Parallel FP-Growth

Iannino Paolo   La Quatra Moreno

Data Science Master

Course: High Performance Computing for Mathematical Models

Lecturer: Christophe Picard

Abstract

In this report we are going to present our work done for the course *High Performance Computing*. The standard algorithm has been parallelized using both *Open MP* and *MPI* to reach very good improvements with respect to the standard sequential version of the algorithm. Extensive tests have been performed in order to validate each choice in the parallelization of the algorithm. The source code of this project is available online at: `https://github.com/PaoloIannino/PFP-Growth`

Contents

1 Introduction

In this section will be illustrated the main algorithm chosen, the way it operates and its goals. Along with the base algorithm are also exposed the possibilities that algorithm offers in terms of parallelization describing the main idea and the reasons behind the parallelization done using the tools at our disposal.

1.1 The algorithm

The algorithm we choose to parallelize is the *Frequent pattern growth* algorithm, best known as *FP-Growth* [1], an association rule learning algorithm used to discover interesting relations between variables in large databases. The goal behind this class of algorithms relies on the interest to better understand the user behaviour in a particular context, for example, one of the most interesting case, is the one of the *Market basket*, where finding items bought together can give a valuable information for the business.

The reason behind our choice relies on the fact that, usually, this methods are applied on large databases that, containing transactions in the order of millions, will benefit from this parallelization.

Amazon case: In case of an internet store, such as Amazon, this kind of algorithm are very useful to find items that usually are bought together such that, an user can receive recommendation on the basis of his/her cart. It can be easily seen that the large number of transactions that the algorithm need to analyze can easily lead to unacceptable performances in the sequential implementation.

1.2 FP-Growth

The FP-Growth method uses a *divide-and-conquer* strategy to find frequent itemsets without requiring candidates generation (typically used by other methods, such as *apriori*). The main idea of the method is to use a special data structure, called *FP-tree* that stores the itemsets association information.

The algorithm follows the steps listed below:

- With an iteration over the full database, computes the support count of each item, sorts it and constructs the *header table* containing all the items with the associated frequency.
- Each transaction that need to be analyzed is sorted using the order obtained from the previous step.
- Constructs the *FP-Tree*.
- For each item in the header table, generates new itemsets using the frequent ones and the informations stored into the *FP-Tree*

[1]Original Paper: `https://www.cs.sfu.ca/~jpei/publications/sigmod00.pdf`

FP-Tree: The FP-Tree is constructed iteratively inserting ordered transactions (with respect to the support of each item present in it) in an unbalanced tree. The result after the processing of each transaction is a final tree that is used then to condition over each item and create the frequent sets of items.

1.3 Parallelization of the algorithm

The approach used for the parallelization of the algorithm has been linked to the weakness of the standard algorithm, above all:

- Long transactions (transactions with a large number of items each)
- Large datasets (dataset with a large number of transactions)

For each machine the parallelization has affected the number of transactions using the API exposed by *Open MP*. In this case using different threads is possible to reduce the time spent for the recursion required to analyze the database (details in the next section).

Having instead the possibility to use different machines, they communicate using *MPI* that offers scalability and high performances. In this case the parallelization has been done conditioning on the items present in the header table.

The next section details the work done for each of the technologies used and the type of architecture obtained.

2 Our Approach

During the parallelization of *FP-Growth* algorithm, have been evaluated different approaches to be able to apply the tools studied during the course and obtain significant improvements with respect to the standard *FP-Growth.* In the following sections are described the ideas behind the workflow of the parallelization of the standard algorithm.

2.1 Open MP

Open MP is an application program interface for multi-threaded parallel processing used on shared-memory multi-core computers. This technology offers the possibility to have parts of the code that are single-thread and others that are multi-threaded.

This tool is used to parallelize two main part of the algorithm:

- The sorting of each transaction retrieved from the database
- The recursion over the FP-tree to extract the conditional tree and the frequent itemsets

In the first case, the algorithm need the transactions to be sorted on the basis of the support of each item on the whole dataset. For this reason, this operation can be easily parallelized because there are not interferences between the sorting of different transactions.

The second part in which *OpenMP* is used is during the iterations over the FP-Tree. In this case for each item extracted from the header table, a recurrent pass is made over the FP-Tree extracting the support of each itemset. Doing so sequentially can require a lot of time, because for each entry in the header table (corresponding to all the items with support greater or equal to a given threshold) a full exploration of the tree need to be done. Having available multiple threads these explorations can be done in a concurrent way obtaining substantially better results (in terms of timing).

It is worth to notice that the parallelization done with *Open MP* is independent to the one done with *MPI*, indeed, also using different machines the parts of the algorithm described in this section can also be executed concurrently improving the time spent for each machine doing the required task.

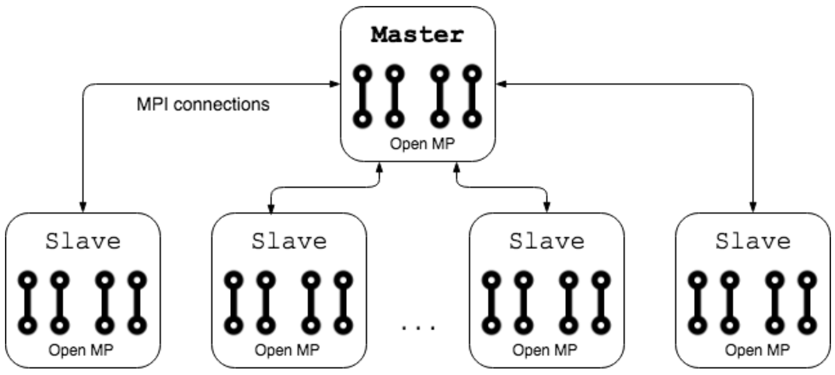
2.2 MPI

MPI (Message Passing Interface) is the specification of a message-passing parallel programming model. In this project, the implementation used is the one named *Open MPI* [2]. This tool offers the possibility to create a net of communicating computers that transfer informations and do calculation to achieve the final goal, in the case of the project, the extraction of frequent itemsets.

[2]High Performance Message Passing Library `https://www.open-mpi.org/`.

The architectural structure achieved can be deduced from the following image, where the connections between machines represent the messages exchanged using using *MPI* for the inter-machine parallelization and parallel flows for each machine represent the concurrent threads of *Open MP* :

Figure 1: *Open MP & MPI* final architecture



In the implementation of the project is possible to distinguish two different behaviours depending on the value assumed by the threshold (t):

- $t = 0$
- $t \geq 1$

The messages exchanged and also the computation done by different machines in these two cases differ. The parallel algorithm proceed with a *master/slave* architecture, where the master is designed to be the machine where the program is launched, and all the remaining ones act as slaves.

2.2.1 Case of Threshold zero

In this case the master machine splits the transactions present in the database equally for all the machines, sending to each of the slaves a subset of them. Each slave than proceed in the same way of before, using *Open MP* for the parallelization. In this case each slave runs the algorithm in the assigned subset and sends back the result to the master who finally merge them and produce the final results.

Under these circumstances the algorithm is following a map/reduce model with multiple mappers (the slaves) that map each itemset to a final value and a single reducer (the master) that merges the results obtained summing up the partial frequencies obtained.

2.2.2 Case of Threshold greater than zero

Having threshold bigger than zero, is not possible to proceed without the knowledge of the main, common *FP-Tree*. In this case in fact, the master makes the first recursion cycle producing the *FP-Tree* and the header table sending them to all the slaves. Each slave, in this case, runs the parallel algorithm (done also in this case with *Open MP* intra-machine) only on the assigned entries of the header table.

For the behaviour presented, is easy to understand that, in this case, the parallelization regards the different items and so the problem addressed here is crosswise related to the length of the transactions. Following this rule in fact, each machine focuses on different entries of the header table doing the recursion only in the conditional trees related to them. This reduces the final timing (scaling up almost linearly with the number of machines) supposing that in the database are present at least a number of items greater or equal to the number of machines used to parallelize.

3 Testing phase

During the testing phase an ablation approach have been used in order to evaluate both the techniques used to parallelize the algorithm. In this case, the tests have been done measuring the execution time of:

- Original sequential *FP-Growth*
- Parallel algorithm using only Open-MP (4 and 8 threads)
- Complete algorithm using Open-MP in conjunction with MPI (using 3, 5 or 9 machines).

These tests have been done over 2 datasets:

Table 1: Datasets details

Dataset	n. transactions	max n. items/transaction
D1	100000	10
D2	50	128

In this case they have been chosen to be of different nature to highlight the actual improvement over both datasets long in size or with a large number of items per transaction. The test on a dataset with both a large number of transactions and an high number of items each has not been possible with the resources at our disposal.

The computers used for to perform the test are all of the same type and are configured as follow:

Table 2: Test machines specifications

CPU	Intel® Core™ i5-3550 CPU - 3.30GHz - Quad-Core
RAM	16 Gb DDR3

To better describe the tests, is possible to recall that the parallel algorithms have different behaviours at different thresholds, for this reason where possible the algorithm has been tested setting this parameter to different values.

In order to be able to compare different tests the following table resume the results obtained for the sequential algorithm:

Table 3: Sequential algorithm detailed timing results

Dataset	Threshold	Execution time
D1	0	159.61 sec.
D1	1	108.25 sec.
D1	2	104.32 sec.
D2	1	156.60 sec.
D2	2	8.93 sec.

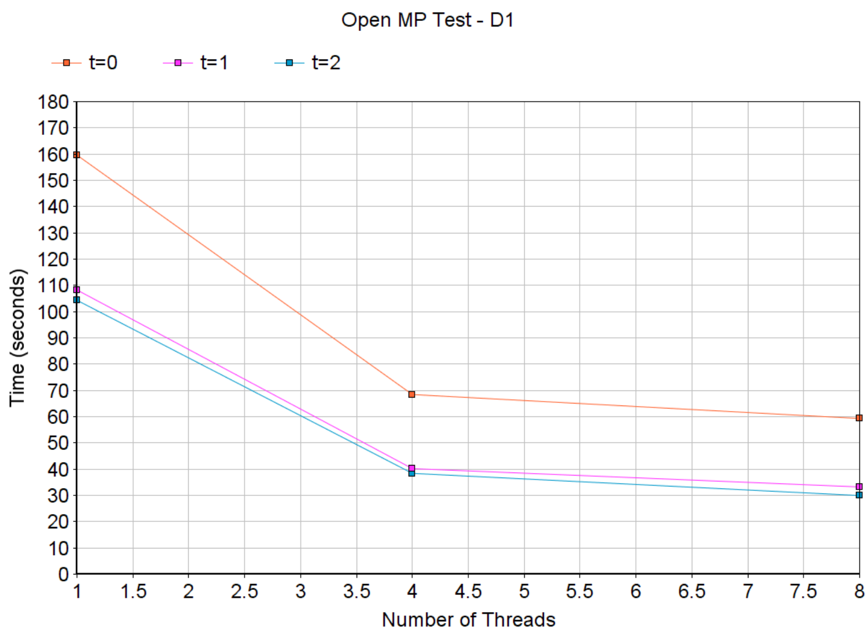
3.1 Open MP test

In this section are reported the results obtained parallelizing *FP-Growth* using only *Open MP* that offers the possibility of creating concurrent threads in a single machine. The following table and graph compare results for this version of the algorithm with the time measured for the sequential one:

Table 4: Open MP detailed timing results

Dataset	Threshold	Threads	Time	Improvement (%)
D1	0	4	68.42 sec.	57.13%
D1	0	8	59.29 sec.	62.85%
D1	1	4	40.20 sec.	62.86%
D1	1	8	33.22 sec.	69.31%
D1	2	4	38.39 sec.	63.19%
D1	2	8	29.94 sec.	71.29%

Figure 2: Open MP timing results plot (Dataset D1)



For this kind of tests the results have been retrieved on the first dataset described on the previous section. These results reveal that there is a strong improvement with respect to the sequential algorithm and the gap passing from 4 to 8 threads give further improvements, considering the execution on a *quad-core* machine.

It is possible to notice from the previous table that, the results are different with respect to the one theorically expected, in particular the decrease in time is not linear with the number of machines. This is related, not only to our algorithm, but also to the underlying hardware architecture of the machines. In this case introducing more threads can lead to an excessive fragmentation of the computation, given that in the real case not all the threads can be executed simultaneously.

3.2 MPI test

This section describe the complete test on the parallel algorithm, done on both datasets, using the configuration of *Open MP* with 8 threads and *MPI* with different number of machines. The following table and plots show the results obtained and the improvement over the baseline (sequential algorithm timing):

Table 5: Complete algorithm detailed test results (both datasets D1 and D2)

Dataset	Threshold	N. machines	Time	Improvement (%)
D1	0	3	41.47 sec.	74.01%
D1	0	5	44.64 sec.	72.03%
D1	1	3	24.96 sec.	76.94%
D1	1	5	20.63 sec.	80.94%
D1	2	3	23.95 sec.	77.04%
D1	2	5	20.27 sec.	80.56%
D2	1	3	30.38 sec.	80.60%
D2	1	5	29.98 sec.	80.85%
D2	1	9	28.26 sec.	81.95%
D2	2	3	1.23 sec.	86.22%
D2	2	5	0.93 sec.	89.59%
D2	2	9	0.71 sec.	92.04%

Figure 3: Timing results plot for the Complete Test on Dataset D1. Single machine sequential, multiple machines MPI + Open MP (8 threads)

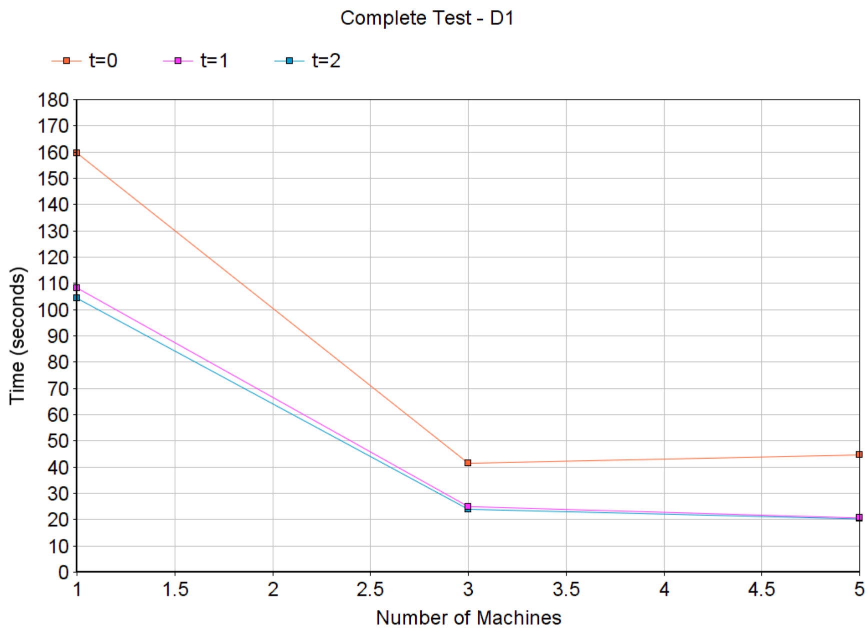
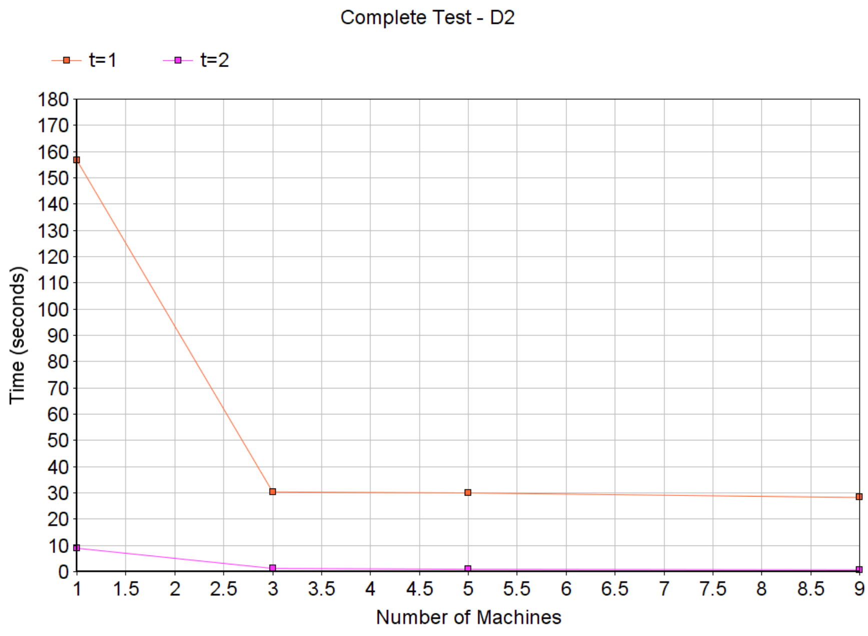


Figure 4: Timing results plot for the Complete Test on Dataset D2. Single machine sequential, multiple machines MPI + Open MP (8 threads)



It is noticeable from these results that is possible to reach further improvements not only with respect to the sequential algorithm, but also with respect to the results obtained using only *Open MP*.

In this case, however, can be seen that, even if the results obtained are really good, the timing reported passing from 3 to 5 or 9 machines (above all in the second dataset *D2*) are not scaling up linearly with the number of machines, this can be due to the relative easy task and the computational requirements that are not so heavy to require a large number of machines; the overhead introduced to share data and results between machines compared to the computational charge may impede the linear scaling of performances.

4 Conclusion

This project has been done in the scope of the *High Performance Computing for Mathematical Models* course. The subject of this algorithm has been chosen because of the interest supposed on this topic. Having the fast-growing internet marketplaces, the result of the analysis of *FP-Growth* are very important for the business and, given the dimension of the databases that is growing together with the number of people using internet shops, the parallelization can bring a lot of benefits.

The results obtained in the testing phase show that the parallelization done has been effective with both the tool used. Using only *Open MP* we reached a timing improvement around 60/70% depending on the number of threads used, showing that the parallelization has been useful. The tests using both *Open MP* and *MPI* instead show saving in time around 72/92%. In this case is possible to conclude that is possible to improve the results of single machine (with only *Open MP*) even if the scaling is not linear with the number of machines proving that the master/slave architecture can be further improved.

We can affirm however that the purposes set at the beginning of the project have been attended, having created a version of the algorithm that can reduce significantly the timing performances of the sequential one both on a single machine and on multiple machines. During the project we learned how to *"think parallel"* to identify possible patterns in the algorithm creating a bottleneck in a sequential algorithm that can be reduced if parallelized in the right way. The project has also been interesting to understand how the parallelization can be done on the same machine or on multiple ones using message passing interfaces.

4.1 Further improvements

Due to timing constraint and the scope of the course some possible improvements have been evaluated but has not been possible to implement them. In particular the project can be further improved considering the following analysis:

In the situation of threshold $t = 0$ is possible to extend the map/reduce paradigm creating more reducers that are executed simultaneously to merge the values computed by the mappers. This can lead to better performances and improved efficiency of the algorithm.

Another consideration can be related to the load balancing. A work in this direction has already been done dividing the entries of the ordered header table among different machines with respect with their position in the table. Other parameters can be also considered to achieve better results for example can be taken into account also the relative frequency of each item for each transaction.