Parallelize Matrix factorization using OpenMP

Arnav Kansal   **Ojas Deshpande**
{ak6806, oad230}@nyu.edu

1 Abstract

In this work, we've implemented and analysed a parallel matrix factorization algorithm, primarly based on the work of $libmf$ [1]. We provide detailed analysis of how increasing the number of threads help. Specifically, we analyze how fast the error goes down, and how many iterations are required.

2 Introduction

Recommender systems have been in wide use in the e-commerce, music and entertainment industry lately. After the Netflix Prize and the work by Koren et.al.2009 [2] matrix factorization methods have been a major component in the recommendation system toolboxes. As the name suggests, the application is an effective method to factorize a matrix into the product of 2 matrices. In this course project we wish to implement matrix factorization using Stochastic gradient descent methods in OPENMP for large matrices.

2.1 Mathematical formulation

Decompose **R** $\in \mathrm{R}^{mxn}$ into product of 2 matrices **X, Y** $\in \mathrm{R}^{mxk}$ and R^{kxn} to compute **XY** $\approx$ **R**.

$$\mathbf{R} = \mathbf{XY}$$

Here **R** may have missing (unknown) entries and thus has wide applications in the domain of recommendation systems for instance, finding the missing entries of **R**. We will not talk about the origin of such matrices in detail. But just as an example i-j th element of **R** may represent i th user's preference for j th element of an e-commerce website.

Different mathematical formulations are used for this problem, and the most general optimization problem that it represents is:

$$arg \min_{X,Y} f_R(X,Y)$$

An example of such a function f_R is:

$$\min_{X,Y} ||\mathbf{R} - \mathbf{XY}||_F^2 + \lambda||\mathbf{X}||_F^2 + \mu||\mathbf{Y}||_F^2$$

One may be tempted to simply apply the k-rank approximation of **R** obtained using SVD, but doing so is not possible because many entries of **R** may be missing.

3 Dataset

Also we plan to create a random dataset of custom size by sampling from a random number generator to produce matrices of varying size to experiment and observe speedup.

Course project : *Multicore Processors: Architecture & Programming*, Spring 2018 at New York University.

4 Survey of related work

The problem of matrix factorization has been solved primarily via three classes of algorithms. The three algorithm classes employed are Alternating Least Squares(ALS) , Stochastic gradient descent(SGD) and Coordinate Descent (CD). The current state of the art methods involve these algorithms as the base idea and evolve on the shortcomings of the degree of parallelization of these algorithms to get better performance on real life datasets. None of these methods can be applied to the problem setting in a vanilla formulation as all of these algorithms have inherent sequential update rules.

ALS is a very simple idea to understand and has excellent theoretical bounds for convergence rates. ALS relies on optimizing for X, given a fixed Y and vice versa alternatively to converge to a perfect pair of matrix factors. Thus only after finding one perfect X leads to optimization for Y and so on. This sequence is inherently sequential and has dependencies between each iteration of the algorithm.

Coordinate descent is another popular algorithm in the field of optimization. Yu, Hsiang-Fu, et al.2012 [3] have applied Coordinate Descent approaches to update the matrix **X** and **Y** in an alternating fashion. So they have utilised both ALS and coordinate descent to formulate their problem.

SGD is a very well known algorithm used in many applications of machine learning because of the minimum memory footprint of the algorithm. Also it is very robust against noise in the data and guarantees fast learning rates. But inherent sequential behaviour of the algorithm limits its use. Recht, Benjamin, et al.2011 [4] have come up with a lock free approach to parallelize SGD and have shown that usually the optimization problem is sparse and that most gradient updates only modify small parts of the decision variable and thus can be parallelized by modifying the sampling scheme of the underlying data. This idea has been further utilized by Chin, Wei-Sheng, et al.2015 [1] for shared memory systems. They have reduced the cache-miss rate and adjusted for the dynamic load balancing of the threads to achieve a state of the art matrix factorization library LIBMF.

5 Approach

We will be mostly be following the approach used by the libMF paper, i.e., using stochastic gradient descent (SGD) to solve matrix factorization. Our loss function would be a non-convex loss function $f_R(X, Y)$ as mentioned above (optionally with some additional regularization terms that minimize the norms of the matrix factors produced).

The main method of parallelizing matrix factorization is based on the assumption that given a matrix **R** to be factorized, different blocks (non-overlapping submatrices that don't share a column or row) of this matrix are independent from each other. If this is true, then different threads can simultaneously update the variables for these blocks in parallel.

The choice of selecting which thread would work on which block is important, and usually the decisive factor in determining the efficiency of this approach. As in libMF, we'll follow the technique of assigning the next free thread the least recently updated free block (no other thread is working on updating this block right now, and this way all blocks get updated periodically). *We implement this using a priority queue, where the priority of a block is the number of updates + random rational between* 0 *and* 1. *This random number is just to ensure that ties are resolved randomly.* It is easy to show that with this approach, using t threads, we need atleast to break **R** into $(s+1)*(s+1)$ blocks.

There's another issue that needs addressing which is how we access the entries in a given block. When a thread accesses the entires of a block discontinuously, it creates a lot of cache misses, which again makes our program less efficient. This is addressed by instead accessing the entries in a fixed order (either row-wise or column-wise), so that one of the matrices **X** or **Y** can now be accessed continuously.

Combining both the above methods, the overall algorithm is as follows:

```
procedure MATRIXFACTORIZE(matrix R, num threads s, maxiter)
    grid R into (s+1) (s+1) blocks of equal size
    construct a scheduler                        ▷ Returns LRU block. Randomly resolves ties.
    launch s working threads
    for fixed number of iterations do          ▷ This code will be run in parallel by each thread
        Get next free block from scheduler
        for entry $(u, v)$ in current block do
            $X_u = X_u + \gamma \cdot (R_{u,v} Y_v - \lambda \cdot X_u)$
            $Y_v = Y_v + \gamma \cdot (R_{u,v} X_u - \mu \cdot Y_v)$
```

6 Experimental Setup

We ran experiments by running our parallel factorization algorithm on 3 datasets:

- Factorize $100 * 100$ matrix into $100 * 10$ & $10 * 100$ factors.
- Factorize $1000 * 1000$ matrix into $1000 * 10$ & $10 * 1000$ factors.
- Factorize $1000 * 1000$ matrix into $1000 * 100$ & $100 * 1000$ factors.

For each of these, we compare the results for $1, 3, 4, 9, 19, 24, 39, 49$ threads (for our algorithm, we need the matrix dimensions to be divisible by the number of threads, thus the choice of these number of threads). All testing is done on the following test bench.

```
model name      : AMD Opteron(TM) Processor 6272
cpu MHz         : 2099.946
Num cores       : 16 [4 sockets on machine *64 cores]
Memory          : 256 GB
```

7 Experiment and analysis

As can be seen from the plots, the experimental results back our expectations. As we increase the number of threads, it takes less time to do the same number of iterations. Also, the same number of iterations now give us a much smaller error.

Also, we've added a cache comparison for the 100*100 matrix. When we store the second factor Y in a row-major fashion, this makes the code cache inefficient, because Y is always accessed in a column major fashion during multiplication. When we change this, and start storing Y in a column-major fashion, we get a speedup over our previous results.

For 100*100 matrices, we observe that the time per iteration decreases as we move from 1 thread to 9 threads, but after that it again starts increasing. This is probably because the matrix size is small enough that the thread creation/waiting overhead is more significant now as compared to the larger matrices. So, when we move from 9 to 19 threads, that's the tipping point after which the thread overhead is the bottleneck and takes the most time. Thus, we see an increasing time pattern from that point onwards. (This increase in time when number of threads are too large is also observed for 1000*1000 matrices, but is less significant and only observed after 40 threads in the plots).

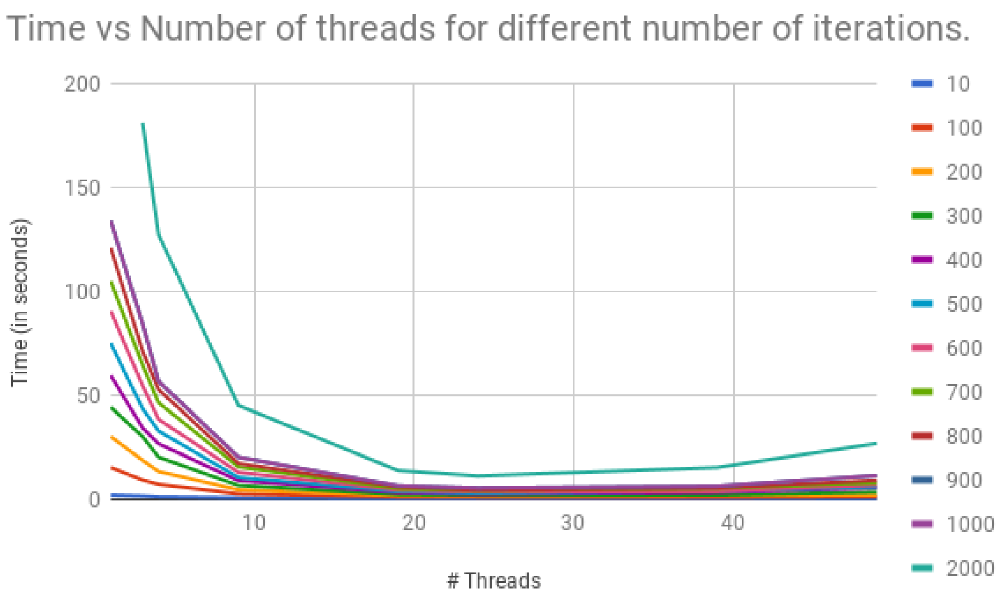

Figure 1: 1000*1000 matrix factorized by latent dimension 10

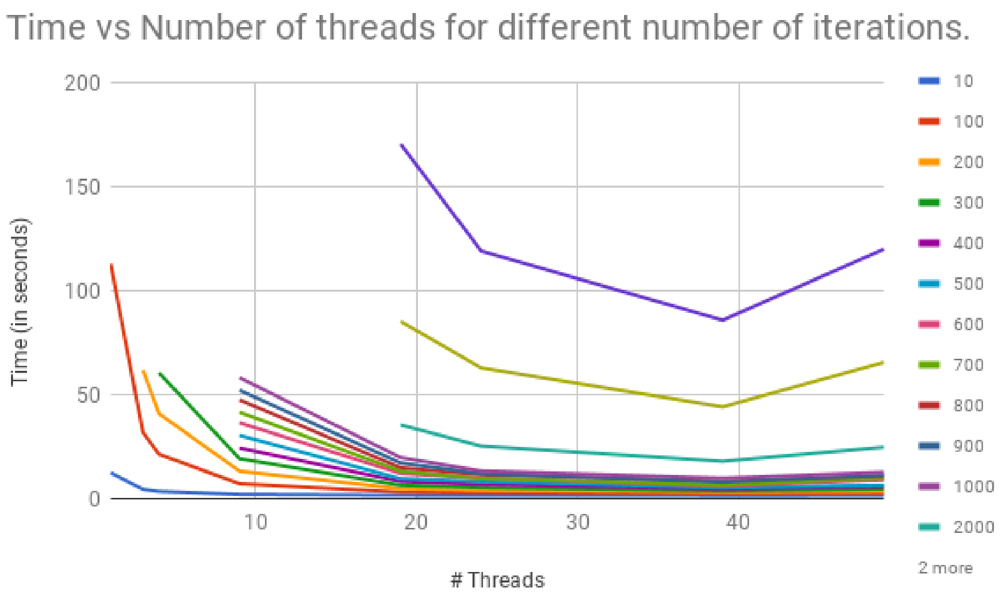

Figure 2: 1000*1000 matrix factorized by latent dimension 100

8 Future Work

- Use Streaming SIMD Extensions instructions on updating arrays.
- Use blocking for minimizing cache-read misses/ align memory on.
- Align the matrices on cache lines.

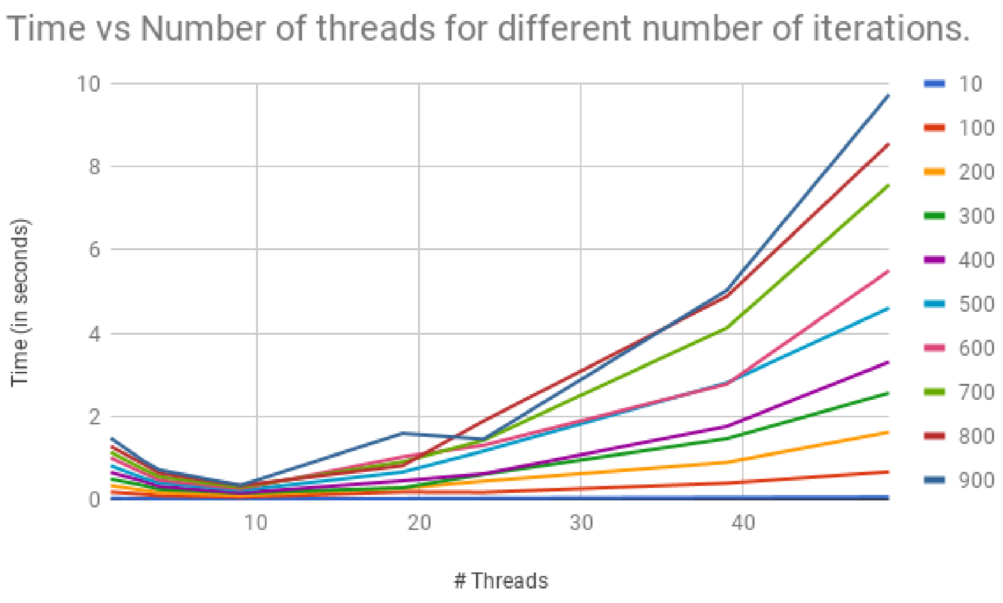

Figure 3: 100*100 matrix factorized by latent dimension 10 (cache inefficient)

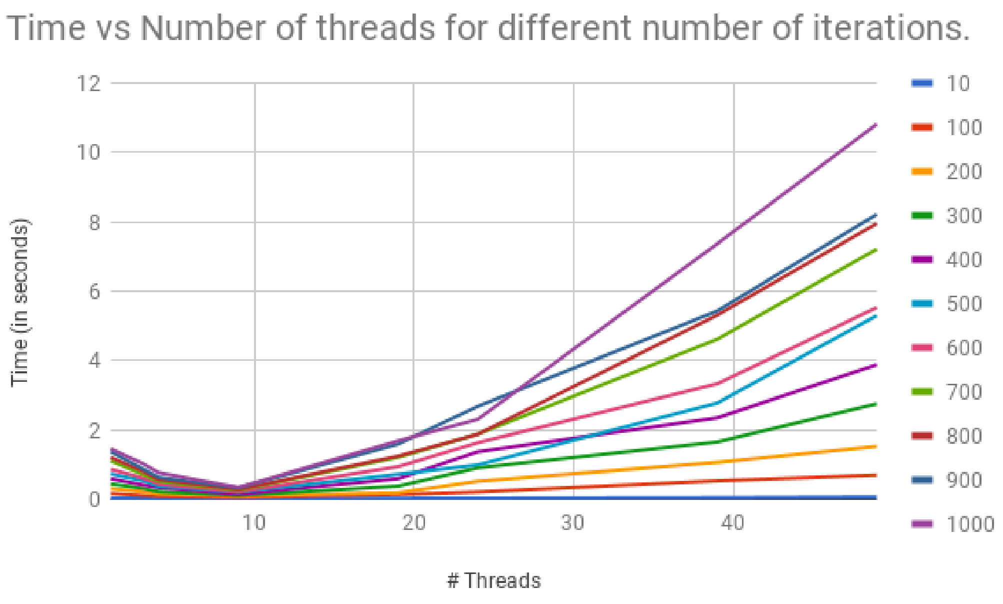

Figure 4: 100*100 matrix factorized by latent dimension 10 (cache efficient)

9 Conclusion

We conclude that as we increase the number of threads, the dimension of the blocks processed by each thread reduces, thus the time per iteration goes down. At the same time, at every iteration, lesser number of the matrix's entries are processed. This means that the error per block reduces faster. Overall, looking at all these factors combined, we conclude that as we increase the number of threads we get the factors with the same error margin much faster.

10 Source code

All the source code and data files used can be found on github. It has been ensured by the authors that the contribution to the project made by each author is equivalent. There has been contribution by both authors in each of the functions enlisted below but for the sake of division more fine contribution can be attributed as follows.

Arnav:

- Data Handling
  1. read_matrix()
  2. dump_matrix()
  3. random_data()
- SGD
  1. calc_error()
  2. factorize_block()
  3. random_data()

Ojas:

- Scheduler
  1. init_sched()
  2. push_block()
  3. get_block()
  4. launch_sched()
- SGD
  1. matrix_factorize()
  2. main()

References

[1] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(1):2, 2015.

[2] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.

[3] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 765–774. IEEE, 2012.

[4] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.