Parallel machine learning using Menthor

Studer Bruno
Ecole polytechnique federale de Lausanne

June 8, 2012

1 Introduction

The algorithms of collaborative filtering are widely used in website which recommend some items, it could be movies, commercial product or even dating site. These website use the users registered and the ratings they gives or the habits they have to form their dataset. Of course the larger the dataset the better result this category of algorithms yield.

Since we want the bigger dataset possible we also need a efficient way to compute the recomendation ; and of course if we cannot find more efficient algorithm, we can parallelize it. The last question remaining is : how well theses algorithms can be parallelize ?

To answer this question we will investigate the scala parallel collection which are pretty straigh-forward ; and we will also try to transform our problem into a graph problem to fit the Menthor framework.

2 Framework

Menthor[1] is a framework developed by Heather Miller and Philipp Haller at the EPFL in Scala. The principle of menthor is to represent the dataset as a graph, and working locally on the vertices. Then there is a communication over the edges using messages ; and finally we have a crunch step who aggregate all the data.

The algorithm must be split into two synchronized steps :
- The update step, where the vertices must locally update themselves using the incoming messages form the adjacent vertices.
- The crunch step, which operate like a huge reduce, and once all the data are processed, make the result available to all the vertices for the next step.

More information on Menthor can be found in the authors publication[2]

[1] Project page: http://lcavwww.epfl.ch/~hmiller/menthor/
[2] Publication: http://lcavwww.epfl.ch/~hmiller/docs/scalawksp11.pdf

3 Algorithm

The algorithm choosen is an algorithm use in collaborative filtering called *Alternating-Least-Squares with Weigthed-λ-regulation* (ALS-WR).
The important idea behind this algorithm is to discretize users and items into a vector of size n_f, the bigger the size the more precise the algorithm will get. This discretization represent a fix numbers of criteria which should (idealy) fully describe a user or an item. And (if the model was perfect) we would only have to calculate the scalar product between a user and an item to find the rating this item would get from this user.
The optimisation method is a Least-Squares approach, the mathematical developpement is done in details in the article *Large-scale Parallel Collaborative Filtering for the Netflix Prize*[3].

Nethertheless we will quickly see the core devloppement ; every vector representing a user (u_i) or a movies (m_j) are put together to form the matrix U and M :

$$\mathbf{u_i} \subseteq \mathbb{R}^{n_f}$$

$$\mathbf{m_j} \subseteq \mathbb{R}^{n_f}$$

$$U = [\mathbf{u_i}]$$

$$M = [\mathbf{m_j}]$$

We can then construct our least square rule, where r_{ij} is a known rating and $< \mathbf{u_i}, \mathbf{m_j} >$ is the scalar product between a user and a movie.

$$\mathcal{L}^{emp}(R, U, M) = \frac{1}{n} \sum_{(i,j) \in I} (r_{ij} - < \mathbf{u_i}, \mathbf{m_j} >)^2$$

This represent the Root Mean square error (RMSE). Now we can add the lambda constraint :

$$\mathcal{L}^{emp}_{\lambda}(R, U, M) = \mathcal{L}^{emp}(R, U, M) + \lambda \left(\|U\Gamma_U\|^2 + \|M\Gamma_M\|^2 \right)$$

where Γ_U and Γ_M are choosen Tikhonov matrices. In our case, we will take one of simpler matrices possible: the identity matrix weighted by the number of user rated or movies rated :

$$\Gamma_U = diag\left(n_{u_i}\right)$$

$$\Gamma_M = diag\left(n_{m_j}\right)$$

To apply this least square optimisation we will use the following update steps for U and M, which will be applied once after another.

[3]Yunhong Zhou. Dennis Wilkinson, Robert Schreiber and Rong Pan. *Large-scale Parallel Collaborative Filtering for the Netflix Prize*. HP Labs, 1501 Page Mill Rd, Palo Alto, CA, 94304

Updating U

$$A_i = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E \tag{1}$$

$$V_i = M_{I_i} R^T (i, I_i) \tag{2}$$

$$\mathbf{u}_i = A_i^{-1} V_i \tag{3}$$

Updating M

$$A_j = U_{I_j} U_{I_j}^T + \lambda n_{m_i} E \tag{4}$$

$$V_j = U_{I_j} R^T (I_j, j) \tag{5}$$

$$\mathbf{m}_j = A_j^{-1} V_j \tag{6}$$

Where :

- I_i is the set of movie that the user i has rated
- n_{u_i} is the cardinality of I_i
- E is the $n_f \times n_f$ identity matrix
- M_{I_i} is the sub-matrix of M where columns $j \in I_i$ are selected
- $R(i, I_i)$ is the row vector where columns $j \in I_i$ of the i-th row of R is taken

3.1 Parallel processing

The important things to understand for parallelizing in this algorithm is that there is three steps

1. Fixing the item, and solving the users
2. Fixing the user, and solving the item
3. Calculating the error

Each of theses steps are dependante from one another, but the first two steps are about updating a huge matrix with independent opration on every line. So the parallelization will apply mainly during linear algebra computation (matrix multiplication, matrix inversion, multi equations solving).

4 Implementation

4.1 Dataset

The first problem that arise is how to keep the dataset in memory. At first our data can be view as a huge list of Tuple3 containing :

1. A unique user id
2. A unique movie id
3. the rating from the user for this movie

Of course as we want to go as fast as possible we need a much more optimal data structure for our problem. For this we go back to the equations 1 and 4 and we see that during all the processing we will need to acces two kind of set for every user or movie:

- the set of all the movie rated by one user.
- the set of all user which have rated a specific movie.

The best construct to acces theses two set, as faster as possible, is to create two object :

- the user map : Map[userId, Map[movieId, rating]]
- the movie map : Map[movieId, Map[userId, rating]]

This is of course not optimal memory-wise since we dublicate all the information ; but this duplication give us almost $\mathcal{O}(n)$ time acces to any of the set from above.

4.2 Matrix

4.2.1 Multiplication

As we seen in the equations 3 and 6 there is an intensive usage of linear algebra operation, thus we need to construct a Matrix class as fast as possible. The main time-consuming operation is from 1 and 4 where we have the product between two potentially huge matrix. Unfortunaly there is not much we can do about this not to become $\mathcal{O}\left(n^3\right)$; the Strassen's algorithm ($\mathcal{O}\left(n^{2.807}\right)$) require square matrix and in our case it would mean to apply a padding which could become very unefficient as the size of the matrix is very variable. However we can use the fact that we are multiplying a matrix with same matrix tranposed which will always result in a square-hermitian matrix. This give us the possibility to calculate only half of the coefficient and thus we got $\mathcal{O}\left(n^3/2\right)$ operations.

4.2.2 Inversion

Another important step of the algorithm are in the equations 3 and 6 where we have a matrix inversion. To calculate this our first choice was the LU-decomposition which is $\mathcal{O}\left(2n^3/3\right)$. But as we seen before the two matrix from 1 and 4 are hermitian and thus we can use the Choelsky decomposition which is $\mathcal{O}\left(n^3/3\right)$.

4.3 Parallel Collections

The implementation using the parallel collection is very simple since we got update step in which all the vector are updated independently. We only need to transform our Map into the parallel equivalent which can be done by writing the three letter 'Par' before our collection and the trick is done. The update operation on the user or movies matrix which was written as a single foreach loop then becomes automatically parallel.

4.4 Menthor

Since menthor is build around machine learning in graphs, we must adapt our algorithm to a graph. We start by collecting all the user into one set and all the movies into another one. Then we simply connect the pair of vertices user-movie if and only if the user rated the movies. With this consruct we can easily see that the constructed graph is indeed bipartite. Furthermore, in this graph representation we got only a few edges between the two sets (which are the given ratings). The goal is now to construct the fully connect bipartite graph with all the calculated ratings which minimize the error.

Now that we got the vertices, we must set the Data type which our graph will use. In our case we need to be able to reduce all the graph during the crunch step representing the RMSE computation. To achieve this we choose the Data type as a Tuple2 of the vector representation of a user/movies and a Float value representing the square error. With this kind of Data type the crunch step become very easy since we will just have to sum all the local errors.

Finally we must construct our update step without inversion of control. We do this using the 'then' combinator, constructing five substeps :

1. the movies vertices send their (updated) values to the users vertices
2. the user vertices read their message, update their values and send them back to the movies
3. the movies vertices update their values using the one's sent by the users vertices
4. calculate the RMSE using the crunch step
5. the monitor vertex receive the result of the aggregration and use it (in our case it's just printed)

These five substeps forms the single menthor step.

4.5 User's perspective

Once the implementation completed, it is good to look back and compare the practicity of the two way to paralellize we used. First the parallel collection, their are very usefull and powerfull once you isolate an idependante set of operation to compute. In this project the idependente set was not very complicated to dig out and thus the parallel collection were simple to use. Also technically you only need to add a 'Par' to your collection to make it parallel, which is very quick and easy to do. On the other hand if this 'extraction' is not easy to see or even impossible, it quickly becomes very hard to use the parallel collection well.

Second, Menthor, as a framework it is not as quick or simple to use it for the first time, but once your problem is transform in a graph problem, the tool of menthor becomes very practical. You can easily creat vertices, connect them and add them to the graph ; and when it's time to form the update steps the

'then' and 'crunch' combinator becomes very handy. In this view menthor take more time to use, but its construction force more clarity in the steps, substeps process.

5 Benchmark

5.1 Protocol

The benchmarking is done using the datasets from *GroupLens research lab*[4]. We are using the two set '100k' and '1m' :

- 100k - 100,000 ratings from 1000 users on 1700 movies.
- 1M - 1 million ratings from 6000 users on 4000 movies.

With these sets we are going to test the tree different version of our algorithm :

1. serial control version
2. parallel using parallel collections
3. parallel using Menthor

For each one of theses, we are going to do three-pass run with the following number of hidden factor $n_f = 50, 100, 200, 300, 500$
Then we will continue using only the two parallel version with $n_f = 700, 1000$

For each of these run we collect the run time of the sum of the three steps ; each step containing three substeps :

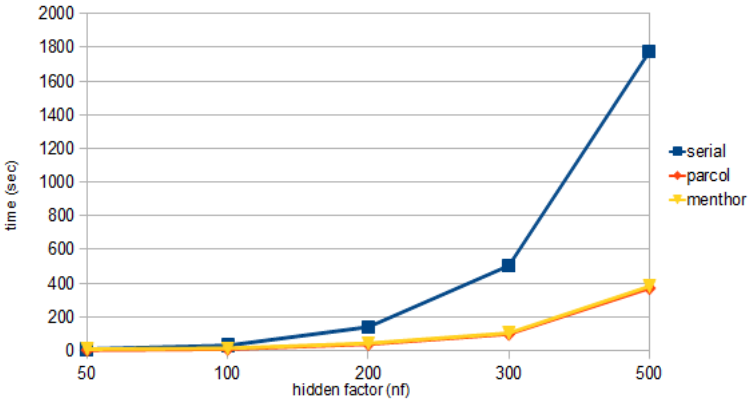
1. updating the users matrix
2. updating the movies matrix
3. computing the error

5.2 Results

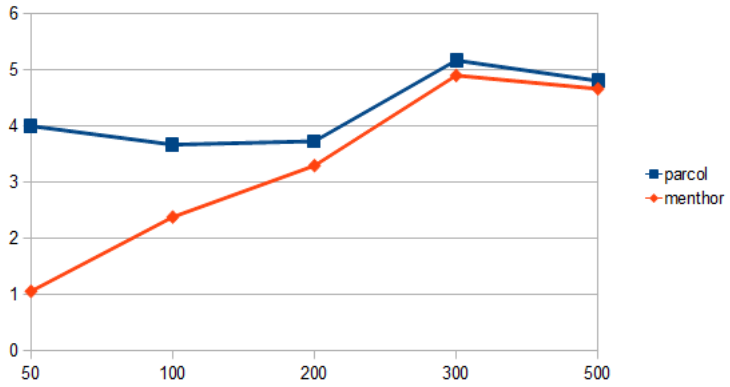
The benchmarking were done on i7-920 (4 physical core with hyperthreading). We also try another machine with 8 physical cores and no hyperthreading, but the result were about the same.

The first set of benchmarks were made on the small database (100k ratings) to get a general idea of how well the parallel collections and menthors would manage the algorithm, and to get an aproximation how the speedup we could hope for.

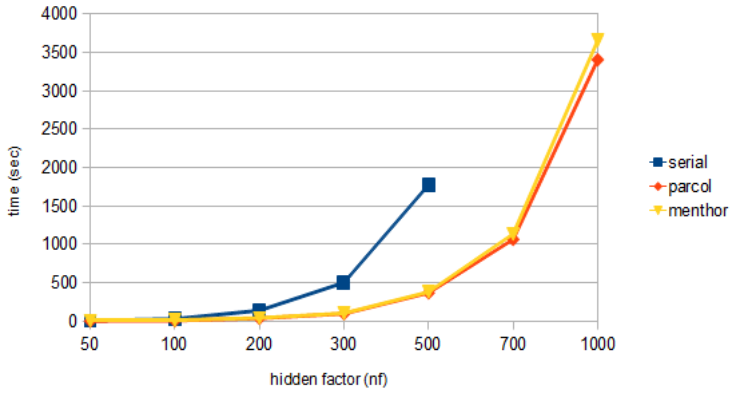
[4]http://www.grouplens.org/node/73



As expected, the parallelization becomes usefull only once we reach a n_f factor large enough. It is also interesting to see that whith the small database the difference between the parallel collection and menthor is almost invisible.
The speedup for this first part is shown in the next graph :

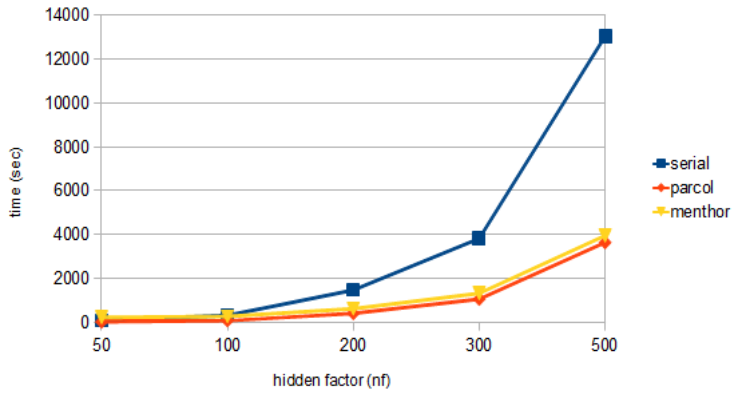


We see that parallel collection are consistent in their speedup. Menthor, by is construction, use much more object and high level construct which, for very small value, diminishes the speedup. But even with average value ($n_f \geq 300$) menthor catch up to the parallel collections.
We now add the remaining part with a higher hidden factor :

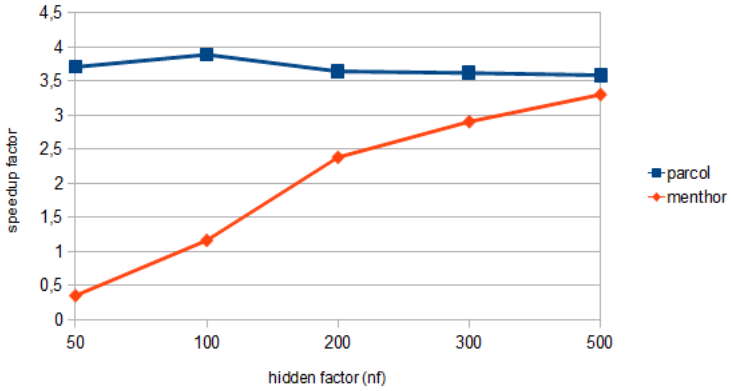


This time we are starting to see a gap beetween menthor and the parallel collections at the highest factor. The difference in about 7% on average. We can see that already in the smaller dataset, we have some emerging tendencies. On the parallelization side, it still works pretty well with this algorithm, and for now the menthor framework is not much behind in terms of performances.

To investigate this a little further we will now look at the next graph, which is from the bigger dataset (1m ratings)

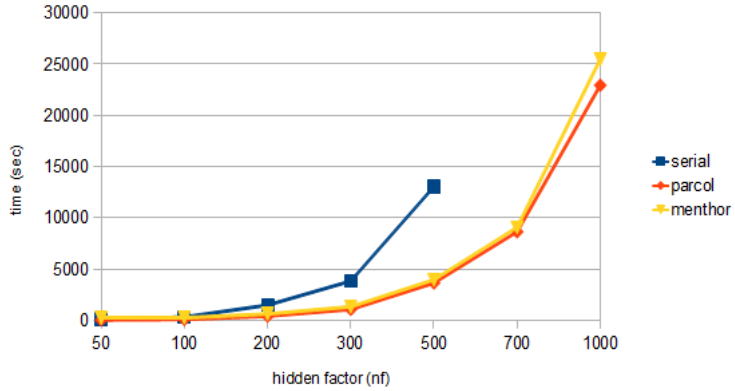


Now as we would expect this is very similar to the smaller dataset but this time we see a little more the difference between menthor and the parallel collections.
Let us see the speedup this time :



The same remarks as the increasing speedup on menthor applies, but this time it is more consistent.

And finally we expand to the full graph of the '1m' dataset :



This last graph confirms two things :
- the parallel collections as well as menthor seems consistently scalable, as theses benchmarks running time were between 8 secondes to 7 hours.
- with this simple algorithm which use mainly basic operation (linear algebra) with very little abstraction, the parallels collections keep an $\sim 10\%$ advantage in running time.

6 Conclusions

The algorithm we choose was a really good test because it was ideal for parallel collections ; all the update can be done independently from one another and the weight of the update is about uniform. As we would have expected the parallel collection performed and scaled very consistently for a wide range of parameters. Furthermore they were easy to use and did not need specific tuning.

The menthor framework, even so it was not the best context for it, still manage to be almost as efficient as the parallel collections and has a very good scaling too. However, one of the difficulty was to understand the algorithm as a graph problem ; but this difficulty is more of a theorical one.

On the implementation side, once you find a good representation, the menthor construction feel natural with its easy vertex construct, vertex connection and of course the update steps which can be split into substeps. It feel more like flow programming with synchronous jump to the next step.

Finally, if we want to go deeper in the menthor's capacity, we could try more complex algorithms or more intensive parallelization (increase number of core, distributed system or GPU parallelization).