# HY-562 - Report for Assignment 3

Gkigkis Petros - A.M. 948 - gigis@csd.uoc.gr

The algorithm was implemented using the Hadoop MapReduce approach.

Below you can find the workflow of the implemented algorithm:

In order to make the algorithm parallel similar to the PSON approach I had to create a phase 0. In this phase a preprocess divides the input file in “N” equal row files.

The goal of this phase to be able to spawn more mappers.

**Phase 1:**

The Phase 1 was implemented according to the exercise. Using the phase 0 I can split the main file in N files (For complexity reason lets assume N = 4) and process each file in a different mapper. Each mapper produces the local candidate itemsets by running the given Apriori algorithm. The apriori algorithm takes as input the data and a local threshold to compute the itemsets. (each mapper runs separately the Apriori algorithm). The output of mappers in this phase returns <Key, Values> where as a key is an itemset and value is 1. (We don’t care at this moment about the value of the pairs)

In the reduce phase it merges the itemsets that were previously produced and returns <Key, Values> where as a key is a local candidate itemset and value is the sum of its occurrences. We may have false positives in this phase but in Phase 2 these false positives will be eliminated.

**Phase 2:**

In phase 2 every mapper receives a copy of the local candidates that were previously generated but also receives a part of the file (chunk). (Using the phase each mapper can get a different partition of the file). Every mapper process the assigned partition and compares each basket with the produced local candidates itemsets. If a local candidate itemset is inside a basket (row) it increases an internal counter for this local candidate itemset. The output of each mapper is a <Key, Value> pair where C is a  candidate frequent itemset and the support for that itemset among the baskets in the input chunk.

Finally, on the reduce phase it calculates the values of each key and compares them against a threshold. This threshold is the global threshold that the user selects. If an itemset is above or equal to the threshold the reducer will write it to the final file.

**Validation of the Algorithm and Experiments:**

In order to test the algorithm performance I did multiple runs against different combinations of local and global partitions but also against different number of partitions. Furthermore, I compared my implementation against the Mahout FPGrowth algorithm.

**Short discussion**

The performance of a frequent itemset mining algorithm depends on various factors.

The most important factor is the the thresholds someone uses. A small local threshold will lead to increasement of the computations but also increases the number of candidate itemsets. As an outcome there will be an increasement on the running time but also it will computes more RAM and CPU.

There is no magic number to use either as local or as global threshold. The threshold depends on the dataset and the items that includes but also the user.

In order to get an idea of the proper threshold we need to have examine the dataset. Some important things to look is the average number of columns, the number of rows and finally estimate the number of distinct items.

Moreover, another important factor is the number of partitions. Using a high number of partitions will lead us to a high number of local candidate itemsets which will lead to more memory usage and need for more computational power. Using a high number of partitions will not make the things run faster and lighter but will have the opposite results. The number of local candidate itemsets will be increased due to false positives. Authors of the PSON paper showed that in order to keep your program fast you need to keep all local candidate itemsets in the main memory of the node. So one approach to select the proper number of partitions is to check after which point the number of local candidates becomes large and goes out of the main memory.

Finally, one the “problems” of my approach is the algorithm that I use to compute the itemsets. For my implementation I used a version of the Apriori algorithm which works but its performance can be significantly improved.

In order to test the proper number of partitions and the combination of local and global thresholds I did multiple runs of the algorithm. I saw that the parallel version works worse than the single mapper.

In my opinion the cause of this is related to two factors.

The most important factor is that inside each node I run an instance of the Apriori algorithm for the given chunk. Despite the fact that each chunk is a small part of the input file it seems that the algorithm works better on a large dataset than n smaller datasets. I couldn' t validate this assumption due to the limited computer resources. Using two cores to simulate 4 Hadoop nodes I can not make a safe and clear claim.
Secondly, the number of  local candidate  itemsets maybe produce overhead.

## Hardware

I assigned two cores and 8 GB RAM to run the cloudera VM.

## Results

### Table 1

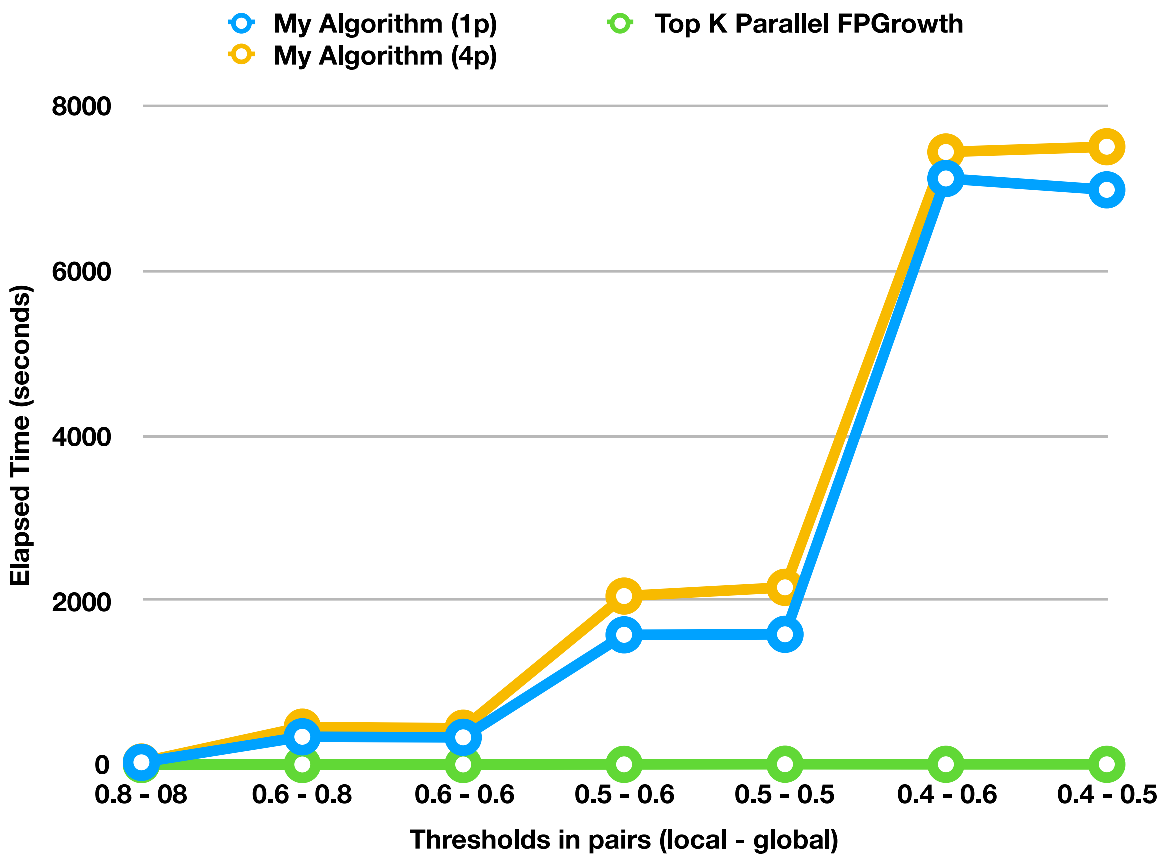
| Local Threshold | Global Threshold | Partitions | Elapsed Time (seconds) | Itemsets |
|---|---|---|---|---|
| 0.8 | 0.8 | 1 | 29 | 138 |
| 0.6 | 0.8 | 1 | 340 | 138 |
| 0.6 | 0.6 | 1 | 333 | 2053 |
| 0.5 | 0.6 | 1 | 1579 | 2053 |
| 0.4 | 0.6 | 1 | 7225 | 2053 |
| 0.5 | 0.5 | 1 | 1585 | 8033 |
| 0.4 | 0.5 | 1 | 6952 | 8033 |

**Table 2**

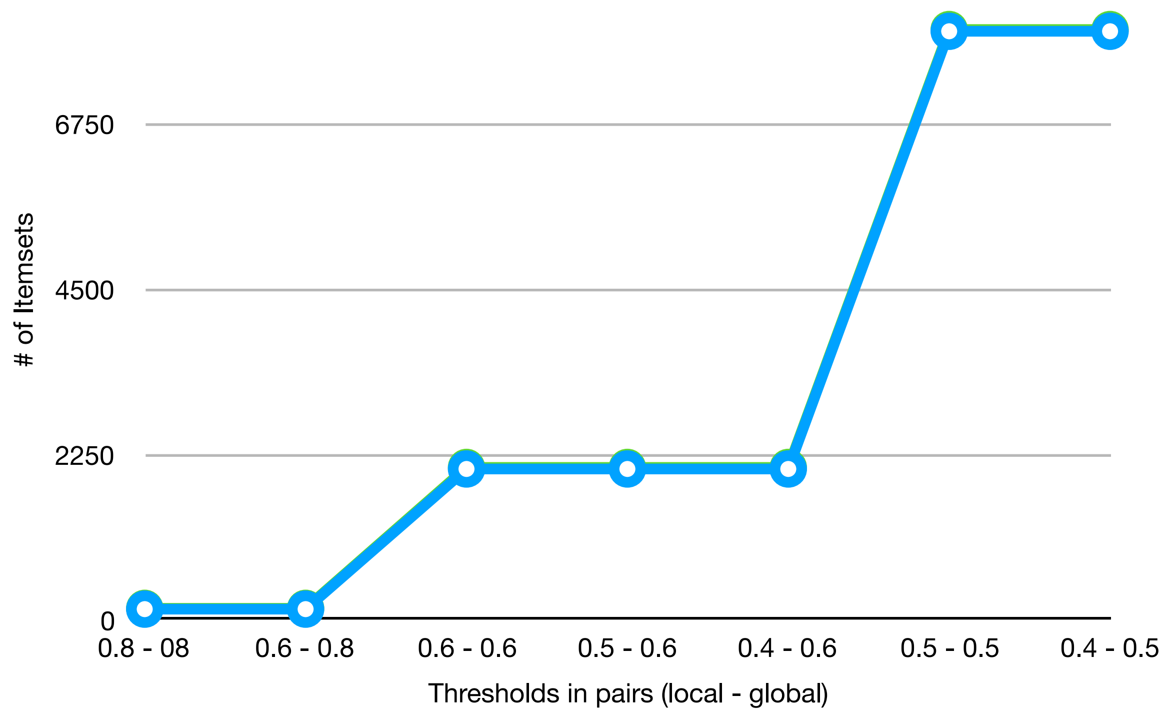
| Local Threshold | Global Threshold | Partitions | Elapsed Time (seconds) | Itemsets |
|---|---|---|---|---|
| 0.8 | 0.8 | 4 | 33 | 138 |
| 0.6 | 0.8 | 4 | 457 | 138 |
| 0.6 | 0.6 | 4 | 444 | 2053 |
| 0.5 | 0.6 | 4 | 2050 | 2053 |
| 0.4 | 0.6 | 4 | 7147 | 8033 |
| 0.5 | 0.5 | 4 | 2156 | 8033 |
| 0.4 | 0.5 | 4 | 7311 | 8033 |

On the above table 1 and 2 we can see different runs of the algorithm using different thresholds (local & global), the elapsed time of each run and the number of the produced itemsets. Table 1 referrers to runs for 1 partition and table 2 for 2 parititons.

We can see that as the local threshold decreases the elapsed time and number of itemsets increases.



On the plot above we see the different threshold pairs (local, global) but also the different number of paritions against the elapsed time. Its clear that when the local threshold decrease the elapsed time increases. Furthermore, I compared My algorithm runs for 1 and 4 partitions respectively against the Top K Parallel FPGrowth (Mahout). Its clear that FPGrowth wins.



Next is plot of the runs using different threshold pairs (local, global) against the number of itemsets. Its clear that as the thresholds decreases  the number of itemsets increases. Furthermore on the table 3 below we can see the different number of itemtsets.  The Top K Parallel FPGrowth  seems to generate more itemsets but this is not true due to the fact that in my approach I exclude itemsets with size 1. If I stop excluding them I will get the same number of itemsets.

| | 0.8 - 08 | 0.6 - 0.8 | 0.6 - 0.6 | 0.5 - 0.6 | 0.4 - 0.6 | 0.5 - 0.5 | 0.4 - 0.5 |
|---|---|---|---|---|---|---|---|
| My algorithm | 138 | 138 | 2053 | 2053 | 2053 | 8033 | 8033 |
| Top K Parallel FPGrowth | 149 | 149 | 2074 | 2074 | 2074 | 8057 | 8057 |

## Memory Usage

Looking on the Hadoop Web UI the most time consuming operation is the phase where the Apriori algorithm runs on each mapper.

Moreover, the phase that the biggest amount of the memory was consumed was the phase where the Apriori algorithm produces the local candidate itemsets.

## Comparison: My algorithm vs Top K Parallel FPGrowth

When it comes to compare the My algorithm against the Top K Parallel FPGrowth we have a clean win of Top K Parallel FPGrowth. It wins in time consumption, memory usage but also CPU.

For the same tasks My algorithm needs hours to run and  Top K Parallel FPGrowth can compute the result in less than 1 minute.

I did all the experiments of My algorithm and saw that the Top K Parallel FPGrowth needed at most 28 seconds.

I tried to use global threshold at 0.1 and run My algorithm
This resulted after 6 hours my VM to crash. (probably due to memory)

However, Top K Parallel FPGrowth delivered the results in 59 seconds.
The number of the computed itemsets was 10691549.