Project Report
Parallel implementation of FP-Growth

Authors:

- Samuele Casarin 862789
- Roberto Perissa 859143

Contents

1 Introduction

The project consists in implementing a solution for the frequent itemsets mining [1] problem and parallelize it. In particular, we have implemented the FP-Growth algorithm [3] in C++, using OpenMP for the model of thread-based parallelism. Finally, we have tested our implementation on the following datasets: `chess` and `accidents` [2].

The code is available at `https://github.com/eleumasc/frequent-itemsets`

2 Sequential implementation as-is (v1)

First of all, we implemented a sequential version of the FP-Growth algorithm as described in literature. Briefly, the algorithm works as follows:

1. read the dataset and sort each transaction by descending frequency of the items (this heuristics reduces the number of nodes in the FP-Tree);
2. build the FP-Tree from the sorted transactions;
3. mine the frequent itemsets recursively by building the corresponding conditional FP-Tree.

Unlike the original version, our implementation has a small improvement for the construction of a conditional FP-Tree: for each node, there is a set called *prefixOf* which enables to know in O(1) time whether an item is present in some node of the subtree rooted on such node. This set is implemented using the STL class `std::set`. Furthermore, we prune nodes associated to infrequent itemsets during the construction of the conditional FP-Tree.

3 Sequential implementation with support memoization (v2)

The most critical performance issue of the direct implementation is that the support of each itemset is very often recomputed for the construction of a conditional FP-Tree. Since the value does not change, this is clearly a waste of time and so it led us to precompute the support of each itemset in the conditional FP-Tree and memoize it. Moreover, in the original version, an FP-Tree has an header which defines the list of nodes associated to each item; these lists are meant to be used to compute the support of the corresponding itemset, by summing up the count of each node of the list. The support memoization allowed us to remove the header.

Another time-consuming factor is the access to the data structure of the *prefixOf* attribute of each node. Instead of using the dense solution provided by `std::set`, we switched to a

sparse, but smart, data structure called `BitRange`, which is a bitmap over a range of indexes. In this case, such indexes are possible items that could be present in a node of the subtree.

4 Sequential implementation with memory reduction (v2.5)

The previous implementations use a lot of memory; in fact, when a conditional FP-Tree is constructed, the actual tree nodes are created in the heap memory.
In order to achieve memory reduction, we took advantage of the fact that if a prefix (i.e. the path to a node) is present in the conditional FP-Tree, then the same prefix is present in the original FP-Tree too. Hence, in this version there is only the original FP-Tree in memory, shared between all the conditional FP-Trees derived from it. The nodes that are actually included in the conditional FP-Tree are indexed by item in a header table: in particular, for each item, this table contains the list of pointers to the corresponding nodes in the original FP-Tree and their current count in the conditional FP-Tree.
We have called *climbing algorithm* the algorithm used to construct the header table of a conditional FP-Tree in relation to a given item. It works as follows: for each record *(node,count)* in the row of the header table associated to such item, "climb" the nodes by following the *parent* pointers as much as possible, and inserting the encountered nodes in the header table of the conditional FP-tree with the respective count. If the node of the next pair is descendant of an encountered node, then the latter is pushed in a stack together with its partial count and extracted when it will be encountered again.
By reducing the amount of memory used, the execution time is also reduced, because many memory accesses have been avoided.

5 Naive parallelization on top-level itemsets (v3)

Our first attempt to parallelize the algorithm consists in splitting the subtrees rooted on the top-level itemsets of the frequent itemsets tree among different threads. In practice, we parallelize the for-loop iterations of the mining function only at the first level of recursion.
The goal of this experiment is to gain some early results to analyze for developing smarter strategies, in order to understand the level of detail we have to reach for a relevant speedup. However, we think that this solution cannot give a great speedup because a thread that has finished to work cannot help another thread which is still working on a huge job.

6 Static task creation (v4)

We tried to distribute the exploration workload of the frequent itemsets tree equally, assuming it is complete. Hence, in this solution each worker explores a subtree of similar size.

Considering the tree as a left-child right-sibling structure, given a number of available workers, from the leftmost itemset of the level:

1. assign n/2 workers to the left child of the itemset, if frequent;
2. assign n/2 workers to the right sibling of the itemset;
3. repeat.

However, this implementation gives a poor speedup, in many cases it is also the slowest compared to the previous ones.

6.1 Workload analysis

To understand why the parallelization was ineffective, we proceeded to check each worker payload. The first evidence of an ill-splitted workload is that the various workers reported to visit a greatly varied number of nodes: one worker visited almost all the tree, a pair some thousands nodes and the remaining just about ten. After this result, we wanted to understand why the division was so unfair and lead us to the visualization of the itemsets tree for each dataset we used for the tests.

The single but critical issue of this approach is that the assumption of a complete tree is wrong. By looking at the first level of the tree, it has the maximum amount of payload under the leftmost itemsets. However, in practice, almost all of that work is cut off by the border, resulting in a very different shape. The maximum amount of work is rooted on the central itemsets and it is shifted depending on the minsup threshold. [Figure 4] [Figure 5]

After this result, it is clear that our approach could have been effective only if we had changed the thread assignment procedure. We considered to redefine the procedure starting from the middle itemsets. This updated approach required to guess the root of the maximum workload as the first step, but all our attempts to design a simple formula to obtain such guess were very inaccurate. Thus, we abandoned this approach and radically changed the method: we switched to a fully dynamic approach to obtain a scalable and adaptive solution.

7 Dynamic task creation (v5)

Since we do not know how the workload is distributed, in the final parallel version we have adopted a solution based on the producer-consumer pattern.
Starting from the root of the frequent itemsets tree, a task is created for each node; in

particular, a task is always created for the right sibling, while a task is created for the left child only if the itemset is frequent.
This first attempt did not give us a significant speedup because creating a task for each node of the frequent itemsets tree generates too much overhead (due to the task creation itself and context switch). In order to solve this issue, the intuition is that if the tree (or a part of it) is sufficiently small, then the sequential solution is faster than the parallel one. We have added a parameter to our algorithm, called *sequential bound* (abbr. *seqBound*): if the minimum item of the itemset is less than *seqBound*, then there is no task creation and so the execution of such part of the tree is handled sequentially by the same thread. This parameter must be not too small, otherwise there is overhead, neither too big, or else the execution is almost sequential. Hence, we have assumed that there is a range of values that makes the execution time optimal. We have varied the value of *seqBound* over the set of items of our datasets and we have found that 20 could be a suitable value. [Figure 6] [Figure 7]

8 Performance analysis

8.1 Testing

We have tested our implementations on a computer with: `Intel Core i7-6700 CPU @ 3.40GHz, Cores 4, Threads 8, L1 Data Cache Size 4 x 32 KBytes, L2 Cache Size 4 x 256 KBytes, L3 Cache Size 8192 KBytes, Memory Size 8192 MBytes @ 1066 MHz`.
On the two datasets, we have carried out the test to acquire the execution time for each version. We have launched the various versions 3 times and kept the minimum value, with timeout after 10 minutes. For both datasets, we have identified a very-low minsup (abbr. v.l.m.) that corresponds to an intensive execution, on `chess` it is equal to 15% and on `accidents` is equal to 2%.

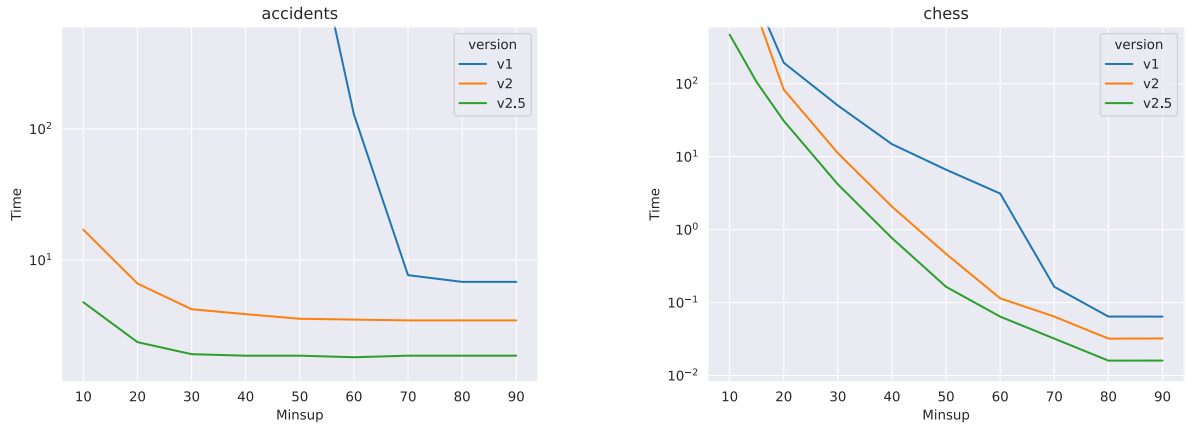

Figure 1: Left: Sequential times on `accidents`. Right: Sequential times on `chess`

Regarding the sequential versions, only v2.5 is capable of handling the lowest minsup for both datasets. On `accidents` v1 goes in timeout very early, on `chess` both v1 and v2 go in timeout just before the v.l.m.

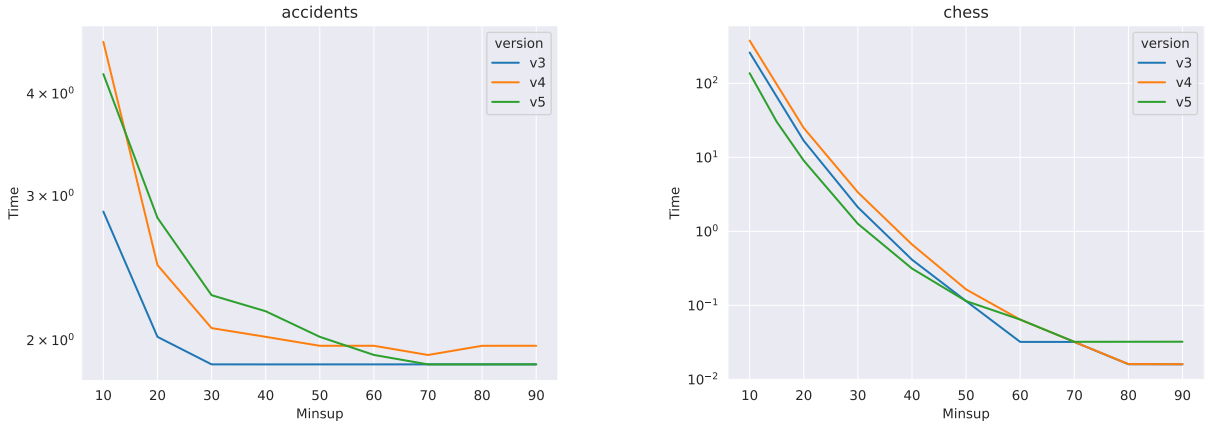

Figure 2: Left: Parallel times on `accidents`. Right: Parallel times on `chess`

For the parallel versions, it is interesting to notice that on `accidents` v3 has the best times for every minsup over the v.l.m., instead on `chess` v5 is the best for all the lower minsups (below 50%).

8.2 Speedup

Comparing v3 and v4 parallel versions to the v2.5, we have computed respectively the following speedups considering only the best results: v2.5 vs v3 $\rightarrow$ 1.8x; v2.5 vs v4 $\rightarrow$ 1.25x.

The two final versions are v2.5 and v5, for the sequential and the parallel implementation respectively. The performance comparison between the two is computed over the v.l.m. of both datasets.

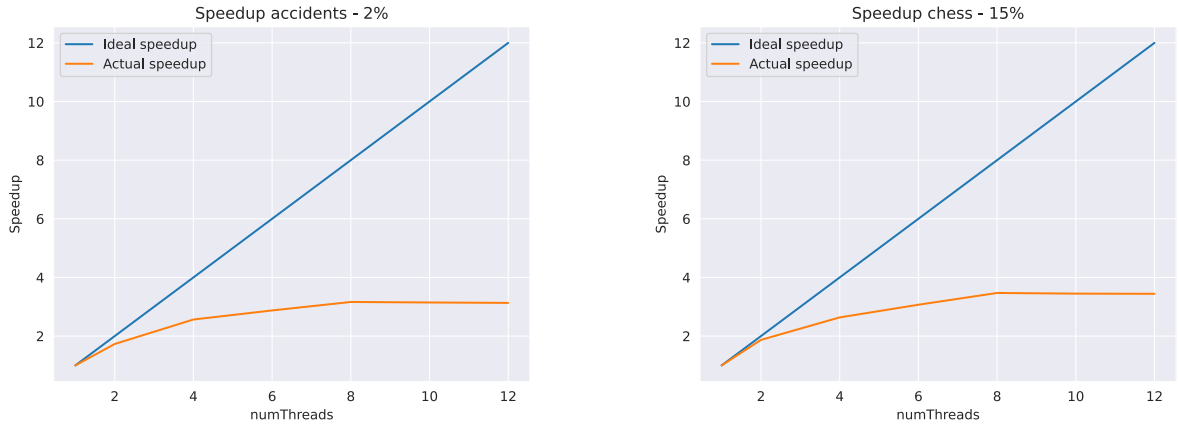

Figure 3: Left: v2.5 vs v5 speedup on `accidents`. Right: v2.5 vs v5 speedup on `chess`

The speedup on `accidents` is 3.16x and on `chess` is 3.47x. Clearly, the speedup is sub-linear as there is overhead due to the task creation and context switch, but we see a good scalability for both datasets and it is noticeable that the speedup stops growing when the number of threads exceeds the level of hardware concurrency.

9 Cache analysis

Although during the design of the algorithm we have not thought about cache efficiency, we still have tested its performance using the Unix tool `perf`.

Version	Dataset	Cache-miss %
v2.5	`accidents`	32,179 %
v2.5	`chess`	13,383 %
v5	`accidents`	22,160 %
v5	`chess`	0,223 %

The percentage of cache misses is very satisfactory, especially for the parallel version.

10 Conclusions

Finally, for this project we have focused on optimizing the FP-Tree data structure through the sequential versions, while the parallel ones are focused on distributing the mining workload in order to get a significant speedup. The final version v5 has a good scalability: compared to v2.5, it has a 3.47x speedup; compared to v1, it has a 21.5x speedup.

11 Appendix

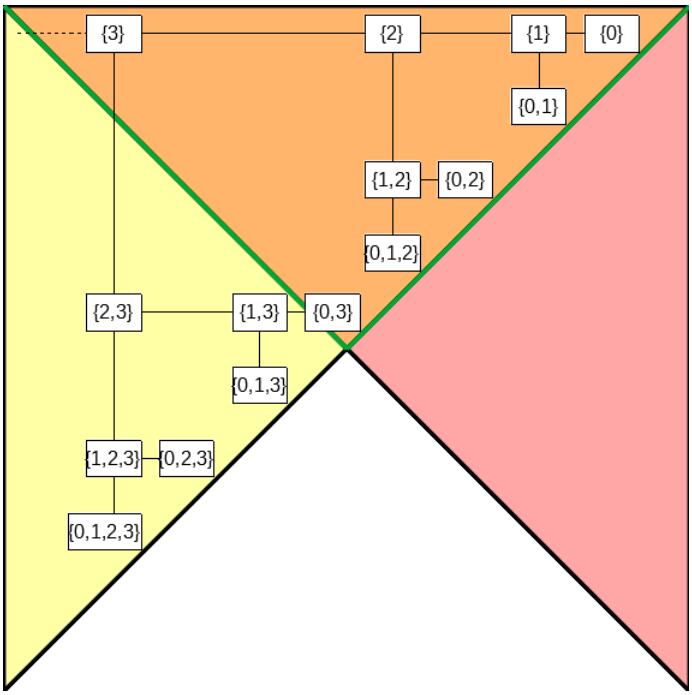

Figure 4: Frequent itemsets tree scheme. Legend - yellow: complete itemsets tree; red: "likelihood" for an itemset to be frequent; orange: frequent itemsets; green line: itemsets above that compose the border

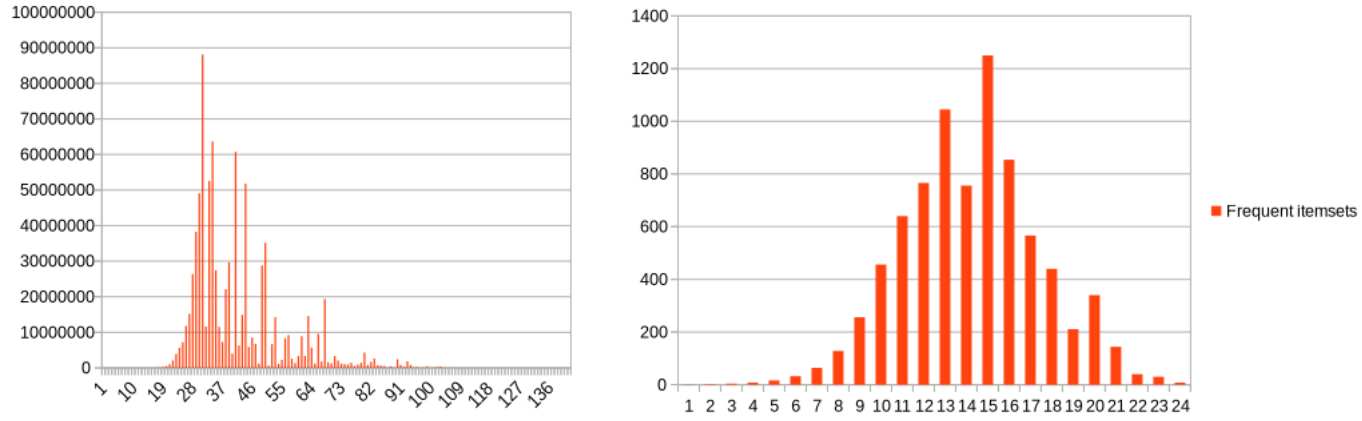

Figure 5: Number of frequent itemsets descendants of the first level singletons on `accidents`. Left: minsup 2% Right: minsup 50%.

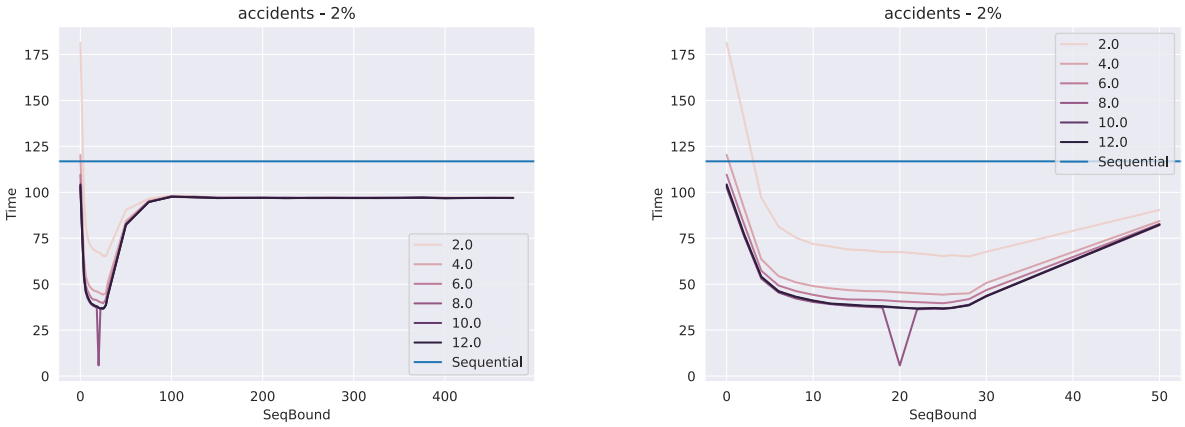

Figure 6: Complete overview of seqBound on `accidents`.

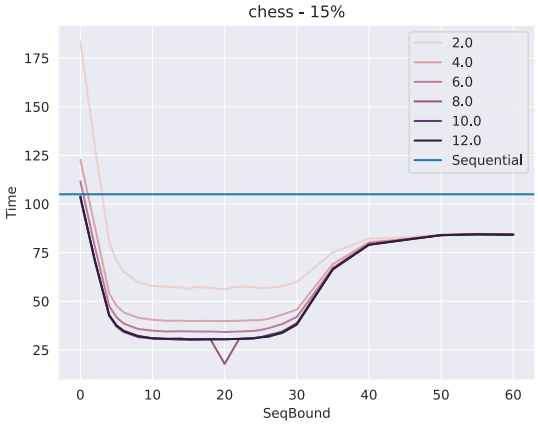

Figure 7: Complete overview of seqBound on `chess`.

References

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining Association Rules between Sets of Items in Large Databases". In: SIGMOD '93 (1993), pp. 207–216. DOI: 10.1145/170035.170072. URL: https://doi.org/10.1145/170035.170072.

[2] *Datasets*. http://fimi.uantwerpen.be/data/. [Online; accessed 8-April-2021]. 2021.

[3] Jiawei Han, Jian Pei, and Yiwen Yin. "Mining Frequent Patterns without Candidate Generation". In: *SIGMOD Rec.* (May 2000), pp. 1–12. DOI: 10.1145/335191.335372. URL: https://doi.org/10.1145/335191.335372.