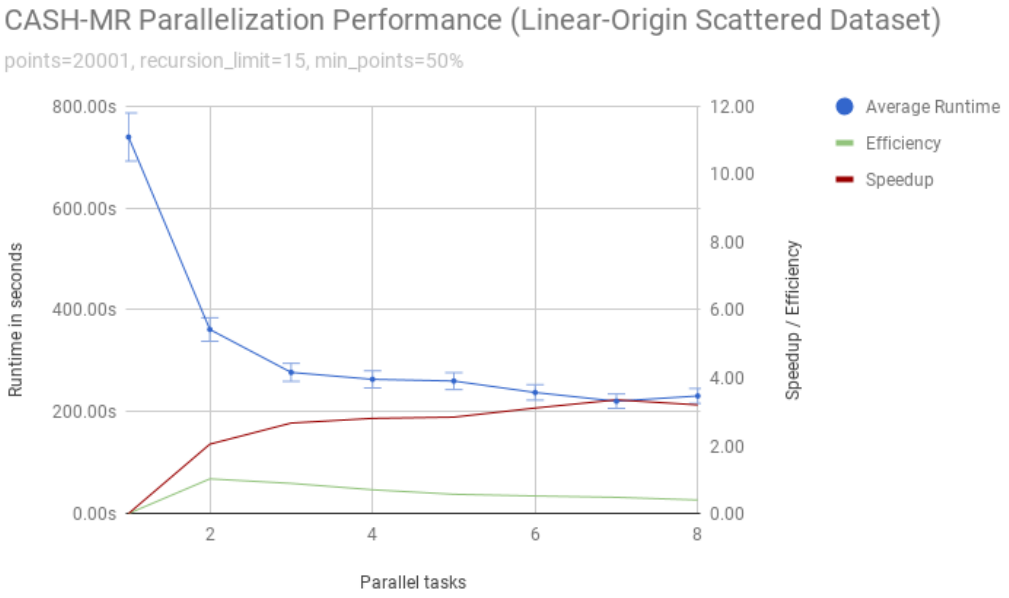

Figure 6.5: Plotted benchmark metrics from applying CASH-MR to the linear-origin scattered dataset.

Clusters Found	Parallelism	Average Runtime	RSD	Speedup	Efficiency
6	1	739.74s	1.41%	0.00	0.00
6	2	361.67s	3.16%	2.05	1.02
6	3	277.32s	4.43%	2.67	0.89
6	4	263.73s	7.01%	2.80	0.70
6	5	260.35s	6.56%	2.84	0.57
6	6	237.85s	9.39%	3.11	0.52
6	7	220.76s	10.83%	3.35	0.48
6	8	231.03s	20.82%	3.20	0.40

Table 6.1: Detailed benchmarks metrics from applying CASH-MR to the linear-origin scattered dataset.

We hypothesize that efficiency is greater with a higher parallelism if fewer cluster candidates exist, and lower when more candidates are found.

6.1.2 Parallelization Performance on Random Dataset

Finding subspace clusters with CASH-MR on the random dataset yields almost linear speedup until a parallelism of five, which supports the hypothesis that fewer cluster candidates result in less CPU saturation. Figure 6.6 shows that fewer cluster candidates exist for this dataset.

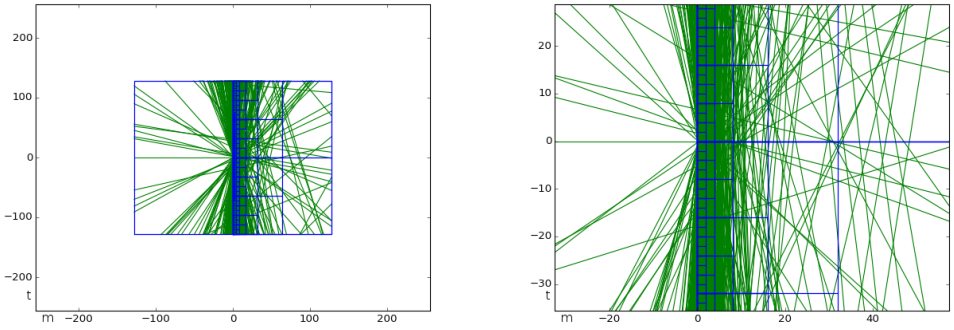

Figure 6.6: Dense grid cells found by CASH-MR. The left image shows the complete search space, the right one is zoomed in.

As fewer candidates are found, the four physical CPUs are not fully saturated and higher parallelism increases speedup significantly compared to Section 6.1.1.

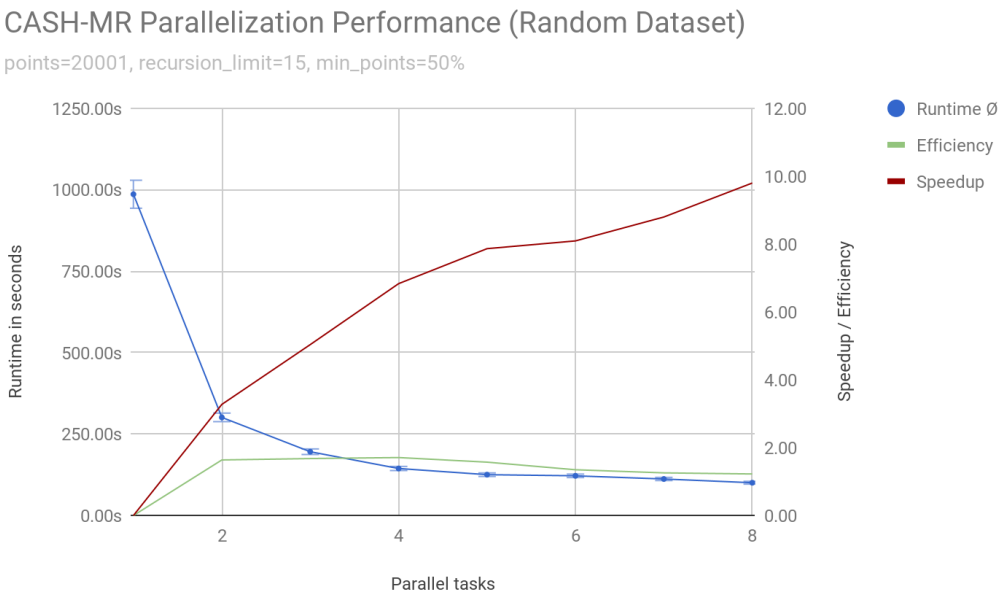

Figure 6.7: Plotted benchmark metrics from applying CASH-MR to the random dataset.

Clusters Found	Parallelism	Average Runtime	RSD	Speedup	Efficiency
10	1	986.27s	1.63%	0.00	0.00
10	2	301.20s	2.51%	3.27	1.64
10	3	195.82s	3.20%	5.04	1.68
10	4	144.33s	11.13%	6.83	1.71
10	5	125.41s	25.30%	7.86	1.57
10	6	121.87s	6.91%	8.09	1.35
10	7	112.11s	5.47%	8.80	1.26
10	8	100.66s	6.86%	9.80	1.22

Table 6.2: Detailed benchmarks metrics from applying CASH-MR to the random dataset.

6.1.3 Parallelization Performance on Linear Dataset

The linear dataset yields a speedup curve similar to the linear-origin scattered dataset from Section 6.1.1, with the steepest increase at $pt = 2$ and gradual increase with $pt > 3$. It further strengthens our hypothesis that more candidates result in faster CPU saturation and thus less speedup when more parallel tasks are used.

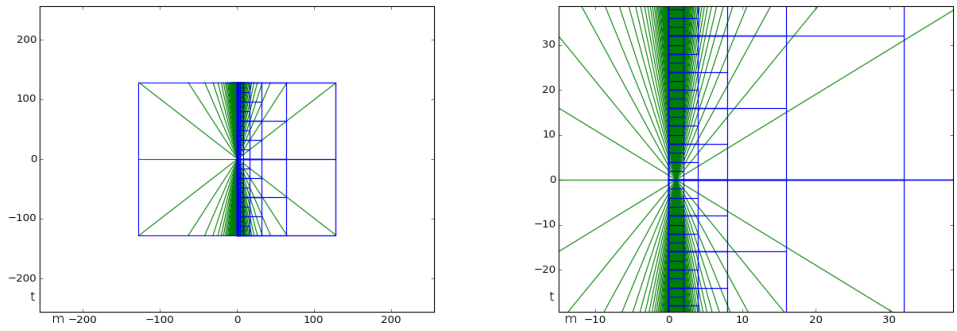

Figure 6.8: Many cluster candidates are found by CASH-MR, resulting in high CPU saturation.

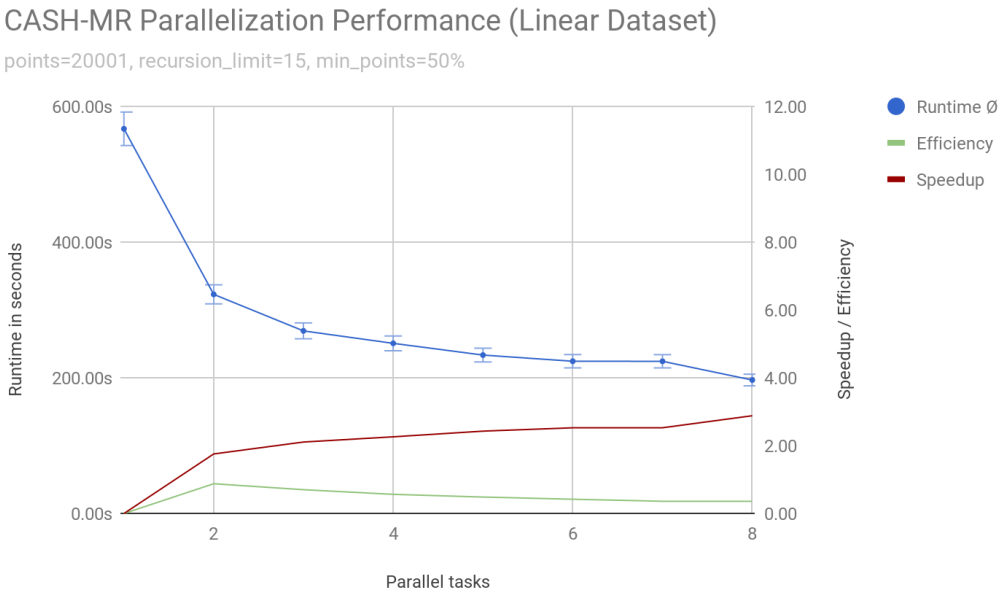

Figure 6.9: Plotted benchmark metrics from applying CASH-MR to the linear dataset.

Clusters Found	Parallelism	Average Runtime	RSD	Speedup	Efficiency
4	1	566.87s	5.14%	0.00	0.00
4	2	322.90s	3.36%	1.76	0.88
4	3	269.09s	5.75%	2.11	0.70
4	4	250.76s	7.61%	2.26	0.57
4	5	233.50s	7.08%	2.43	0.49
4	6	224.38s	24.18%	2.53	0.42
4	7	224.30s	6.16%	2.53	0.36
4	8	196.73s	21.22%	2.88	0.36

Table 6.3: Detailed benchmarks metrics from applying CASH-MR to linear dataset.

6.1.4 Conclusion

We conclude that our parallelized implementation of CASH achieves significant speedup and efficiency regardless of the shape of the data. We observe that the amount of cluster candidates impacts speedup and efficiency negatively due to saturation of physical CPUs. We assume that CASH-MR achieves higher speedup and efficiency when more physical CPUs are available.

6.2 BRELMAR-CASH Parallelization Performance

We apply BRELMAR-CASH to the three datasets (linear, random, linear-origin scattered) defined in this chapter with recursion limit of 0, the minimum points required to identify a cluster set to 40%, and the initial boundary $\{(-128, -128, 128, 128)\}$. Each benchmark is repeated three times each with parallel tasks $pt \in \{1, 2, 3, ..., 7, 8\}$.

6.2.1 Parallelization Performance on Linear-Origin Scattered Dataset

Applying the algorithm to the linear-origin scattered dataset shows steady speedup increase until $pt \geq 5$, further strengthening our hypothesis that MapReduce is applicable to correlation clustering. We suspect that speedup decrease at $pt \geq 5$ is caused by all four physical CPUs being saturated. Thus, additional parallel tasks add overhead and increase the total workload, which causes the drop in speedup. One explanation for this behavior could be the implementation described in Section 5.4.4, which uses *collect* to union results from spatial descent iterations.

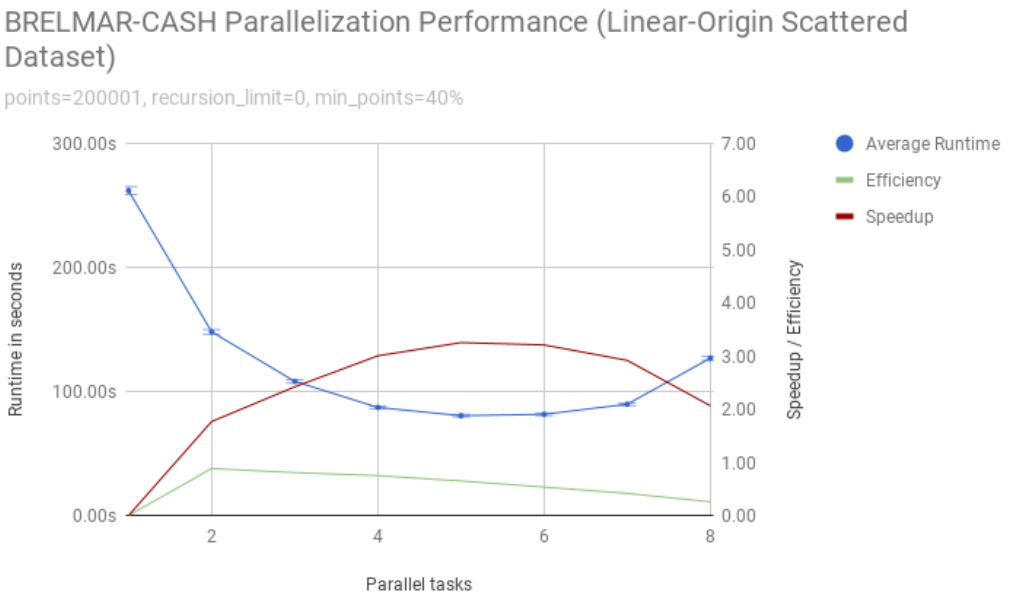

Figure 6.10: Plotted benchmark metrics from applying BRELMAR-CASH to the linear-origin scattered dataset.

Clusters Found	Parallelism	Average Runtime	RSD	Speedup	Efficiency
257	1	261.90s	0.53%	0.00	0.00
257	2	148.09s	1.41%	1.77	0.88
257	3	108.17s	7.29%	2.42	0.81
257	4	87.15s	3.15%	3.01	0.75
257	5	80.50s	0.48%	3.25	0.65
257	6	81.61s	3.93%	3.21	0.53
257	7	89.73s	0.61%	2.92	0.42
257	8	126.84s	11.75%	2.06	0.26

Table 6.4: Detailed benchmark metrics from applying BRELMAR-CASH to the linear-origin scattered dataset.

6.2.2 Parallelization Performance on Random Dataset

Applying BRELMAR-CASH to the random dataset returns similar results to the previous section, with performance increasing until $pt = 6$. Although similar trends in average run-time, speedup, and efficiency surface, fewer clusters are found (32 in Table 6.4 versus 257 in Table 6.5). We deduce that the number of clusters found does not impact overall performance when recursion limit is very low, or zero. We evaluate this theory in Section 6.3.

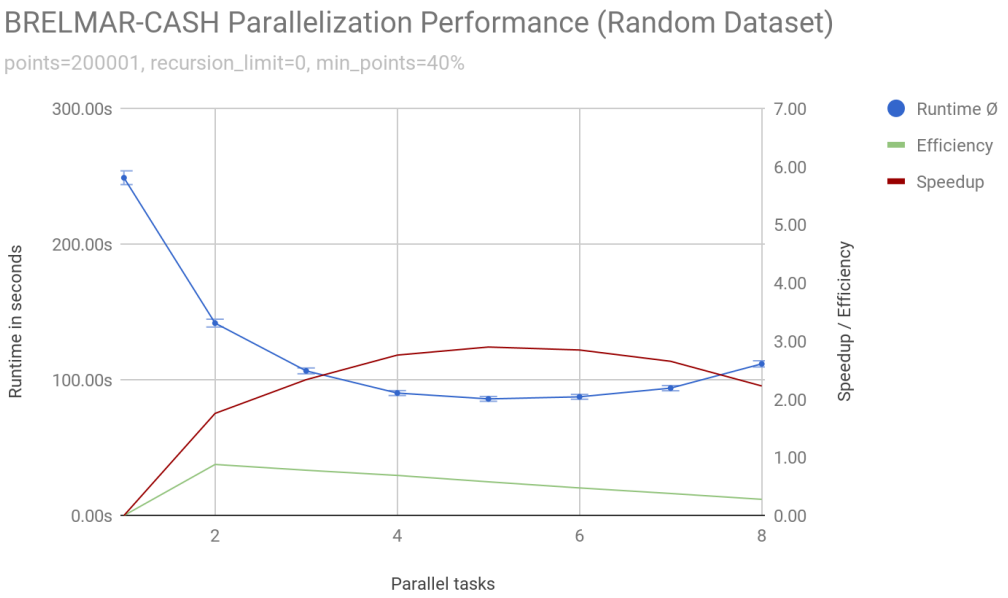

Figure 6.11: Plotted benchmark metrics from applying BRELMAR-CASH to the random dataset.

Clusters Found	Parallelism	Average Runtime	RSD	Speedup	Efficiency
23	1	248.85s	0.95%	0.00	0.00
23	2	141.76s	1.00%	1.76	0.88
23	3	106.55s	3.93%	2.34	0.78
23	4	90.23s	1.29%	2.76	0.69
23	5	85.91s	1.05%	2.90	0.58
23	6	87.45s	1.88%	2.85	0.47
23	7	93.86s	0.79%	2.65	0.38
23	8	111.74s	0.20%	2.23	0.28

Table 6.5: Detailed benchmark metrics from applying BRELMAR-CASH to the random dataset.

6.2.3 Parallelization Performance on Linear Dataset

BLREMAR-CASH produces similar results to previous sections when being applied to the linear dataset. We derive that BRELMAR-CASH's run-time is less sensitive to data variance than CASH-MR when the recursion limit is low.

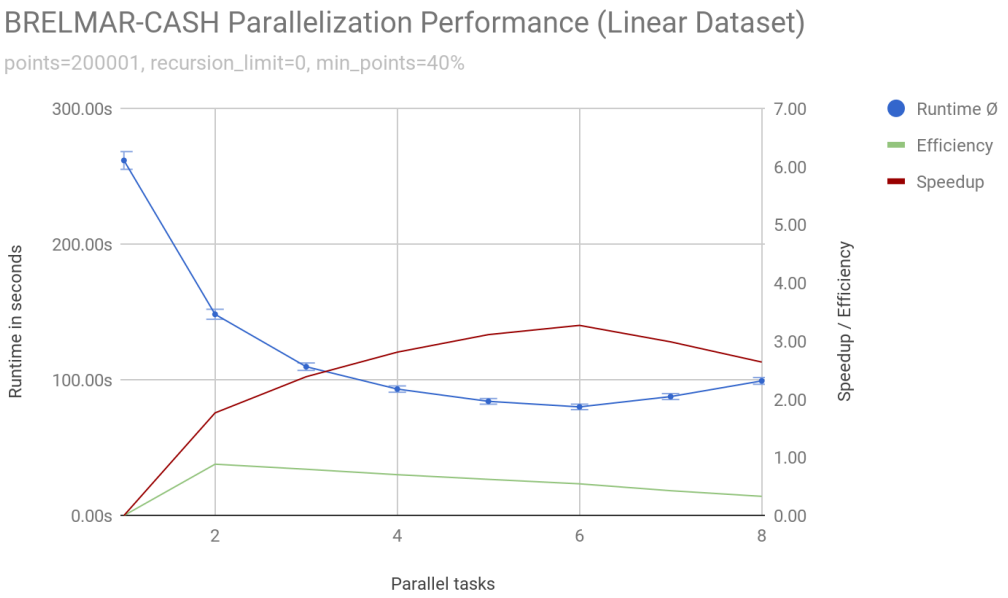

Figure 6.12: Plotted benchmark metrics from applying BRELMAR-CASH to the linear dataset.

Clusters Found	Parallelism	Average Runtime	RSD	Speedup	Efficiency
257	1	261.61s	0.38%	0.00	0.00
257	2	148.26s	4.02%	1.76	0.88
257	3	109.64s	1.18%	2.39	0.80
257	4	93.16s	1.22%	2.81	0.70
257	5	84.13s	0.70%	3.11	0.62
257	6	80.03s	0.91%	3.27	0.54
257	7	87.62s	0.77%	2.99	0.43
257	8	99.17s	0.66%	2.64	0.33

Table 6.6: Detailed benchmark metrics from applying BRELMAR-CASH to the linear dataset.

6.2.4 Conclusion

We conclude that BRELMAR-CASH achieves significant speedup and efficiency when executed in a distributed environment. Due to various limitations in our implementation, such as the one described in Section 5.4.4, speedup decreases when the physical CPUs are fully saturated. Additionally, we observe that the algorithm is less sensitive to cluster candidates with regards to execution time compared to CASH-MR. However, BRELMAR-CASH finds more

clusters of potentially lower quality than CASH-MR when applied to the same data with similar parametrizations.

6.3 Comparing Performance of CASH-MR with BRELMAR-CASH

The two algorithms take different approaches to solve the task of correlation clustering. A comparison of both algorithms is unlikely to yield universal truth, as one algorithm may perform better than the other in certain scenarios, and the other way around in others. Therefore, we first must understand how different parametrizations impact algorithm performance before comparing execution times.

First, we evaluate different parametrizations for BRELMAR-CASH and CASH-MR. Second, we apply both algorithms to the same datasets and compare their average run-times. Third, we interpret the results and draw a conclusion.

6.3.1 BRELMAR-CASH Performance Impact of Maximum Recursion Depth, Minimum Points, and Initial Scale Factor

Section 6.2 shows that the number of clusters found has no significant impact on the algorithm's performance. In this section, we apply BRELMAR-CASH to the linear dataset defined in this chapter with various recursion depths and initial scale factors. Using a higher initial scale factor with a simultaneously lower recursion limit finds dense grid cells (clusters) of the same area and volume. We parametrize BRELMAR-CASH in a fashion which returns clusters with area of $0.2 \times 0.2 = 0.04$ (delta of 0.2), for example a cluster spanning $\{(1, 1), (1.2, 1.2)\}$.

Figure 6.13 shows the algorithm's run-time for various parametrizations and the number of clusters found in each execution. We applied BRELMAR-CASH with parallelism of 2, maximum recursion $mr \in \{0, 2, 4\}$, and initial scale factor $sf \in \{0, 2, 4\}$ to the linear dataset.

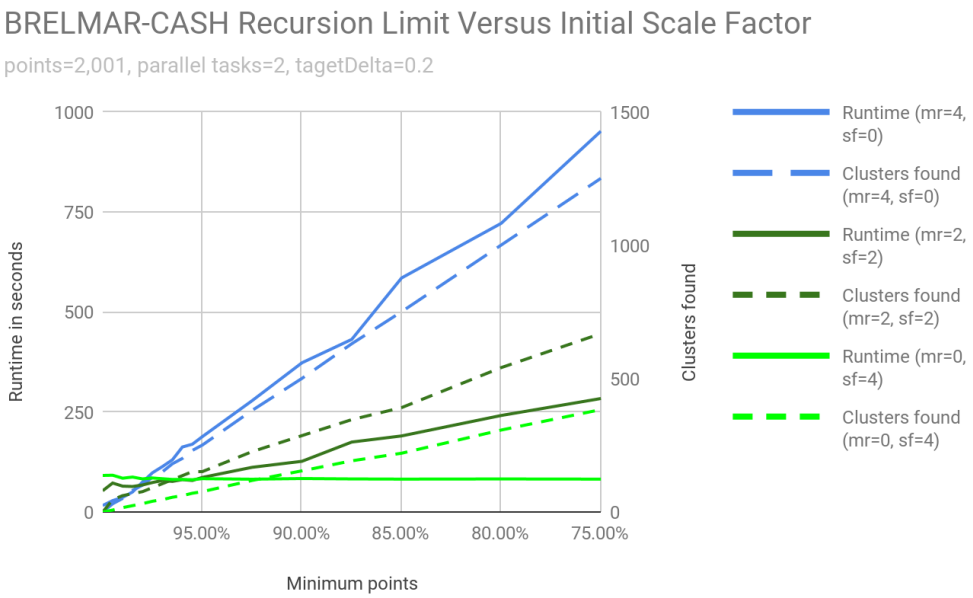

Figure 6.13: Applying BRELMAR-CASH with maximum recursions $mr \in \{0, 2, 4\}$ and initial scale factor $sf \in \{0, 2, 4\}$ to the linear dataset from.

Minimum Points	**Clusters Found**	**Average Runtime**	**RSD**
2000 (99.95%)	0	16.38s	1.04%
1990 (99.45%)	31	28.57s	1.82%
1980 (98.95%)	51	36.55s	0.40%
1970 (98.45%)	81	50.19s	4.68%
1960 (97.95%)	101	75.04s	8.49%
1950 (97.45%)	131	98.26s	2.37%
1940 (96.95%)	151	113.70s	3.85%
1930 (96.45%)	181	130.58s	1.33%
1920 (95.95%)	201	162.54s	3.16%
1910 (95.45%)	231	169.53s	1.06%
1900 (94.95%)	251	188.38s	3.61%
1850 (92.45%)	381	278.35s	1.05%
1800 (89.96%)	501	373.02s	3.52%
1750 (87.46%)	631	431.55s	9.41%
1700 (84.96%)	751	585.06s	7.01%
1600 (79.96%)	1001	721.61s	3.33%
1500 (74.96%)	1251	951.83s	3.11%

Table 6.7: Benchmark results for $mr = 4 \wedge sf = 0$.

Minimum Points	Clusters Found	Average Runtime	RSD
2000 (99.95%)	0	53.19	0.30%
1990 (99.45%)	31	72.49	22.91%
1980 (98.95%)	41	64.75	0.76%
1970 (98.45%)	46	63.84	0.72%
1960 (97.95%)	51	67.68	0.38%
1950 (97.45%)	61	73.83	2.14%
1940 (96.95%)	71	78.75	0.49%
1930 (96.45%)	81	76.58	12.34%
1920 (95.95%)	91	80.99	16.48%
1910 (95.45%)	101	78.86	0.48%
1900 (94.95%)	101	86.56	4.08%
1850 (92.45%)	151	111.80	0.66%
1800 (89.96%)	191	126.81	5.64%
1750 (87.46%)	231	174.94	2.76%
1700 (84.96%)	261	190.23	0.52%
1600 (79.96%)	361	241.35	7.27%
1500 (74.96%)	446	283.68	0.41%

Table 6.8: Benchmark results for $mr = 2 \wedge sf = 2$.

Minimum Points	Clusters Found	Average Runtime	RSD
2000 (99.95%)	0	91.47	2.28%
1990 (99.45%)	5	92.03	11.53%
1980 (98.95%)	11	84.60	2.52%
1970 (98.45%)	16	87.40	4.84%
1960 (97.95%)	21	82.85	1.86%
1950 (97.45%)	27	85.18	2.00%
1940 (96.95%)	31	83.24	2.91%
1930 (96.45%)	37	81.46	1.32%
1920 (95.95%)	41	81.40	0.26%
1910 (95.45%)	47	80.85	1.98%
1900 (94.95%)	51	83.37	1.64%
1850 (92.45%)	79	82.01	1.28%
1800 (89.96%)	103	83.73	3.36%
1750 (87.46%)	128	82.72	1.89%
1700 (84.96%)	147	82.25	0.47%
1600 (79.96%)	205	82.67	0.32%
1500 (74.96%)	256	82.24	2.39%

Table 6.9: Benchmark results for $mr = 0 \wedge sf = 4$.

We make the following observations:

- Given $mr = 0 \wedge sf = 4$ (Table 6.7), run-time remains constant over mp and the fewest clusters are found, compared to $mr \in \{2, 4\}$
- Given $mr = 4 \wedge sf = 0$ (Table 6.9), less time is needed to find clusters when $mp > 95\%$, but run-time increase is - at least - linear. Compared to $mr \in \{0, 2\}$, the most clusters are found.

It is apparent that BRELMAR-CASH performs extraordinarily well with few recursions, regardless of density criteria. We assume that the difference of clusters found is caused by approximation errors as explained in Section 5.2.3. Improving this behavior and identifying steps for purging, prioritizing, or filtering cluster candidates in recursive descent iterations is not included in this work due to brevity but should be explored in future work.

We hypothesize that BRELMAR-CASH could be used to identify regions of interest in applications where $mp < 50\%$. The results could be used as input to CASH-MR which would find clusters accurately and filter out noisy data and uninteresting regions.

6.3.2 CASH-MR Performance Impact of Recursion Depth and Minimum Points

In Section 5.1 we describe worst case scenarios with regards to run-time complexity. Here, we explore this property by applying CASH-MR to the random dataset with parallelism of 2, a total of 2001 data points, maximum recursion $mr \in \{5, 10, 15\}$, and minimum points $7.5\% < mp < 100\%$.

Figure 6.14 shows a significant exponential increase in total run-time when $mr = 15$ and a more graduate exponential increase when $mr = 10$, compared to $mr = 5$.

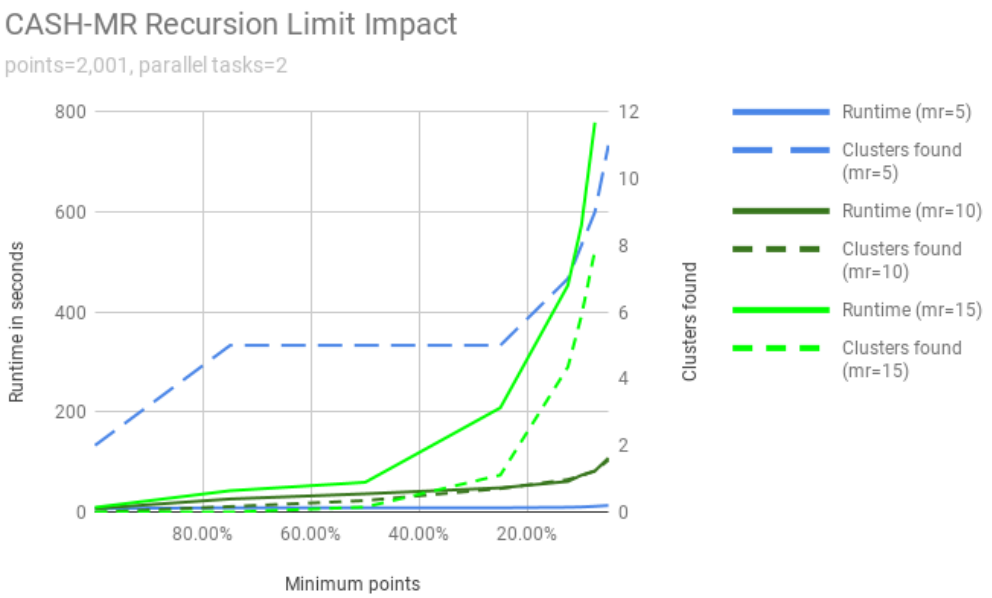

Figure 6.14: Applying CASH-MR with maximum recursions mr $\{5, 10, 15\}$ to the random dataset.

Minimum Points	**Clusters Found**	**Average Runtime**	**RSD**
2000 (99.95%)	2	7.02s	0.56%
1500 (74.96%)	5	8.61s	1.20%
1000 (49.98%)	5	8.64s	2.12%
500 (24.99%)	5	8.58s	2.14%
250 (12.49%)	7	9.57s	3.58%
200 (10.00%)	8	10.21s	1.22%
150 (07.50%)	9	11.55s	1.61%
100 (05.00%)	11	13.16s	0.36%

Table 6.10: Applying CASH-MR with $mr = 5$.

Minimum Points	Clusters Found	Average Runtime	RSD
2000 (99.95%)	0	7.11s	0.85%
1500 (74.96%)	11	26.24s	1.49%
1000 (49.98%)	23	36.53s	2.68%
500 (24.99%)	47	48.38s	18.32%
250 (12.49%)	65	61.25s	4.31%
200 (10.00%)	73	73.67s	2.11%
150 (07.50%)	82	82.28s	16.11%
100 (05.00%)	104	107.92s	11.93%

Table 6.11: Applying CASH-MR with $mr = 10$.

Minimum Points	Clusters Found	Average Runtime	RSD
2000 (99.95%)	0	9.42s	4.01%
1500 (74.96%)	0	42.72s	2.57%
1000 (49.98%)	10	59.53s	1.59%
500 (24.99%)	74	208.50s	3.05%
250 (12.49%)	290	453.31s	0.06%
200 (10.00%)	392	572.78s	5.85%
150 (07.50%)	523	778.85s	16.56%

Table 6.12: Applying CASH-MR with $mr = 15$.

It is apparent that CASH-MR has exponential runtime when many cluster candidates exist.

6.3.3 CASH-MR versus BRELMAR-CASH Performance

We apply CASH-MR and BRELMAR-CASH to the three datasets defined in this chapter. The expected cluster area is set to $0.25 \times 0.25 = 0.0625$, which guarantees that both algorithms find clusters of the same size.

BRELMAR-CASH yields constant time when $mr = 0$ and is thus expected to perform better than CASH-MR when the minimum points threshold mp is small. Furthermore, we forecast that BRELMAR-CASH may perform worse than CASH-MR when $mr > 0$, as BRELMAR-CASH suffers the same $O(n^2)$ spatial descent run-time complexity and finds more cluster candidates on average. We benchmark different parametrizations of BRELMAR-CASH in order to get a balanced and fair comparison. We run benchmarks on $pt = \{1, 4\}$ and apply BRELMAR-CASH with parameters $sf = 0 \wedge mr = 3$ and $sf = 3 \wedge mr = 0$.

6.3.3.1 Random Dataset Benchmarks

Applying both algorithms to the random dataset results in the various observations.

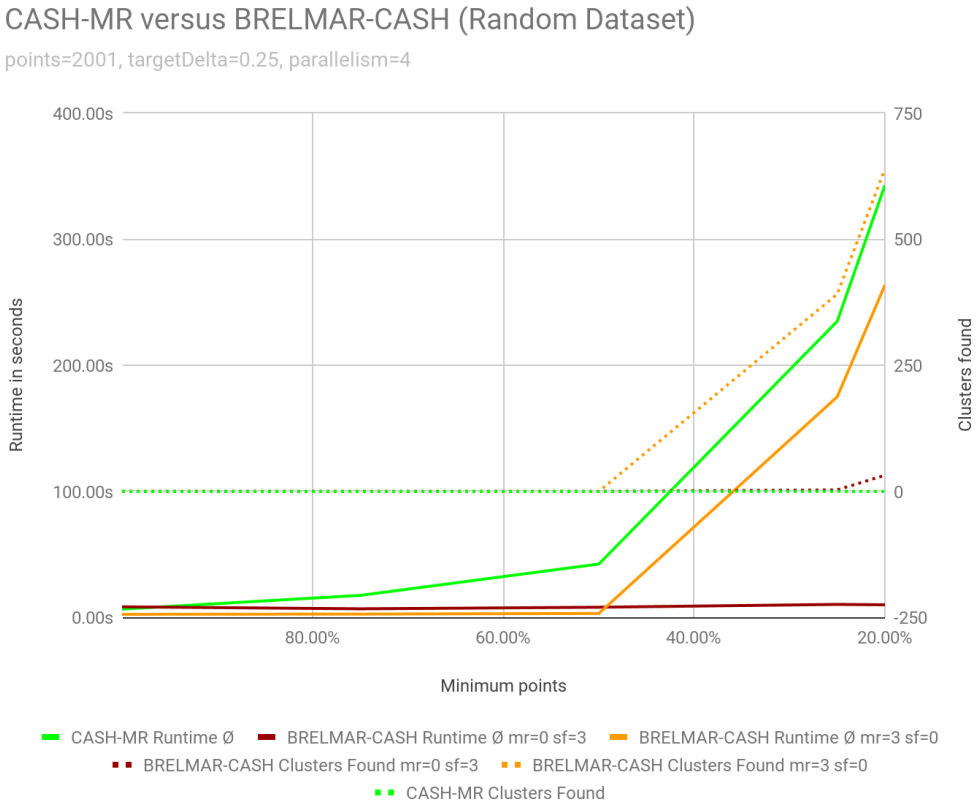

Figure 6.15: Plotted benchmark results for applying BRELMAR-CASH and CASH-MR to the random dataset.

First, BRELMAR-CASH, with a high mp threshold, requires more execution time with $sf = 4 \wedge mr = 0$ than with $sf = 0 \wedge mr = 3$. Second, BRELMAR-CASH executes in constant time when $sf = 3 \wedge mr = 0$, regardless of how mp is chosen. Overall we observe that BRELMAR-CASH is capable of processing low mp thresholds in constant time when recursion is disabled.

The experiments show that CASH-MR is generally slower than BRELMAR-CASH when applied to this dataset, regardless of the latter's configuration. The most likely cause is that CASH-MR finds many regions of interest which result in exponential run-time. While the same applies BRELMAR-CASH as well, the lower recursion depth results in fewer search branches and thus less

execution time.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	6.63s
1500 (74.96%)	0	17.56s
1000 (49.98%)	0	42.40s
500 (24.99%)	0	234.87s
400 (19.99%)	0	342.70s

Table 6.13: Detailed benchmark results for CASH-MR when applied to the random dataset.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	8.48s
1500 (74.96%)	0	6.90s
1000 (49.98%)	0	8.15s
500 (24.99%)	3	10.38s
400 (19.99%)	32	10.06s

Table 6.14: Detailed benchmark results for BRELMAR-CASH with $mr = 0 \wedge sf = 3$ when applied to the random dataset.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	2.56s
1500 (74.96%)	0	2.71s
1000 (49.98%)	0	3.21s
500 (24.99%)	391	175.02s
400 (19.99%)	639	263.60s

Table 6.15: Detailed benchmark results for BRELMAR-CASH with $mr = 3 \wedge sf = 0$ when applied to the random dataset.

6.3.3.2 Linear-Origin Scattered Dataset Benchmarks

We observe similar patterns as in Section 6.3.3.1 with regards to the constant execution time of BRELMAR-CASH. In this experiment however, CASH-MR outperforms BRELMAR-CASH when configured with $mr = 3 \wedge sf = 0 \wedge min\_points < 25\%$.

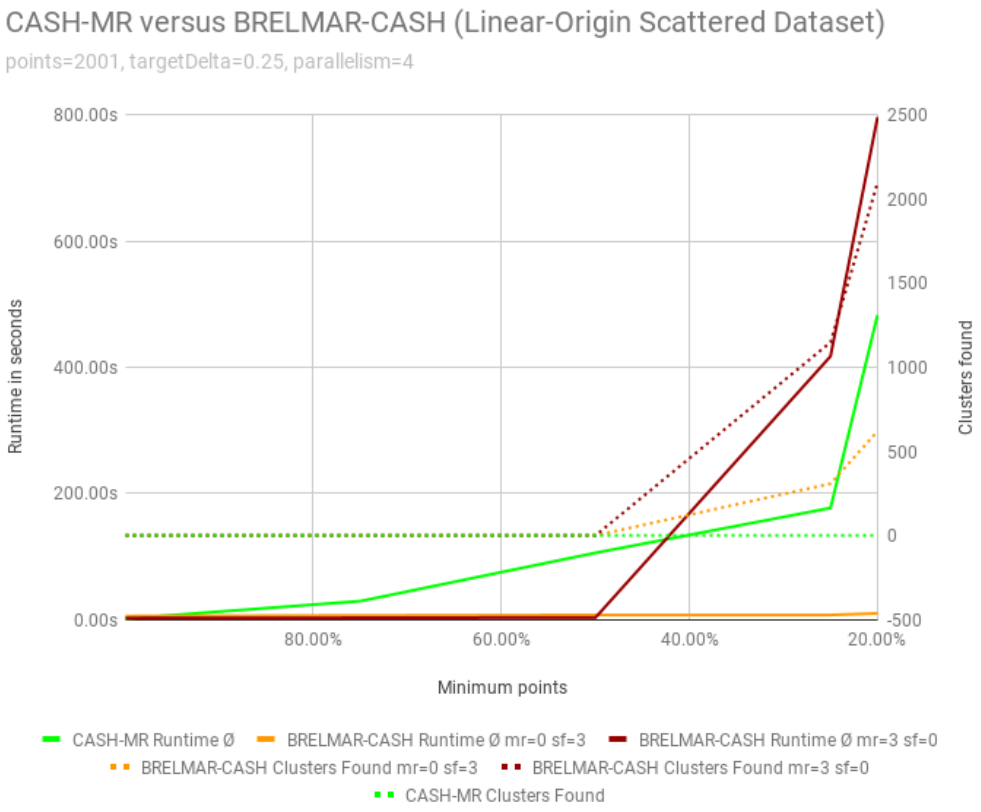

Figure 6.16: Plotted benchmark results for applying BRELMAR-CASH and CASH-MR to the linear-origin scattered dataset.

Minimum Points	**Clusters Found**	**Average Runtime**
2000 (99.95%)	0	1.50s
1500 (74.96%)	0	29.26s
1000 (49.98%)	0	105.81s
500 (24.99%)	0	176.95s
400 (19.99%)	0	483.09s

Table 6.16: Detailed benchmark results for CASH-MR when applied to the linear-origin scattered dataset.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	5.56s
1500 (74.96%)	0	6.66s
1000 (49.98%)	0	7.16s
500 (24.99%)	308	7.32s
400 (19.99%)	620	9.73s

Table 6.17: Detailed benchmark results for BRELMAR-CASH with $mr = 0 \wedge sf = 3$ when applied to the linear-origin scattered dataset.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	2.39s
1500 (74.96%)	0	2.73s
1000 (49.98%)	0	3.25s
500 (24.99%)	1149	417.54s
400 (19.99%)	2098	796.68s

Table 6.18: Detailed benchmark results for BRELMAR-CASH with $mr = 3 \wedge sf = 0$ when applied to the linear-origin scattered dataset.

6.3.3.3 Linear Dataset Benchmarks

CASH's spatial descent search strategy finds many potential cluster candidates (see Figure 6.17) in the linear dataset when split indices are low. In this benchmark, both CASH-MR and BRELMAR-CASH fail to compute results when $mp < 40\%$ due to this effect. Configuring BRELMAR-CASH without spatial descent ($mr = 0 \wedge sf = 3$) is insensitive to the number of cluster candidates and is thus the only configuration which provided meaningful results. Additionally, we observe that CASH-MR executes faster than BRELMAR-CASH with $mr = 3$ because it finds fewer cluster candidates during spatial descent.

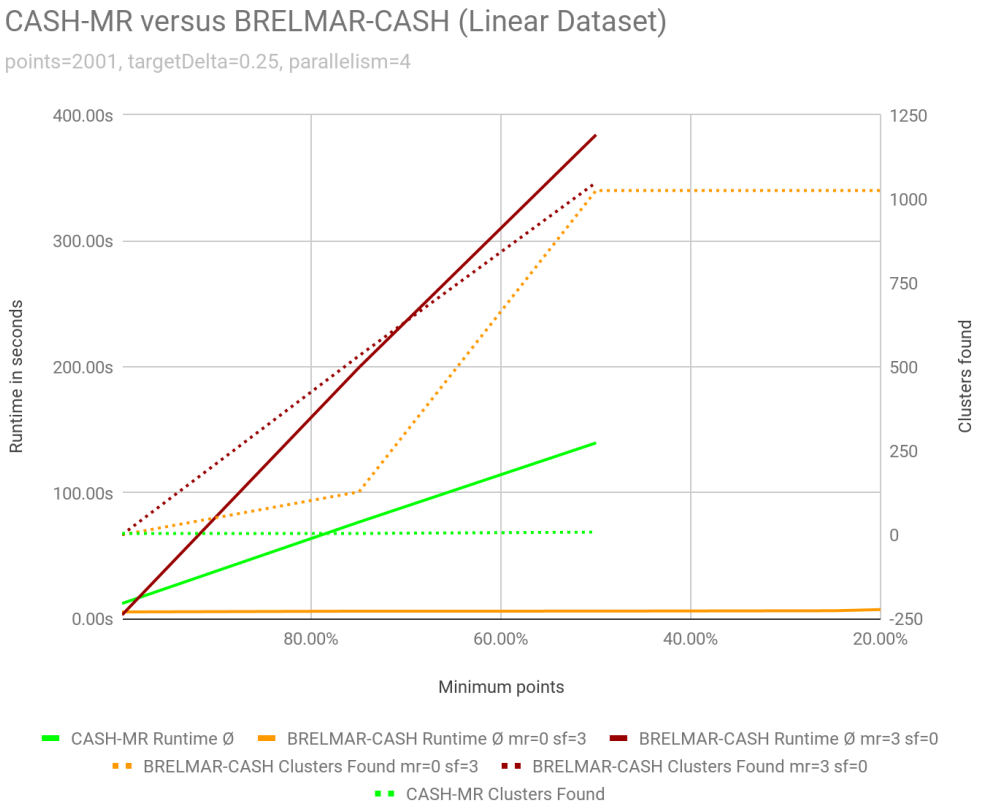

Figure 6.17: Plotted benchmark results for applying BRELMAR-CASH and CASH-MR to the linear dataset.

Minimum Points	**Clusters Found**	**Average Runtime**
2000 (99.95%)	4	12.13s
1500 (74.96%)	4	76.77s
1000 (49.98%)	8	139.64s
500 (24.99%)		
400 (19.99%)		

Table 6.19: Detailed benchmark results for CASH-MR when applied to the linear dataset. Red cells indicate that the program exited with an error code.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	5.41s
1500 (74.96%)	128	6.04s
1000 (49.98%)	1025	6.06s
500 (24.99%)	1025	6.36s
400 (19.99%)	1025	7.31s

Table 6.20: Detailed benchmark results for BRELMAR-CASH with $mr = 0 \wedge sf = 3$ when applied to the linear dataset.

Minimum Points	Clusters Found	Average Runtime
2000 (99.95%)	0	3.01s
1500 (74.96%)	533	199.52s
1000 (49.98%)	1049	384.24s
500 (24.99%)		
400 (19.99%)		

Table 6.21: Detailed benchmark results for BRELMAR-CASH with $mr = 3 \wedge sf = 0$ when applied to the linear dataset. Red cells indicate that the program exited with an error code.

6.3.4 Conclusion

We applied both algorithms to synthetic data with various parametrizations in this chapter. We conclude that both are capable of solving the problem of correlation clustering in the context of BigData efficiently by leveraging a MapReduce framework. In comparison, none of the two algorithms is explicitly better, with regards to run-time, than the other. Instead, they execute more efficiently than their counterpart in certain scenarios.

It is notable that BRELMAR-CASH finds, on average, more cluster candidates than CASH-MR. This may be caused by effects described in Sections 5.2.3 and 5.2.2. Most of the clusters found are neighbors which impact output quality negatively. Future work could examine if using BRELMAR-CASH's results as input to CASH-MR reduces run-time complexity in certain scenarios. We hypothesize that BRELMAR-CASH could be used to roughly identify regions of interest. These regions could be passed to CASH-MR to exactly determine clusters, with reduced run-time.

Chapter 7

Discussion and Future Work

This chapter summarizes the thesis, discusses its findings, points out limitations of the approach explained in this paper, and outlines directions for future research.

Based on our hypothesis that linear correlation clustering can be parallelized, we introduced parallelization to the CASH algorithm by exploring and identifying MapReduce-able operations within the algorithm. We further hypothesized that CASH requires, in worst case scenarios, exponential runtime, and proposed an adaptation called BRELMAR-CASH that reduces the needed amount of spatial descent iterations by leveraging Bresenham's line algorithm.

We implemented CASH-MR and BRELMAR-CASH using Python and the Apache Flink Java SDK. We showed that CASH-MR is capable of significantly reducing total runtime when running on multiple worker nodes. We further demonstrated that BRELMAR-CASH is parallelizable as well, and that the algorithm is capable of reducing the overall runtime in specific scenarios. Additionally, we showed in particular that, due to Apache Flink's technical limitations, some parallelization concepts do not work.

For brevity, this paper's scope introduces various limitations. The developed algorithms work with 2-dimensional datasets only and use Cartesian coordinates instead of polar coordinates. These limitations do not impact the applicability of our findings to CASH. Instead, we identified further areas where MapReduce could be beneficial, for example in recursive descent operations.

We explored approximation issues with Bresenham's line algorithm in spatial descent iterations. We discussed that improving BRELMAR-CASH's pruning strategies and approximation errors, and finding optimal scale models should be explored in future work, as well as implementing CASH-MR with support for n-dimensional datasets and the polar coordinate system.

We discovered that BRELMAR-CASH finds more clusters of lower quality

compared to CASH-MR due to BRELMAR-CASH's constant execution time when spatial descent is disabled. We hypothesized that both algorithms could be combined to reduce execution time in future work.

Distributed linear correlation clustering could become an essential component of finding meaningful patterns in huge, high-dimensional data. By developing CASH-MR and BRELMAR-CASH, we present the theory and validation for developing extremely efficient algorithms capable of addressing the task.

Linear correlation clustering in high-dimensional data has been solved for over a decade by algorithms such as CASH, 4C, and ORCLUS. However, none of these algorithms, to our knowledge, was successfully parallelized previously. By applying MapReduce to CASH, we provide novel methods for finding linearly correlated clusters in the context of BigData. Our results demonstrate that both methods - CASH-MR and BRELMAR-CASH - achieve significant speedup ratio and parallel efficiency. We postulate that the generalization of our findings, specifically with respect to high-dimensional data, will lead to improvements in the broader context of finding interesting patterns in large databases.

7.1 Acknowledgment

I want to thank the Database and Data Mining Research Institute at the Ludwig-Maximilians-Universität München and specifically Daniyal Kazempour for their support in writing this thesis. I would also like to thank my parents for supporting me in my studies.

Appendix A

CASH-MR Python Source Code

The source code is also available on GitHub[1] and the attached USB stick.

```
\% matplotlib notebook

from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
import functools
from time import time

# This is a constant that can be used to approach 0
close_to_zero = 1.0 / 10.0 ** 200


# in the form of (mins, maxes). For example (2d dataset):
#
#  min_x, min_y  max_x, max_y
# [[  -1,  -2], [2,     1]]
def compute_min_max(points):
  # [x, y]
  minima = []
  maxima = []

  for p in points:
    for k, v in enumerate(p):

      if k >= len(minima):
```

[1] https://github.com/arekkas/cash-mr/blob/master/cash/cash-mr.ipynb

```
        minima.append(v)
      elif v < minima[k]:
        minima[k] = v

      if k >= len(maxima):
        maxima.append(v)
      elif v > maxima[k]:
        maxima[k] = v

  return [tuple(minima), tuple(maxima)]


def draw_image_space(points):
  fig = plt.figure()
  sub = fig.add_subplot(111)
  bounds = compute_min_max(points)
  sub.axis([bounds[0][0] - 1, bounds[1][0] + 1, bounds[0][1] -
   ↪ 1, bounds[1][1] + 1])
  draw_2d_points(points, sub, "xb")


def draw_2d_points(points, sub, style):
  sub.plot(
    list(map(lambda p: p[0], points)),
    list(map(lambda p: p[1], points)),
    style
  )


def create_plot(axis):
  fig = plt.figure()
  sub = fig.add_subplot(111)
  delta_x = axis[1][0] - axis[0][0]
  delta_y = axis[1][1] - axis[0][1]
  sub.axis([
    axis[0][0] - delta_x / 2,
    axis[1][0] + delta_x / 2,
    axis[0][1] - delta_y / 2,
    axis[1][1] + delta_y / 2
  ])
  return sub
```

```
# Takes a point and a boundary. Then, it computes the parameter
    ↪ function of said point. Next, the intersections
# of said function with the given boundary are returned. If no
    ↪ intersection with the boundary exists, an empty
# list is returned.
#
# Given y = mx + t,
# the horizontal axis (first tuple element) is the parameter m
# and the vertical axis (second tuple element) is the parameter
    ↪ t,
# for example: (1,0) -> y = 1*x + 0
#
# Usage of this function:
#
# list_parameter_space_intersections_with_boundaries((1,1),
    ↪ [(-1, -1), (1, 1)])
# [([1, 0], [0, 1])]
#
# Meaning that we have two intersections, the first at m=1 and t
    ↪ =0, and the second at m=0 and t=1
def list_parameter_space_intersections_with_boundaries(point,
    ↪ bounds):
  x = point[0]
  y = point[1]

  # We can't divide by 0 so let's use a value which is very
    ↪ close to zero instead.
  if x == 0:
    x = close_to_zero

  bound_left_m = bounds[0][0]
  bound_right_m = bounds[1][0]

  bound_lower_t = bounds[0][1]
  bound_upper_t = bounds[1][1]

  # uip = upper intersection point
  uip_t = y - bound_right_m * x
  uip_m = (y - bound_upper_t) / x
```

```
  # lip = lower intersection point
  lip_t = y - bound_left_m * x
  lip_m = (y - bound_lower_t) / x

  bx1, by1 = bounds[0]
  bx2, by2 = bounds[1]

  pts = np.array([
    (bound_right_m, uip_t),
    (uip_m, bound_upper_t),
    (bound_left_m, lip_t),
    (lip_m, bound_lower_t),
  ])
  lower_left_intersections = np.array([bx1, by1])
  upper_right_intersections = np.array([bx2, by2])

  indices = np.all(np.logical_and(lower_left_intersections <=
   ↪ pts, pts <= upper_right_intersections), axis=1)
  # To compute the points which lay outside of our boundary, use
   ↪ :
  # outbox = pts[np.logical_not(indices)]

  intersections = list(map(tuple, pts[indices]))

  # de-duplicate any points
  intersections = list(set(intersections))

  return intersections


#
#
#
# CASH-MR specific logic Barrier
#
#
#

def draw_parameter_space(points, boundaries, plot):
  for p in points:
```

```
    intersections =
    ↪ list_parameter_space_intersections_with_boundaries(p,
    ↪ boundaries)
    draw_2d_points(intersections, plot, "g")


# this splits a room into two, either vertically or horizontally
def split_room(bounds, index):
  lx, ly = bounds[0]
  ux, uy = bounds[1]

  if index % 2 == 0:
    # split left right
    mx = (ux - lx) / 2
    return [
      [(lx, ly), (ux - mx, uy)],
      [(lx + mx, ly), (ux, uy)],
    ]

  my = (uy - ly) / 2
  return [
    [(lx, ly), (ux, uy - my)],
    [(lx, ly + my), (ux, uy)],
  ]


# counts intersection in an area
def count_intersections_in_area(intersections):
  return functools.reduce(
    lambda l, n: l + n,
    map(
      lambda p: p[1],
      intersections
    ),
    0
  )


def run_original_recursive_descent_iteration(point, boundaries):
  intersections =
   ↪ list_parameter_space_intersections_with_boundaries(point,
```

```
    ↪ boundaries)

  return [boundaries, 0 if len(intersections) == 0 else 1, [
    ↪ point]]


# runs recursive descent, returns a list of areas of intrest.
def run_original_recursive_descent(points, boundaries,
    ↪ min_points, recursion_limit, current_recursion, plot,
    ↪ visualize):
  intersections = list(map(
    lambda point: run_original_recursive_descent_iteration(point
    ↪ , boundaries),
    points
  ))

  intersection_count = count_intersections_in_area(intersections
    ↪ )
  if intersection_count < min_points:
    # print "exiting", bounds, "because intersections is lower
    ↪ than minf", intersections, '<', minf
    return []

  # for p in intersections_per_point:
  #   print("Bresen intersections:", p)

  # satisfy_min_points = list(filter(lambda point: point[1] >=
    ↪ min_points, intersections_per_point))

  if current_recursion >= recursion_limit:
    combined_intersections = list(functools.reduce(
      lambda p, n: [n[0], p[1] + n[1], p[2] + n[2]],
      intersections,
      [[], 0, []]
    ))

    return [combined_intersections]

  if visualize:
    # Visualize the boundaries (aka areas of interest) for each
    ↪ recursion
```

```
    plot.plot(
      [boundaries[0][0], boundaries[1][0], boundaries[1][0],
   ↪ boundaries[0][0], boundaries[0][0]],
      [boundaries[0][1], boundaries[0][1], boundaries[1][1],
   ↪ boundaries[1][1], boundaries[0][1]],
      "b"
    )

  rooms = split_room(boundaries, current_recursion)
  next_recursion = current_recursion + 1
  branches = map(
    lambda room: run_original_recursive_descent(
      points=points,
      boundaries=room,
      min_points=min_points,
      recursion_limit=recursion_limit,
      current_recursion=next_recursion,
      plot=plot,
      visualize=visualize
    ),
    rooms
  )

  non_empty_branches = list(filter(lambda branch: len(branch) >
   ↪ 0, branches))

  if (len(non_empty_branches)) == 0:
    return []

  combined_branches = list(functools.reduce(
    lambda p, n: p + n,
    non_empty_branches
  ))

  # print("got branches", combined_branches)

  return combined_branches


def run_cash_mr_bresenham(points, boundaries, min_points,
   ↪ recursion_limit, visualize):
```

```
  plot = None
  if visualize:
    # draw_image_space(points)
    plot = create_plot(boundaries)
    draw_parameter_space(points, boundaries, plot)

  areas = run_original_recursive_descent(
    points=points,
    boundaries=boundaries,
    min_points=min_points,
    recursion_limit=recursion_limit,
    current_recursion=0,
    plot=plot,
    visualize=visualize
  )

  for area in areas:
    delta = area[0][1][0] - area[0][0][0]
    print("Found subspace cluster", area[0], "with delta", delta
    ↪ , "and frequency", area[1], "and", len(area[2]),
        "points belonging to it")


data = [(x, x) for x in range(-10, 11)]
min_points = len(data)
recursion_limit = 10

start_time = time()
run_cash_mr_bresenham(
  boundaries=[(-2, -2), (2, 2)],
  points=data,
  min_points=min_points,
  recursion_limit=recursion_limit,
  visualize=True
)
elapsed_time = time() - start_time
print("CASH-MR-Bresenham took ", elapsed_time)
```

Appendix B

BRELMAR-CASH Python Source Code

The source code is also available on GitHub[1] and the attached USB stick.

```
\% matplotlib notebook

from __future__ import division

import matplotlib.pyplot as plt
import numpy as np
from math import ceil
from time import time
import functools

# This is a constant that can be used to approach 0
close_to_zero = 1.0 / 10.0 ** 200


# in the form of (mins, maxes). For example (2d dataset):
#
#  min_x, min_y  max_x, max_y
# [[  -1,  -2], [2,     1]]
def compute_min_max(points):
  # [x, y]
  minima = []
  maxima = []
```

[1] https://github.com/arekkas/cash-mr/blob/master/cash/cash-mr-bresenham.ipynb

```
  for p in points:
    for k, v in enumerate(p):

      if k >= len(minima):
        minima.append(v)
      elif v < minima[k]:
        minima[k] = v

      if k >= len(maxima):
        maxima.append(v)
      elif v > maxima[k]:
        maxima[k] = v

  return [tuple(minima), tuple(maxima)]


def draw_image_space(points):
  fig = plt.figure()
  sub = fig.add_subplot(111)
  bounds = compute_min_max(points)
  sub.axis([bounds[0][0] - 1, bounds[1][0] + 1, bounds[0][1] -
   ↪ 1, bounds[1][1] + 1])
  draw_2d_points(points, sub, "xb")


def draw_2d_points(points, sub, style):
  sub.plot(
    list(map(lambda p: p[0], points)),
    list(map(lambda p: p[1], points)),
    style
  )


def reduce_intersections(prev, nex):
  last = prev[-1]
  if last[0] == nex[0]:
    return prev[:-1] + [[nex[0], last[1] + nex[1], last[2] + nex
   ↪ [2]]]
  return prev + [nex]
```

```
def create_plot(axis):
  fig = plt.figure()
  sub = fig.add_subplot(111)
  delta_x = axis[1][0] - axis[0][0]
  delta_y = axis[1][1] - axis[0][1]
  sub.axis([
    axis[0][0] - delta_x / 2,
    axis[1][0] + delta_x / 2,
    axis[0][1] - delta_y / 2,
    axis[1][1] + delta_y / 2
  ])
  return sub


# Takes a point and a boundary. Then, it computes the parameter
   ↪ function of said point. Next, the intersections
# of said function with the given boundary are returned. If no
   ↪ intersection with the boundary exists, an empty
# list is returned.
#
# Given y = mx + t,
# the horizontal axis (first tuple element) is the parameter m
# and the vertical axis (second tuple element) is the parameter
   ↪ t,
# for example: (1,0) -> y = 1*x + 0
#
# Usage of this function:
#
# list_parameter_space_intersections_with_boundaries((1,1),
   ↪ [(-1, -1), (1, 1)])
# [([1, 0], [0, 1])]
#
# Meaning that we have two intersections, the first at m=1 and t
   ↪ =0, and the second at m=0 and t=1
def list_parameter_space_intersections_with_boundaries(point,
   ↪ bounds):
  x = point[0]
  y = point[1]

  # We can't divide by 0 so let's use a value which is very
```

```
 ↪ close to zero instead.
if x == 0:
  x = close_to_zero

bound_left_m = bounds[0][0]
bound_right_m = bounds[1][0]

bound_lower_t = bounds[0][1]
bound_upper_t = bounds[1][1]

# uip = upper intersection point
uip_t = y - bound_right_m * x
uip_m = (y - bound_upper_t) / x

# lip = lower intersection point
lip_t = y - bound_left_m * x
lip_m = (y - bound_lower_t) / x

bx1, by1 = bounds[0]
bx2, by2 = bounds[1]

pts = np.array([
  (bound_right_m, uip_t),
  (uip_m, bound_upper_t),
  (bound_left_m, lip_t),
  (lip_m, bound_lower_t),
])
lower_left_intersections = np.array([bx1, by1])
upper_right_intersections = np.array([bx2, by2])

indices = np.all(np.logical_and(lower_left_intersections <=
 ↪ pts, pts <= upper_right_intersections), axis=1)
# To compute the points which lay outside of our boundary, use
 ↪ :
# outbox = pts[np.logical_not(indices)]

intersections = list(map(tuple, pts[indices]))

# de-duplicate any points
intersections = list(set(intersections))
```

```
  return intersections


#
#
#
# CASH-MR-Bresenham logic barrier
#
#
#

# Returns the bresenham points given two points and a grid
   ↪ resolution
def bresenham_transform(points, resolution):
  return list(
    bresenham(
      points[0][0], points[0][1],
      points[1][0], points[1][1],
      resolution
    )
  )


# This map function takes a point and boundaries and computes
   ↪ the intersections of the hough transform of said point
   ↪ with said boundaries.
# It returns an array where the first key is the point and the
   ↪ second key is an array of bresenham points.
# [(1,1), [(-10,-10],(-9,-9), ...]
def compute_bresenham_points(point, boundaries, plot,
   ↪ bresenham_resolution, visualize):
  ht_boundary_intersections =
   ↪ list_parameter_space_intersections_with_boundaries(point,
   ↪ boundaries)

  if len(ht_boundary_intersections) < 2:
    # print("Data point",p,"with Hough Transforms",
   ↪ ht_boundary_intersections,"does not have a Hough Transform
   ↪  function that intersects with bounds",bounds)
    return [point, []]
```

```
  if visualize:
    draw_2d_points(ht_boundary_intersections, plot, "b")

  ht_bresenham_points = bresenham_transform(
   ↪ ht_boundary_intersections, bresenham_resolution)

  if visualize:
    draw_2d_points(ht_bresenham_points, plot, "gx")

  return [point, ht_bresenham_points]


def find_areas_of_interest_with_bresen(points, boundaries, plot,
   ↪  min_points, bresenham_resolution, visualize):
  # a list of points to plot so we get an idea what's going on
  # plotht = bounds
  bresenpoints = list(filter(
    lambda point: len(point[1]) > 0,
    map(
      lambda point: compute_bresenham_points(
        point=point,
        boundaries=boundaries,
        plot=plot,
        bresenham_resolution=bresenham_resolution,
        visualize=visualize
      ),
      points
    )
  ))

  if len(bresenpoints) == 0:
    return []

  # This flattens the bresenpoints, the result is a list of:
  # [[bresenpoint_1, 1, [original_datapoint_1]], [bresenpoint_2,
   ↪  1, [original_datapoint_1]], ...]
  flattened_bresenpoints = functools.reduce(
    lambda prev, nex: prev + list(map(
      lambda p: [p, 1, [nex[0]]],
      nex[1]
    )),
```

```
    bresenpoints,
    []
  )

  # print ("Pre-proccessed bresenham points",
   ↪ flattened_bresenpoints)
  sorted_bresenpoints = sorted(flattened_bresenpoints, key=
   ↪ lambda p: (p[0][0], p[0][1]))
  # for p in sorted_bresenpoints:
  #   print("Sorted bresenham:", p)

  intersections_per_point = functools.reduce(
    reduce_intersections,
    sorted_bresenpoints[1:],
    [sorted_bresenpoints[0]]
  )

  # for p in intersections_per_point:
  #   print("Bresen intersections:", p)

  satisfy_min_points = list(filter(lambda point: point[1] >=
   ↪ min_points, intersections_per_point))

  for smp in satisfy_min_points:
    # print ("Bresen intersection satisfying min points", minf,
   ↪ "found:", p)
    if visualize:
      draw_2d_points([smp[0]], plot, "or")

  # print("bresen points", satisfy_min_points)
  return satisfy_min_points


def bresenham_recursive_descent_iteration(area, min_points,
   ↪ bresenham_resolution, visualize, force_boundaries=None):
  points = area[2]

  boundaries = [
    (area[0][0] - 1, area[0][1] - 1),
    (area[0][0] + 1, area[0][1] + 1),
  ]
```

```
  if force_boundaries is not None:
    boundaries = force_boundaries

  plot = None
  if visualize:
    plot = create_plot(boundaries)

  return find_areas_of_interest_with_bresen(
    points=points,
    boundaries=boundaries,
    plot=plot,
    min_points=min_points,
    bresenham_resolution=bresenham_resolution,
    visualize=visualize
  )


# Areas is an argument that looks like [(0, 0), 2, [(1, 1), (2,
    ↪ 2)]] where (0,0) is a parameter-space point, 2 is
# the number of intersections and [(1, 1), (2, 2)] is a list (
    ↪ len always equals the second element) of points that cross
    ↪  said point.
def run_bresenham_recursive_descent(
    areas, min_points, recursion_limit, current_recursion,
    ↪ visualize,
    initial_boundaries=None
):
  if current_recursion > recursion_limit:
    return areas

  bresenham_resolution = next_recursion = current_recursion + 1

  new_areas = list(
    functools.reduce(
      lambda p, n: p + n,
      map(
        lambda area: bresenham_recursive_descent_iteration(
          area=area,
          min_points=min_points,
          bresenham_resolution=bresenham_resolution,
          visualize=visualize,
```

```
          force_boundaries=initial_boundaries
        ),
        areas
      ),
      []
    )
  )

  # Let's run the recursion. Do not pass the initial boundaries
   ↪ to consecutive recursions
  return run_bresenham_recursive_descent(
    areas=new_areas,
    min_points=min_points,
    recursion_limit=recursion_limit,
    visualize=visualize,
    current_recursion=next_recursion
  )


def bresenham(x0, y0, x1, y1, resolution):
  x0 = int(round(x0))
  x1 = int(round(x1))
  y0 = int(round(y0))
  y1 = int(round(y1))

  dx = x1 - x0
  dy = y1 - y0

  sx = 1.0 if dx >= 0 else -1.0
  sy = 1.0 if dy >= 0 else -1.0

  sx /= resolution
  sy /= resolution

  dx = abs(dx)
  dy = abs(dy)

  if dx >= dy:
    d = 2 * dy - dx
    delta_a = 2 * dy
    delta_b = 2 * dy - 2 * dx
```

```
    x = 0
    y = 0

    for i in range(int(ceil(dx * resolution) + 1)):
      yield (x + x0, y + y0)
      if d > 0:
        d += delta_b
        x += sx
        y += sy
      else:
        d += delta_a
        x += sx
  else:
    d = 2 * dx - dy
    delta_a = 2 * dx
    delta_b = 2 * dx - 2 * dy
    x = 0
    y = 0
    for i in range(int(ceil(dy * resolution) + 1)):
      yield (x + x0, y + y0)
      if d > 0:
        d += delta_b
        x += sx
        y += sy
      else:
        d += delta_a
        y += sy


def run_cash_mr_bresenham(points, min_points, recursion_limit,
    ↪ visualize, initial_boundaries):
  if visualize:
    draw_image_space(points)

  areas = run_bresenham_recursive_descent(
    areas=[[(0, 0), 0, points]],
    min_points=min_points,
    recursion_limit=recursion_limit,
    current_recursion=0,
    visualize=visualize,
```

```
    initial_boundaries=initial_boundaries
  )

  delta = (1.0/recursion_limit)
  for area in areas:
    boundary = [(area[0][0], area[0][1]), (area[0][0] + delta,
    ↪ area[0][1] + delta)]
    print("Found subspace cluster", boundary, "with delta",
    ↪ delta, "and frequency", area[1], "and", len(area[2]),
        "points belonging to it")


data = [(x, x) for x in range(-10, 11)]
min_points = len(data)
recursion_limit = 4

start_time = time()
run_cash_mr_bresenham(
  initial_boundaries=[(-2, -2), (2, 2)],
  points=data,
  min_points=min_points,
  recursion_limit=recursion_limit,
  visualize=True
)
elapsed_time = time() - start_time
print("CASH-MR-Bresenham took ", elapsed_time)
```

Appendix C

CASH-MR & BRELMAR-CASH Apache Flink Source Code

The source code is also available on GitHub[1] and the attached USB stick.

```
package org.arekkas.cashmr;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.util.Collector;

import java.io.Serializable;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.LongStream;

class Boundary2D implements Serializable {
  Point2D lower;
  Point2D upper;
```

[1] https://github.com/arekkas/cash-mr-java

```
  Boundary2D() {
  }

  Boundary2D(Double xl, Double yl, Double xu, Double yu) {
    this.lower = new Point2D(xl, yl);
    this.upper = new Point2D(xu, yu);
  }

  public String toString() {
    return String.format("[%s, %s]", lower, upper);
  }

  public String toKey() {
    return String.format("[%s, %s]", lower.toKey(), lower.toKey
    ↪ ());
  }
}

class Point2D implements Serializable {
  Double x;
  Double y;

  Point2D(Double x, Double y) {
    // In Java (because s) 0 and -0 exist. So we need to make
    ↪ sure we
    // only have 0 or the keys won't work
    if (x == 0.0) {
      x = Math.abs(x);
    }
    if (y == 0.0) {
      y = Math.abs(y);
    }

    this.x = x;
    this.y = y;
  }

  public String toString() {
    return String.format("(%f, %f)", x, y);
  }
```

```
  public String toKey() {
    return String.format("(%f; %f)", x, y);
  }
}

class ParameterSpaceIntersectionFinder {
  private Double closeToZero = 1.0 / Math.pow(10.0, 300);

  public static <T> Predicate<T> distinctByKey(Function<? super
   ↪ T, ?> keyExtractor) {
    Set<Object> seen = ConcurrentHashMap.newKeySet();
    return t -> seen.add(keyExtractor.apply(t));
  }

  List<Point2D> list(Point2D point, Boundary2D boundary) {
    if (point.x == 0) {
      point.x = this.closeToZero;
    }

    // y = t
    // x = m
    Double upperIntersectionT = point.y - boundary.upper.x *
   ↪ point.x;
    Double lowerIntersectionT = point.y - boundary.lower.x *
   ↪ point.x;

    Double upperIntersectionM = (point.y - boundary.upper.y) /
   ↪ point.x;
    Double lowerIntersectionM = (point.y - boundary.lower.y) /
   ↪ point.x;

    List<Point2D> points = Arrays.asList(
        new Point2D(boundary.upper.x, upperIntersectionT),
        new Point2D(upperIntersectionM, boundary.upper.y),
        new Point2D(boundary.lower.x, lowerIntersectionT),
        new Point2D(lowerIntersectionM, boundary.lower.y)
    );
    List<Point2D> intersections = new ArrayList<>();

    points.forEach((ixp) -> {
```

```
      if ((ixp.x >= boundary.lower.x && ixp.y >= boundary.lower.
    ↪ y)
          && (ixp.x <= boundary.upper.x && ixp.y <= boundary.
    ↪ upper.y)) {
        intersections.add(ixp);
      }
    });

    // Dedupe so we avoid this scenario:
    // Found intersection for (-1,00, -1,00) with [(2,00, 1,00),
    ↪  (2,00, 1,00), (0,00, -1,00), (-0,00, -1,00)]
    return intersections.stream()
        .filter(distinctByKey(Point2D::toKey))
        .collect(Collectors.toCollection(ArrayList::new));
  }
}

class Bounded2DArea {
  Boundary2D boundary;
  List<Point2D> points;
  Long intersections;

  Bounded2DArea() {

  }

  Bounded2DArea(Boundary2D boundary, Long intersections, List<
    ↪ Point2D> points) {
    this.boundary = boundary;
    this.points = points;
    this.intersections = intersections;
  }

  public String toString() {
    return String.format("[%s, %d, %s]", this.boundary.toString
    ↪ (), this.intersections, this.points);
  }

  public String toKey() {
    return this.boundary.toKey();
  }
```

```
}

class Bresenham2DArea {
  Point2D point;
  List<Point2D> points;
  Long intersections;

  Bresenham2DArea(Point2D point, Long intersections, List<
   ↪ Point2D> points) {
    this.point = point;
    this.points = points;
    this.intersections = intersections;
  }

  public String toString() {
    return String.format("[%s, %d, %s]", this.point.toString(),
   ↪ this.intersections, this.points);
  }
}

class BresenhamTransfomrations {
  Point2D point;
  List<Point2D> bresenhams;

  BresenhamTransfomrations(Point2D point, List<Point2D>
   ↪ bresenhams) {
    this.point = point;
    this.bresenhams = bresenhams;
  }

  public String toString() {
    return String.format("[%s, [%s]]", this.point, this.
   ↪ bresenhams);
  }
}

class Bresenham {
  List<Point2D> compute2D(Point2D p0, Point2D p1, Double
   ↪ resolution) {
    int x0 = (int) Math.round(p0.x);
    int x1 = (int) Math.round(p1.x);
```

```
    int y0 = (int) Math.round(p0.y);
    int y1 = (int) Math.round(p1.y);

    int dx = x1 - x0;
    int dy = y1 - y0;

    Double sx = dx >= 0 ? 1.0 : -1.0;
    Double sy = dy >= 0 ? 1.0 : -1.0;

    sx /= resolution;
    sy /= resolution;

    dx = Math.abs(dx);
    dy = Math.abs(dy);

    List<Point2D> result = new ArrayList<Point2D>();
    if (dx >= dy) {
      Double d = (2.0 * dy) - dx;
      Double deltaA = 2.0 * dy;
      Double deltaB = (2.0 * dy) - (2.0 * dx);

      Double x = 0.0;
      Double y = 0.0;

      int limit = (int) Math.ceil(dx * resolution) + 1;
      for (int i = 0; i < limit; i++) {
        // Ideally, this should use the resolution to decide
↪ where to look
        // result.add(new Boundary2D(x + x0, y + y0, x + x0 + sx
↪ , y + y0 + sy));
        result.add(new Point2D(x + x0, y + y0));

        // yield
        if (d > 0) {
          d += deltaB;
          x += sx;
          y += sy;
        } else {
          d += deltaA;
          x += sx;
        }
```

```
      }
    } else {
      int d = (2 * dx) - dy;
      int deltaA = 2 * dx;
      int deltaB = (2 * dx) - (2 * dy);

      Double x = 0.0;
      Double y = 0.0;

      int limit = (int) Math.ceil(dy * resolution) + 1;
      for (int i = 0; i < limit; i++) {
        result.add(new Point2D(x + x0, y + y0));

        if (d > 0) {
          d += deltaB;
          x += sx;
          y += sy;
        } else {
          d += deltaA;
          y += sy;
        }
      }
    }
    return result;
  }
}

class IDMapper implements MapFunction<Bounded2DArea,
    ↪ Bounded2DArea> {
  @Override
  public Bounded2DArea map(Bounded2DArea bounded2DArea) throws
    ↪ Exception {
    return bounded2DArea;
  }
}

class CashBresenhamAlgorithm2D {
  private Long minPoints;
  private Integer recursionLimit;
  private Integer initialBresenhamResolution;
  private ExecutionEnvironment env;
```

```
static <T> Predicate<T> distinctByKey(Function<? super T, ?>
 ↪ keyExtractor) {
  Set<Object> seen = ConcurrentHashMap.newKeySet();
  return t -> seen.add(keyExtractor.apply(t));
}

class BresenFlatMapper implements FlatMapFunction<
 ↪ BresenhamTransfomrations, Bresenham2DArea> {
  @Override
  public void flatMap(BresenhamTransfomrations b, Collector<
 ↪ Bresenham2DArea> out) {
    b.bresenhams.forEach(p -> {
      List<Point2D> ps = Arrays.asList(b.point);
      out.collect(new Bresenham2DArea(p, 1l, ps));
    });
  }
}

class Bounded2DAreaSelector implements KeySelector<
 ↪ Bresenham2DArea, String> {
  @Override
  public String getKey(Bresenham2DArea w) {
    return String.format("(%f; %f)", w.point.x, w.point.y);
  }
}

public CashBresenhamAlgorithm2D(ExecutionEnvironment env, Long
 ↪  minPoints, Integer recursionLimit, Integer
 ↪ initialBresenhamResolution) {
  this.minPoints = minPoints;
  this.recursionLimit = recursionLimit;
  this.initialBresenhamResolution = initialBresenhamResolution
 ↪ ;
  this.env = env;
}

public DataSet<Bounded2DArea> findAreasOfInterest(
    DataSet<Point2D> points,
    Boundary2D boundary,
    Double bresenhamResolution,
```

```
    Integer currentRecursion
) throws Exception {

 DataSet<BresenhamTransfomrations> bresenpoints = points.map
↪ ((Point2D point) -> {
   ParameterSpaceIntersectionFinder finder = new
↪ ParameterSpaceIntersectionFinder();
   List<Point2D> intersections = finder.list(point, boundary)
↪ ;

   if (intersections.size() < 2) {
     return new BresenhamTransfomrations(point, new ArrayList
↪ <>());
   }

   Bresenham b = new Bresenham();
   List<Point2D> bresenhams = b.compute2D(intersections.get
↪ (0), intersections.get(1), bresenhamResolution);

   return new BresenhamTransfomrations(point, bresenhams);
 }).filter((BresenhamTransfomrations transforms) ->
↪ transforms.bresenhams.size() > 0);

 if (bresenpoints.count() == 0) {
   // Hack to make types work
   return bresenpoints.map((BresenhamTransfomrations t) ->
↪ new Bounded2DArea());
 }

 DataSet<Bresenham2DArea> flattenedBresenPoints =
↪ bresenpoints.flatMap(new BresenFlatMapper());

 DataSet<Bresenham2DArea> areas = flattenedBresenPoints
     .groupBy(new Bounded2DAreaSelector())
     .reduce((Bresenham2DArea prev, Bresenham2DArea next) ->
↪ new Bresenham2DArea(prev.point, prev.intersections + next.
↪ intersections, new ArrayList<>()));

 Long mp = minPoints;
 DataSet<Bresenham2DArea> areasSatisfyingMinPoints = areas.
↪ filter((Bresenham2DArea b) -> b.intersections >= mp);
```

```
  return areasSatisfyingMinPoints.map((Bresenham2DArea b) ->
 ↪ new Bounded2DArea(
      new Boundary2D(b.point.x - 1, b.point.y - 1, b.point.x +
 ↪  1, b.point.y + 1),
      b.intersections,
      b.points
  ));
}

List<Bounded2DArea> recursiveDescentIteration(
    List<Bounded2DArea> areas,
    DataSet<Point2D> points,
    Integer currentRecursion,
    Boundary2D initialBoundary,
    boolean useInitialBoundary
) throws Exception {
  // Set up bresenhamResolution and recursions
  int nextRecursion = currentRecursion + 1;
  Double bresenhamResolution = initialBresenhamResolution + (
 ↪ double) nextRecursion;

  List<Bounded2DArea> newAreas = new ArrayList<>();

  for (Bounded2DArea area : areas) {
    Boundary2D boundary = initialBoundary;
    if (!useInitialBoundary) {
      boundary = area.boundary;
    }

    DataSet<Bounded2DArea> branchAreas = findAreasOfInterest(
        points,
        boundary,
        bresenhamResolution,
        currentRecursion
    );

    newAreas.addAll(branchAreas.collect());
  }

  newAreas = newAreas.stream()
```

```
        .filter(distinctByKey(Bounded2DArea::toKey))
        .collect(Collectors.toCollection(ArrayList::new));

    // If the recursion limit was hit, return the current area
    if (currentRecursion >= recursionLimit) {
      return newAreas; //results.collect();
    }

    // Run recursive descent on consecutive areas
    return runRecursiveDescent(
        //results.collect(),
        newAreas,
        points,
        nextRecursion
    );
  }

  private static class Bounded2DAreaDistinctSelector implements
   ↪ KeySelector<Bounded2DArea, String> {
    // private static final long serialVersionUID = 1L;
    @Override
    public String getKey(Bounded2DArea t) {
      return t.toKey();
    }
  }

  List<Bounded2DArea> runRecursiveDescent(
      List<Bounded2DArea> areas,
      DataSet<Point2D> points,
      Integer currentRecursion
  ) throws Exception {
    return recursiveDescentIteration(areas, points,
   ↪ currentRecursion, new Boundary2D(), false);
  }

  List<Bounded2DArea> runRecursiveDescent(
      List<Bounded2DArea> areas,
      DataSet<Point2D> points,
      Integer currentRecursion,
      Boundary2D initialBoundary
  ) throws Exception {
```

```
    return recursiveDescentIteration(areas, points,
    ↪ currentRecursion, initialBoundary, true);
  }
}

/**
 * The original CASH-MR algorithm
 */
class CashAlgorithm2D {
  /**
  * Defines the minimum points that an area needs to fulfill in
    ↪ order to be considered of interest.
  */
  private Long minPoints;

  /**
  * Defines how deep the recursive descent will step.
  */
  private Integer recursionLimit;

  CashAlgorithm2D(Long minPoints, Integer recursionLimit) {
    this.minPoints = minPoints;
    this.recursionLimit = recursionLimit;
  }

  DataSet<Bounded2DArea> findIndIntersectionsWithBoundary(
    ↪ DataSet<Point2D> points, Boundary2D ogboundary) {
    Double lx = ogboundary.lower.x, ly = ogboundary.lower.y;
    Double ux = ogboundary.upper.x, uy = ogboundary.upper.y;

    return points.map((point) -> {
      Boundary2D boundary = new Boundary2D(lx, ly, ux, uy);
      ParameterSpaceIntersectionFinder finder = new
    ↪ ParameterSpaceIntersectionFinder();
      List<Point2D> intersections = finder.list(point, boundary)
    ↪ ;
      List<Point2D> sources = Arrays.asList(point);
      Long count = 0L;

      if (!intersections.isEmpty()) {
        count = 1L;
```

```
    }

    return new Bounded2DArea(boundary, count, sources);
  });
}

Long countIntersections(DataSet<Bounded2DArea> areas) throws
 ↪ Exception {
  return areas.reduce((Bounded2DArea prev, Bounded2DArea next)
 ↪  -> {
    prev.intersections += next.intersections;
    return prev;
  }).collect().get(0).intersections;
}

DataSet<Bounded2DArea> combineIntersections(DataSet<
 ↪ Bounded2DArea> areas) throws Exception {
  return areas.reduce((prev, next) -> {
    List<Point2D> points = new ArrayList<>();
    points.addAll(prev.points);
    points.addAll(next.points);
    return new Bounded2DArea(
        next.boundary,
        prev.intersections + next.intersections,
        points
    );
  });
}

List<Boundary2D> splitRoom(Boundary2D boundary, Integer
 ↪ recursion) {
  if (recursion % 2 == 0) {
    Double mx = (boundary.upper.x - boundary.lower.x) / 2.0d;
    return Arrays.asList(
        new Boundary2D(boundary.lower.x, boundary.lower.y,
 ↪ boundary.upper.x - mx, boundary.upper.y),
        new Boundary2D(boundary.lower.x + mx, boundary.lower.y
 ↪ , boundary.upper.x, boundary.upper.y)
    );
  }
```

```
    Double my = (boundary.upper.y - boundary.lower.y) / 2.0d;
    return Arrays.asList(
        new Boundary2D(boundary.lower.x, boundary.lower.y,
    ↪ boundary.upper.x, boundary.upper.y - my),
        new Boundary2D(boundary.lower.x, boundary.lower.y + my,
    ↪ boundary.upper.x, boundary.upper.y)
    );
  }

  List<Bounded2DArea> runRecursiveDescent(
      DataSet<Point2D> points,
      Boundary2D boundaries,
      Integer currentRecursion
  ) throws Exception {
    DataSet<Bounded2DArea> intersections = this.
    ↪ findIndIntersectionsWithBoundary(points, boundaries);

    Long intersectionCount = this.countIntersections(
    ↪ intersections);
    if (intersectionCount < this.minPoints) {
      return new ArrayList<>();
    }

    if (currentRecursion >= recursionLimit) {
      return Arrays.asList(this.combineIntersections(
    ↪ intersections).collect().toArray(new Bounded2DArea[0]));
    }

    Integer nextRecursion = currentRecursion + 1;
    List<Boundary2D> rooms = this.splitRoom(boundaries,
    ↪ currentRecursion);
    List<Bounded2DArea> newAreas = new ArrayList<>();

    for (Boundary2D room : rooms) {
      newAreas.addAll(this.runRecursiveDescent(points, room,
    ↪ nextRecursion));
    }
    return newAreas;
  }
}
```

```
public class CashMR2D {
  private static void printOriginalUsage() {
    System.out.println("Usage: cashmr [original] [parallelism] [
   ↪ maxRecursion] [minPoints] [generatorAlgorithm] [
   ↪ generatorLower] [generatorUpper] [boundaryLowerX] [
   ↪ boundaryLowerY] [boundaryUpperX] [boundaryUpperY]");
    System.out.println("parallelism (integer): How many parallel
   ↪  tasks to run");
    System.out.println("algorithm (string): Can be ‘original‘, ‘
   ↪ bresenham‘, ‘random‘, ‘random_linear‘");
    System.out.println("maxRecursion (integer): Deepest allowed
   ↪ recursion");
    System.out.println("minPoints (integer): Minimum points
   ↪ required to be area of interest");
    System.out.println("generatorAlgorithm (string): Can be ‘
   ↪ linear‘, ‘linear_abs‘");
    System.out.println("generatorLower (integer): Start point
   ↪ for x value");
    System.out.println("generatorUpper (integer): End point for
   ↪ x value");
    System.out.println("boundaryLowerX (integer): Value for left
   ↪  boundary x");
    System.out.println("boundaryLowerY (integer): Value for left
   ↪  boundary y");
    System.out.println("boundaryUpperX (integer): Value for
   ↪ right boundary x");
    System.out.println("boundaryUpperY (integer): Value for
   ↪ right boundary y");
  }

  private static void printBresenhamUsage() {
    System.out.println("Usage: cashmr [bresenham] [parallelism]
   ↪ [maxRecursion] [minPoints] [generatorAlgorithm] [
   ↪ generatorLower] [generatorUpper] [boundaryLowerX] [
   ↪ boundaryLowerY] [boundaryUpperX] [boundaryUpperY] [
   ↪ initialBresenhamResolution]");
    System.out.println("parallelism (integer): How many parallel
   ↪  tasks to run");
    System.out.println("algorithm (string): Can be ‘original‘, ‘
   ↪ bresenham‘, ‘random‘, ‘random_linear‘");
    System.out.println("maxRecursion (integer): Deepest allowed
```

```
  ↪ recursion");
   System.out.println("minPoints (integer): Minimum points
  ↪ required to be area of interest");
   System.out.println("generatorAlgorithm (string): Can be '
  ↪ linear', 'linear_abs'");
   System.out.println("generatorLower (integer): Start point
  ↪ for x value");
   System.out.println("generatorUpper (integer): End point for
  ↪ x value");
   System.out.println("boundaryLowerX (integer): Value for left
  ↪  boundary x");
   System.out.println("boundaryLowerY (integer): Value for left
  ↪  boundary y");
   System.out.println("boundaryUpperX (integer): Value for
  ↪ right boundary x");
   System.out.println("boundaryUpperY (integer): Value for
  ↪ right boundary y");
   System.out.println("initialBresenhamResolution (integer):
  ↪ The initial bresenham resolution, must be >= 0");
}

public static void main(String[] args) throws Exception {
  // set up the execution environment
  final ExecutionEnvironment env = ExecutionEnvironment.
  ↪ getExecutionEnvironment();

  if (args.length == 0) {
    System.out.println("Usage: cashmr [original|bresenham]");
    return;
  }

  switch (args[0]) {
    case "original":
      if (args.length != 11) {
        CashMR2D.printOriginalUsage();
        return;
      }
      break;
    case "bresenham":
      if (args.length != 12) {
        CashMR2D.printBresenhamUsage();
```

```
        return;
      }
      break;
    default:
      System.out.println("Not supported, use ‘original‘ or ‘
↪ bresenham‘");
      return;
 }

 int parallelism = Integer.parseInt(args[1]);
 env.getConfig().disableSysoutLogging();
 env.setParallelism(parallelism);

 // Parse basic arguments
 int maxRecursion = Integer.parseInt(args[2]);
 long minPoints = Long.parseLong(args[3]);
 Boundary2D boundary = new Boundary2D(Double.parseDouble(args
↪ [7]), Double.parseDouble(args[8]), Double.parseDouble(args
↪ [9]), Double.parseDouble(args[10]));
 Random r = new Random(10l);
 // Populate the dataset with the given algorithm.
 DataSet<Point2D> data;
 long startdp = System.nanoTime();

 final List<Point2D> points = new ArrayList<>();

 switch (args[4]) {
   case "linear_abs":
     LongStream
         .rangeClosed(Long.parseLong(args[5]), Long.parseLong
↪ (args[6]))
         .boxed().forEach((in) -> points.add(new Point2D((
↪ double) in, (double) Math.abs(in))));
     data = env.fromCollection(points).setParallelism(
↪ parallelism);
     break;
   case "linear":
     LongStream
         .rangeClosed(Long.parseLong(args[5]), Long.parseLong
↪ (args[6]))
         .boxed().forEach((in) -> points.add(new Point2D((
```

```
↪ double) in, (double) in)));
    data = env.fromCollection(points).setParallelism(
↪ parallelism);
    break;
  case "random_linear":
    LongStream
        .rangeClosed(Long.parseLong(args[5]), Long.parseLong
↪ (args[6]))
        .boxed().forEach((in) -> points.add(new Point2D((
↪ double) in * r.nextFloat(), (double) in * r.nextFloat())))
↪ ;
    data = env.fromCollection(points).setParallelism(
↪ parallelism);
    break;
  case "random":
    LongStream
        .rangeClosed(Long.parseLong(args[5]), Long.parseLong
↪ (args[6]))
        .boxed().forEach((in) -> points.add(new Point2D((
↪ double) 20 * r.nextFloat(), (double) 20 * r.nextFloat())))
↪ ;
    data = env.fromCollection(points).setParallelism(
↪ parallelism);
    break;
  default:
    System.out.println("Unknown algorithm: " + args[4]);
    return;
}
long enddp = System.nanoTime();

// Declare outputs
List<Bounded2DArea> result;
long start, end;

switch (args[0]) {
  case "original":
    // Runs the original CASH-MR algorithm
    CashAlgorithm2D a = new CashAlgorithm2D(minPoints,
↪ maxRecursion);

    start = System.nanoTime();
```

```
      result = a.runRecursiveDescent(data, boundary, 0);
      end = System.nanoTime();
      break;
    case "bresenham":
      // Runs the bresenham adaption of CASH-MR
      Integer initialBresenhamResolution = Integer.parseInt(
↪ args[11]);
      CashBresenhamAlgorithm2D ab = new
↪ CashBresenhamAlgorithm2D(env, minPoints, maxRecursion,
↪ initialBresenhamResolution);

      start = System.nanoTime();
      result = ab.runRecursiveDescent(Arrays.asList(new
↪ Bounded2DArea(boundary, 0l, new ArrayList<>())), data, 0,
↪ boundary);
      end = System.nanoTime();
      break;
    default:
      System.out.println("Not supported, use `original` or `
↪ bresenham`");
      return;
  }

  for (Bounded2DArea area : result) {
    // A small hack to properly compute resulting grid sizes
↪ in bresenham
    if (args[0].equals("bresenham")) {
      Double scale = 1.0 / (maxRecursion + 1 + Integer.
↪ parseInt(args[11]));
      area.boundary.lower.x = area.boundary.lower.x + 1.;
      area.boundary.lower.y = area.boundary.lower.y + 1.;
      area.boundary.upper.x = area.boundary.lower.x + scale;
      area.boundary.upper.y = area.boundary.lower.y + scale;
    }

    System.out.println(String.format("Found area of interest
↪ with targetDelta=\%.4f: [\%s, \%d, \%d]", area.boundary.
↪ upper.x - area.boundary.lower.x, area.boundary, area.
↪ intersections, area.points.size()));
  }
```

```
    System.out.println(String.format("Found \%d areas of
   ↪ interest.", result.size()));
    System.out.println("Took " + ((enddp - startdp) / 1000000d /
   ↪  1000d) + "s to generate " + data.count() + " datapoints")
   ↪ ;
    System.out.println("Took " + ((end - start) / 1000000d /
   ↪ 1000d) + "s for " + args[0] + " to compute " + data.count
   ↪ () + " datapoints with minPoints=" + minPoints + " and
   ↪ maxRecursion=" + maxRecursion + " and boundary=" +
   ↪ boundary + " and parallelism=" + env.getParallelism());
  }
}
```

List of Figures

List of Tables

Bibliography

[ABD+08] Elke Achtert, Christian Böhm, Jörn David, Peer Kröger, and Arthur Zimek. *Global Correlation Clustering Based on the Hough Transform*. Statistical Analysis and Data Mining, 1:111–127, 2008.

[AX10] H. Abdi and L. J. Xu. *Principal Component Analysis*. In Wiley Interdisciplinary Reviews: Computational Statistics, 2010.

[AY00] Charu C. Aggarwal and Philip S. Yu. *Finding generalized projected clusters in high dimensional spaces*. ACM SIGMOD Record, 29(2):70–81, 2000.

[Bec05] Hila Becker. *A Survey of Correlation Clustering*, 2005.

[BKKZ04] Christian Böhm, Karin Kailing, Peer Kröger, and Arthur Zimek. *Computing Clusters of Correlation Connected objects*. Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04, page 455, 2004.

[Bre65] J. E. Bresenham. *Algorithm for computer control of a digital plotter*. IBM Systems Yournal, 4:25–30, 1965.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. *Apache FlinkâĎć: Stream and Batch Processing in a Single Engine*. IEEE Data Eng. Bull., 38:28–38, 2015.

[DEFI06] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. *Correlation Clustering in General Weighted Graphs*. Theor. Comput. Sci., 361(2):172–187, September 2006.

[DG08] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51:107–113, 2008.

[DH72] Richard O. Duda and Peter E. Hart. *Use of the Hough transform to detect lines and curves in pictures*. Commun. ACM, 15:11–15, 1972.

[EKSX96] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*. In Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96), Portland, OR, pages 226–231, 1996.

[Fou18] Apache Foundation. *Apache Flink 1.3 Documentation*, 2018.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. 1988.

[Joy99] I. K. Joy. *Bresenham's Algorithm*. 1999.

[KMKS17] Daniyal Kazempour, Markus Mauder, Peer Kröger, and Thomas Seidl. *Detecting Global Hyperparaboloid Correlated Clusters Based on Hough Transform*. In Proceedings of the 29th International Conference on Scientific and Statistical Database Management, SSDBM '17, pages 31:1–31:6, New York, NY, USA, 2017. ACM.

[SRB+18] Matthias Schubert, Matthias Renz, Felix Borutta, Evgeniy Faerman, Christian Frey, Klaus Schmid, Daniyal Kazempour, and Julian Busch. *Batch Processing Systems*, 2016-2018.

[Zim08] Arthur Zimek. *Dissertation: Correlation Clustering*. LMU München: Fakultät für Mathematik Informatik und Statistik, 2008.