Parallelization of a collaborative filtering algorithm with Menthor

Semester Project
Final Report

Louis Bliss, Olivier Blanvillain
louis.bliss@epfl.ch, olivier.blanvillain@epfl.ch

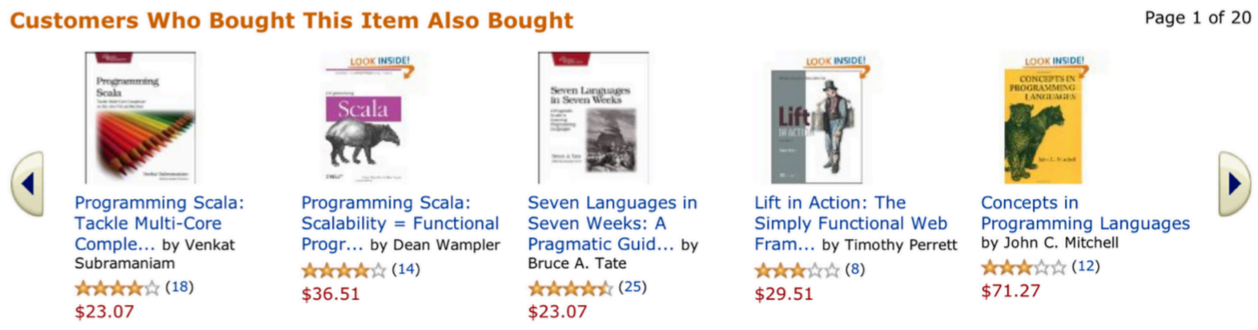
1. Summary

2. Introduction

Collaborative filtering applications are widely used on most websites which offer personalized contents: social network, e-commerce, movie databases, dating sites. These applications recommend items (the “filtering” part), usually with marketing interests, but with the specificity that these recommendations are personalized for the specific tastes of the user. They are made based on a database of ratings from a large amount of clients (the “collaborative” part) and the ratings of the user.

The category of algorithms behind the recommender we have designed further on, uses the concept of similarity, which can be either user-based or item-based: we respectively compute the similarities between users and make recommendations to the client based on items liked by other users similar to him, or compute the similarities between items and recommend items similar to those the client likes. The second item-based approach is used.

An very simple example of recommender:



Customers Who Bought This Item Also Bought

Page 1 of 20

Programming Scala: Tackle Multi-Core Comple... by Venkat Subramaniam (18) $23.07

Programming Scala: Scalability = Functional Progr... by Dean Wampler (14) $36.51

Seven Languages in Seven Weeks: A Pragmatic Guid... by Bruce A. Tate (25) $23.07

Lift in Action: The Simply Functional Web Fram... by Timothy Perrett (8) $29.51

Concepts in Programming Languages by John C. Mitchell (12) $71.27

Very large datasets are typically used as database for this kind of application. Consequently the computations involved are quite important: distributing and parallelizing to dispatch the data and drastically improve the performances is therefore of great interest. According to this the utilization of a framework such as Menthor which implements these two mechanisms seems a good idea. We have focused our project on parallelization, implying that the splitting of a specific dataset for a potential distributed version was not covered.

An important part of the project is about performance, especially the scaling with the number of cores. The analysis of these benchmarks gives information on how well our implementation was done and some possible issues with it, and ideally about the framework itself.

Other important points which will be reviewed are how this specific algorithm suits Menthor, some superficial comparisons with other existing implementations and more generally, the use of our application as an example to show the Menthor’s mechanisms.

3. Framework

Menthor[1] is a framework developed by Heather Miller and Dr Philipp Haller at the EPFL in Scala. The basic mechanisms are similar to the ones of the traditional map-reduce paradigm, but with the important difference that the whole dataset is represented as a graph: its vertices locally execute some procedures and communicate over the edges by messages.

The algorithm is split into synchronized steps of two kinds:
- *Update*[2], which is comparable to map: each vertex has a list of possible incoming messages (from the precedent step), executes locally the update function code and finally sends some messages.
- *Crunch*, similar to reduce: a reduce operation is applied on the whole graph, with the specificity that the result is available in every vertex in the next step.

More information on Menthor can be found in the author's publication[3].

[1] Project page: http://lcavwww.epfl.ch/~hmiller/menthor/
[2] Is called "then" in the code, which is a little bit confusing.

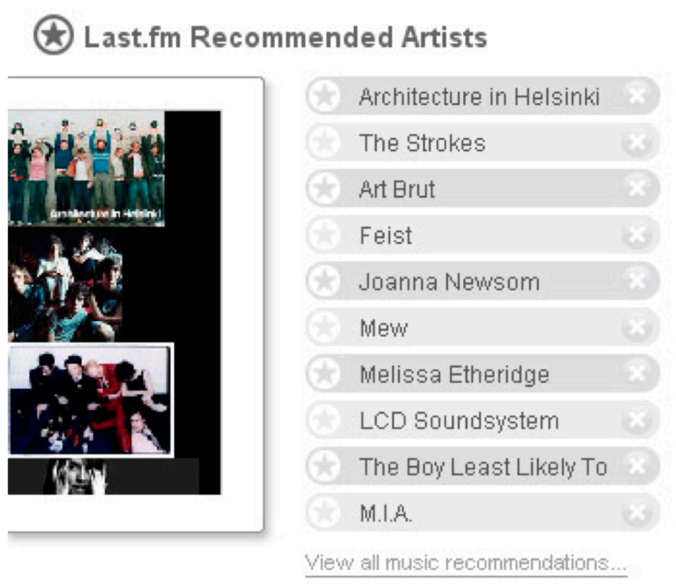
[3] Philipp Haller and Heather Miller. *Parallelizing Machine Learning– Functionally: A Framework and Abstractions for Parallel Graph Processing*.

4. Algorithm

We used the algorithm described in Sarwar et al.: *Item-Based Collaborative Filtering Recommendation Algorithms*[4]. The main goal is to compute for every user a “top-k” recommendations list containing new items that the user will theoretically appreciate the most.

The database is a dataset containing “ratings”. A rating consist of a 3-tuple: a user ID, an item ID and a grade. These are the definitions of “rating” and “grade” that will be used further on. The grade could be as simple as a 0 or 1 boolean value for “has bought the item”, or an actual grade measuring how much the client liked the item.

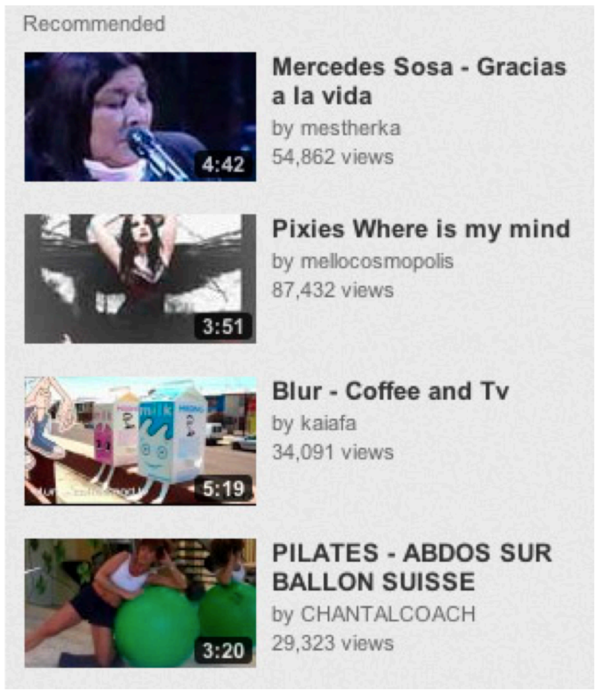
Two examples of recommendations based on collaborative filtering:



Last.fm Recommended Artists

- Architecture in Helsinki
- The Strokes
- Art Brut
- Feist
- Joanna Newsom
- Mew
- Melissa Etheridge
- LCD Soundsystem
- The Boy Least Likely To
- M.I.A.

View all music recommendations...



Recommended

Mercedes Sosa - Gracias a la vida
by mestherka
54,862 views
4:42

Pixies Where is my mind
by mellocosmopolis
87,432 views
3:51

Blur - Coffee and Tv
by kaiafa
34,091 views
5:19

PILATES - ABDOS SUR BALLON SUISSE
by CHANTALCOACH
29,323 views
3:20

4.1. Abstract

The item-based recommender algorithm consists of two main computations: the similarity for each pair of items and the predictions for each user.

[4] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. *Item-based collaborative filtering recommendation algorithms*. In Proceedings of the 10th international conference on World Wide Web which took place in may 2001 (WWW ‘01).

4.1.1. Computing the similarity

There are different ways of computing the similarity. The cosine-based similarity works with the items as vectors of grades, where the coordinate *i* of the vector corresponds to the grade given by the user *i*. The similarity between two items is equal to the cosine of the angle between their respective grade vectors. This value can be easily derived from the dot product:

$$\vec{a} \cdot \vec{b} = ||\vec{a}|| \; ||\vec{b}|| \cos\theta \Rightarrow S(a,b) = \frac{\sum a_i b_i}{\sqrt{\sum {a_i}^2}\sqrt{\sum {b_i}^2}}$$

A slightly modified version of this formula, the adjusted cosine similarity, produces more accurate results[5]. An issue not taken into account with the basic cosine similarity is that users might grade with different bias: some could give 1/10 to a movie they didn't liked, and 6/10 to a movie they liked and another might instead give 5/10 and 10/10 in the same situation. To compensate these variations and to give the same average grade to each user, the adjusted cosine similarity formula weights the grades by the user's average:

$$AdjustedCosineSim(a,b) = \frac{\sum_{u \in U} (R_{u,a} - \overline{R_u})(R_{u,b} - \overline{R_u})}{\sqrt{\sum_{u \in U} (R_{u,a} - \overline{R_u})^2}\sqrt{\sum_{u \in U} (R_{u,b} - \overline{R_u})^2}}$$

Where $R_{u,a}$ denotes the grade given by the user *u* to the item *a*, $\underline{R}_u$ the average grade of the user *u*, and *U* the set of all the users which graded both *a* and *b*. Our implementation uses this formula.

4.1.2. Computing the prediction

Once the similarity of each pair of items is computed, the algorithm generates the recommendation list for each user. This is done by looking at all the items a user has rated, selecting new items similar to them and predicting the grade that our user would give to each of these new items. Several ways can be used to compute this prediction: the method we chose is simply a weighted sum:

$$Prediction(u,i) = \frac{\sum_{N \in items\ similar\ to\ i} Sim(i,N) R_{u,N}}{\sum_{N \in items\ similar\ to\ i} |Sim(i,N)|}$$

[5] See section 4.3.1 Effect of Similarity Algorithms in the referenced paper.

This formula models the predictions under the assumption that the users interest towards an item is a linear function on how much they like similar items.

4.2. Complexity

The similarity has to be computed for every pair of items and the adjusted cosine similarity formula for a specific pair *(a, b)* goes through all the users which rated both *a* and *b*, therefore, overall, the similarities computation runs in *O(#Items * #Items * #Users)*.

The prediction has the same complexity: for each user all the items not graded have to be considered, and then the computation of the predictions runs through all the items graded by the user, therefore having an *O(#Items * #Items * #Users)*.

This leads to an overall complexity of O(n^3). in the size of the dataset (assuming that the number of items and users scale linearly with the dataset). This algorithm works far better when the number of items is small compared to the number of users.

However the real complexity is much less than this worst case scenario, mainly since the ratings matrix is very sparse and intelligent implementations might take advantage of it.

One important characteristic is the possibility of giving relevant results (though less precise) using only a subset of the users (or of the ratings). Therefore the computation of the recommendations could be done for a sole user using a precomputed similarity table. This is correct because the similarity between two items tends to show a stable value. For instance, in the case of a movie recommendation systems, if two movies are considered very similar based on a sufficient amount of ratings the chances are high that they stay that way.

5. Implementations

5.1. Hadoop-Mahout

We started off by looking at another existing implementation[6] of an item-based recommender, part of the open source Apache Mahout library. Mahout is built on top of the Apache Hadoop framework and uses the map/reduce paradigm.

The entire execution consists of matrices manipulations which are accessed via the Hadoop distributed file system. The algorithm starts by loading the dataset and performing a multitude of transformations (maps and reduces) in order to prepare the data for later use.
A simplified description of each operation of the mahout implementation is listed below.

Part 1 is the creation preference matrix:

a) Job itemIDIndex
  Map and reduce: Converts the dataset ItemID to an internal index.
b) Job toUserVectors
  Map: Creates (UserID, (ItemID, Preference)) pairs
  Reduce: Groups them as (UserID, List((ItemID, Preference)))
c) Job toItemVectors
  Map: Creates the vector array[ItemID] := Preference
  Reduce: Merges all the vectors in a matrix

Part 2 is the computation of the matrix of similarities:

a) Job normsAndTranspose
  Map: Computes the euclidian norm of each item vector.
  Reduce: Sums the norms
b) Job pairwiseSimilarity
  Map and Reduce: Computes for every pair the similarity using the vectors and the norm.
c) Job asMatrix
  Map: Takes the similarities for each pair.
  Reduce: Merges the similarity values in a matrix.

[6] Project page: https://cwiki.apache.org/confluence/display/MAHOUT/Itembased+Collaborative+Filtering

Part 3 is the use of the similarities matrix to predict the recommendations.

a) Job prePartialMultiply1, prePartialMultiply2, partialMultiply
   Map: Maps a row of the similarity matrix to a VectorOrPrefWritable, a type that can hold either a similarity vector or a preference.
   Map: Filters the data according to the set of usersToRecommendFor.
   Also removes non significative preferences.
   Reduce: Merges VectorOrPrefs to VectorAndPrefs.

b) Job itemFiltering
   Map: For each user takes his preferences and the similarities needed in the predictions.
   Reduce: Applies the prediction formula.

c) Job aggregateAndRecommend
   Map and reduce: Aggregates the results of the last phase and return top-k recommendation for each user.

Every step takes for input the result of the previous one. These accesses are made using Mahoot distributed file system and therefore are done accessing the hard drive, the opposite of Menthor where everything is stored in the RAM memory. This approach allows the data structure to be changed from one step to another, what is not achievable with Menthor. However the obvious cost is the time spent to load the input and write the output for every single step.

The actual code of Mahout is quite “cryptic” (as in “hardly readable at all”). The most unfortunate parts are the general complexity of the Hadoop framework and the use of a lot a internal tools inside Mahout; the given impression is that the code spreads out in all directions making the understanding of what is actually going on very awkward.

The lack of expressiveness is also caused by the Java language itself which doesn't allow things to be expressed as clearly as they could be. Functional code for a “map-reduce” based program would be a lot more readable.

To illustrate this, an example of the class that defines a “map” operation (3.a, one of the simplest):

```
package org.apache.mahout.cf.taste.hadoop.item; import ...
public final class SimilarityMatrixRowWrapperMapper extends
    Mapper<IntWritable,VectorWritable,VarIntWritable,VectorOrPrefWritable> {
  @Override
  protected void map(
      IntWritable key,
      VectorWritable value,
      Context context) throws IOException, InterruptedException {
    Vector similarityMatrixRow = value.get();
    /* remove self similarity */
    similarityMatrixRow.set(key.get(), Double.NaN);
    context.write(new VarIntWritable(key.get()), new
VectorOrPrefWritable(similarityMatrixRow));
} }
```

And the code from the main calling this “function” used by the map:

```
Job prePartialMultiply1 = prepareJob(
  similarityMatrixPath,
  prePartialMultiplyPath1,
  SequenceFileInputFormat.class,
  SimilarityMatrixRowWrapperMapper.class,
  VarIntWritable.class,
  VectorOrPrefWritable.class,
  Reducer.class,
  VarIntWritable.class,
  VectorOrPrefWritable.class,
  SequenceFileOutputFormat.class);
prePartialMultiply1.waitForCompletion(true);
```

In addition to the parallelization, Mahout implements a lots of small optimizations and options. Most of them consist of thresholds used to drop unsignificative data and, as consequence, decrease the amount of computation. However some really lower the code readability, for instance the use of *null* or *NaN* as non-error values. Incidentally one of the few comments[7] in the project warned the reader about a “// dirty trick” (without further information), which we never completely understood...

Because of the Mahout complexity, the amount of optimizations and fundamental differences in the data structure, we decided to disregard this code and to start from scratch our own implementation using only the publication we used as reference.

5.2. Menthor

In this section you will find a detailed description of our implementation on top of the Menthor framework. Our project source code[8] is hosted on git as a fork of the menthor project.

5.2.1. Data structure

First, it is of note to mention that a Menthor-based application won’t have the same structure at all as another map-reduce implementation based on Mahoot for one important reason: our subjacent data structure is a static graph and consequently not mutable. Therefore a major part of our work consisted of finding a suitable graph model for the Menthor framework. The vertices require to be as self-sufficient as possible to minimise the amount of overhead due to communication and to realize most of the computation vertex locally.

Any solution will need at least one vertex per user in order to compute the average grade, the predictions and the list of recommendations. The pairwise similarity part will have to be done in vertex which somehow represents the items. Our first approach consisted of having one node

[7] org.apache.mahout.math.hadoop.similarity.cooccurrence.RowSimilarityJob.java, line 206

[8] https://github.com/LBliss/menthor/tree/recommender. In addition to the code you will also find the datasets we used for benchmarking and the alternative implementation that used the java API.

per pair of item and containing the appropriate information to compute the similarity. This is the most intuitive choice when looking at the Adjusted Cosine Sim expression but unfortunately leads to a graph with a terrifying amount of vertices: having *#Items*^{2} nodes occupy a lot of memory and the amount of messages needed to dispatch the ratings was very problematic.

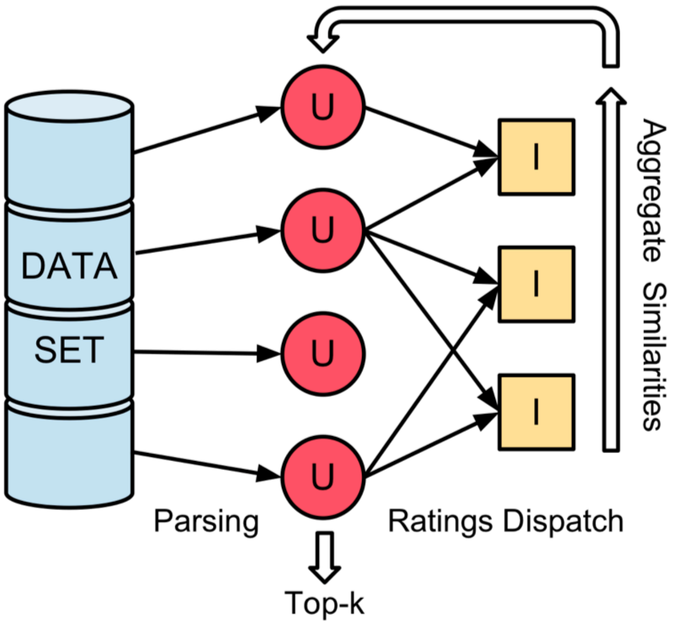
Our second and more conclusive graph structure was to use one node per item. Each user sends to the items all the ratings they need to realize their computation. Having one node per item also implies that each similarity is calculated twice. A simple strategy was needed to decide in which item the computation would be done.

A last point we had to deal with is an issue related to the data structure: the Menthor framework requires that all nodes and messages carry objects of the same type. Since they are two kinds of nodes which require back and forth communication with different type of information we used an abstract *DataWrapper* case class. Carried objects are subclass of the *DataWrapper* class and simple pattern matching is used to retrieve their content.

5.2.2. Data flow

Having the basic structure we need to figure out how the data will flow through the graph.

The entry point of the algorithm is the dataset parsing. The datasets we used were sorted having all the ratings of a user grouped together. The graph is initialized with all the ratings stored in their respective user vertex, and all the items nodes are empty. Afterwards the users will need to transfer appropriately their ratings to the items, in addition to their average grade which is locally computed. With that information the items will be able to compute the similarities and communicate the results with the users. The similarities are “crunched” in a hashmap which will be available in the next step for the users. Finally each user calculates his own top-k recommendations.



5.2.3. Execution flow

According to the data flow our program contains several steps which are executed sequentially as we need the results of the previous one in order to carry on. Menthor has two kinds of step: *update* or *crunch*.

1) The first step is an update which operates only in the user vertices and contains two operations:
- Firstly the average grade of the user is computed (we'll use it later to weight the grades), this average is simply calculated by the sum of all the grades the user gave divided by the number of grades.
- Secondly we dispatch the ratings to the item vertices. To do so, we use the mechanism of messages: each user node prepares a list of mesgrade. sages with an item vertex as destination containing some ratings and the user average These ratings will be used by the recipient to compute its pairwise similarity with other items and therefore will be needed only if the user rated it (i.e. if user12 didn't grade item5, item5 won't need user12's ratings to compute its similarity with other items). To further reduce the quantity of data transfer and the amount of computation, only one item of each pair will compute the similarity. The strategy used is that the item with the smaller ID will compute it (i.e. the calculation of the similarity between item8 and item13 will be done only in the item8 vertex).

2) Step two takes place in the item vertices, it's an update which will compute the pairwise similarities. It consists of two phases:
Part one is the treatment of all incoming messages. These messages contain the ratings given by the users and must be temporarily stored in different local sets and maps.
The second part is the actual computation of all the pairwise similarities including the current item and other ones with higher ID. We simply apply the adjusted cosine formula using the data we received. All the similarities are finally stored in the vertex for the next step.

3) The third step is a crunch step: the similarities of every item vertex are aggregated in a map which will later be used to predict the recommendations.

4) The forth and last step is an update in the user vertices consisting of the creation of the recommendations. Using the personal ratings of the user and the global map of similarities, we apply the weighted sum formula and take the top-k of the most similar items (excluding the ones already graded by the user).

5.2.4. Optional thresholds

In order to accelerate the process we included two optional optimizations which trade precision for speed:

The algorithm predict the recommendations by finding items similar to the items the user liked. One can set the number of items which will be used for finding the similar ones instead of taking every single one the user graded. As one with a higher grade will have a stronger influence on the prediction than one with a poor grade (due to the weighted sum used for prediction), it makes sense to take only some items by descending order of the grades.

Pair of items with low similarity value won't have much influence on the final output. To improve performance one can set a threshold for a minimal similarity (similarities below will be ignored).

Two other thresholds are also available which can be used to make the final output safer and more stable.

One can choose to recommend only items that are at least similar to a certain number of items the user already graded. This threshold avoids recommending an item that is by chance very similar to a very small number of items the users like (therefore having their prediction based on a small number of similarities, typically one) instead of items that are a little bit less similar, but based on many similarities (and therefore a safer recommendation).

The similarity value is computed from the users ratings, but is sometimes based on only a few ratings. One can set a minimum number of ratings that a pair of items must commonly possess to have their similarity computed (else the pair won't have a similarity value and will be later ignored).

However, as our project is not really on the performance of our specific implementation but more on the use and the scaling offered by Menthor, we did not extensively test these different thresholds. They must be considered as tools, as options and are very dependent on the dataset (a very sparse dataset will behave very differently than a dense one).

5.2.5. Correctness

Three different solutions have been used to check the correctness of our implementation.

We randomly generated a dataset with a special characteristic. Every user tends to like an item for which the ID is equivalent *mod m* with the user ID, and dislike the other ones. The consequences are that items with equivalent ID (*mod m*) will have a high similarity value and therefore users only get recommended items from the same congruence class.

The second check was to manually compute the similarities and the recommendation with a very small dataset. Obtaining the exact same values for the similarities and the predictions ensure the correctness of the calculations and of the use of the formulas.

Our ultimate test consisted of comparing the final output with a large dataset against another implementation of the algorithm. The agreement of the results convinced us that our implementation was correct.

5.3. Java API

In addition to the Mahout implementation and the one we built as the core of our project, we have a third implementation in Java. We were very lucky as we were asked to realize a similar project as part of our Software Engineering course[9]. However we had to adapt it in order to obtain the same version of the algorithm. The reference paper was exactly the same one we used but some details about the algorithm were "freely interpreted". The input format was also quite different.

We extended the project with a parallel version using the tools provided by the Java API. First we substituted each data structure with it's concurrent alternative, *HashMap* became *ConcurrentHashMap* and *new ArrayList()* became *Collections.synchronizedList(new ArrayList())*. Afterwards the instructions of the two main loops were wrapped and executed by a *ThreadPoolExecutor*.

6. Benchmark

6.1. Procedure

The benchmarking has been realised on *mtcquad* EPFL server, with 8 physical cores and no hyper-threading. We were interested in two different aspects: the raw performances and the scalability.

The set up of the necessary environment to run the Mahout implementation was unfortunate and almost impossible on the benchmark server. Furthermore the execution in itself is very complex because of all the options to set, the obfuscated way to actually run it as well as to provide the input, and it leads us to give up the benchmarking with this implementation. Therefore we only performed benchmarks with the Menthor and the Java ones.

We prepared several datasets for benchmarking, and chose two of them based on test results

[9] Course page: http://sweng.epfl.ch/

and duration (both have grades between 0 and 9):
- The first one contains 350’000 ratings, 5000 different users and 250 items.
- The second one has 1’000’000 ratings, 25’000 users and 1000 items.

We performed 11 runs for each dataset, every run consisting of seven benchmarks using from 1 to 7 cores. We kept one core inactive to limit the influence of system and external solicitations.

With the small dataset the variance of the measures between different runs is quite high, the difference between the lowest and the highest values being around 15% and increasing up to 25% with the number of cores (the more cores are used, the larger the variance is). This is the reason why we took the median to plot the graphs, excluding the “abnormal” measures. This variability could indicate a high sensibility to external events and to internal thread schedule and load distribution, which obviously is more sensitive when a higher number of cores are active.

With the bigger dataset the variance is less problematic, the difference between measurements rarely exceed 10% (the biggest is 14%) and are overall quite small.

6.2. Raw Performances

The raw performances analysis provides information on how well our implementation is written and on the cost induced by the use of Menthor.

6.2.1. 350k ratings dataset

With the 350k dataset, the Java implementation terminates in about 60 seconds when using one core, while the Menthor took 155 seconds. With 7 cores enabled the ratio of these two remains approximately the same, 11 seconds for Java and 25 for Menthor. Independently of the number of cores used, the Java implementation performs about twice faster than ours.
The Java implementation is clearly more efficient in terms of memory: a matrix consumes less memory than a graph representation which induces a lot of overhead. With this small dataset, Java uses 0.5GB and Menthor 1.1GB.

The main explanation for the disparity of execution times resides in the intricate architecture we had to design to fit the algorithm in Menthor graph structure. Important overhead results from the use of messages to send the appropriate ratings from the users to the items. It actually accounts for 2/3 of the time spent in phase n°2 and thus for more than 1/4 of the total time. This overhead represents the time spent by the framework from creation of the messages to their arrival in the destination nodes. You will find more information on this in section 6.4.

6.2.2. 1000k ratings dataset

For a dataset 3 times larger, the running time of our implementation is 1150 seconds using one core and 270 seconds with 7 cores. The time required is about 8 times more than for our first data set, which is quite reasonable considering the worst-case complexity of our algorithm.

The comparative results with this 1000k dataset are quite different. The memory usage still favors Java (2.6GB against 4.8GB), but the Menthor implementation is this time faster. With one core Java is 3 times slower (3900 seconds), but with 7 cores this factor is decreases to 2 (630 seconds).

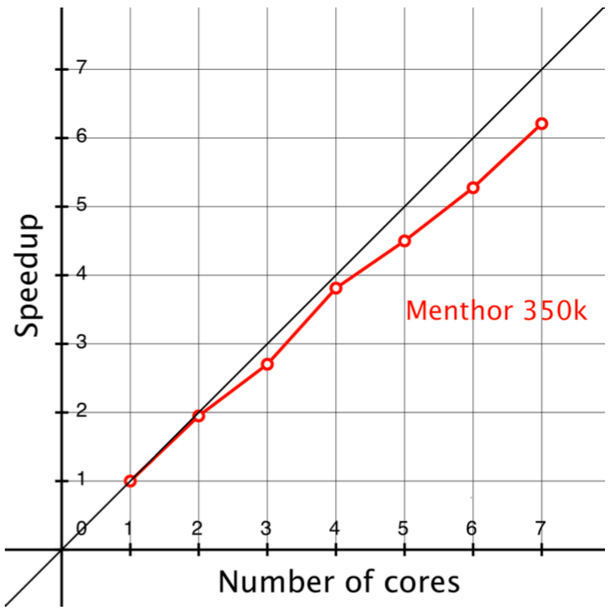
The Menthor implementation scales better with the dataset size because it takes advantage of the fact that the matrix containing the ratings is very sparse (we only send the minimal amount of ratings to the items nodes). The Java one, more straightforward, runs each computation through the entire vector of grades even if it contains mostly zeros.

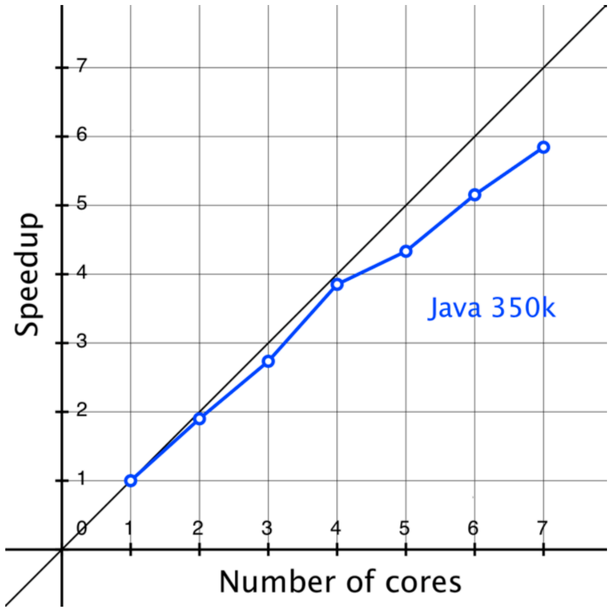
On a broader outlook no definitive statements can be made based solely on this short analysis, mostly because the performance really depends on the implementation and because comparing two designs with different languages, strategies and data structures, is almost impossible to achieve without doing it in a superficial manner.

6.3. General Speedup

The real point of interest of this whole benchmark resides in the scalability. The analysis will show the possible bottlenecks and the effectiveness of the parallelization of our specific implementation offered by the framework.

Speedup graphs of the Menthor and the Java implementations on the 350k dataset.

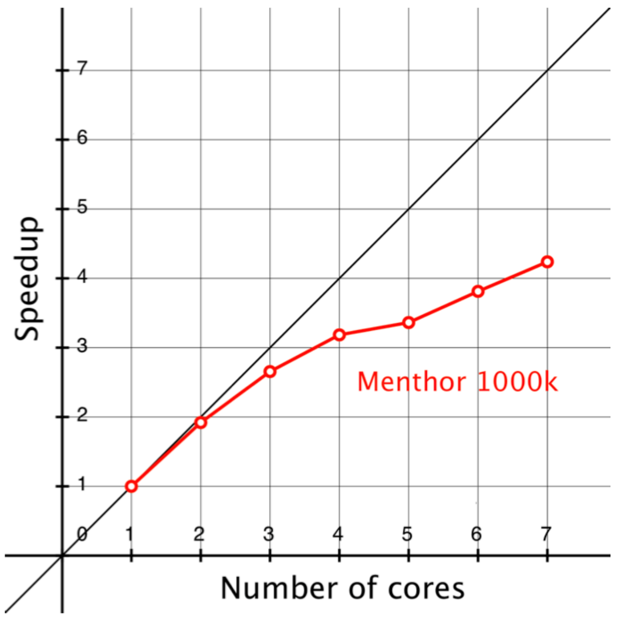


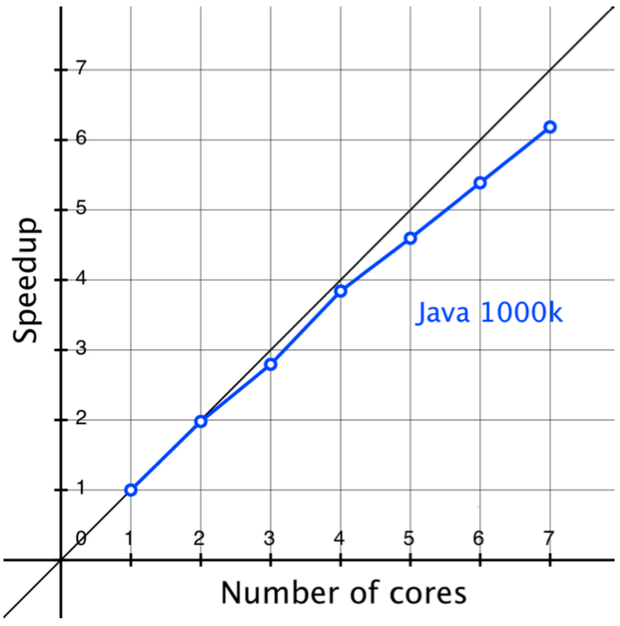


We can see an almost perfect scalability for both implementations. Menthor seems to behave a little bit better with a higher number of cores, but it could also simply be by chance (according to the high variance). To figure this out a server with more cores would be needed.

These speedups shows no bottleneck as the scalability is very good. However as the simpler Java API tools works almost as well as Menthor, the use of a library for this particular case may not really be necessary.

Speedup graphs of the Menthor and the Java implementations on the 1000k dataset.



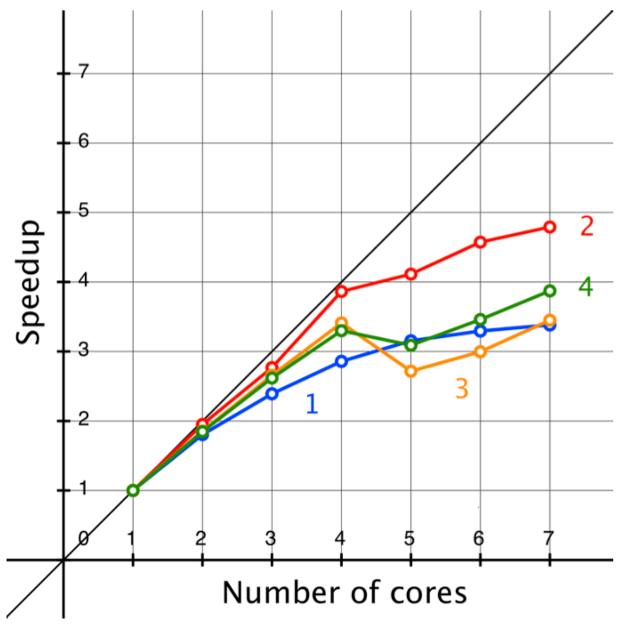


However, the general situation using the 1000k drastically change. The speedup with Menthor is far less effective and this encouraged us to perform a more detailed benchmark in order to understand what prevents our implementation to scale correctly. On the other hand, the Java application has no problem related to the scalability, although, as seen in the section about raw performances, the running time is actually higher.

6.4. Detailed Speedup

In order to realize more detailed benchmarking, we inserted lines of code in the specific places responsible for step transitions inside the Menthor source code to calculate and print timestamp (using basic *System.currentTimeMillis*) from which interesting outputs resulted.

Speedup graphs for the 4 steps of Menthor implementation using the 1000k dataset.



We used this to figure out the load distribution between the steps and their respective scalability. The distribution of the running time between the four steps is roughly 2:50:1:65. Consequently the speedup curves for the steps 1 and 3 are insignificant regarding to the global speedup. They also represent very short (time-wise) steps and are potentially more influenced by events that could happened simultaneously and by the framework in itself. This benchmark also shows that none of the steps are a unique bottleneck. All contribute (although more or less) to the decrease effectiveness of speedup that we observed with the 1000k dataset.

The running time with 5 cores is very intriguing, a drop in performance is repeatedly measured there, and the scaling afterwards remains far less efficient. We didn't find an explanation for this strange event. A similar curiosity in all graphs is a slight performance loss when odd numbers of cores are used. The reason is unknown, but it could hypothetically be caused by either the hardware, the OS or the framework.

With the detailed timing we were able to calculate the overhead induced by message handling in step 2, which takes the surprisingly amount of 66% (350k) and 90% (1000k) of the time spent in the second step. This overhead was hidden in our comparison with the Java implementation by the fact that this message handling also scales.

The most important explanation revealed by our benchmarks is that the different threads tend to complete their task with substantial delay between one another. This phenomena grows with the number of cores up to more than 20%.

For instance with 7 cores run on the 1000k dataset the first core idled at the end of the second step after 81.605 seconds of usage, whereas the last core only finished after 99.884 seconds. This means that the faster core was idling for about 18% of the time dedicated to this step. In this example a performance increase of 13% would have been realized if all the threads would have finished at the same time.

This issue is related to the data dispatch when sending the ratings to the items. Some item vertices have far more similarities to compute than others. In addition to the inherent heterogeneity of the dataset, the strategy used (only one item of each pair performs the computation) also increases the load disparity between the nodes. A more refined way to accomplish the load distribution would speed up this important slowing down, but a thorough analysis of the dataset would be required to produce a well balanced distribution, as the effective load of an item can not be trivially estimated.

As the 350k dataset is more dense than the 1000k, and contain far fewer items, this issue is much less pronounced.

Apart from there they might have been other factors contributing to the non-scalability of our implementation, the overuse of messages is certainly an instance of them.

7. Conclusion

The framework offers great predispositions to scaling. However certain points must be known in order to obtain efficiency.

First of all the use of messages should remain reasonable, and it consequently means that the algorithm topography shouldn't change too much: the data structure as well as the data distribution in the graph should remain as static as possible in order to achieve good performances.

Secondly if the application has different loads on the vertices (e.g. they have more or less work to do, or realise different jobs), especially if you have nodes of different nature (as we do), the distribution is extremely important. If this load factor cannot be known in advance or only approximately, the scaling will be far less efficient.

As the distribution of the nodes to the different cores is static, external solicitation of a core used by Menthor will drastically slow the whole program, as all the other cores have to wait for the last one to finish. Related to this, since the use of steps are synchronized, unnecessary steps should really be avoided: if the second of two consecutive steps doesn't need the result of the first one, they should be merged.

According to these points, some algorithms are definitely better suited to Menthor than others. Algorithms having a natural graph representation will perform far better than others that have to be moulded by force in the graph pattern. As previously stated, a balanced distribution of load between the nodes is also primordial to take the best out of Menthor.

Unfortunately our collaborative filtering algorithm wasn't perfectly suitable for these two reasons. The messages overhead resulting from the "moulding" in the graph structure and the inability of a balanced nodes distribution without a deep dataset analysis, prevented the optimal use of the multi-core architecture we worked with.

However with smaller datasets where these two issues were not a significant problem the framework performed very efficiently.

Exploring more in detail the framework would be a really attractive opportunity, but a more complex architecture, with a larger number of cores (in the two digit range) or even a distributed infrastructure, would be required to study more deeply the potential of Menthor.