Parallelizing Incremental Grid Density Clustering Algorithm for Real-time Network Anomaly Detection

Jagatheesan Kunasaikaran, Roslan Ismail, Abdul Rahim Ahmad

Abstract— Cyberattacks are a growing threat to the internet. These cyberattacks are targeted to exploit the vulnerabilities in a network. Often, the intention of such cyberattacks is malicious. The growing number of devices that are connected to the Internet also increases the vulnerability of such devices to cyberattacks. Cyberattacks will usually exhibit anomalous characteristics in a network traffic flow. These anomalies could be identified in real-time and investigated further to minimize the impact of cyberattacks. Many researchers today use a wide array of tools to detect cyberattacks in real-time which comes with a high complexity cost in terms of the system architecture and maintenance. In this research, a simple method of parallelizing the incremental grid density clustering algorithm used to detect the anomalies is implemented and studied. The simplified approach used in this research is practical to be used in a real-life environment.

Index Terms— clustering, cybersecurity, machine learning, network anomaly detection, real-time

1 Introduction

IN today's world, devices are being connected to large networks at an accelerating pace. By 2021, it is expected that 28 billion devices will be connected to the network, predominantly, the Internet [1].

A report produces by Symantec Security says that malware is being prompted in approximately 8 percent of gadgets connected to the Internet [2]. Also, a huge jump of 54 percent in the variety of malware targeting mobile gadgets with an almost equal increase of 46 percent in the type of ransomware are being widely seen today. The explosive growth of devices connected to the Internet in developing regions such as Africa and Asia where the growth of the number of such devices has been recorded as 9941 percent, and 1670 percent from 2000 to 2018 respectively [3] shows the huge potential for vulnerability exposed to malicious cybercriminals.

Hence, implementing a system capable of detecting network intrusion in real-time which is easily deployable and scalable is an important challenge to be solved as non-real time network intrusion detection reduces the possibility of taking security measures to defend a system promptly and has a lower chance of detecting novel attacks. In this research, upon studying, a crucial part of the network intrusion detection system which is the network anomaly detection module is enhanced. The enhancement is focused on improving the performance of the clustering algorithm used to detect anomalies.

This paper is divided into six main sections, a discussion on related works, a brief overview of clustering algorithms, a brief discussion on the Go programming language focusing on it's concurrency features, the implementation of the experiment, a discussion on the results obtained and finally, conclusions and future direction to continue to work done in this research.

2 Related Work

The authors of [4] have proposed to use Apache Hadoop [5], Hive [6] and Apache Spark [7] to process the network flow data in real-time. Apache Hadoop is used to store the network packets in a Hive table that is accessible by Apache Spark for real-time processing of the features. The implementation of the researchers can process a large amount of network flow data to detect anomalies in a short span of time as the computation was distributed among a cluster of Spark processes. On the flip side, fine-tuning configuration settings for Hadoop is highly challenging [8]. Apache Spark also comes with own set of challenges when especially on the ease of debugging [9].

In [10], the authors have used a combination of the CluStream [11] clustering algorithm and a decision tree to detect anomalies. CluStream is used to cluster the training data into anomaly or non-anomaly. Then, the clusters formed are used to create a decision tree. The features of the data that streams into the system is reduced by using an online feature selection method which in turns reduces the time to detect the anomalies. The system also has a high precision rate. However, the data set used, KDD99, is known for its limitation regarding a high number of redundant and different level of difficulties in the distribution of the dataset [12]. Besides, the authors have not tested the system using the whole KDD99 dataset.

Besides that, in [13], the authors has proposed a multi-stage anomaly detection system. The system is divided into two engines. Whenever the first engine detects a DDOS attack, the second engine proceeds to find the potential Bot-Master. Dividing the processing of detection using in a multi-stage process decreases the computation resource needed. The ensuing work [14] from this research paper though did not state the computational performance of the system in detail.

- *Jagatheesan Kunasaikaran is currently pursuing masters degree program in information technology in Universiti Tenaga Nasional. E-mail: st22489@utn.edu.my*
- *Roslan Ismail is an associate professor in Universiti Tenaga Nasional. His area of interest is in software testing and cybersecurity. E-mail: Roslan@uniten.edu.my*
- *Abdul Rahim Ahmad is an assocate professor in Universiti Tenaga Nasional. His area of interest is in networking and machine learning. Email: Abdrahim@uniten.edu.my*

A two-stage NIDS whereby the first stage is to form a rule set using unsupervised learning methods and the second stage of classifying network traffic in an online manner using the rule set forms is done in [15]. The first stage of form rules uses k-means clustering. The proposed method has its advantage as it eliminates the need to have a labeled training dataset. The downside of this method is that the rule set needs to be re-formed to detect novel attacks.

Focusing on the perspective of performance of the proposed system, the approach suggested in [16] to identify anomalies by clustering over sub-spaces using an incremental grid clustering algorithm is well suited to a real-time setting as the algorithm can include streaming network traffic to update the clusters. The use of sub-spaces also further decrements the size of the feature space that needs to be clustered. However, clustering over the sub-spaces is done in a sequential manner whereby a loop is running to cluster each of the sub-spaces. The result of clustering all the sub-spaces is then accumulated to identify anomalies. To improve the performance of the clustering, the process of clustering over all the sub-spaces can be done concurrently. This is because the sequential clustering of the sub-spaces will not be able to utilize the multiprocessor architecture found commonly in many hardware setups. Concurrency can be achieved by processing the data as streams using the concept proposed in [4]. The downside of the method is the multiple big data technologies that need to be integrated to achieve the concurrency. Notably, the authors have proposed Apache Hadoop, Apache Spark and Hive and the core big data tools to be used to process the streams. These tools have its advantages in being very scalable. However, this technological stack could quickly become unwieldy to maintain as more features are added into the system. One possible way to mitigate this is to use a cloud-hosted variation of this tech stack which is provided by cloud hosting provider which also comes with a need to have a deep understanding on the cloud-hosted variant of these tools. A point must be taken note that spinning a cluster of Apache Hadoop, Apache Spark and Hive will also incur an additional cost which the system administrator needs to manage carefully.

The authors of [10] have proposed to use semi-supervised network anomaly detection system which can increase the performance of the system. The increase in performance is given by the fact that a decision tree is constructed beforehand by clustering the network traffic data. Then, anomalies in incoming network traffic is identified by evaluating the network traffic with the decision tree constructed. A similar approach has also been proposed in [15] whereby a rule engine is trained, and classification of network traffic as anomalous or otherwise is done using the rule engine. Both these methods increase the complexity of the system because it involved two different data mining approaches to enable the system to detect anomalies reliably. The need to devise a model to identify anomalies also raises the question whether these methods can accommodate to novel attacks quickly. The data set used in the researches are also not the standard data set that has real-time network traffic properties encapsulated such as MAWILab data set. In [10], only a subset of KDD99 dataset was used in the evaluation and in [15], the data set used is not identified clearly and only stated vaguely as a real-life web server data. Hence, it yields the question whether the results will be reproducible if a larger and a widely available data set is used. Similar impediments is found in validating the result obtained in [13] and [14] where the performance results were not clearly reported.

Regarding accuracy, the authors of [16] did not state specifically which type of anomalies was being targeted in their accuracy evaluation which yields the ROC curve reported in the related work. Instead, a high-level description of the aggregation level and the variable threshold used to generate the ROC curve is described.

3 Clustering Algorithms

Among different types of machine learning algorithms available, clustering algorithms are a promising solution to solve the challenge of real-time network anomaly detection. Clustering algorithms that use the concept of a grid, density or a combination of both have the characteristics of anomalies embedded in the algorithm. This characteristic offers a key advantage of using these algorithms to detect network anomalies instead of other available algorithms.

Grid-based clustering algorithms partitions objects in the data space into different grids. The computational complexity is reduced through this method as the clustering algorithm, later, does not need to cluster the individual objects. Instead, the clustering algorithm uses the characteristics of the grid to do the clustering. The widely used algorithm that belongs to the grid-based clustering algorithm is STING (A Statistical Information Grid Approach to Spatial Data Mining) [17] and CLIQUE (Clustering In QUEst) [18]. STING introduces a method to execute clustering without the need to traverse through individual objects by using a hierarchical statistical grid which in turn yields a reduces computational cost and a higher chance of able to parallelize. CLIQUE solves the problem associated with clustering called the curse of dimensionality by automatically looking for clusters in the subspaces derived from the original data space.

Density-based algorithms focus on grouping the objects in a data space into connected dense regions which are made up objects placed close together. A cluster is formed when these dense regions are found. DBSCAN [19] is a famous algorithm used in this class of algorithm. Two key ideas used in DBSCAN is density reachability and density connectivity of the algorithm. The algorithm can detect clusters in any shapes and find noises identified through the clustering process. These noises work robustly in pointing to the anomalies in the data space. The number of clusters does not need to be specified for this algorithm which makes it very flexible for many use cases.

Grid density algorithms such as the one used in [11] use both the concepts of the grid and density clustering to detect anomalies in the data fed into the algorithm. The usage of grids and basing the clustering on the density of the points reduces the data points that need to be processed which in turn increases the performance of the algorithm.

4. Go Programming Language

Go programming language, commonly referred as Golang, is a programming language created at Google by Robert Griesemer, Rob Pike and Ken Thompson [20]. It was released to the public on November 10, 2009, as an open source programming language that is statically typed compiled. Notable features of Go are memory safety, garbage collection, structural typing and concurrent programming is done using Hoare's Communication Sequential Processes (CSP) [21] style.

Often, concurrency in a multi-threaded environment has high complexity due to the need of synchronizing access to the memory space. This is achieved by a complex amalgamation of concepts such as mutexes, semaphores and condition locks. CSP circumvents all these by providing a model for high-level linguistic capabilities in implementing concurrency support.

Goroutines are central to the concept of implementing CSP inspired concurrency in Go. Go runtime intelligently manages the independently executing function, coroutines, across a set of threads. For example, if a coroutine is blocked by another coroutine executing a long running process such as input/output (I/O) communications, the Go runtime moves this coroutine to a different runnable thread. This concept of a managed group of coroutines is known as goroutines or in its singular form as goroutine.

Goroutines are cheap because each goroutine that is created is given a few kilobytes in a bounded stack that is resizable. The Go run-time manages the size of the stack as per the need of the goroutines that are being executed while keeping the CPU resource needed for this shrinking and expansion of the stack as minimal as possible. As a direct result of this efficient use of memory space and CPU resource, it is feasible to create hundreds of thousands of goroutines in the same address space. If the goroutines were created as normal operating system thread, the number of goroutines that could be created would be far smaller.

Listing 0.1 shows the syntax to start a goroutine. It is a simple as calling a function or method that needs to be run concurrently with the go keyword. The Go runtime will manage the complexities of creating and managing the threads to run the goroutine.

```
1. go functionToExecute()
```

Listing 0.1. Syntax to start a goroutine

Go runtime allows multiple goroutines to access the same memory space by providing type conduits to ensure only one goroutine has access to a memory space at a given time. These type conduits are known as channels. Channels can be used to send and receive values.

Listing 0.2 shows the syntax used for communicating over channels in the Go programming language. The direction of the arrow indicates the data flow direction.

```
1. ch <- v
2. v := <- ch
```

Listing 0.2. Syntax for channels

Line 1 shows that value *v* is sent to a channel, *ch*. Line 2 shows value is received from channel *ch* and this value is assigned to *v*. The design of channels ensures that a sender is blocked until a receiver is available to receive the value and vice-versa. This ensures that data can be shared across goroutines without the need for any locking mechanisms or condition variables. However, should the programmer decide that a sending channel should not be blocked for a limited number of values, then the programmer can use buffered channels. Buffered channels do not block senders until the buffer is full. On the other hand, receive operations on the buffered channel is blocked when the buffer is empty. The simplicity offered by these syntaxes is evident in the few numbers of lines needed to write a reliable concurrent implementation. Go runtime handles the rest of the execution details to ensure the concurrent execution is reliable.

5. Experiment Implementation

In [16], the authors have used the incremental grid density clustering algorithm (IGDCA) algorithm introduced in [22] in network anomaly detection domain.

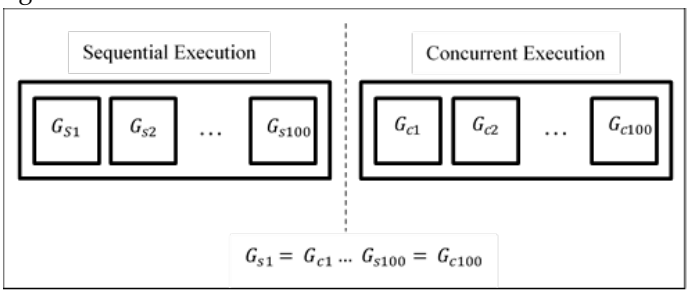
Algorithm 1: Subspace clustering in [16]

for i = 1: Number of subspaces do
Execute IGDCA on subspace i
end for

Algorithm 1 shows the subspaces clustering in [16]. In this research, the algorithm is tested in four different architectural methods, sequential execution as per the referred work, concurrent execution with a thread created for every subspace clustered, a worker pool communicating on a shared messaging channel and a worker pool communicating on a unique messaging channel for every thread.

The performance of the IGDCA is evaluated by executing the algorithm sequentially and in three concurrent manners. To avoid discrepancies in the result, two parameters which are the minimum number of points to consider a unit as dense and the minimum number of points to consider a group of units as a cluster is kept constant with the value of 10 and 100 respectively.

Fig. 1**Error! Reference source not found.** illustrates the method



of construction of the subspaces for IGDCA evaluation. 100 grids which equate to 100 subspaces were initialized and, in each grid, there are 100 units with the sides being the length of 0.1. The grids used for each method's evaluation is kept constant by initializing the same random seed to generates the number of points to be clustered.

Fig. 1. Construction of subspaces for IGDCA evaluation

With the same seed used for each method, each grid will have the same distribution of points. To simulate the data space that will be encountered in real-life network traffic, the distributions among the units inside a grid is varied.

The four methods that were tested are sequential execution, concurrent execution types which are made up of concurrent execution with new goroutine for each grid in an iteration, worker pool with a shared channel for results and a worker pool with a different channel for results for each worker. In sequential execution, the IDGCA algorithm was run on a single CPU. In the rest of the three methods, the IDGCA algorithm was run concurrently across the available CPUs.

The experiments for this research are conducted with a system equipped with Windows 7 operating system, 3.40 GHz Intel ® Core ™ i7-3770 CPU processor and 8.00 GB RAM. The system is entirely written in Go programming language.

6 Result

Each implementation is evaluated in two different ways, by varying the number of points in a subspace and varying the number of subspaces.

6.1 Varying number of points

Fig. 2 shows that among the four methods test, the sequential execution exhibits the fastest execution time. The worker pool that communicates on a shared and unique channel follow closely. The concurrent execution which creates a new goroutine for each subspace clustering is the slowest. A trend that can be seen is that as the number of points increases, the sequential execution tends to reduce in performance. In comparison to that, the worker with shared and unique channels exhibit an almost consistent execution time. When the number of points used to test ranged from 1000 to 10000 points, the worker pool with unique channels for each subspace also shows a dip below the 0.025ms line. On the other hand, the worker pool with a shared channel shows a graduals increase in execution time as the number of points increases.

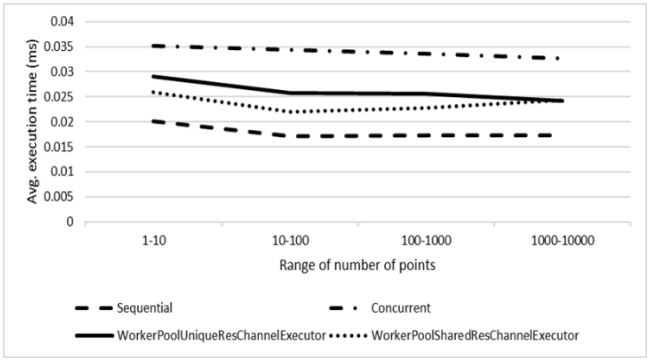


Fig. 2. Average execution time of four methods

The sequential execution is the fastest executing method in this test because it does not need to create goroutines and channels to communicate concurrently. However, as a direct consequence of a lack of concurrency, the sequential execution slows down as the number of points increases. It does not use the available CPUs to the best of its ability. The worker pool with shared and unique channels, on the other hand, doesn't degrade much in performance when the number of points of increases because these methods use all the available CPUs. Lastly, the concurrent execution with new goroutines created for each grid is the slowest because there is a performance hit when a goroutine needs to be created for every subspace, and it also needs to be garbage collected by the runtime later.

6.2 Varying number of subspaces

Fig. 3 shows that the average execution time of the worker pool that communicates its results on a shared channel being the lowest among the four methods. The average execution time of the rest of the three methods was almost the same until 64 subspaces. Post 64 subspaces, the sequential execution saw an exponential increase in the average execution time. The concurrent execution with new goroutines created for each subspace also showed a similar increase in the average execution time. The fastest two methods were the worker pool that communicates its result on a shared channel and unique channels. The average execution times were just slightly increased above 2ms after 1024 subspaces.

The ability of the worker pool methods to distribute computation across multiple processors allowed these methods to be the fastest methods among the methods tested. The use of a fixed number of goroutines in these methods also removed the overhead of creating new goroutines which in turn reduced the computational cost which hit the concurrent execution where new goroutines were created for each subspace clustered. After 64 subspaces, sequential execution slows down dramatically as all its computation is ran on a single CPU.

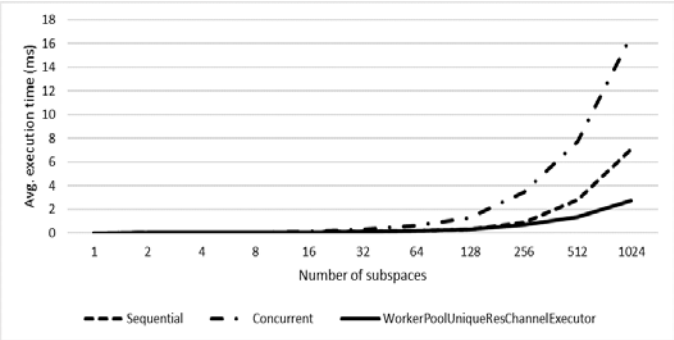


Fig. 3. Average execution time of four methods

6.3 Result Analysis

The sequential execution used in this research is referred from the algorithm implemented in [16]. Go programming language's profiling tool was used to find out the value of the p parameter in the Gustafson-Barsis' Law. The value of p which equals to the percentage of code that is improved to run in parallel is computed as ~40% using the Go profiling tool.

Fig. 4 shows that the sequential execution is the best execution method when two criteria are present. Firstly, the number of subspaces to be clustered is small and secondly, the number of points in these subspaces is large. The parallel implementations are slowed due to the performance cost imposed by the computation overhead in creating Goroutines. For example, due to this performance hit, the concurrent implementation showed a decrease in performance of almost 0.2 factor when compared to the sequential execution. The other two implementation's performance hovered in between the sequential and concurrent execution.

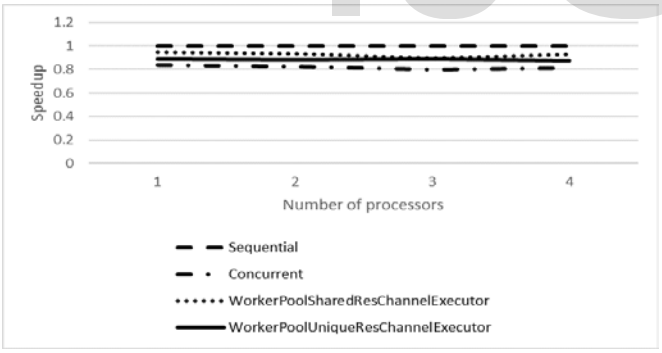


Fig. 4. Evaluating system performance under total point in a subspace being between 1000 to 10,000 using Gustafson-Barsis' Law

In contrast, Fig. 5 shows that when the number of subspaces increases, the worker pool methods shows a marked performance increase compared to the other two methods. There is an improvement of 2 times when compared to the base case of sequential execution as referenced from [16]. The result proves that the parallel executions gives a solution to the challenge of finding network anomalies as the number of subspaces in the data space increases which is a common occurrence in network traffic data today.

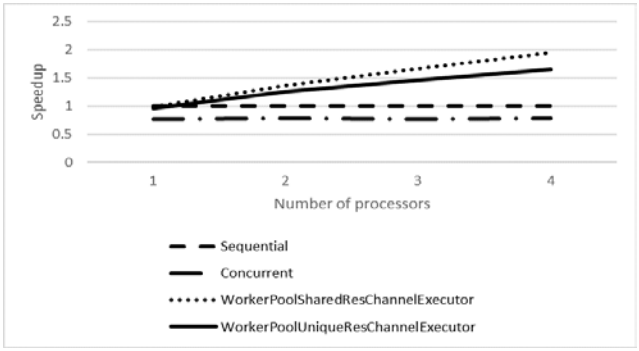


Fig. 5. Evaluating system performance under 1024 subspaces using Gustafson-Barsis' Law

7. Conclusion

In this research work, the IGDCA algorithm used in [16] which exhibits useful characteristics such as the ability to incrementally update the data space and reducing the data space to be clustered by using grid and density concepts is further improved in performance with the introduction of parallelization. The usage of parallelized architecture when there is an increase in the number of points in the data space does not yield significant improvements over a sequential architecture. However, when the number of subspaces to be clustered increases, parallelized architecture shows a marked improvement in performance when compared to the sequential execution. This improvement has great potential to be enhanced and used in network anomaly detection as this could lead to a robust real-time network anomaly detection system. In future work, the impact of the improvements to the accuracy of the system needs to be studied further.

7 References

[1] A. Nordrum, "Popular internet of things forecast of 50 billion devices by 2020 is outdated," *IEEE Spectr.*, vol. 18, 2016.
[2] Symantec, "ISTR Internet Security Threat Report Volume 23," 2018.
[3] I. W. Stats, "World Internet Users and 2018 Population Stats," 2018. .
[4] K. Kato and V. Klyuev, "Development of a network intrusion detection system using Apache Hadoop and Spark," in *2017 IEEE Conference on Dependable and Secure Computing*, 2017, pp. 416–423.
[5] A. Hadoop, "Hadoop," *http://hadoop.apache.org*, 2018. .
[6] A. Hive, "Apache Hive," *https://hive.apache.org/*, 2018. .
[7] A. Spark, "Apache Spark: Lightning-fast cluster computing," *http://spark.apache.org*. .
[8] S. B. Joshi, "Apache hadoop performance-tuning methodologies and best practices," *Proc. third Jt. WOSP/SIPEW Int. Conf. Perform. Eng. - ICPE '12*, p. 241, 2012.
[9] M. Armbrust, M. Zaharia, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, and R. Xin, "Scaling spark in the real world," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1840–1843, 2015.
[10] Z. Cataltepe, U. Ekmekci, T. Cataltepe, and I. Kelebek, "Online feature selected semi-supervised decision trees for network intrusion detection," in *Proceedings of the NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, 2016, pp. 1085–1088.
[11] C. C. Aggarwal, T. J. Watson, R. Ctr, J. Han, J. Wang, and P. S. Yu, "A Framework for Clustering Evolving Data Streams," *Proc. 29th int. conf.*

Very large data bases, pp. 81–92, 2003.

[12] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, 2009, pp. 1–6.

[13] P. V. Amoli and T. Hämäläinen, "A real time unsupervised NIDS for detecting unknown and encrypted network attacks in high speed network," in *Proceedings - M and N 2013: 2013 IEEE International Workshop on Measurements and Networking*, 2013, pp. 149–154.

[14] P. Vahdani Amoli, "Unsupervised network intrusion detection systems for zero-day fast-spreading network attacks and botnets," *Jyväskylä Stud. Comput.*, vol. 10, no. 231, pp. 1–13, 2015.

[15] A. Juvonen and T. Sipola, "Combining conjunctive rule extraction with diffusion maps for network intrusion detection," in *Proceedings - International Symposium on Computers and Communications*, 2013, pp. 411–416.

[16] J. Dromard, G. Roudière, and P. Owezarski, "Online and Scalable Unsupervised Network Anomaly Detection Method," *IEEE Trans. Netw. Serv. Manag.*, vol. 14, no. 1, pp. 34–47, 2017.

[17] W. Wang, J. Yang, and R. Muntz, "STING : A Statistical Information Grid Approach to Spatial Data Mining," *23rd VLDB Conf. Athens*, pp. 186–195, 1997.

[18] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 94–105, 1998.

[19] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," *Proc. 2nd Int. Conf. Knowl. Discov. Data Min.*, pp. 226–231, 1996.

[20] G. Team, "Frequently Asked Questions (FAQ) - The Go Programming Language," 2009. [Online]. Available: https://golang.org/doc/faq. [Accessed: 02-Jun-2018].

[21] C. A. R. Hoare, "Communicating Sequential Processes."

[22] Ã. C. Ning, C. An, and Z. Long-Xiang, "An Incremental Grid Density-Based Clustering Algorithm," vol. 1313, no. 101, pp. 1–7, 2002.