High Performance Computing for Data Science

Group 8

Project Report

Parallelization of the Frequent Pattern Growth algorithm

June 2021

Gabriele Masina
220496

Giovanni Zotta
223898

Contents

1 Introduction

This document explains what we have done for the first part of the project for the course on High Performance Computing for Data Science. The steps that we have taken resemble the outline that we defined on the Project Roadmap document, with a few modifications. This time we are going to dive deep on the implementation details in order to explain the choices that we have made and how we decided to parallelize the Frequent Pattern Growth algorithm. The entire codebase of our project is hosted on GitHub at the following link: `https://github.com/GiovanniZotta/hpc4ds/tree/main/project`. The code of our project is in the `/src` folder, and it is thoroughly documented. We generated a PDF with the documentation of the code via Doxygen, which is available in `/doc/doxygen/latex/refman.pdf`.

1.1 Acknowledgements

In this section we want to make clear what we have reused from other projects and what is original work. As C does not have a standard library with common data structures, we took and adapted an implementation for dynamic vectors from c-vector and of hashmaps from c_hashmap. For what concerns specific algorithms, we referenced them in their respective sections.

2 Problem Analysis

2.1 Serial version of the algorithm

The main focus of the FP Growth algorithm is to find associations between items [1][2]. The algorithm transforms the transaction database into a Frequent Pattern tree, by doing the following steps:

1. enumerate how many times each item appears in the whole database. This is called the *support* of the item;
2. sort the items in each transaction by decreasing support;
3. execute Algorithm 1.

Algorithm 1: build_fp_tree (serial)

```
Input : LIST $support\_count$
        SET $transactions$
1  $FP\_tree \leftarrow$ new FP_TREE($\perp$)
2  foreach $t \in transactions$ do
3      $sort(t, support\_count)$        // Sort $t$ by their support count in decreasing order
4      foreach $item \in t$ do
5          if $item \notin FP\_tree.children$ then
6              $FP\_tree.add\_children(item, 1)$
7          else
8              $FP\_tree.increase\_children\_count(item)$
9          $FP\_tree \leftarrow FP\_tree.children[item]$
10     $FP\_tree \leftarrow FP\_tree.root$
11 return $FP\_tree$
```

3 Main steps to parallelize the algorithm

Here we discuss how we parallelized the algorithm in order to speed up the performance. We distribute the load among N machines in the cluster using the MPI library and we exploit also multiprocessing via OpenMP for a couple of different shared-memory tasks.

We are now going to explain how we parallelized every part of the FP Growth algorithm, namely:

1. reading the dataset in a distributed manner;

2. counting how many times each item appears in the whole dataset;
3. sorting the list of items by support count;
4. building the FP Tree.

3.1 Reduce operation

Every task that we had to parallelize follows the same pattern. First, every process elaborates information regarding its own work, which is just a shard of the whole computation. Then, the goal is to spread the knowledge that has been computed in parallel by all the workers to every process in the cluster. However, in order to do this it is not sufficient to broadcast the information to everyone, since the outputs of the computation done by different processes have first to be combined.

To do this, we follow a reduce-like approach. Every worker, after completing its own work, sends its information to an upper-level process, which will then combine the information that it already has to the information that it has received. This pattern is executed recursively, until only one process remains. At this point, the so called *root* process has the complete information and is now ready to broadcast it to every process.

We have looked into using the `MPI_Allreduce` function in order to implement such approach through a User-Defined MPI Reduce operation. However, we have not found a suitable way to implement an User-Defined MPI operation which uses variable length buffers of data to conduct the computation. This is a problem for us, since for instance we do not know how many different items we have, so we can not allocate in advance an array to store the support of each. The same goes for the construction of the FP Tree, as when we combine two FP Trees we do not know the size of the result. This will be explained better in the following sections where we describe how these steps are actually implemented.

Therefore, our solution has been to implement a reduce-like operation from scratch, by using only `MPI_Send`, `MPI_Recv` and `MPI_Bcast` operations, in order to be able to send dynamic amounts of data without having to resort on static structures. When we need to send some data, we do that in two steps:

1. send an integer representing the number of items that is going to be sent;
2. send as many structures as previously declared.

Depending on the reduce operation, the type of the structure that we are sending varies. However, we either send an array of integers or a custom-built MPI Datatype.

In Figure 1, we go through an example of how this is done.

3.2 Distributed reading from file

We assume that our dataset contains a transaction in each row[1]. A transaction is made of a list of space-separated items, each item being a string of characters. In order to distribute the load among the processes, 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.

This has a series of subtle implications that we had to be aware of. Since with MPI there is not an option to read a text file by rows, we had to specify the bytes read by each process. However, we wanted to make sure everyone eventually had a fair amount of data with respect to the other processes in the cluster and, most importantly, that a process does not read just partially a transaction or an item. Since we are aware of the total size of the dataset in bytes, we make each process read twice as much as the bytes that it should process (Equation 1), and then we make sure that each transaction gets processed by exactly one process.

$$B = \frac{\text{File size}}{\text{Number of processes}} \tag{1}$$

[1]The dataset format is explained in Section 4.1

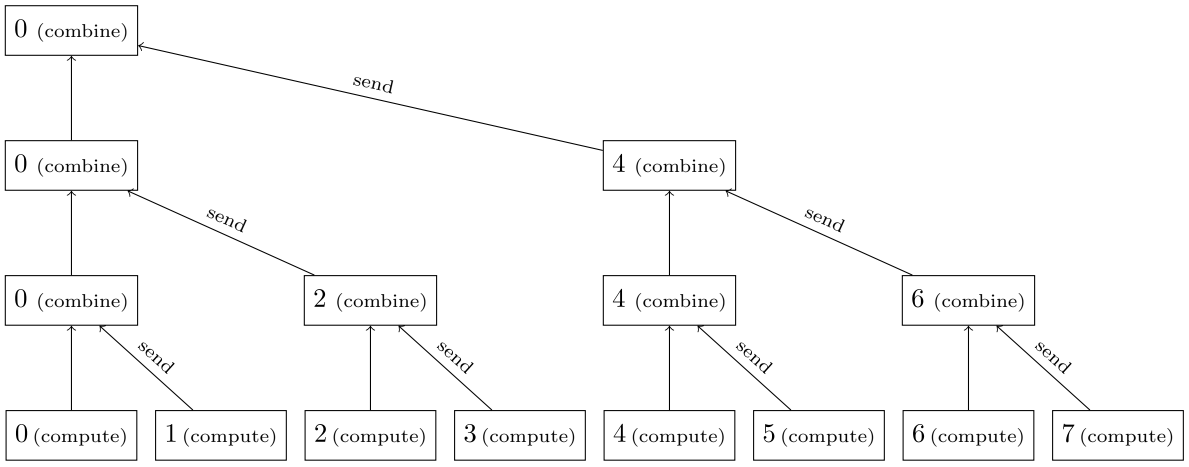


Figure 1: Here we illustrate the pattern followed by the reductions we implemented. Assume we have 8 processes, numbered from 0 to 7. At the bottommost level, each process performs some operation on its chunk of data. Then, each process with an odd rank i sends its output to the process with rank $i-1$. Then, this operation is repeated with processes with even rank i such that $i \mod 4 \neq 0$ sending their output to the processes with rank $i-2$. We keep going on until we reach the root of this binary tree, which has rank 0. The root has all the information, so it can broadcast it to every other process to share its knowledge.

3.3 Finding the support of the items

Now that every process has the chunk of data of the original dataset that is assigned to it, it creates a dictionary of the items that it has read along with their local support count. Up until this point, every process only has partial knowledge, indeed it only knows about the items that happened to be in the chunk assigned to it. Now, our objective is to have global knowledge of the support count of each item worldwide. To achieve this goal, we use the above explained reduce-like operation. In this case, the combine step consists of summing the support of each received item to the support already contained in the map of the target process. To be more detailed about the inner workings of this procedure, on the sending process we extract an array of hashmap elements (item-support pairs) from its map, then we send it to the target process by implementing a custom built `MPI_Datatype`, which is needed for the transmission through MPI. Once the target process receives this array of item-support pairs, it updates its support map as explained above.

At the end of the reduce operation, the process number 0 has global knowledge about the support of each item of the original dataset and it can broadcast it to every process in the world through a call to `MPI_Bcast`.

3.4 Sorting the items by support

In order to speed up this step, we implemented a distributed QuickSort algorithm. Our goal is to distribute the load as much as possible among our processes, so we decided to make each process sort only a single slice of the whole dataset. To do this, we have assigned a portion of items to each process, which will sort them independently. After this operation, each process has sorted its own sub-array, but is unaware of the order of items that are outside of its domain. In order to proceed to the next step of our algorithm, everyone has to have global knowledge of the order of the items, and therefore we adopt the same reduce-like approach as before. Before getting into the details of how it works, we have to talk about how each process sorts its own sub-array in an efficient and parallel manner.

We took our chance to use an hybrid parallelization approach by employing OpenMP to sort the sub-array of each process.

3.4.1 Parallel Quicksort with OpenMP

After a brief survey of existing parallel sorting algorithms, we decided to go for a balanced option in terms of performance and complexity. We opted for the implementation of version *sort_omp_1.0* of

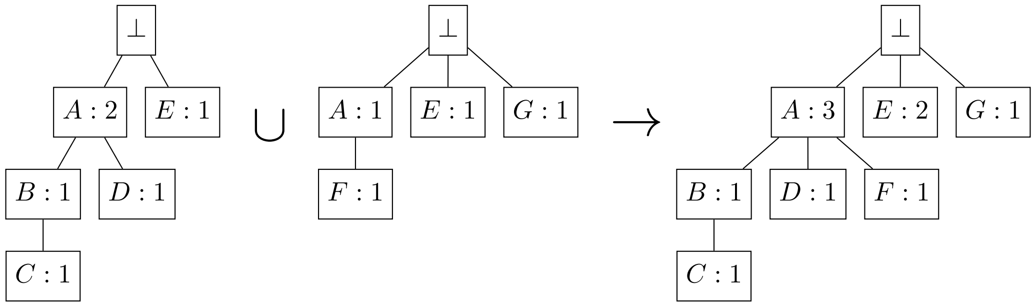


Figure 2: Merging two Frequent Pattern sub-trees

Quicksort suggested by Süß and Leopold [3].

To get started, we have a pool of threads that are ready to work. Let us say that we have an array of items that we want to sort by support. Initially, one thread chooses a pivot, it moves the elements of the array that are smaller than the value of the pivot to its left and the elements that are greater than the value of the pivot to its right. Then, the two parts of the array are left waiting to be processed until a thread takes the responsibility to process one of them. The entire procedure is conceptually recursive, reaching the base case when the size of the array to sort reaches a certain threshold, at which point the sub-array gets sorted via Insertion sort, which is very efficient for sorting small arrays.

3.4.2 Merging the sub-arrays with MPI

Now that each process has sorted its own subarray with OpenMP, it is time to get the global knowledge to every process. To do this, we adopt the usual reduce-like pattern that we have already seen before. The compute step for each process has already been discussed, so we will focus on how we conduct the combine phase. As a compute operation, we want to merge two sorted sub-arrays into a larger, sorted array. To achieve this, we use the merge operation of the classical Merge sort algorithm.

At the end, the process number 0 will have an array of every item in the dataset sorted by its support count, and can therefore broadcast its knowledge to every other process via `MPI_Bcast`.

3.5 Growing the tree

This is the crucial part of our algorithm, where most of the time is spent computing. Therefore, we had to make sure to parallelize the work among our cluster as much as possible. We adopt a strategy similar to the one used for sorting the items by support count; we take advantage of both MPI and OpenMP in order to parallelize the computation.

The goal is to have a Frequent Pattern Tree which represents the initial dataset in a more compact and useful manner. The high level idea of our implementation is the following: we subdivide the original dataset into subsets of transactions and initially we assign each subset to an MPI process. The latter will then proceed to split the work among N OpenMP threads, and each thread will be given a subset of the transactions that were assigned to the MPI process.

3.5.1 Frequent Pattern Tree merging

Before proceeding with the explanation of how the algorithm is parallelized, it is important for the reader to understand how two Frequent Pattern Trees can be merged together to form a bigger Frequent Pattern Tree. If we have two sub-trees grown on two disjoint subsets of the transactions, we can combine them to build the Frequent Pattern 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 simply added to the tree. We show an example in Figure 2.

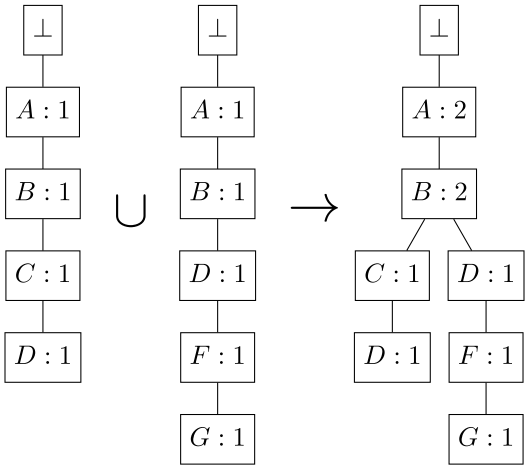


Figure 3: Merging two linked lists, each representing a transaction

3.5.2 Subset processing with OpenMP

Now, we will focus on how a single process parallelizes the computation of its assigned subset of transactions. For the ease of the reader, the entire procedure is depicted in Figure 4, so we suggest to keep it in mind throughout the explanation. We start from the concept that each transaction can be represented as a special case of a tree, namely a linked list. Then, we can merge the linked lists in order to form a tree as in Figure 3. If we imagine to merge every transaction of the subset together, we will end up with a Frequent Pattern Tree representing the subset of transactions that were assigned to the original process. Afterwards, we will use this to be able to merge Frequent Pattern Trees among MPI processes, which will be part of the next section.

Starting with the big picture in mind, it should be natural to think about a parallel pattern similar to the ones that we have already described in the previous chapters, as also this part of the computation serves itself well to be parallelized.

The building of the tree is performed by levels, as shown in Figure 4. At first, each thread is assigned a subset of transactions and it builds the FP Tree of each of such transactions. Then, at level 1, each thread merges pairs of transaction trees; this operation can safely be done in parallel by different threads as long as the pair of trees they merge are disjoint. Once we have done this step, we proceed to level 2, where we repeat the same operations merging pairs of trees generated at level 1. This sequence is repeated until we have merged all trees into a single one, which is a Frequent Pattern Tree of the chunk of transactions assigned to the process. The procedure follows the pseudocode shown in Algorithm 2.

Algorithm 2: build_openmp_transactions_tree

```
 Input : SET $transactions$
1  #pragma omp parallel num_threads($N$)
2  for $pow = 1; pow < 2 * transactions.size; pow \leftarrow pow * 2$ do
3      $start \leftarrow 0$
4      if $pow > 1$ then
5          $start \leftarrow \frac{pow}{2}$
6      #pragma omp for
7      for $i = start; i < transactions.size; i \leftarrow i + pow$ do
8          if $pow > 1$ then
9              $tree\_merge(trees[i - \frac{pow}{2}], trees[i])$
10         else
11             $trees[i] \leftarrow build\_tree\_from\_transaction()$
12 return $trees[0]$
```

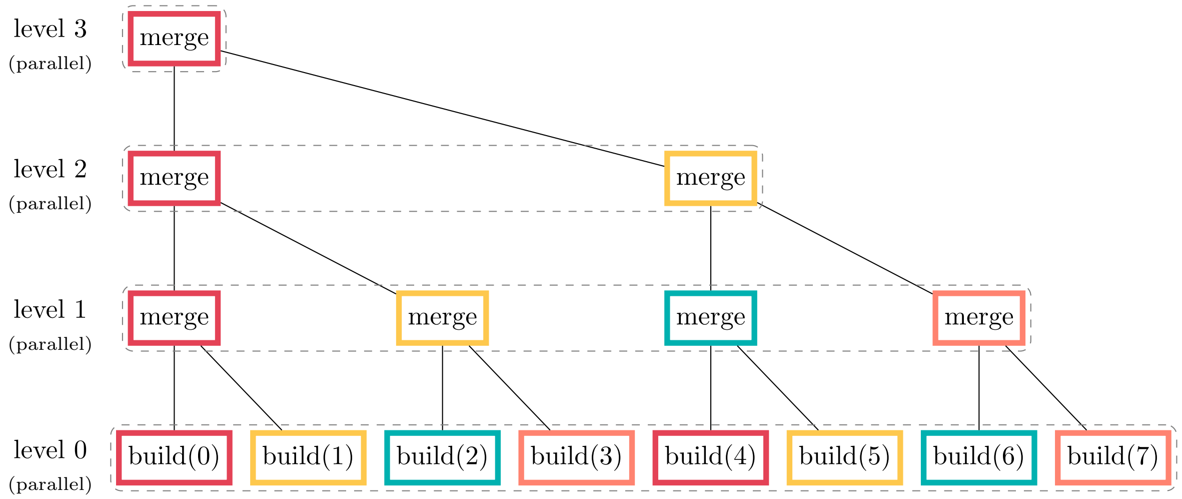


Figure 4: In this figure we show how the construction of the local tree is parallelized using OpenMP. The nodes represent the tasks that have to be performed, and their color represents the thread by which they are carried out. In this example, we assume to have 8 transactions, numbered from 0 to 7, and 4 threads T0,T1,T2 and T3. The tasks that belong to the same level of the tree are performed in parallel. Here we show the behaviour assuming we are using a static scheduling with chunksize 1.

3.5.3 Merging Frequent Pattern Trees with MPI

If we take a step back, we realize that now every process has created a Frequent Pattern Tree of the transactions that were assigned to it. This means that the processes now just need to spread the knowledge among themselves in order to all have the complete Frequent Pattern Tree representing the whole dataset. Once again, to do this as efficiently as possible, we adopt the familiar reduce-like pattern shown in Figure 1. We can think of the compute step as the one conducted previously with OpenMP, where the combine step is exactly identical as the one of Figure 2. At the end of this reduce operation, the process number 0 has the Frequent Pattern Tree representing the whole original dataset, and can broadcast its knowledge to every other process in the world.

4 Benchmarking

After implementing the algorithm, we wanted to understand how our parallelization impacts the performance of the algorithm. Therefore, we devised a series of parameters that we wanted to tweak to understand how the running time would have been affected.

4.1 Dataset

Before getting to the performance analysis part, we are going to define how the dataset on which we tested our algorithm is structured. After scouting different dataset formats, we settled for [4], which is a dataset made to test the performance of text mining algorithms [5]. Each file is composed of `5000000` transactions and `50000` items, and each transaction is composed of up to `100` items. The items are strings shorter than `16` characters, which was a requirement due to how hashmaps are implemented in our project. To benchmark our algorithm, we used 4 files from the dataset, with the following itemset densities[2]: `0.2, 0.4, 0.6, 0.8`. Each of those files is about 800MB in size. Table 1 shows an example of input file from our dataset.

[2]Dataset density specifies the percentage of baskets that intentionally contain frequent itemsets

Transaction 1	`I71 I13 I91 I89 I34`
Transaction 2	`F6 F5 F3 F4`
Transaction 3	`I39 I16 I49 I62 I31 I54 I91`
Transaction 4	`F4 F3 F1 F6 F0 I69 I44`
Transaction 5	`I22 I31`

Table 1: Example of input file

4.2 Minimum support

The FP Growth algorithm requires a parameter that indicates the minimum number of times an itemset has to appear in order to be considered frequent. Since we only perform the growth of the tree, for our purposes this means that we discard the items that are encountered fewer times than the minimum. We set this value empirically at the 0.1% of the number of transactions in each dataset, in order for the resulting tree to be big enough to allow us extract some significant statistics from the execution times.

4.3 Configurations

In order to understand how our parallelization improved the running time, we devised a series of configurations that would have helped us to understand what were the most efficient parameters to adopt. We wrote a script that allowed us to submit thousands of jobs to the HPC cluster of UniTrento across several days, in order to have good and reliable results to analyze. The number of tests that we have done amount to a total of approximately `3700` runs. Each run takes on average 5 minutes, so we had to spread the submission of jobs across several days. Every submission was run on 16 HPC nodes, each with 8 CPU cores and 64GB of RAM, apart from the serial jobs which were run on nodes with 1 CPU core and 64GB of RAM. Figure 5 shows the number of runs that we made for each configuration. Since the results had a non-negligible variance between runs, we had to make a good amount of trials for each configuration. We ran about 30 runs for static and guided scheduling policies, 50 trials for serial runs (1 process, 1 thread) and about 100 runs for the dynamic scheduling policy.

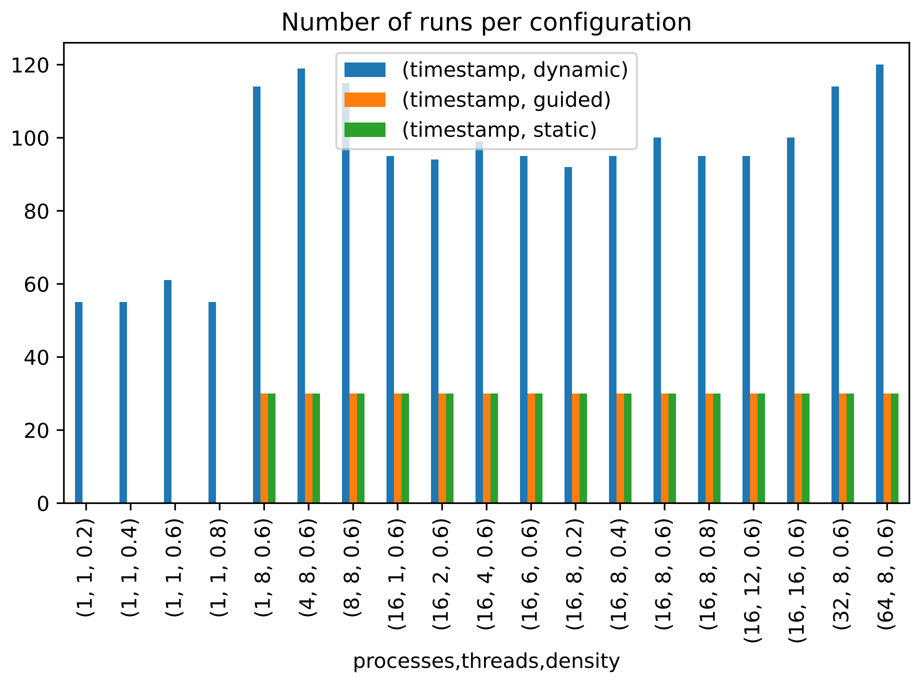


Figure 5: Number of runs made for each configuration

Since we had 128 cores at our disposal for each run, we defined a default configuration with 16 MPI processes and 8 OpenMP threads per process, totaling to 128 OpenMP threads. We then used this default configuration to iterate over different parameters in order to understand how they were impacting the performance of our algorithm, such as:

- number of MPI processes;
- number of OpenMP threads;
- itemset density;
- OpenMP scheduling policy.

In every configuration, the creation of the local FP Trees was tested with three different OpenMP schedules: static, dynamic and guided, each with chunksize 1. The sorting algorithm has been left with the default (dynamic) schedule since we noticed that its impact on the global running time of our algorithm was almost negligible, as shown in Section 4.4.

4.3.1 Number of MPI processes

We wanted to understand how the number of MPI processes would have affected our algorithm. Each of these tests has been made with 8 OpenMP threads and an input file with `0.6` itemset density. As we expected, the running time of our algorithm improved a lot by scaling our algorithm from 1 process to 16 processes, but we were still surprised by the improvements that we had by having 32 and 64 MPI processes. Figure 6a shows the running time of each configuration by manipulating the number of processes.

4.3.2 Number of OpenMP threads

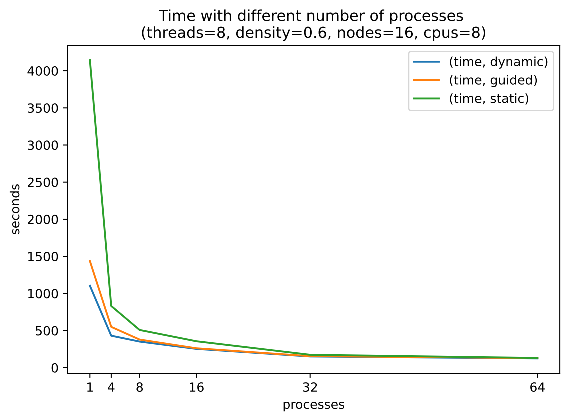
As we did in Section 4.3.1, we wanted to understand the performance impact that the number of OpenMP threads had on our parallelization. All the tests related to this category have been run with 16 MPI processes and an input file with `0.6` itemset density. The results were interesting: we can see that having 1 thread is actually not bad. Indeed, having 2, 4 or 6 threads is actually much worse than having only one thread. However, with the dynamic and guided scheduling policies and 8 or more threads we can see that OpenMP helps improving the running time of our algorithm, as we can see in Figure 6b.

The worsening of the running time may be due to the large amount of memory needed to build the local tree in parallel, since every thread has to store a large amount of information when building its local tree. A possible explanation for this poor performance is the higher number of page faults that is experienced by having many threads on the same node. Another important consideration is that, when we have more than 8 threads, we actually have more threads than CPUs, since we are running our tests with 16 MPI processes and 128 CPU cores, so it is expected not to experience any improvement. For future benchmarking, it would be interesting to test this on larger memory machines with more cores in order to test 32 and 64 OpenMP threads.

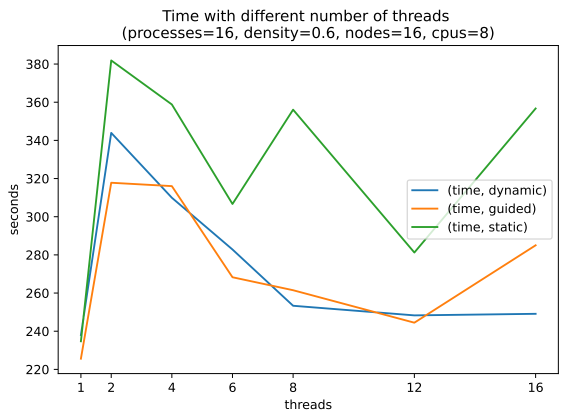
4.3.3 Itemset density

In this section of our benchmark, we wanted to know how our algorithm would have behaved with different itemset densities. Therefore, we tested it on four different densities (`0.2, 0.4, 0.6, 0.8`). As we can see in Figures 7a and 8a, the running time is inversely proportional to the itemset density. This is due to the fact that, the higher the itemset density in the dataset, the fewer nodes our final FP Tree has. These are the number of nodes of the final tree for the different itemset densities:

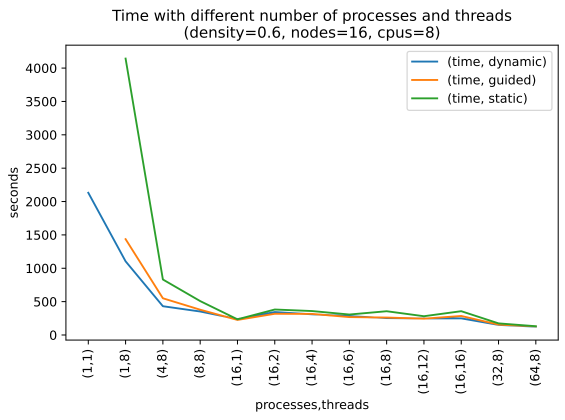
- `0.2: 112086275`
- `0.4: 104568723`
- `0.6: 93117403`
- `0.8: 80613921`



(a) Running time with different numbers of MPI processes



(b) Running time with different numbers of OpenMP threads



(c) Running time with different configurations

Figure 6: Measurements of running times with different configurations

4.3.4 OpenMP scheduling policy

Each of the above tests has been run with three OpenMP scheduling policies for the local tree creation. We tested the static, the dynamic and the guided policies, all with chunksize set to 1. Theoretically, the dynamic schedule with chunksize 1 is the most balanced policy, since it assigns the jobs on the fly. However, it could have a non-negligible runtime overhead. The static policy could be good due to its non-existent overhead if the load is well spread out from the start, while the guided could be a compromise between the two.

The results show that the dynamic and guided scheduling policies are better than the static one by a fair margin, and between them the dynamic scheduling policy has the best performance overall. We think that this is due to the fact that the workload can often be not very well distributed from the beginning, since it depends on the dataset characteristics.

4.4 Measuring the steps of the algorithm

We wanted to understand how much time each part of our algorithm takes to be computed. Therefore, we measured the time needed for the following steps to execute:

- `read transactions`: read the dataset and compute the local support;
- `received global map`: reduce operation to get the global map to every process;
- `sorted local items`: sort the local portion of items;
- `received sorted global items`: reduce operation to get the sorted items to every process;
- `built local tree`: build the local FP Tree;
- `received global tree`: reduce operation to get the global FP Tree to every process.

In order to have an understanding of how much each of these parts impacted the final running time, we have measured these times in all our configurations. By looking at Figure 7b, it is immediately clear that when we have only one process, the part that takes the most time to execute is the `built local tree`. Instead, for the configuration with 16 processes and 8 threads as shown in Figure 8b, the most time consuming part is the reduce operation for merging the FP Trees, closely followed by `built local tree`. This is due to the fact that, despite having only `50000` items, the final tree can have hundreds of millions of nodes, due to the large number of transactions in the dataset.

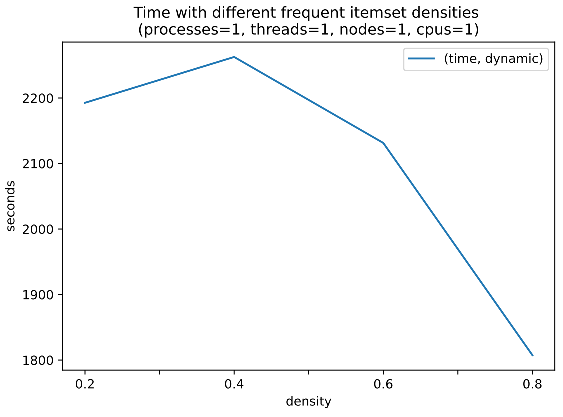
Moreover, when we have just 1 process, also the `read transaction` step is pretty heavy, due to the I/O required to read a 800MB file, as shown in Figure 7c. Instead, this step is not so impactful when we have more processes, as the I/O is splitted among many processes, as shown in Figures 8c and 8d. The other parts of the algorithm are almost negligible in comparison to the two sections that we just discussed, as we can see in Figures 7d and 8d.

5 Conclusions and further steps

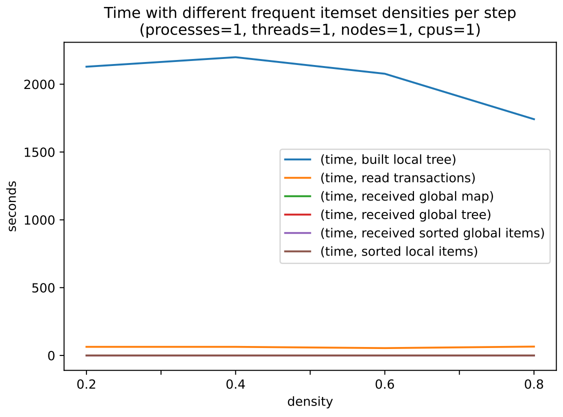
Up until this point, we have basically morphed the database of the transactions into a tree data structure that can be used to efficiently extract meaningful information from the dataset. The complete algorithm would continue with some more steps aimed at the creation of association rules between the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

However, the scope of our project was to implement the above descripted parallel algorithm, which already posed significant challenges, like distributed sorting and distributed merging of trees.

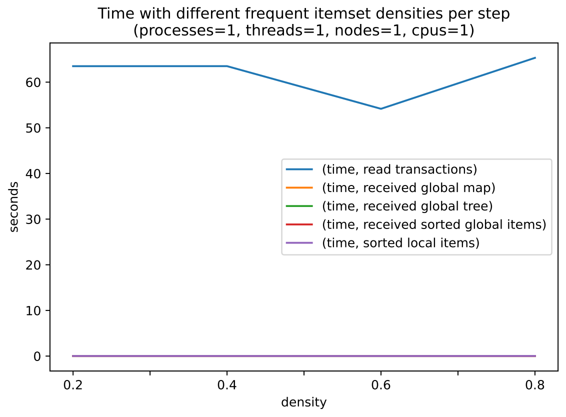
The parallelization of the FP Growth algorithm turned out to be very interesting to implement even if the problem is not perfectly parallelizable in itself, since some heavy operations still have to be run by a few processes. A possible improvement for the future would be to parallelize even the merging of two trees, by assigning different branches to different threads. To achieve this, our algorithm for FP Tree merging would have to be transformed from recursive to iterative.



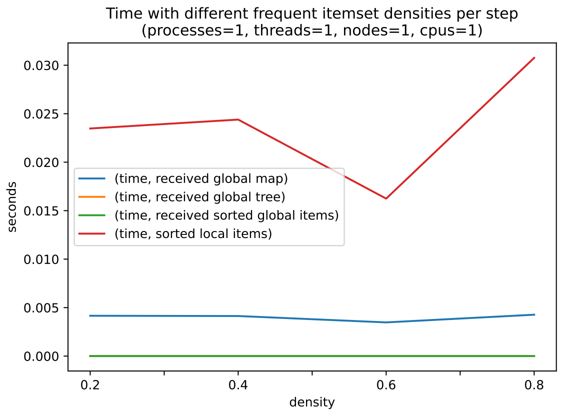
(a) Running time of different itemset densities



(b) Here we can see the dominance of the `built local tree`

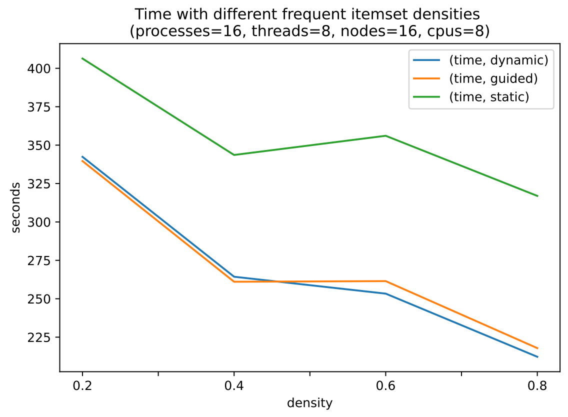


(c) Excluding the `built local tree` phase, the `read transaction` phase is the most expensive

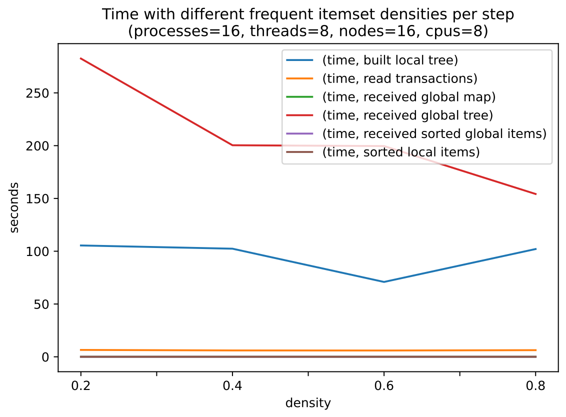


(d) Excluding `read transactions` and `built local tree`, we can see that the other parts of the algorithm take very little time

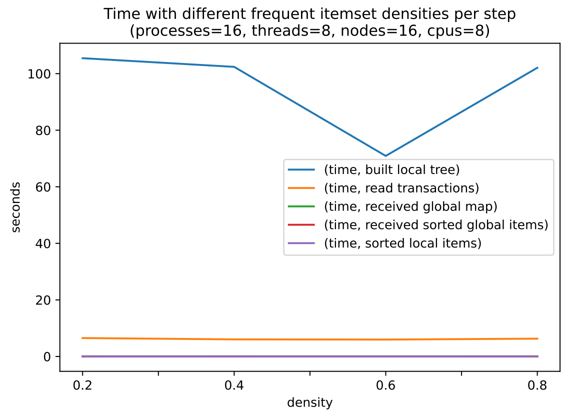
Figure 7: Measurements of running times of different steps with the serial algorithm



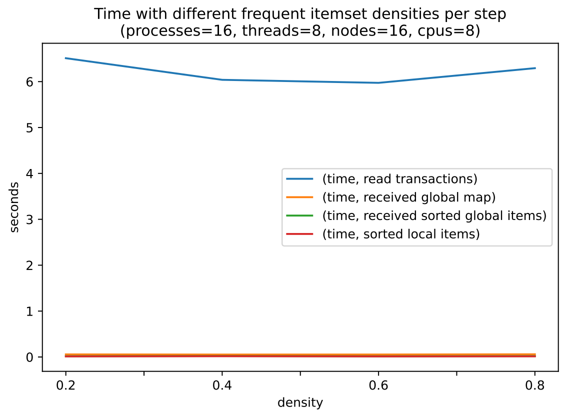
(a) Running time of different itemset densities



(b) With more than one process and one thread, the **received global tree** phase starts to be impactful



(c) Excluding the **received global tree** phase, the **built local tree** phase is the most expensive



(d) Excluding **received global tree** and **built local tree**, we can see that **read transactions** is the most expensive phase. However, when compared to 7c, we can see that the running time is much lower since the load is being spread on many processes

Figure 8: Measurements of running times of different steps with the parallel algorithm with 16 processes and 8 threads

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] G. L. Team, "Understanding (frequent pattern) fp growth algorithm." https://www.mygreatlearning.com/blog/understanding-fp-growth-algorithm/, 2020. Accessed: 2021-06-01.

[3] M. Süß and C. Leopold, "A user's experience with parallel sorting and openmp," in *Proceedings of the Sixth European Workshop on OpenMP-EWOMP'04*, pp. 23–38, Citeseer, 2004.

[4] J. Heaton, "Characteristics that favor freq-itemset algorithms." https://www.kaggle.com/jeffheaton/characteristics-that-favor-freqitemset-algorithms, 2016. Accessed: 2021-06-13.

[5] J. Heaton, "Comparing dataset characteristics that favor the apriori, eclat or fp-growth frequent itemset mining algorithms," *CoRR*, vol. abs/1701.09042, 2017.