# The Netflix Prize: Alternating Least Squares in MPI

Sean Harnett

March 5, 2010

## 1 Introduction

In 2006 Netflix announced a million dollar prize to the first team that could beat their Cinematch recommendation system by 10% on a particular test data set. Specifically, given over 100 million ratings, 1-5, from 480,189 different users and 17,770 different movies, the goal was to produce predictions for the test set that minimize the root mean square error. Cinematch scored an rmse of .9525, so the goal was to score less than or equal to .8572. My goal in this project was to simply beat Cinematch, in parallel.

The prize was won in September, 2009 by a team using a blend of many different techniques. One such technique that played a large part in the winning blend was matrix factorization. The 480,000x17,000 (sparse) ratings matrix, $R$, is approximated by a product of two much smaller matrices. A singular value decomposition can produce the two matrices and conveniently minimizes the square norm. After choosing a number of "features", f, you want an fx480,000 user matrix $U$ and an fx17,000 movie matrix $M$ whose product, $U^T M$ comes as close as possible to the given training data in $R$. A number of algorithms exist to find these two matrices; I used an approach called alternating least squares with weighted-$\lambda$-regularization. [1]

## 2 My Approach

I started with freely available code [2] that used a stochastic gradient descent algorithm to find $U$ and $M$. I modified this to use the ALS approach detailed in [1], then parallelized it with MPI. The ALS algorithm is rather simple conceptually: initialize $M$, and solve a linear algebra problem for $U$, holding $M$ fixed. Next hold $U$ fixed and solve for $M$. Repeat until convergence.

The parallelization is also straightforward: the linear algebra problem to solve for $U$ can be done column by column; in other words, solving for the features for each user is independent of the other users. Thus the $U$ matrix can be partitioned by columns across many processes, solved for, and then a simple "allgather" to collect the updates is sufficient to prepare for the next step of solving for $M$. And likewise when solving for $M$. The data for the 100 million ratings can be distributed to just the processes that need it for their updates, and needs no communicating. Thus, the only communication is the "allgathering" of the feature matrices $U$ and $M$ at each iteration.

See [1] for the details of the linear algebra problems. To solve for column $i$ of $U$, the relevant paragraph is this:

$$u_i = A_i^{-1} V_i \quad \forall i$$

> where $A_I = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E, V_i = M_{I_i} R^T(i, I_i)$, and $E$ is the $n_f \times n_f$ identity matrix. $M_{I_i}$ denotes the sub-matrix of $M$ where columns $j \in I_i$ are selected, and $R(i, I_i)$ is the row vector where columns $j \in I_i$ of the $i$-th row of $R$ is taken.

Here, $n_f$ is the number of features. Solving for $M$ is done similarly. There are only three operations:

1. a symmetric matrix-matrix multiply then sum ($A_I = M_{I_i} M_{I_i}^T + \lambda n_{u_i} E$)
2. a matrix vector multiply ($V_i = M_{I_i} R^T(i, I_i)$)
3. a linear solve with a symmetric matrix ($u_i = A_i^{-1} V_i$)

I used BLAS for the first two and LAPACK for the third. The sequential code is quite fast and produces predictions that beat Cinematch in under 20 minutes of run time, signicantly faster than the stochastic gradient descent code in [2].

## 3 Organizing the data

### Text files

The data from Netflix is given in thousands of text files, one per movie. This of course was painfully slow to load, so it was necessary to create binary files with the data. I created several of these. The first was for the data in the original, movie-sorted order. This was in the form of an array, $R_1$, of "Data" structs, which contained the customer ID, the movie ID, and the rating. The ALS algorithm also needs the data in customer-sorted order, so I made one of those as well, $R_2$. The sort in <algorithm.h> was fine for this as a one-time job; sorting each time was too slow when using all the data, but was ok when testing on just a fraction of the data.

[2] also used an array of movie metrics (number of ratings, average rating, etc) and user metrics, which I continued to use for convenience. These resulted in small binaries, less than 6MB total.

### $R$ isn't actually a matrix

Because the conceptual 480000x17000 (roughly 8 billion) ratings matrix is impractical to actually store like that, [2] used an array of structs containing only the 100 million actual ratings. Unfortunately, distributing this array in an intelligent way for use with the ALS algorithm was a bit tricky. I assigned each process a range of movies and a range of users it was responsible for updating. So each process needed a corresponding piece of the movie-sorted ratings array, $R_1$ and the user-sorted ratings array, $R_2$.

To help manage this, I created arrays of indices, that reported where in $R_1$ or $R_2$ each new movie or user began. For example, say processor 3 is responsible for movies 3500-4999. I can use movieIndexArray[3500] and movieIndexArray[5000] to find the piece of $R_1$ that processor 3 needs. Perhaps movieIndexArray[3500] is 30 million, and movieIndexArray[5000] is 40 million.

Then the 1500 movies for which processor 3 is responsible have a total of 10 million ratings between them. The corresponding data can be found in $R_1$[30 million] to $R_1$[40 million].

I also used the index arrays to determine the size of the ratings arrays. In full, they are 800MB each, so it wasn't realistic for each process to have a full size array, considering it only needs a fraction. I allocated these dynamically on each process after computing the size of each piece from the index arrays.

### (not) Broadcasting the ratings data

In [1] they loaded the data on one process and broadcast it. On catapult, this approach was prohibitively slow for the ratings data (though I did this for smaller things, such as the index arrays). Broadcasting on just 10% of the 800MB ratings array took several minutes. I didn't bother trying it on all of the data. Instead I had the processes take turns loading from the same binary file, using the idea Francois posted on the message board for "writing stuff in order". This approach resulted in loading/broadcasting time for all the data of around ten seconds. I'm not even sure it was necessary for the processors to take turns; I assumed having them all try to load different parts of the same file simultaneously wouldn't work, but I didn't bother trying since it was so fast already.

### Leftovers

Of course the number of users and movies isn't a convenient multiple of all the small integers I'd like to use as the number of processors. So there is no perfectly even splitting of $U$ and $M$ across the processors. My solution to this was simply to tack on the remainder to the last process. For instance, if I wanted four processes, I'd assign the first three 4442 movies, and the fourth process gets 4444, for a total of 17770. I created what I thought was some clever logic to handle the communication for this, when I realized it must be a common problem with a solution. Indeed, MPI_Allgatherv was what I ended up using.

## 4 Pseudocode

The overall picture looks like this:

```
begin
    if (root)
        load index, movies, and users arrays
        intialize movie features matrix M
    endif
    broadcast
    load ratings arrays R1 and R2
    loop n times (or until convergence)
        solve for user features matrix U
        allgather
        solve for M
        allgather
    endloop
    compute predictions and rmse
end
```

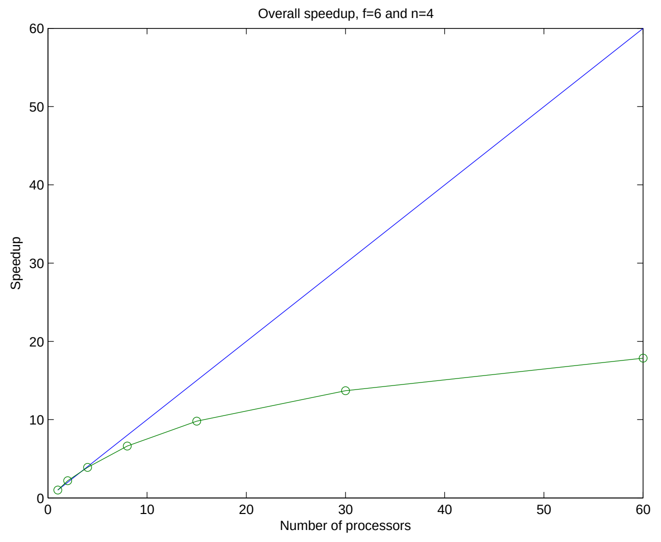

Figure 1: Total time is dominated by the overhead for a small problem; poor speedup

The loading using Francois' idea:

```
for i = 0 to numjobs
    if (rank==i) load R1 and R2
    MPI_Barrier()
end
```

## 5 Results

### RMSE

I was able to beat Netflix in under a minute. Using 6 features, 4 iterations of ALS, and 60 processes, the algorithm produced an rmse of 0.9511, just edging out Cinematch's score of 0.9525. Loading took 8.0 seconds, ALS 18.1 seconds, and computing the prediction set 33.5 seconds, for a total of 59.6 seconds.

Going for the kill with 100 features, 40 iterations, and 60 processes produced an rmse of .9278 in just under two hours. [1] did a bit better for this number of features, but they ran the algorithm until it stopped improving instead of a fixed number of iterations.

Relatively simple methods to further reduce rmse include bias correction and blending. Bias correction is simply shifting each prediction from the model by a constant, so that the model's average rating aligns with the average rating of the data set. Blending is taking two or more sets of predictions and forming the linear combination that minimizes rmse. I didn't investigate either of these techniques.

### Speedup

Speedup was very close to linear for the ALS iterations. There is a fixed overhead for loading and processing of around 45 seconds. The only other factor preventing perfect linear speedup was the small smount of communication at each iteration of ALS. For the small problem of 6 features and 4 iterations, the overhead dominated the total time when using many processors,

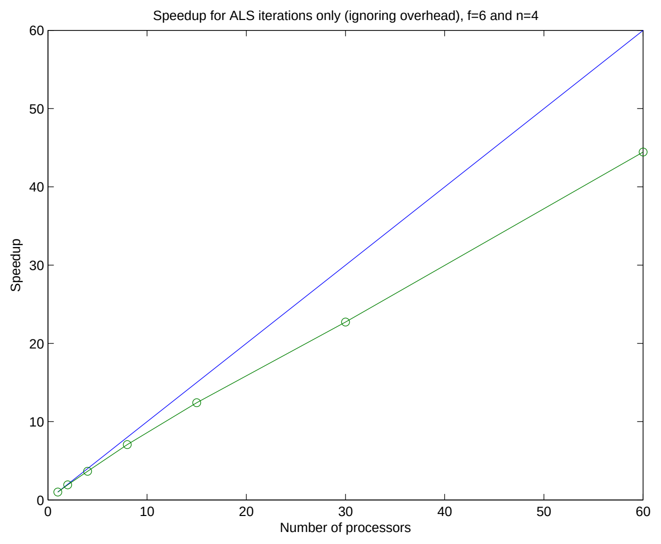

Figure 2: Ignoring the overhead, speedup is close to linear

and speedup was poor (Figure 1). In the worst case, as described above, ALS took only 18 seconds compared to the 41 seconds of overhead. But ignoring the overhead, speedup was almost linear (Figure 2).

For a larger problem of 40 features and 12 iterations, this overhead issue goes away. Figure 3 shows the speedup curve relative to p=2. The sequential version for this problem size took unusually long; I suspected this was just an anomaly, but running it twice resulted in times of roughly 12 hours in each case. Figure 4 shows the amazing superlinear speedup computed using this sequential time.

## 6 Miscellaneous Thoughts

### Customer ID distribution

The data is given by Netflix sorted by movie, with movie IDs between 1 and 17770. But the customers are in no order, and furthermore have IDs ranging in the millions, for only 480189 customers. So [2] remapped these IDs to the compact range 1-480189, storing the old IDs in a hash table, and used the new IDs as array indices. The mapping algorithm simply assigned the compact IDs in ascending order as the program scanned through the data. This naturally resulted in the low ID users having many, many ratings while those with high IDs having very few. Using a naive splitting of the user matrix with this distribution would result in a terrible load imbalance. My solution: continue with the naive splitting, but shuffle the mapping. If you look at my code listing, you'll see a function "shuffle" I found with google that worked nicely.

### Custom datatypes in MPI

This was necessary for communication using the Data, Movie, and Customer structs I used. I found the available documentation to be lacking in good, simple examples on how to create custom MPI datatypes. I got something to work, but I don't really understand what it's doing or why it's so complicated.

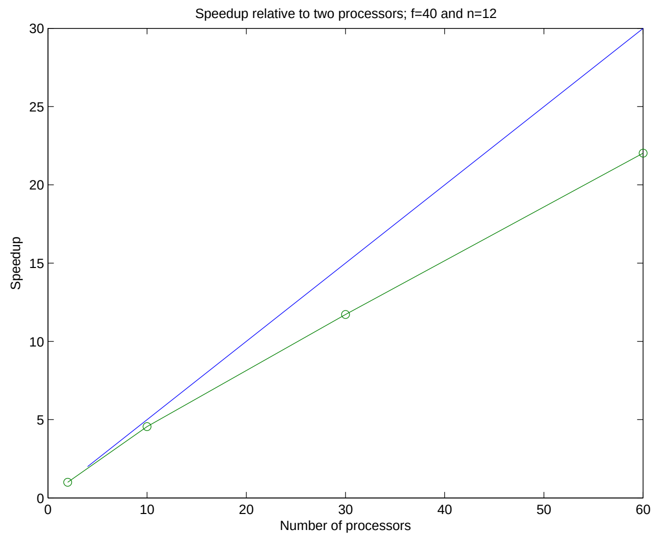

Figure 3: Compare to figure 2; for the larger problem, overhead is negligible

## LAPACK in C++

This took some figuring out. Catapult is equipped with cblas, so using the BLAS routines was as simple as #include<cblas.h> and compiling with the -lcblas flag. Unfortunately it doesn't seem to have something equivalent for LAPACK; it took some time to figure out the necessary compilation flags and how to declare the routine in my code. The routines themselves were a little tricky to wrap my head around as well; I needed to play around with them on some toy matrices for a while before I figured out how to use them correctly.

## Debugging

Apparently using something like gdb with MPI code is possible, but it looks pretty involved, requiring a decent amount of knowledge of the system you're trying to use it on. So I had to resort to lots of screen and file print-outs to debug, which was cumbersome to say the least. In fact, the main lesson I've taken away from this project is that I seriously need to learn C++, good coding practices, and how to use real development tools. Debugging was an absolute nightmare at times, and I can only imagine that someone who actually knew what they were doing would laugh at the silly hacks I strung together to get things to finally work.

## Machine Learning

As I was mainly concerned with performance issues, I took some shortcuts. I iterated ALS a fixed number of times, instead of looping until convergence. Ideally, you'd set a tolerance to some minimum level of improvement, and keep repeating until the improvement in rmse is below that tolerance. This of course requires computing the rmse at each iteration, slowing down the algorithm. I also trained on all of the given data. I should have trained on only a portion (say 90%) and then tested that model against the remainder (10%). This is known as cross-validation and is used to prevent overfitting of the model to the training data. The parameters that result in the lowest rmse againt the *testing* set (10%) are the best, and would be submitted to Netflix for the contest.

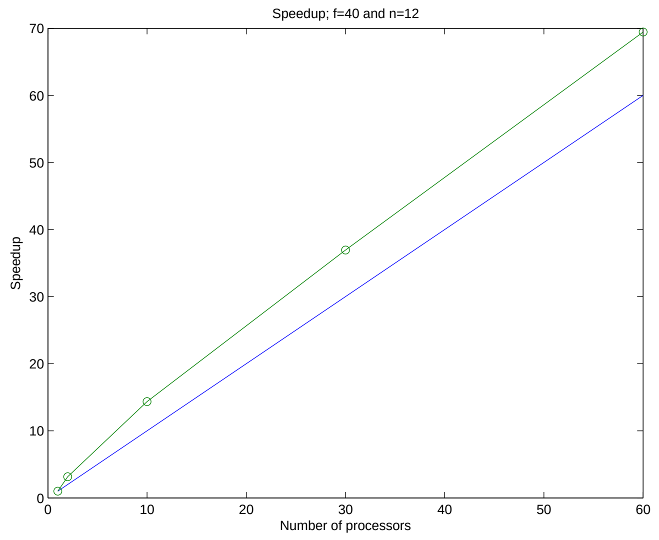

Figure 4: Superlinear speedup; something was likely wrong with the sequential timing

## 7 Conclusion

I was able to achieve my goal of beating Cinematch in parallel, and I was surprised at how little computational time it took to achieve that score. Being able to beat Netflix in 60 seconds suggests that perhaps their algorithm before this contest wasn't very good. Though Cinematch of course has other motivations besides strictly minimizing rmse.

In any case, the ALS method produces a low rmse and seems to parallelize very well. For Netflix, this scaling property is a necessity as they must have billions of data points to consider when making their movie recommendations. A sequential algorithm on a single machine is simply too slow; getting a low rmse even with 60 processors took hours of computation. Indeed, from the Netflix Prize FAQ:

> And while we are on the topic of system performance we should note that it took Cinematch a non-trivial amount of time to produce its predictions for the quiz and test subset. And were not even talking about the time it took to learn the training dataset. Think days. Were talking serious horsepower here. Just a heads up.

So 60 seconds was pretty good.

## References

[1] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM, LNCS 5034*, pages 337-348, 2008.

[2] "Netflix prize results and source code." Timely Development, LLC. 2008. Web. <*http://www.timelydevelopment.com/demos/NetflixPrize.aspx*>.