Distributed Bayesian Probabilistic Matrix Factorization

Tom Vander Aa and Imen Chakroun
ExaScience Life Lab, Belgium
IMEC, Leuven, Belgium

Tom Haber
ExaScience Life Lab, Belgium
UHasselt, Belgium

***Abstract*—Matrix factorization is a common machine learning technique for recommender systems. Despite its high prediction accuracy, the Bayesian Probabilistic Matrix Factorization algorithm (BPMF) has not been widely used on large scale data because of its high computational cost. In this paper we propose a distributed high-performance parallel implementation of BPMF on shared memory and distributed architectures. We show by using efficient load balancing using work stealing on a single node, and by using asynchronous communication in the distributed version we beat state of the art implementations.**

***Index Terms*—probabilistic matrix factorization; collaborative filtering; machine learning; distributed systems; multi-core;**

I. Introduction

Recommender Systems (RS) have become very common in recent years and are useful in various real-life applications.

The most popular ones are probably suggestions for movies on Netflix and books for Amazon. However, they can also be used in more unlikely area such drug discovery where a key problem is the identification of candidate molecules that affect proteins associated with diseases. One of the approaches that have been widely used for the design of recommender systems is collaborative filtering (CF). This approach analyses a large amount of information on some users' preferences and tries to predict what other users may like. A key advantage of using collaborative filtering for the recommendation systems is its capability of accurately recommending complex items (movies, books, music, etc) without having to understand their meaning. For the rest of the paper, we refer to the items of a recommender system by movie and user though they may refer to different actors (compound and protein target for the ChEMBL benchmark for example [1]).

To deal with collaborative filtering challenges such as the size and the sparseness of the data to analyze, Matrix Factorization (MF) techniques have been successfully used. Indeed, they are usually more effective because they take into consideration the factors underlying the interactions between users and movies called *latent features*. As sketched in Figure 1, the idea of these methods is to approximate the user-movie rating matrix R as a product of two low-rank matrices U and V (for the rest of the paper U refers to the users matrix and V to the movie matrix) such that $R \approx U \times V$. In this way U and V are constructed from the known ratings in R, which is usually very sparsely filled. The recommendations can be made from the approximation $U \times V$ which is dense. If $M \times N$ is the dimension of R then U and V will have dimensions $M \times K$ and $N \times K$. K represents then number of latent features characterizing the factors, $K \ll M$, $K \ll N$.

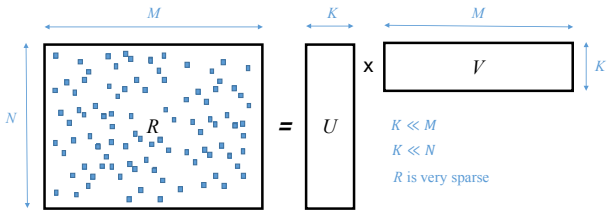

Fig. 1. Low-rank Matrix Factorization

Popular algorithms for low-rank matrix factorization are alternating least-squares (ALS) [2], stochastic gradient descent (SGD) [3] and the Bayesian probabilistic matrix factorization (BPMF) [4]. Thanks to the Bayesian approach, BPMF has been proven to be more robust to data-overfitting and released from cross-validation (needed for the tuning of regularization parameters). In addition, BPMF easily incorporates confidence intervals and side-information [5], [6]. Yet BPMF is more computational intensive and thus more challenging to implement for large datasets. Therefore, the contribution of this work is to propose a parallel implementation of BPMF that is suitable for large-scale distributed systems.

The remainder of this paper is organized as follows. Section II describes the BPMF algorithm. In Section III, the shared-memory version and in Section IV the distributed version of BPMF are described. The experimental validation and associated results are presented in Section V. Conclusions are drawn in Section VI

II. BPMF

The BPMF algorithm [4] puts matrix factorization in a Bayesian framework by assuming a generative probabilistic model for ratings with prior distributions over parameters. It introduces common multivariate Gaussian priors for each user of U and movie in V. To infer these two priors from the data, BPMF places fixed uninformative Normal-Wishart hyperpriors on them. We use a Gibbs sampler to sample from the prior and hyperprior distributions.

This sampling algorithm can be expressed as the pseudo code shown in Algorithm 1. Most time is spent in the loops updating U and V, where each iteration consist of some

relatively basic matrix and vector operations on $K \times K$ matrices, and one computationally more expensive $K \times K$ matrix inversion.

Algorithm 1 BPMF Pseudo Code

```
for sampling iterations do
    sample hyper-parameters movies based on V
    for all movies m of M do
        update movie model m based on ratings (R) for this
        movie and model of users that rated this movie, plus
        randomly sampled noise
    end
    sample hyper-parameters users based on U
    for all users u of U do
        update user u based on ratings (R) for this user
        and model of movies this user rated, plus randomly
        sampled noise
    end
    for all test points do
        predict rating and compute RMSE
    end
end
```

These matrix and vector operations are very well supported in Eigen [7] a high-performance modern C++11 linear algebra library. Sampling from the basic distributions is available in the C++ standard template library (STL), or can be trivially implemented on top. As a results the Eigen-based C++ version of Algorithm 1 is a mere 35 lines of C++ code with good performance.

In the next sections we describe how to optimize this implementation to run efficiently on a shared memory multi-core system and on a distributed system with multiple compute nodes.

III. Multi-core BPMF

The main challenges for performing BPMF in parallel is how to distribute the data and the computations amongst parallel workers (threads and/or distributed nodes). For the shared memory architectures, our main concern is using as many cores as possible, keeping all threads as busy as possible and minimizing memory discontinuous accesses. Since the number of users entries (resp. movie entries) is very large and since all items can be computed in parallel, it makes sense to assigned a set of items to each thread.

Next, balanced work sharing is a major way of avoiding idle parallel threads. Indeed, if the amount of computations is not balanced some threads are likely to finish their tasks and stay idle waiting for others to finish. Some items (users or movies) have a large number of ratings and the amount of compute is substantially larger for those items. To ensure a good load balance, we use a cheaper but serial algorithm for items with less than 1000 ratings. For items with more ratings, we use a parallel algorithm containing a full Cholesky decomposition. This choice is motivate by Figure 2 which shows the time to update one item versus the number of ratings for the three possible algorithms. By using the parallel algorithm for more expensive users/movies we effectively split them up in more smaller tasks that can utilize multiple cores on the system.

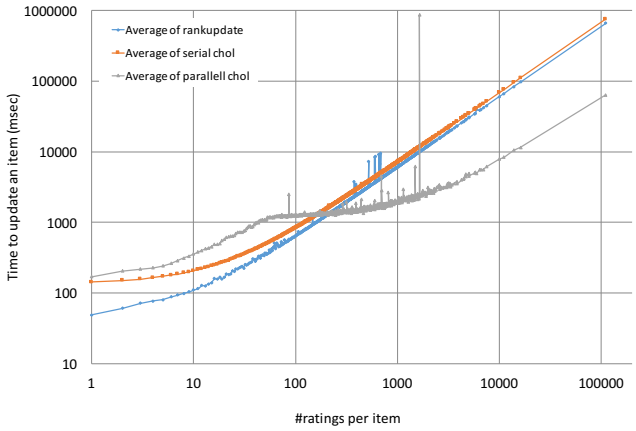

Fig. 2. Compute time to update one item for the three methods: sequential rank-one update, sequential Cholesky decomposition, and parallel Cholesky decomposition

IV. Distributed parallel BPMF

The multi-core BPMF implementation presented above has been extended to distributed systems using MPI [8]. In this section we first describe the MPI programming model, next how the data is distributed across nodes, how the work per node is balanced and how communication is handled.

A. Distributed Programming using MPI

Message Passing Interface (MPI) is a standardized and portable message-passing system for distributed systems. The latest standard MPI-3.0 includes features important for this BPMF implementation, for example: support for asynchronous communication, support for hybrid application combining message passing with shared memory level parallelism like OpenMP [9] or TBB [10].

B. Data Distribution

We distribute the matrices U and V across the system where each nodes computes their part. When an item is computed, the rating matrix R determines to what nodes this item needs to be sent.

Our main optimization concern on how to distribute U and V is to make sure the computational load is distributed as equally as possible and the amount of data communication is minimized. Similarly to the cache optimization mentioned above, we can reorder the rows and columns in R to minimize the number of items that have to be exchanged, if we split and distribute U and V according to consecutive regions in R.

Additionally we take work balance in to account when reordering R. For this we use a workload model derived from Figure 2: we approximate the workload per user/movie with fixed cost, plus a cost per movie rating.

C. Updates and data communication

To allow for communication and computation to overlap we send the updated user/movie as soon as it has been computed. For this we use the asynchronous MPI 3.0 routines `MPI_Isend` and `MPI_Irecv`. However, the overhead of calling these routines is too much to individually send each item to the nodes that need it. Additionally, too many messages would be in flight at the same time for the runtime to handle this efficiently. Hence we store items that need to be sent in a temporary buffer and only send when the buffer is full.

V. Validation

In this section, we present the experimental results and related discussion for the proposed parallel implementations of the BPMF described above.

A. Hardware platform

We performed experiments on Lynx a cluster with 20 nodes, each equipped with dual 6-core Intel(R) Westmere CPUs with 12 hardware threads each, a clock speed 2.80GHz and 96 GB of RAM, and on Fermi, an IBM BlueGene/Q system with 10240 nodes, each equipped with 16 cores running at 1.2Ghz and 16 GB of memory.

B. Benchmarks

Two public benchmarks have been used to evaluate the performances of the proposed approaches: the ChEMBL dataset [1] and the MovieLens [11] database.

The ChEMBL dataset is related to the drug discovery research field. It contains descriptions for biological activities involving over a million chemical entities, extracted primarily from scientific literature. Several version exist since the dataset is updated on a fairly frequent basis. In this work, we used a subset of the version 20 of the database which was released on February 2015. The subset is selected based on the half maximal inhibitory concentration (IC50) which is a measure of the effectiveness of a substance in inhibiting a specific biological or biochemical function. The total ratings number is around 1023952 from 483500 compounds (acting as users) and 5775 targets (acting as movies).

The MovieLens dataset (ml-20m) describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. It contains 20M ratings across 27278 movies. These data were created by 138493 users between January 09, 1995 and March 31, 2015.

For all the experiments, all the versions of the parallel BPMF reach the same level of prediction accuracy evaluated using the root mean square error metric (RMSE) which is a used measure of the differences between values predicted by a model or an estimator and the values actually observed [12].

C. Results for Multi-core BPMF

In this section, we compare the performance of the proposed multi-core BPMF with the Graphlab library which is a state of the art library widely used in machine learning community.

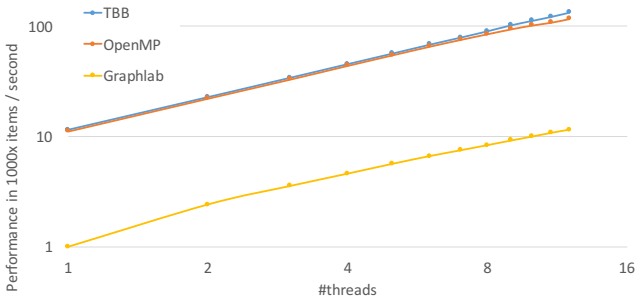

Fig. 3. Performance of the multi-core BPMF on the ChEMBL dataset in number of updates to U and V versus the number of parallel threads.

We have chosen GraphLab because it is known to outperform other similar graph processing implementations [13].

The results presented in Figure 3 report the performance in number of updates to U and V per second for the ChEMBL benchmark suite on a machine with 12 cores for three different version, using TBB, OpenMP and GraphLab.

The results show that all parallel implementations of the BPMF scale with the increasing number of used cores. The TBB version performs better than the OpenMP version because TBB's support for nested parallelism and because TBB uses a work-stealing scheduler that can better balance the work. The higher-level GraphLab library focuses less on performance and more on programmer productivity and this gap is clearly visible in the graph.

D. Distributed BPMF

Figure 4 shows strong scaling results for the distributed MPI version of BPMF on a large system. Scaling is good, even super-linear, up to 32 nodes, which is one node rack on this system. Once the application's allocation exceeds this one rack, performance degrades significantly. The overlap of communication and computation is displayed in Figure 5. The term 'both' in this figure means time spent communicating *and* computing. This figure shows that it is possible to overlap computation and communication with MPI at small core count. It seems that this overlap does not help much for large core count, most probably due to a large overhead in the MPI library itself.

VI. Conclusion and future work

This work proposed a high-performance distributed implementation of the Bayesian probabilistic matrix factorization algorithm. We have shown that load balancing and asynchronous communication are essential to achieve good parallel efficiency, clearly outperforming more common synchronous approaches like GraphLab. The achieved speed-up allowed us to speed up machine learning for drug discovery on an industrial dataset from 15 days for the initial Julia-based version to 30 minutes using the distributed version.

In future work we will try to improve scaling results of the distributed version of BPMF by using a more light-weight multi-threaded communication library [14].

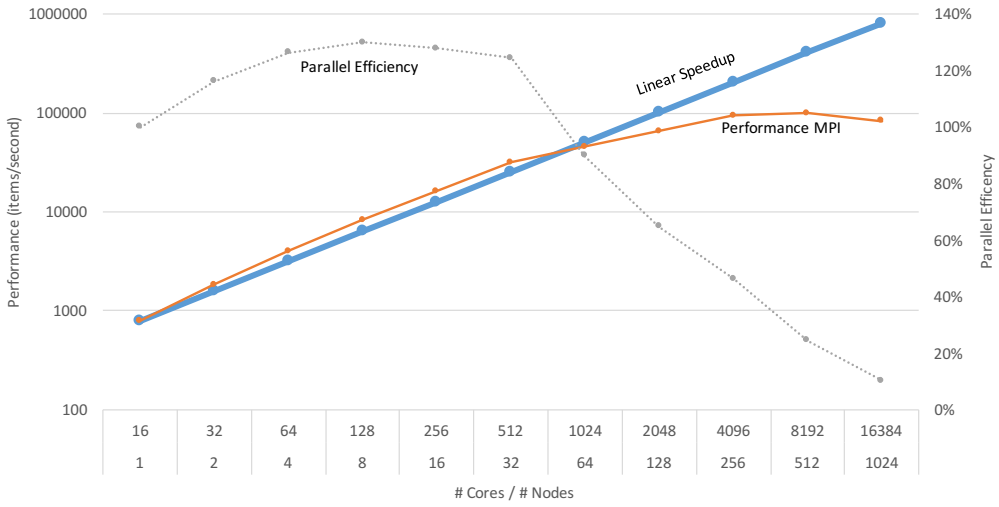

Fig. 4. Performance of the distributed BPMF on the MovieLens dataset in number of updates to U and V per second versus the number of cores used.

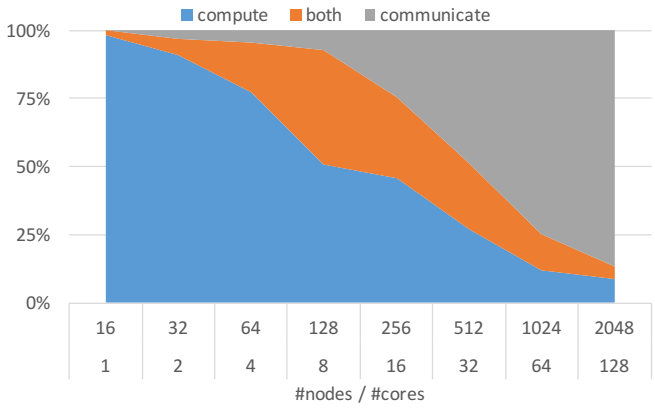

Fig. 5. Time spent in computing, communication and in both for the distributed MPI implementation.

Acknowledgments

This work is partly funded by the European projects EXA2CT (EXascale Algorithms and Advanced Computational Techniques) and ExCAPE (Exascale Compound Activity Prediction Engine) with references 610741 and 671555. We acknowledge PRACE for awarding us access to resource Fermi based in Italy at CINECA and IT4I for providing access to the Anselm and Salamon systems.

References

[1] A. Bento, A. Gaulton, A. Hersey, L. Bellis, J. Chambers, M. Davies, F. Kruger, Y. Light, L. Mak, S. McGlinchey, M. Nowotka, G. Papadatos, R. Santos, and J. Overington, "The ChEMBL bioactivity database: an update." *Nucleic Acids Res.*, vol. 42, pp. 1083–1090, 2014.

[2] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix prize," in *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, ser. AAIM '08. Berlin, Heidelberg: Springer-Verlag, pp. 337–348.

[3] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug 2009.

[4] R. Salakhutdinov and A. Mnih, "Bayesian probabilistic matrix factorization using Markov chain Monte Carlo," in *Proceedings of the International Conference on Machine Learning*, vol. 25, 2008.

[5] J. Liu, C. Wu, and W. Liu, "Bayesian probabilistic matrix factorization with social relations and item contents for recommendation." *Decision Support Systems*, vol. 55, no. 3, pp. 838–850, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/dss/dss55.html#LiuWL13

[6] J. Simm, A. Arany, P. Zakeri, T. Haber, J. K. Wegner, V. Chupakhin, H. Ceulemans, and Y. Moreau, "Macau: Scalable bayesian multi-relational factorization with side information using MCMC," 2015.

[7] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[8] MPI Forum, "MPI: a message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[9] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[10] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[11] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2827872

[12] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting*, vol. 22, no. 4, pp. 679 – 688, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169207006000239

[13] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "Benchmarking graph-processing platforms: A vision," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 289–292. [Online]. Available: http://doi.acm.org/10.1145/2568088.2576761

[14] D. Grünewald and C. Simmendinger, "The GASPI API specification and its implementation GPI 2.0," in *7th International Conference on PGAS Programming Models*, vol. 243, 2013.