Fast Matrix Factorization for Recommendation Systems

15-418/618 Project Final Report

Zhewei Shi, Weijie Chen

Andrew ID: zheweis, weijiec

Dec 15, 2018

1 Summary

We implemented two parallel algorithms for matrix factorization in recommendation systems using MPI and OpenMP frameworks. The implementation and analysis is based on Xeon CPUs and Xeon Phi platforms. Our implementations of both parallel algorithms can achieve nearly linear speedup using up to 16 workers.

2 Problem Formulation

In mathematics, matrix factorization (or matrix decomposition) is to represent a matrix by a product of matrices. It is one of the most popular collaborative filtering techniques used for recommendation systems.

The problem in matrix factorization approach can be described in the following way: Assume that we have m users and n items. We are provided with a rating matrix R and its entries can be an interaction value or a missing value (invalid). The rating matrix R encodes the preference of the u-th user on the v-th item at the (u, v) entry $r_{u,v}$. To estimate some of the missing values, we construct two matrices $P \in R^{m\times k}$ and $Q \in R^{n\times k}$ for some rank k as the latent features of users and items. So the estimated interaction value becomes the inner product of two feature vectors $\hat{r}_{u,v} = {p_u}^T q_v$. Then, we are required to find matrices P and Q that best represent the existing rating values in R. In practice, we will try to minimize a loss function to obtain the optimal matrices.

In this project, we study the mean-square loss function with a regularization term. It can be represented as:

$$\min_{P,Q} \sum_{(u,v)\in R} (r_{u,v} - {p_u}^T q_v)^2 + \lambda_P ||p_u||^2 + \lambda_Q ||q_v||^2$$

We will address two parallel algorithms to solve the matrix factorization problem: alternating least squares (ALS) and stochastic gradient descent (SGD).

3 Algorithms

3.1 ALS

The basic steps of ALS algorithm is shown as following:

Algorithm 1 ALS algorithm

```
Data: rating matrix $R$
Result: user matrix $P$, item matrix $Q$
Initialize matrix $Q$ by assigning average rating values
Initialize matrix $P$ with random small values
while stopping criterion is not satisfied do
    Fix $Q$, solve $P$ by minimizing the loss function
    Fix $P$, solve $Q$ by minimizing the loss function
end
```

There are three main challenges associated with ALS algorithm:

Firstly, there exists dependency between two adjacent update steps. After updating one matrix, the next step will be based on the updated matrix. Therefore, the updating process is inherently serial. Without modifying the algorithm, we can only explore parallelization within a single update step.

Secondly, while updating one matrix, a worker should keep a complete copy of the other matrix. If work is distributed across multiple threads, there will be massive communications between workers when an iteration step starts or ends.

Thirdly, the rating matrix can be extremely large if we have many users and items. Therefore, in order to fit data into memory, we need to compress matrices efficiently.

3.2 SGD

The basic steps of SGD algorithm is shown as following:

Algorithm 2 SGD algorithm

```
Data: rating matrix $R$
Result: user matrix $P$, item matrix $Q$
Initialize matrices $P$, $Q$ by random small values
while stopping criterion is not satisfied do
    Randomly select a $(u, v)$ entry
    $e_{u,v} = r_{u,v} - {p_u}^T q_v$
    $p_u := p_u + \gamma(e_{u,v} q_v - \lambda_P p_u)$
    $q_u := q_u + \gamma(e_{u,v} p_u - \lambda_Q q_v)$
end
```

SGD algorithm is inherently sequential due to the iterative process of updating parameters. Therefore, we are required to explore possible parallelization among these iterative steps. Another problem is poor data locality. Because each entry is randomly selected from the dataset, there will be many random memory access which may lead to a high cachemiss rate.

4 Our Approaches

4.1 Parallelizing ALS

The parallel ALS algorithm is implemented in C language. It will be running on Xeon CPUs using MPI framework.

As we have discussed, each update step in ALS algorithm is dependent on the previous step, making it hard to parallelize. However, we can still parallelize ALS algorithm within a single update step. For example, to update item matrix Q, all item vectors in Q will be evenly distributed to workers (as blocks). The worker who stores the feature vector of item j will be responsible to update the corresponding vector in Q. By this way, the work of one update step is distributed to and completed by many workers. Then, we can apply the similar method to update P. After each update step, workers need to communicate with each other to obtain a complete and updated copy of the matrix.

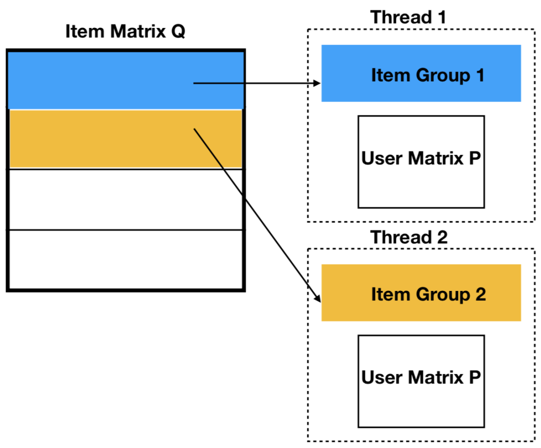

Figure 1: updating Q, item vectors are evenly distributed to workers

4.1.1 Data Compression

Instead of storing the rating matrix R as a 2D array, we utilize the Compressed Sparse Row format (CSR) to record ratings for each user/item. We maintain three arrays for users (similar arrays for items): `userStartIdx[]` stores the start index in the other two arrays for each user, so for user i its data will be placed in `userStartIdx[i]` ... `userStartIdx[i+1]`-1; `movieId[]` will store the index of a movie and `movieRating[]` stores the rating given by a user to the corresponding movie. Then, after creating these arrays, we can efficiently extract all ratings for each user/item. Because the rating matrix is extremely sparse, using the compressed format can save a great amount of space and improve locality in data access.

4.1.2 Matrix Operations

In the updating process, the worker will optimize the feature vector for one user/item at a time. A lot of matrix operations are required while generating the optimal values for a feature vector. To handle numerical computing and matrix operations efficiently, we employ the GNU Scientific Library (GSL) in our project. At the beginning of each update step, we need to move rating and feature data to `gsl_vector` or `gsl_matrix` variables and then we can start the computation. Inverting a matrix is the most time-consuming part in ALS algorithm and we apply LU decomposition to calculate the inverse. After obtaining the optimal feature vector, we are required to move the result back to the array, so workers can communicate and share their results for this update step.

4.2 Parallelizing SGD

The parallel SGD algorithm is implemented in C++ language. It will be running on Xeon Phi machines using OpenMP framework.

Although SGD algorithm is a sequential process, we can take advantage of the property that some blocks in the rating matrix R are mutually independent so the corresponding feature vectors can be updated in parallel without affecting the results. Each thread in OpenMP framework will act as a worker and each of them will update an independent block in the rating matrix. As all workers will update the user/item feature matrices in parallel, the rating, user, and item matrices (R, P, Q) will be shared among all threads. Because different threads will be accessing and modifying different entries of the shared matrices, there is no need to add additional locks to ensure synchronization.

4.2.1 Basic Parallel Version

We uniformly grids the rating matrix R into $s \times s$ sub-matrices, where s is the number of working threads. Then, SGD algorithm can be applied to independent blocks simultaneously.

Two blocks are independent only if they share neither any common columns nor any common rows in the rating matrix. Therefore, if all updating blocks are independent, the updating user/item will be different for all working threads and it greatly reduces the contention between working threads.

In each iteration, all blocks in the rating matrix will be covered. We generate s patterns and each pattern includes s independent blocks. Then, each iteration can be broken into s sub-loops: in each sub-loop, s independent blocks in the corresponding pattern will be assigned to different working threads.

The details of the algorithm is shown as following:

Algorithm 3 Parallel SGD

```
Data: rating matrix $R$, thread number $s$, iteration number $T$
Result: user matrix $P$, item matrix $Q$
Grid $R$ into $s \times s$ blocks $B$ and generate $s$ patterns
1: for $t = 1 \dots T$ do
2:     Decide the order of s patterns sequentially
3:     for each pattern of $s$ independent blocks of $B$ do
4:         Assign $s$ selected blocks to $s$ threads
5:         for $b \in \{1, \dots, s\}$ do
6:             Parallel randomly select ratings from block $b$ and apply SGD
7:         end for
8:     end for
9: end for
```

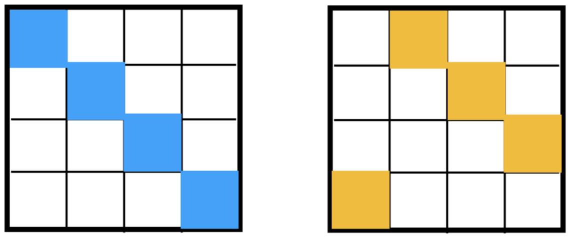

Figure 2: thread number = 4, two consecutive iterations in line 3

The above figure shows two consecutive iterations in line 3. With 4 working threads, the rating matrix R is gridded into 4×4 sub-matrices. Blue and yellow blocks are two sets of independent blocks that can be updated in a batch. There are 4 such patterns, which cover all sub-matrices in the rating matrix.

4.2.2 Random Shuffling

We notice that the locking problem exists in the initial version of parallel SGD algorithm. Because s independent blocks are updated in a batch, if the running time for each block varies, a thread that finishes its job earlier has to wait for other threads. The problem will be more severe when the rating matrix R is unbalanced.

In order to balance the workload, we adopt the method of random shuffling. Before starting the computation, we randomly permute user and item IDs to make the rating matrix more balanced in each single block. In most cases, the number of valid ratings in each block will be evenly distributed after random shuffling.

4.2.3 Data Locality

Most SGD solvers randomly pick instances from the rating matrix for update. Such random method is likely to discontinuously access data in memory. To address this locality problem, we use ordered access pattern in our design: we sequentially select rating instances by user IDs and then we can have continuous access to the user feature matrix.

However, after comparing the cache performance of ordered access and random access, we discover that the ordered access model does not improve much in speedup and convergence speed. Cache misses still occur as frequently as in the random access model. We believe this is mainly due to the highly sparse rating matrix and, after gridding, sparsity is further aggravated in sub-matrices. In our final implementation, we use the ordered access model as it performs better when the thread number is small.

4.2.4 Merging (Increase Granularity)

Due to the overhead of synchronization, the parallelization performance is not ideal when the thread number is large. All s threads need to synchronize after finishing their update jobs. Even if the workload is relatively balanced after random shuffling, frequent synchronization will still be very expensive.

We try to minimize the synchronization cost by introducing a merge factor M into our algorithm. In each iteration, we will merge the updates of M patterns to one sub-loop. As synchronization occurs only after each sub-loop, reducing the number of sub-loops can effectively reduce the synchronization overhead.

One drawback of larger granularity is that the updating process of each thread may not be independent any longer. However, under the assumption that the rating matrix is highly sparse and randomly sampled, the probability of conflicts will be very low. Furthermore, even if some data becomes stale (not updated synchronously), we can tolerate it as long as this does not seriously compromise the convergence performance.

In our final implementation, we also adjust the value of merge factor based on the thread number. Because our algorithm can achieve nearly linear speedup when the thread number is small, the merge factor will be set to 1 in this case. When the thread number becomes larger, the merge factor will increase accordingly as the synchronization overhead will increase significantly under this condition.

5 Experiments

5.1 Experimental Settings

Datasets. Two MovieLens datasets, `ml-100k` and `ml-1m`, are used for experiments. Approximately, `ml-100k` contains 100,000 ratings from 600 users on 9,000 movies; `ml-1m` contains

1 million ratings from 6,000 users on 4,000 movies. All ratings are made on a 5-star scale.

Platforms. All experiments are performed on the Latedays cluster. ALS algorithm is tested on Xeon CPUs using MPI. Each node (machine) will run up to 16 processes (threads) and we will use up to 4 nodes. SGD algorithm is tested on Xeon Phi (offload mode) using OpenMP. There are 60 cores and we will use up to 4 threads on each core.

Parameters. For ALS algorithm, we set the default iteration number to be 5, default latent feature number to be 16, and coefficients of the regularization term $\lambda_P = \lambda_Q = 0.065$. For SGD algorithm, we set the default iteration number to be 30, default latent feature number to be 10 (an empirical value according to many papers), and $\lambda_P = \lambda_Q = 0.001$. The learning rate of SGD is set to be 0.0001. As we mainly focus on speedup, the adaptive learning rate will not be used here.

Initialization. As the user and movie IDs are not continuous in the datasets, we preprocess the rating file and assign new continuous IDs to each user and movie. For feature matrices initialization, we set all values in user and item matrices to be random small values in $[0, 1]$. Because all real ratings are from 1 to 5, this random initialization can achieve relatively good performance and also accelerate convergence.

Metrics. To measure the prediction accuracy of the model, we use root-mean-square error (RMSE) as the metric. RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{\#\ ratings} \sum_{(u,v) \in R} (r_{u,v} - \hat{r}_{u,v})^2}$$

To measure the effectiveness of parallelism, we use speedup as the metric. The computational time includes input file processing and initialization. Speedup is defined as:

$$Speedup = \frac{single\ thread\ computational\ time}{N\ threads\ computational\ time}$$

5.2 Results on Parallel ALS

5.2.1 Correctness of Implementation

We use the same initialization values (pseudo-randomness) for all experiments on parallel ALS algorithm, so the optimal results should be fixed using more processes. The training errors on multiple processes have the identical results as those from a single process. Therefore, we can ensure that our implementation of parallel MPI version is correct.

5.2.2 Convergence Rate

We record RMSE after each iteration to show the convergence rate of ALS algorithm. As the results of each iteration is fixed and can be predetermined, we simply run this experiment once on 4 processes.

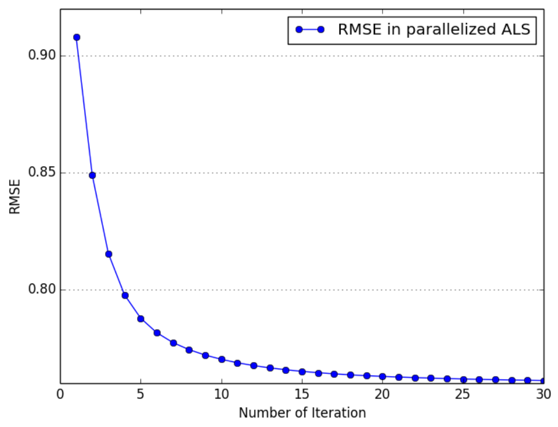

Figure 3: RMSE for ALS on `ml-1m` dataset

As RMSE after 5 iterations (0.78782) is sufficiently good compared to RMSE after 30 iterations (0.761138), we set the default iteration number to be 5 for ALS algorithm.

5.2.3 Effectiveness of Parallelism

Impact of # Nodes/Processes. We run our parallel ALS algorithm on 1, 2, and 4 nodes and then use different number of processes (1, 2, 4, 8, 16) on each node to test speedup performance of the parallel algorithm.

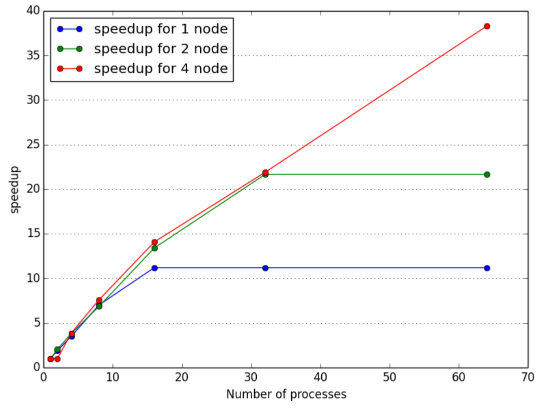

Figure 4: Speedup of parallel ALS on `ml-1m` dataset

Our parallel ALS algorithm can achieve nearly linear speedup (x14.07) using 16 processes and the maximum speedup can reach x38.26 while using 64 processes.

Impact of # Features. We run our parallel ALS algorithm using 8, 16, and 32 latent feature sizes on `ml-1m` dataset.

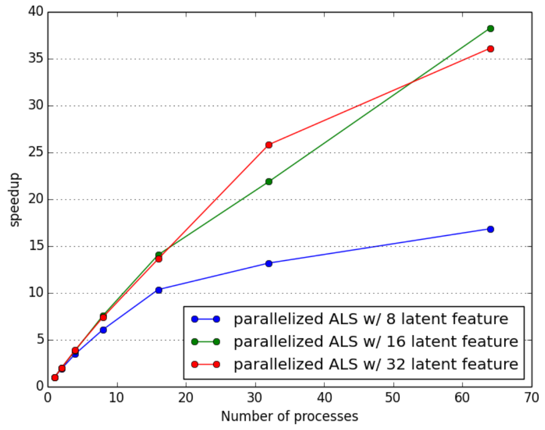

Figure 5: Speedup comparison on different feature numbers

While the feature size is sufficiently large (16 vs 32), increasing the number of features will not improve speedup performance much. However, if the feature size is relatively small (8 vs 16), the speedup performance will drop significantly as computational tasks can be completed very fast and synchronization overhead may dominate and harm the performance.

5.3 Results on Parallel SGD

5.3.1 Influence on Accuracy

In order to make sure that merging (larger granularity) will not compromise accuracy of the model, we compute RMSE after each iteration to monitor the influence. The following table shows RMSE after 0/10/20/30 iterations:

Iteration Number	0	10	20	30
RMSE w/ merging	2.180674	1.076381	0.945193	0.898162
RMSE w/o merging	2.167226	1.078101	0.947153	0.899789

As we can see, merging does not have noticeable effect on RMSE, but it can effectively make SGD algorithm run much faster.

5.3.2 Convergence Rate

As our focus in this project is speedup instead of prediction accuracy (RMSE), we only need to ensure that our parallel algorithm does not compromise the convergence performance. As the baseline, we run the sequential version of code, and then compare the results with the parallel version (128 threads) using the same experimental settings.

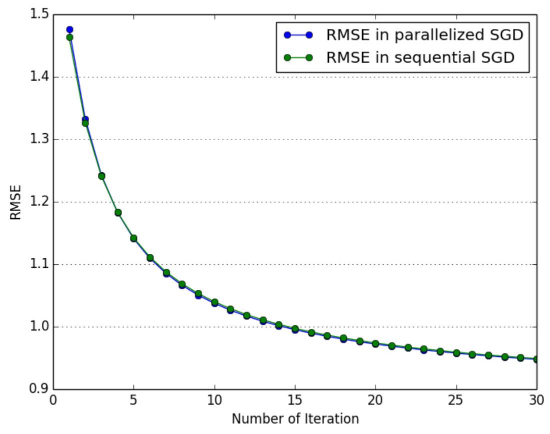

Figure 6: RMSE for sequential and parallel SGD on `ml-1m` dataset

According to above experimental results, our algorithm can achieve nearly the same convergence performance after all iterations. It proves that our parallel algorithm does not harm the convergence rate.

5.3.3 Effectiveness of Parallelism

Different Models. In the approach section, we have discussed three different versions of parallel SGD algorithm: naive, with random shuffling, and with merging factor. In order to prove the effectiveness of parallelism, we compare the speedup of these three versions.

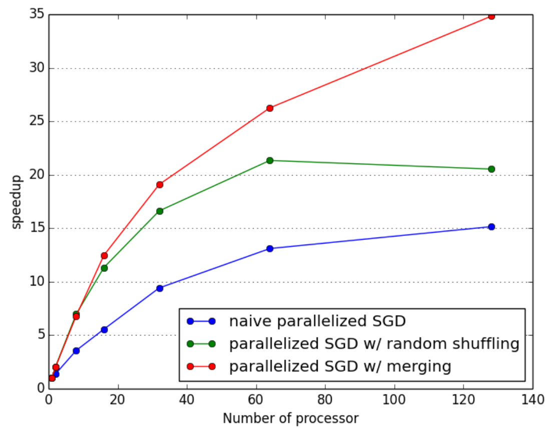

Figure 7: Speedup of three parallel versions

After adding random shuffling, the speedup increases because the workload is more balanced and each thread spends less time waiting for others. As we can see from the figure, the final version could achieve nearly linear speedup when the thread number is relatively small ($\leq$ 16). With appending the merging factor for large thread number, the number of sub-loops decreases, which results in less synchronization overhead and higher parallelism effectiveness.

Impact of Problem Size. The input dataset size has a huge impact on the speedup performance. We run the final parallel version on `ml-100k` dataset.

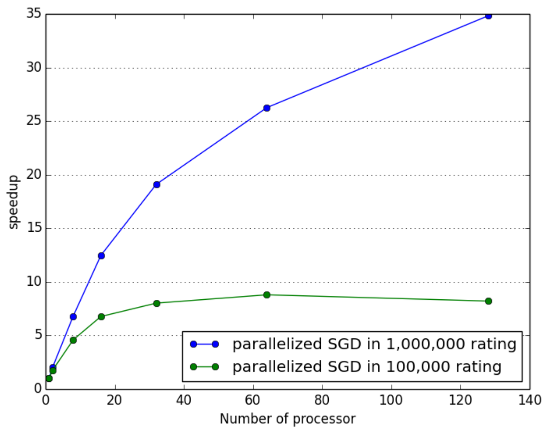

Figure 8: Comparison between small/large datasets

Based on `ml-1m` dataset, we also compare the effect of different latent feature numbers.

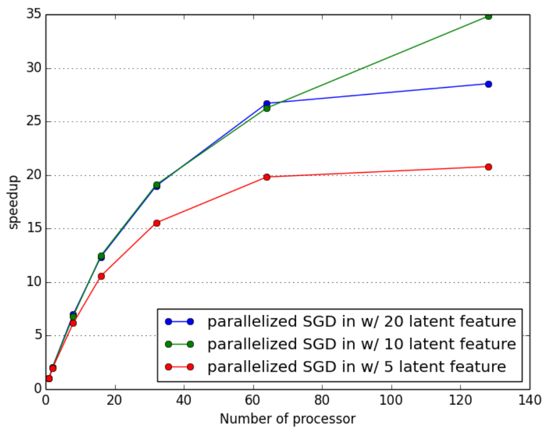

Figure 9: Comparison between different feature numbers

The speedup performance is not ideal when the number of ratings is small. There are several reasons: the synchronization overhead impact is larger when computation in each sub-loop is faster; when dataset size is smaller, the rating matrix is more likely to be sparse, so even random shuffling cannot well balance the workload. In addition, our algorithm would perform better on larger feature number as the overhead impact decreases. Therefore, we can conclude that our algorithm works better for larger dataset with more ratings and features.

5.3.4 Speedup Limit & Further Analysis

There are 2 factors that limit the speedup. Firstly, the synchronization overhead after each sub-loop will keep many threads waiting and idling. According to the results, we can achieve linear speedup for small thread number and speedup performance becomes worse for larger thread number (more sub-loops). After adding the merging factor, which decreases the sub-loop number and synchronization cost, speedup for larger thread number has been greatly improved. Secondly, the cache miss problem is not well addressed in our design. Because the rating matrix is highly sparse (especially after gridding), changing access mode from random to ordered does not help much with the cache miss problem.

In SGD algorithm, every worker should keep updated user and item matrices, and both of them will be updated for each training example. Due to the large number of training examples, MPI framework will certainly result in high communication cost. Therefore, we believe OpenMP's shared memory model is a sound choice to implement the parallel SGD algorithm.

6 References

[1] Zhuang, Yong, et al. "A fast parallel SGD for matrix factorization in shared memory systems." Proceedings of the 7th ACM conference on Recommender systems. ACM, 2013.

[2] Zhou, Yunhong, et al. ”Large-scale parallel collaborative filtering for the netflix prize.” International Conference on Algorithmic Applications in Management. Springer, Berlin, Heidelberg, 2008.

[3] Gemulla, Rainer, et al. ”Large-scale matrix factorization with distributed stochastic gradient descent.” Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011.

[4] He, Xiangnan, et al. ”Fast matrix factorization for online recommendation with implicit feedback.” Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval. ACM, 2016.

[5] Zinkevich, Martin, et al. ”Parallelized stochastic gradient descent.” Advances in neural information processing systems. 2010.

7 Work and Total Credit Distribution

Zhewei Shi: ALS implementation

Weijie Chen: SGD implementation

Both participate in optimization, experiments, and analysis.

Credit distribution is 50 : 50.