High Performace Computing for Data Science

Group 8

Project Roadmap

May 2021

UNIVERSITY OF TRENTO

Gabriele Masina
220496

Giovanni Zotta
223898

1 Introduction

This document explains our plans for the first project of the Course on High Performance Computing for Data Science.

For the first part of our project, we decided to work on a text mining algorithm called FP Growth algorithm [1], which stands for Frequent Pattern Growth algorithm. First of all, we are going to explain what problem this algorithm solves and how the serial version works, in order to have a bird's-eye view of what we are working with.

2 Problem Analysis

2.1 Serial version of the algorithm

The main focus of the FP Growth algorithm is to find associations between items. As an example for this explanation, we will take into consideration a database of transactions that represent items purchased by customers in a shop. As explained in [1], the serial algorithm compresses the transaction database into a Frequent Pattern tree, for which it performs the following steps:

1. enumerate how many times an attribute appears in the whole database. This is called *support count*;
2. sort the newly generated list of support counts by number of occurrences;
3. execute algorithm 1.

Algorithm 1: build_fp_tree

```
Input : List support_count
        Set transactions
FP_tree ← new FP_Tree(⊥)
foreach t ∈ transactions do
    sort(t, support_count)        // Sort t by their support count in decreasing order
    foreach item ∈ t do
        if item ∉ FP_tree.children then
            FP_tree.add_children(item, 1)
        else
            FP_tree.increase_children_count(item)
        FP_tree ← FP_tree.children[item]
    FP_tree ← FP_tree.root
return FP_tree
```

3 Proposed parallelization of the algorithm

Now we discuss how we plan to parallelize the algorithm in order to speed up the performance. We will distribute the load among N pc in the cluster using the MPI library and possibly exploit also multi-processing via OpenMP.

The solution we propose considers a tree-like hierarchy of processes, where there is a *root* process that coordinates k level-1 children, which in turn coordinates their children and so on. We still do not know how deep and wide this structure will be, we plan to decide it as we develop and test the solution. Anyway, the objective is to make it easily scalable and adaptable according to the dataset.

We will refer to the lowest-level processes as *workers*, and to the process "above" another as its *coordinator*. The processes managed by a coordinator are called *employees*.

3.1 Find the support of the items

We assume we have a file containing a transaction in each row. A transaction is made by a list of comma-separated items, each item being a string of characters. In order to distribute this step, we let each worker read a partition of the file (a set of contiguous rows) and, using a map, count how many times an item is seen. Afterwords, each worker sends the computed map to its coordinator, which combines the maps of all its employees by summing up the counts of each item. Then, it passes the combined map to its coordinator which in turns combines all the maps received. This process is repeated until we reach the root, which can reconstruct the whole map and send it back to all the processes which will need it later.

3.2 Sort the items by support

In order to speed up this step, we are going to implement a distributed QuickSort algorithm. Each coordinator performs the pivot step and then passes the generated sub-arrays to its employees, which will apply the algorithm recursively (if we have more than two children, we could possibly use Multi-pivot QuickSort in order to divide evenly the array among the employees [2]). The leaf processes sort the received sub-array with some algorithm, possibly using multi-threading.

Here we would have a nice opportunity to integrate MPI and OpenMP in a hybrid parallelization architecture, because if we were to use only MPI we would likely have to divide the array in many parts, leading to a large overhead caused by network transmission and copying of the data structure. Instead, if we use a few MPI processes and give them a comparatively big portion of the data, we can minimize the network overhead and exploit the fact that each portion of the array would be managed by a single MPI process. This would mean to have that portion of the array in shared memory, where we can use OpenMP to parallelize that section of the computation in a more efficient manner.

3.3 Grow the tree

We notice that also the growth of the algorithm can be performed following a divide-et-impera pattern. For instance, if we have two sub-trees grown on two disjoint subsets of the transactions, we can combine them to build the FP tree of the union of the two transaction subsets as follows. Starting from the root, we merge the common paths of the two trees, by summing up the values of the common nodes. The paths that are present in only one tree are the added to the tree. We show an example in Figure 1.

3.4 Further steps

Up until this point, we have basically compressed the database of the transactions into a tree data structure that can be used to efficiently extract meaningful information from the dataset. The complete algorithm would continue with some more steps aimed at the creation of association rules between the frequent itemsets, which would be the final product of the algorithm.

However, for the scope of our project we will try to implement the above descripted parallel algorithm, which already poses significant challenges, like distributed sorting and distributed merging of trees. If at a certain point in the implementation we understand that this goal is well within reach,

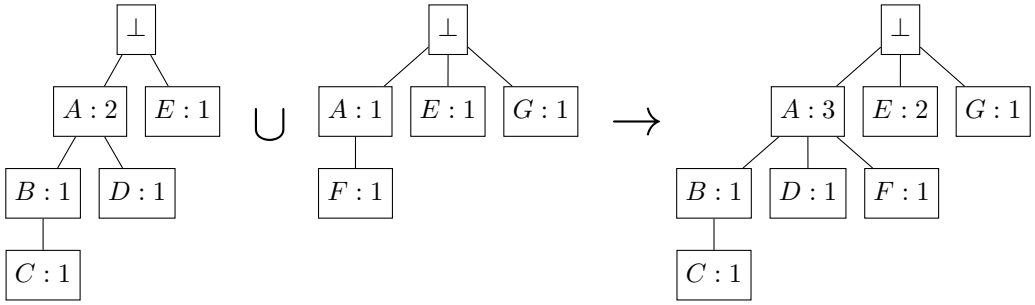

Figure 1: Merging two sub-FP trees

we could also think about expanding the algorithm with the successive steps that would lead to the creation of association rules.

References

[1] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, p. 1–12, May 2000.

[2] S. Kushagra, A. López-Ortiz, A. Qiao, and J. I. Munro, "Multi-pivot quicksort: Theory and experiments," in *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 47–60, SIAM, 2014.