# Implementation of PSCAN Algorithm for Big Graph Networks using PySpark

*CSC 502 PROJECT REPORT*

**Lepeng Zhou** *(V01045967) lepeng@uvic.ca*

## ABSTRACT

This project is about understanding and implementing the Parallel Structural Clustering Algorithm for Networks (PSCAN) [1] with PySpark. PySpark can manage big data well with the MapReduce model and the Resilient Distributed Dataset. Networks, like those on social media, are getting bigger and more complex. It's important to understand and spot groups within these networks. I have made the PSCAN algorithm work on PySpark. This uses many computers (or many CPU cores if there is only a single machine) to deal with big data better. This project is important because it could make finding groups more precise and faster. This helps us know more about large networks.

## INTRODUCTION

In the era of big data, networks that represent social interactions, biological systems, and information flows have expanded in both size and complexity. This growth calls for efficient tools for analysis and understanding. Clustering networks, a method that groups the vertices of a network into clusters, is essential for revealing community structures. These structures are characterized by groups of vertices that are more densely connected to each other than to the rest of the network. This report introduces the implementation of the PSCAN algorithm using PySpark, with the aim of utilizing distributed computing to achieve scalable clustering of networks. The selection of PySpark as the platform for this task is driven by its scalability, flexibility, and fault tolerance capabilities in managing structured and unstructured data within a distributed computing environment.

## BACKGROUND

The core of our implementation is the PSCAN algorithm, which extends the SCAN algorithm by introducing parallelism in the computation of structural similarities across edges and efficiently identifying clusters in large networks. Unlike conventional clustering algorithms that may struggle with the scale and complexity of modern datasets, PSCAN's design allows for scalable and accurate identification of not just clusters but also hubs and outliers within the network. PySpark, a unified analytics engine for large-scale data processing, provides the necessary infrastructure to implement PSCAN over distributed systems. By utilizing PySpark's resilient distributed datasets (RDDs) and its comprehensive API, we can efficiently parallelize the computation required by PSCAN, thus handling large networks more effectively.

## METHODOLOGY

The methodology section talks about the specifics of implementing PSCAN within the PySpark framework. The process begins with data preparation, where networks are represented as graphs with vertices and edges, followed by the main phases of the PSCAN algorithm.

### Toy Dataset

For this project, I developed a toy dataset to test and validate the PSCAN algorithm's implementation using PySpark. This dataset is designed to be both realistic and controlled in terms of network properties. It includes a randomly generated and manually cleaned network dataset. This network has 100 nodes and 349 edges, showing the complexity and details you would find in real-world networks but on a smaller scale. This makes it easier to analyze in detail and debug the clustering algorithm.

| UserID | Friends |
|---|---|
| 1 | 72,18,7 |
| 2 | 3,7,9,13,16,83,20 |
| 3 | 2,4,5,38,7,6,11,29,19,59,61 |
| 4 | 16,19,3,12 |
| 5 | 3,8,10,11,13,17,19,20,86 |
| 6 | 3,7,14,15,16,20,23 |
| 7 | 1,2,3,6,9,10,46,15,16,18,53 |
| 8 | 5,9,10,79,19 |
| 9 | 2,37,7,8,11,14,17,18,20 |
| 10 | 5,7,8,11,14,84,20,57 |
| 11 | 3,68,5,9,10,75,12,14,15,49,19,53 |
| 12 | 19,11,4,14 |
| 13 | 97,2,5,75,15,17,19,55 |
| 14 | 6,9,10,11,12,15,20,86,31 |
| 15 | 6,7,11,13,14,16 |

I started by generating the dataset through a process that tries to copy the structure of social networks. This includes clusters (communities), hubs (nodes with many connections), and outliers (nodes with few connections). To do this, the graph generation process used methods that make some nodes more likely to connect to others, showing how in real networks, popular nodes get even more connections. It also added random connections to make sure there were outliers.

### PSCAN Algorithm Overview

The PSCAN algorithm operates in 2 primary phases: calculation of structural similarities then pruning of edges based on these similarities (PCSS), and Label Propagation for Connected Components (LPCC).

1. #### Parallel Calculation of Structural Similarity and Pruning (PCSS)

   Structural similarity between two nodes is calculated based on their shared neighbors, employing the Jaccard similarity coefficient as the metric. This step is parallelized using PySpark's map-reduce capabilities:

#### Mapper:

```
[412] kv_rdd2 = sc.parallelize(kv_rdd)
```

```
[413] def pcss_mapper(edge):
          node, neighbors = edge
          for neighbor in neighbors:
              yield ((min(node, neighbor), max(node, neighbor)), neighbors)

      pcss_mapped = kv_rdd2.flatMap(pcss_mapper)
```

```
[414] pcss_mapped.take(5)
```

```
[((1, 72), [72, 18, 7]),
 ((1, 18), [72, 18, 7]),
 ((1, 7), [72, 18, 7]),
 ((2, 3), [3, 7, 9, 13, 16, 83, 20]),
 ((2, 7), [3, 7, 9, 13, 16, 83, 20])]
```

The mapper takes each node and then emits a pair for every neighbor of that node. The replication rate is the average number of neighbors each node has.

## Reducer:

```
def calculate_structural_similarity(adjacency_list1, adjacency_list2):
    set1 = set(adjacency_list1)
    set2 = set(adjacency_list2)

    intersection = set1.intersection(set2)
    union = set1.union(set2)

    if not union:
        return 0  # Avoid division by zero
    jaccard_similarity = len(intersection) / len(union)

    return jaccard_similarity
```

```
def pcss_reducer(adjacency_lists):
    global threshhold
    adjacency_lists = list(adjacency_lists)
    if len(adjacency_lists) == 2:
        adjacency_list1, adjacency_list2 = adjacency_lists
        structural_similarity = calculate_structural_similarity(adjacency_list1, adjacency_list2)
    else:
        structural_similarity = 0
    if structural_similarity > threshhold:
      return structural_similarity
    else:
      return None

pcss_reduced = pcss_mapped.groupByKey().mapValues(pcss_reducer).filter(lambda x: x[1] is not None)
```

```
[417] pcss_reduced.take(10)
```

```
[((3, 5), 0.1111111111111111),
 ((3, 19), 0.2352941176470588),
 ((3, 59), 0.1333333333333333),
 ((4, 12), 0.1428571428571428),
 ((5, 11), 0.16666666666666666),
```

For each edge, the reducer takes each node on 2 ends of that edge then computes their Jaccard similarity. However this similarity can be many other suitable functions. But Jaccard is the best and simple to demonstrate here. As J = set difference / set union, and J is between 0 and 1. Then the reducer will remove edges that have J less than a preset threshold.

Convert network from edge format to (node, neighbors) format:

```
[418]
    pruned_neighbors_rdd = (pcss_reduced
                            .flatMap(lambda x: [(x[0][0], x[0][1]), (x[0][1], x[0][0])])
                            .distinct()
                            .groupByKey()
                            .mapValues(list))

    updated_kv_rdd = kv_rdd2.leftOuterJoin(pruned_neighbors_rdd)

    updated_kv_rdd = updated_kv_rdd.mapValues(lambda x: x[1] if x[1] is not None else [])
    sorted_kv_rdd = updated_kv_rdd.sortByKey()
```

```
sorted_kv_rdd.take(5)
```

```
[(1, [18]),
 (2, [7, 9]),
 (3, [5, 19, 59]),
 (4, [12, 19]),
 (5, [3, 11, 13, 19, 8, 10])]
```

This is necessary as we need to follow the format required for the following algorithms.

2. Label Propagation for Connected Components (LPCC)

   The final step involves identifying connected components within the pruned network. Each connected component represents a cluster or community within the network. To parallelize the identifying process, we can use a smart distributed label propagation algorithm, iterate it until all clusters are identified.

Mapper:

```
# initialize each node with its own ID as the label and mark it as status
lpcc_rdd = sorted_kv_rdd.map(lambda x: (x[0], {'label': x[0], 'neighbors': x[1], 'status': True}))
```

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()

def lpcc_mapper(node):
    yield (node[0], (node[1], 'state'))

    if node[1]['status']:
        for neighbor in node[1]['neighbors']:
            yield (neighbor, (node[1]['label'], 'label'))
```

For every node that has not converged yet, the mapper emits its structure information and neighbor's label.

## Reducer:

```
def lpcc_reducer(values):
    current_state = None
    min_label = float('inf')

    for value, valueType in values:
        if valueType == 'state':
            current_state = value
        elif valueType == 'label':
            min_label = min(min_label, value)

    if min_label < current_state['label']:
        return {'label': min_label, 'neighbors': current_state['neighbors'], 'status': True}
    else:
        return {'label': current_state['label'], 'neighbors': current_state['neighbors'], 'status': False}

# loop until no more updates
while True:
    propagated_labels = lpcc_rdd.flatMap(lpcc_mapper)

    updated_lpcc_rdd = (propagated_labels
                        .groupByKey()
                        .mapValues(lpcc_reducer))

    any_updates = updated_lpcc_rdd.filter(lambda x: x[1]['status']).count() > 0

    lpcc_rdd = updated_lpcc_rdd

    if not any_updates:
        break  # Exit
```

```
[448] lpcc_rdd.sortByKey().take(10)
```

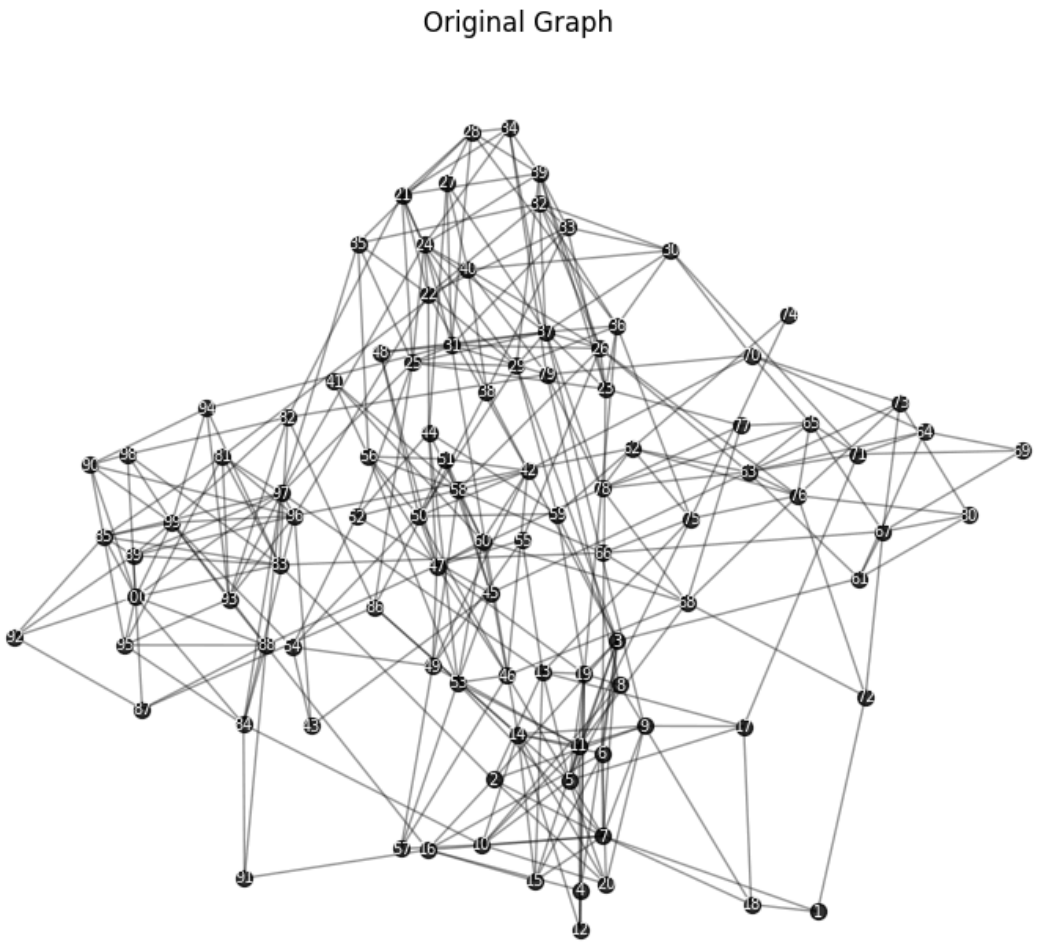
```
[(1, {'label': 1, 'neighbors': [18], 'status': False}),
 (2, {'label': 1, 'neighbors': [7, 9], 'status': False}),
 (3, {'label': 1, 'neighbors': [5, 19, 59], 'status': False}),
 (4, {'label': 1, 'neighbors': [12, 19], 'status': False}),
 (5, {'label': 1, 'neighbors': [3, 11, 13, 19, 8, 10], 'status': False}),
```

The reducer identifies the smallest label from its neighbors, then updates its own label under if the label coming from neighbors is smaller than its own.

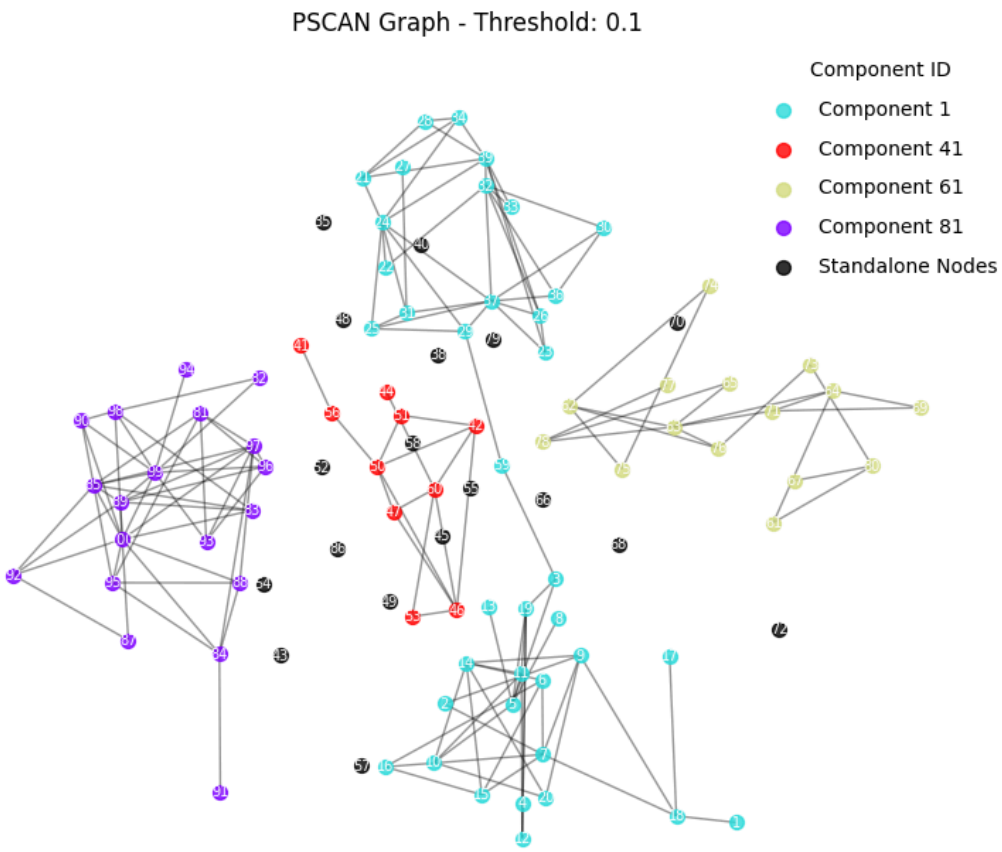
3. Visualization of the graph for analysis

   To visualize and compare how different threshold values affect the clustering results with the PSCAN algorithm using a network graph, we can use Python's NetworkX library.

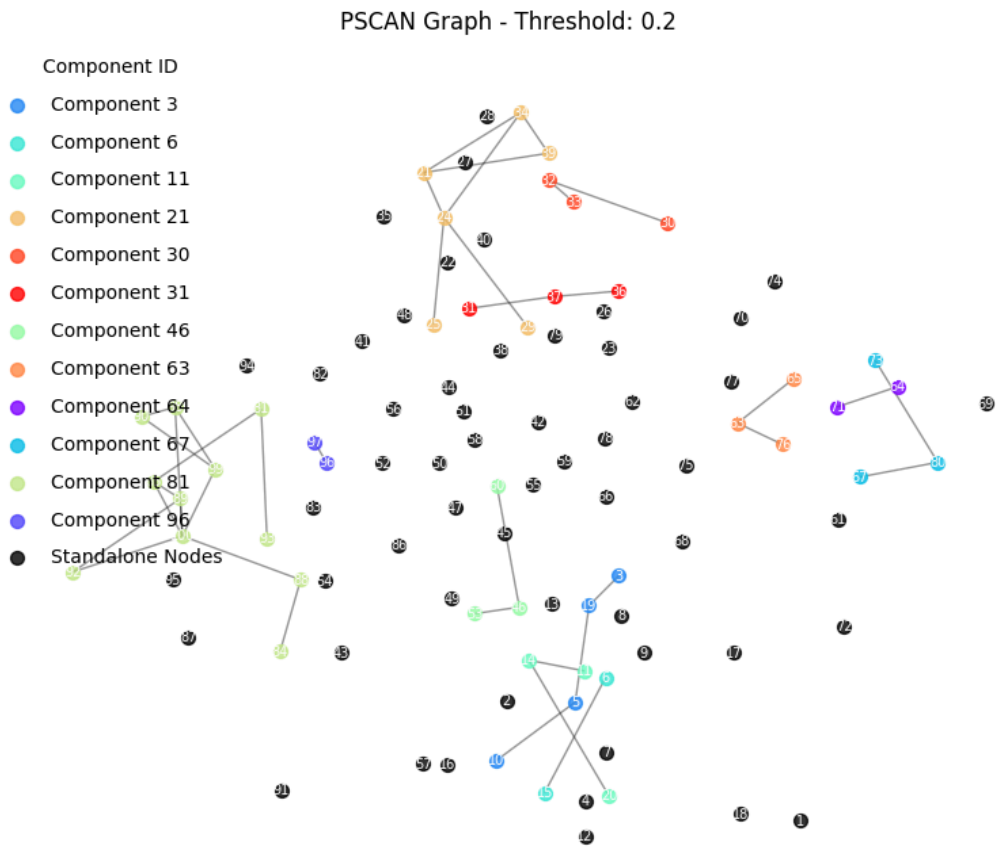
Original Network:




Network with Threshold = 0.1:




Network with Threshold = 0.2:




## CONCLUSION

My implementation and experiment showed that for this toy dataset, a structural similarity threshold of 0.1 works best. It finds a good balance, keeping enough connections to show community structures without losing too much data. But, raising the threshold to 0.2 cut too many connections, almost breaking the network's connectivity. This shows how crucial it is to choose parameters carefully for the success of the algorithm. It also shows how sensitive network analysis results are to these settings.

Encouraged by these initial findings, I plan to work with bigger and more complex datasets. The next step is to look at larger networks to see how well the algorithm scales and performs. Datasets from sources like the Stanford Large Network Dataset Collection, like the one at https://snap.stanford.edu/data/ego-Facebook.html, are ideal for further study. These datasets reflect real social network structures and will be great for testing the algorithm's speed and efficiency with more data.

Also, I'm interested in upgrading my computer resources. I want to build a "parallel machine system" - a powerful setup of many cheap computers working together. This could greatly increase our processing power and reduce the time it takes to analyze big datasets, making it possible to work with much larger datasets more efficiently.

## REFERENCE

[1] W. Zhao, V. Martha and X. Xu, "PSCAN: A Parallel Structural Clustering Algorithm for Big Networks in MapReduce," 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), Barcelona, Spain, 2013, pp. 862-869, doi: 10.1109/AINA.2013.47.