Scaling Collaborative Filtering with PETSc

Alister Johnson
Department of Computer and Information Sciences
University of Oregon
ajohnson@cs.uoregon.edu

***Abstract*—Machine learning and recommendation systems have become extremely popular and widely used in recent years, with several major companies using recommenders to help their users sort through the vast array of products offered. Naturally, we want these recommender systems to give accurate predictions of products a user might like, but we also want these predictions quickly, based on the user's past preferences and newer preferences they may have just expressed. Many recommendation algorithms have been created, including neural networks, nearest neighbor algorithms, and various latent factor models. While neural networks have received the most attention lately, these other algorithms can produce results that are just as accurate, and in many cases better understood. This paper explores how an optimized, highly scalable recommendation system can be written using scientific libraries and the scalability of such a the system by implementing implicit alternating least squares with PETSc.**

***Index Terms*—recommender systems, PETSc, alternating least squares, collaborative filtering**

I. Introduction

Recommender systems use machine learning techniques to make sense of user preference data, such as product ratings, website page views, or TV shows watched. At a high level, recommender systems try to find patterns in users' preferences for items, like movies or other products, and predict more items they might like. Recommender systems are most commonly used by online retailers or streaming services to help users find new products among the wide variety that are available.

There are many kinds of recommender systems, the main two being latent factor models and neighborhood-based models. Latent factor models, such as matrix factorization and principal component analysis, try to find underlying characteristics or properties that can accurately describe users' preferences. Matrix factorization, for example, tries to decompose a matrix of user preferences for items into two latent factors matrices, one for users and one for items, where each user and item is described by a vector containing its various properties. Each factor corresponds to a specific property, such as whether a movie is a comedy or tragedy, although these properties might not have any easily understood meaning.

Neighborhood-based models don't try to discover any underlying logic in the system, but operate on the principle that similar people usually like similar things. For example, if there are two people who both like X, Y, and Z, and one of them also likes W, it's a good bet the other will like W, too. These models try to compute similarity scores for users, based on how many of the same things each has expressed a preference for, and recommend items other similar users like. Variations on this model find similar items to recommend or use some combination of user and item similarities.

Regardless of how our recommender system works, we want it to work fast. Our recommendations do us no good if we get them after the user has moved on to other things. We want to be able to train our system quickly, especially at the start when we have very little data to work with, but also when we have a large database of preferences, which is the more challenging problem.

We can get a head start, so to speak, on optimizing and scaling our recommender system by using scientific libraries to write it. Many recommender systems rely on linear algebra operations, as do several scientific computing applications, so scientific libraries have the basic algorithms we need and are already heavily optimized. We can take advantage of that to avoid optimizing low level code and instead spend our time optimizing the main algorithm. Libraries like PETSc that are built on a communication system such as MPI can also give us optimized communication for "free," again leaving us with more time to work on the rest of the recommender system. Since these libraries are designed to be run on highly parallel systems, we want to choose an easy-to-parallelize algorithm, such as implicit alternating least squares (iALS). This paper uses PETSc to implement an optimized, highly scalable version of iALS.

The contributions of this paper are as follows.

- A high performance, scalable implementation of iALS using PETSc and Python.
- Benchmarks of this implementation, showing that it possesses good strong scaling.
- Optimizations to the iALS algorithm that have not been previously published, to the author's best knowledge.

The rest of this paper is organized as follows. Section II discusses background information about the iALS algorithm and libraries involved. Sections III and IV describe the implementation and how it was optimized. Section V describes scalability and accuracy results, and finally Section VI contains references to related research.

II. Background

A. Collaborative Filtering

Collaborative filtering is a class of algorithms used for making recommendation systems. In general, for any given user, these algorithms find other users with similar tastes and recommend items well-liked by those users. For example, if you buy a keyboard from an online shopping website, the site may start recommending other products commonly bought with a keyboard, like a desktop monitor or mouse. Other variants of the algorithm will instead find similar items to things a user has liked, so if you bought multiple books by a particular author, the recommender would begin showing you more books by that author. Some algorithms, like the one implemented in this paper, use a combination of both.

1) Implicit Alternating Least Squares: The specific algorithm implemented in this paper is implicit alternating least squares (iALS). iALS is a collaborative filtering algorithm designed to work with implicit data. Implicit data by nature contains uncertainty, since it may not reflect users' true preferences. Instead, it contains information about what a user has seen, like page views or products they've bought, but not necessarily whether they liked it. However, we can associate confidences with these values to represent this uncertainty. For example, if a user viewed a web page twice, we can assume they like it, but only with a low confidence; if they viewed the page twenty times, we can say they like it with high confidence.

At a high level, iALS decomposes the matrix of user-item preferences into two matrices of latent factors, one for users and one for items. It alternates between fitting the user and item factors matrices to the data, which avoids overfitting on either one. This algorithm was first described by Koren, Hu, and Volinsky in [1]. In their original paper, Koren et al. denoted the user factors matrix as X and the item factors matrix as Y, and recomputed each row x_u of the user factors matrix with

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u),$$

where C^u is a diagonal matrix with user u's confidence values along the diagonal, $p(u)$ is a vector of the user's preferences, λ is a scalar regularization term, and I is the identity matrix. Each row of the item factors matrix is computed analogously. User u's confidence values for each item i are a function of their observed preferences, as described earlier. One such function proposed by Koren et al. is

$$c_{ui} = 1 + \alpha \log(1 + r_{ui}/\epsilon),$$

where c_{ui} is the confidence value for user u and item i, α and ϵ are scaling factors, and r_{ui} is a binarized preference value (1 if a user has viewed an item, 0 if they have not), but many other functions are also suitable.

Using this confidence function, as Koren et al. did, the diagonal of C^u is generally not sparse, so computing $Y^T C^u Y$ is expensive. To optimize this multiplication, they instead compute $Y^T Y + Y^T(C^u - I)Y$, which is equivalent, and has the advantage of allowing them to pre-compute $Y^T Y$ for all rows of X, since that term does not depend on the user. Since c_{ui} is 1 when a user has not viewed an item, $C^u - I$ is sparse and fewer operations are required to compute $Y^T(C^u - I)Y$. Pseudocode for this algorithm can be seen in Alg. 1, where $conf$ is a confidence function applied to the user's preference vector, and $diag(M)$ denotes the diagonal of matrix M.

Algorithm 1 Original iALS

```
user-items = $users \times items$ matrix {preferences matrix}
item-users = user-items$^T$
$X$ = $users \times factors$ matrix {user factors}
$Y$ = $items \times factors$ matrix {item factors}
for $i = 0$ to max-iteration do
  calculate $Y^T Y$
  for $x_u$ in $X$ do
    $C^u$ = $items \times items$ matrix
    $diag(C^u)$ = $conf$(user-items[$u$])
    $A = Y^T Y + Y^T(C^u - I)Y + \lambda I$
    $b = Y^T C^u \cdot$ user-items[$u$]
    solve $Ax = b$ for $x$
    set $x_u = x$
  end for
  calculate $X^T X$
  for $y_i$ in $Y$ do
    $C^i$ = $users \times users$ matrix
    $diag(C^i)$ = $conf$(item-users[$i$])
    $A = X^T X + X^T(C^i - I)X + \lambda I$
    $b = X^T C^i \cdot$ item-users[$i$]
    solve $Ay = b$ for $y$
    set $y_i = y$
  end for
end for
```

B. PETSc

PETSc [2], [3] is the Portable, Extensible Toolkit for Scientific Computation, a library of high performance, scalable data structures and algorithms for scientific applications. PETSc includes MPI algorithms, as well as GPU algorithms using CUDA or OpenCL and hybrid MPI-GPU algorithms. There are also Python bindings for PETSc, provided by petsc4py [4], which were used to implement iALS for this paper. PETSc contains linear algebra routines, ODE solvers, and linear solvers and preconditioners. PETSc was chosen because it is extremely high performance and widely used in scalable applications. This paper primarily uses PETSc's matrix and vector algebra routines.

III. Optimization

A. User Preferences as a Proxy for Confidence

As mentioned earlier, in the original paper, Koren et al. optimized their computation of $Y^T C^u Y$ by transforming it into $Y^T Y + Y^T(C^u - I)Y$, which has a sparse diagonal for the center matrix. However, if we use user preference values instead of confidence values for the diagonal of C^u, this optimization becomes unnecessary. Confidence values are meant to express uncertainty in our data, but the data itself can do this in many ways. For example, if a cable company is keeping track of how many hours of any given TV show a user has watched, they can conclude that the show that user has watched 500 hours of is one that user likes, and the show they've only watched 5 hours of is one they don't care for. In general, the more a user views an item, the more confident we can be that they like it, so users' view counts can be used as both preferences and confidences.

When we put these preference values in the diagonal of C^u, we can substitute zeros for preferences we have no data for, and the diagonal of C^u becomes sparse. This allows us to further optimize computing $Y^T C^u Y$.

B. Matrix Reduction

This optimization turns a very large, sparse matrix-matrix multiplication into a comparatively small, dense multiplication, and greatly reduces the number of FLOPS wasted on multiplications by zero. It also replaces a very expensive transpose operation (transposing all of Y) with many much smaller transpose operations, which may be more efficient, depending on the relative sizes of Y and the various reduced Ys. Even if these many small transposes are less efficient than a single large transpose, the gains from smaller matrix-matrix multiplications more than make up for it. The fill factors for these user-item matrices are usually very low – the average person can only view a very small percentage of the products on offer – so the fill factor for the diagonal is similarly very low. The two data sets used by this paper both had user-item fill factors of 1% or lower. The dense multiplication is therefore usually on the order of 1% the size of the original, sparse multiplication.

When we multiply a matrix on the left by a diagonal matrix, this has the effect of scaling the rows of the original matrix by the values on the diagonal. If many of those diagonal values are zero, this will zero out several rows of the original matrix. Furthermore, if we right-multiply a matrix by another matrix with many zero rows, the corresponding columns of the original matrix will only be multiplied by zero. We can optimize this multiplication by only multiplying the non-zero rows and their corresponding columns, since any multiplication by zero is a waste of computing power. We use this to compute $Y^T C^u Y$ as follows.

First, we remove all the rows of Y that would become zero and only keep a copy of the others. We then transpose this reduced Y, since those rows are exactly the columns of Y^T that would not become zero when multiplied by $C^u Y$ – we must reduce both Y and Y^T so the inner dimensions match for the last multiplication. We compute $C^u Y$ by scaling our reduced Y by the non-zero values in C^u, avoiding many multiply-by-zeros. Finally, we multiply our reduced Y^T by the reduced $C^u Y$ to get $Y^T C^u Y$.

A small example of this is shown in Fig. 1, where Y consists of row vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$ of length n, and C^u has two zeros and a one on the diagonal. Since $\vec{x}$ and $\vec{y}$ become zero vectors, the full multiplication is equivalent to a smaller multiplication with only $\vec{z}$ and its transpose.

The full iALS algorithm, including this optimization, is shown in Alg. 2.

$$\begin{bmatrix}\vec{x}^T & \vec{y}^T & \vec{z}^T\end{bmatrix}\begin{bmatrix}0 & & \\ & 0 & \\ & & 1\end{bmatrix}\begin{bmatrix}\vec{x}\\ \vec{y}\\ \vec{z}\end{bmatrix} = \begin{bmatrix}\vec{x}^T & \vec{y}^T & \vec{z}^T\end{bmatrix}\begin{bmatrix}\vec{0}\\ \vec{0}\\ \vec{z}\end{bmatrix} = \begin{bmatrix}z_1\vec{z}\\ z_2\vec{z}\\ \vdots\\ z_n\vec{z}\end{bmatrix}$$

$$\vec{z}^T \cdot 1 \cdot \vec{z} = \begin{bmatrix}z_1\\ z_2\\ \vdots\\ z_n\end{bmatrix} \cdot \vec{z} = \begin{bmatrix}z_1\vec{z}\\ z_2\vec{z}\\ \vdots\\ z_n\vec{z}\end{bmatrix}$$

Fig. 1. A small example of the optimization described in Sec. III-B. The vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$ are row vectors of length n (thus their transposes are column vectors of length n).

C. Parallelization

The original algorithm described by Koren et al. is serial, as is the version described in Alg. 2. This algorithm is easily parallelizable by noting that calculating each row of the user factors matrix is independent of calculating any other row of the matrix, and the same is true for the item factors. We can thus distribute the rows of the user and item factors matrices between processes, so these recalculations can be done in parallel. Afterwards, the full matrices can be reconstructed by broadcasting each processes' rows to all the other processes.

Algorithm 2 Serial iALS

```
user-items = $users \times items$ matrix {preferences matrix}
item-users = user-items$^T$
$X$ = $users \times factors$ matrix {user factors}
$Y$ = $items \times factors$ matrix {item factors}
for $i = 0$ to max-iteration do
    for $x_u$ in $X$ do
        $r$ = number of non-zeros in user-items$[u]$
        $C^u$ = $r \times r$ matrix
        $diag(C^u)$ = non-zeros in user-items$[u]$
        $Y'$ = $\{Y[i]$ | user-items$[u][i]$ is non-zero$\}$
        calculate $Y'^T$
        $A = Y'^T C^u Y' + \lambda I$
        $b = Y'^T C^u \cdot$ user-items$[u]$
        solve $Ax = b$ for $x$
        set $x_u = x$
    end for
    for $y_i$ in $Y$ do
        $r$ = number of non-zeros in item-users$[i]$
        $C^i$ = $r \times r$ matrix
        $diag(C^i)$ = non-zeros in item-users$[i]$
        $X'$ = $\{X[u]$ | item-users$[i][u]$ is non-zero$\}$
        calculate $X'^T$
        $A = X'^T C^i X' + \lambda I$
        $b = X'^T C^i \cdot$ item-users$[i]$
        solve $Ay = b$ for $y$
        set $y_i = y$
    end for
end for
```

IV. Implementation

Scientific libraries provide many benefits for implementing programs like this one, beyond the built-in parallelism mentioned earlier. We can save a great deal of time by using a library with the basic operations we need, since we can trust that those operations are correct and already optimized, and spend that time on higher level optimizations, such as the ones just described. Scientific libraries often come with optimized data structures as well, which saves us the time of implementing our own. This section describes how PETSc's linear algebra operations and matrix data structures were used to implement iALS.

A. Mapping iALS to PETSc

All the operations of iALS are linear algebra. The two most computationally expensive are the Y^TC^uY multiplication and solving the final linear system to recalculate each row of X and Y. In this implementation, all of the matrices were one of PETSc's sequential matrix types (as opposed to the MPI versions[1]), and parallelism was instead implemented with mpi4py [5]–[7]. Each processor owns a block of rows in both the user and item factors matrices, and after the recalculation of that matrix is done, each process broadcasts its new values for its rows to all the other processes.

The Y^TC^uYmatrix multiplication was implemented with only a single full matrix multiplication; the C^uY part is done using PETSc's `diagonalScale` operation, which is much more efficient than a full matrix multiplication. Instead of doing a full multiply, it simply scales the rows or columns of the dense matrix by the corresponding elements on the diagonal of the diagonal matrix, eliminating all the multiply-by-zeros that would have resulted from the off-diagonal elements of the diagonal matrix. Since the matrix reduction optimization described above removes all zeros from the diagonal, this effectively removes all zero multiplications from the Y^TC^uYcalculation.

The linear solver used to recalculate the user and item factors is PETSc's Cholesky direct solver. The linear system is a square matrix of size $factors \times factors$; since $factors$ is usually chosen to be a comparatively small number, on the order of one hundred, and usually not more than one thousand, a direct solver will be just as accurate and likely faster than an iterative solver. Cholesky solvers are designed to work with real, symmetric, positive-definite matrices. $Y^TC^uY$is certainly real, so we must only show that the system is symmetric and positive-definite. Consider a diagonal matrix C, whose diagonal contains the square roots of the diagonal of C^u, and let $A = CY$. The diagonal entries of C will scale the rows of Y, and, similarly, if we right-multiply Y^T by C, the diagonal entries will scale the columns of Y^T. This means that $Y^TC = A^T$, so $Y^TC^uY = (Y^TC)(CY) = A^TA$. The product of a matrix and its transpose is always symmetric, therefore $Y^TC^uY$is symmetric. Furthermore, since it is unlikely that any of the elements of Y are zero, and the matrix reduction optimization removes all zeros from C^u, A is also likely non-singular (i.e., its determinant is non-zero). Since A is non-singular, $A^TA = Y^TC^uY$ is positive-definite, because for any non-zero column vector z, $z^TA^TAz = (A^Tz)^T(A^Tz) > 0$, the definition of non-singularity. This product is strictly greater than zero because it is equivalent to the dot product of A^Tz with itself. It will always be greater than zero if A^Tz is non-zero, which it is. Since $Y^TC^uY$is real, symmetric, and positive-definite, a Cholesky solver is appropriate.

[1]PETSc's MPI matrix types distribute the matrix across all the processors, but iALS requires both X and Y to be completely resident on every processor for each iteration.

B. Memory Management

Although Python is increasingly popular in data science and machine learning, which both often require applications to be high performance, Python is not a high performance language in and of itself. This can become a problem when working with large data sets, like the Netflix Prize data set. In these cases, it can be useful to fall back on memory management efficiency techniques that are commonly used in high performance languages like C.

One such technique is pre-allocating arrays, instead of the Python idiom of creating an empty list and appending. Repeatedly appending to a list in Python is slow and memory inefficient, since Python over-allocates when growing an array. When that array gets large, the over-allocation can be far more than necessary, to the point where the program uses two times more memory than is truly needed. C-style memory allocation can be achieved using the NumPy package [8], which provides array data structures and operations and is written in C to improve efficiency.

The other main memory management technique used in this implementation was storing data, both in memory and on disk, in CSR format. CSR format is used to store sparse data, and keeps the data in three one-dimensional arrays, two of which store the non-zeros and their column indices, and the third of which indexes into the first two, showing where each new row starts. PETSc provides an "AIJ" matrix type that stores sparse data like this, but it can also be advantageous to read in the data this way. Using CSR format can improve both run-time memory usage and I/O times, since less data needs to be stored and read in at start-up.

V. Experimental Results

A. Data Sets

The first data set used is a collection of link counts between pages on TVTropes.org, scraped from the website by the author in January and February of 2018. The pages are divided into two categories, which are works of fiction, such as movies, books, and TV shows, and tropes, which are narrative devices used in those works, such as common plot points, character archetypes, and setting features. In the context of a recommender system, works are analogous to users and tropes are analogous to items. The user-item matrix contains link counts between work and trope pages and is approximately square, with 63,705 works and 60,290 tropes. The matrix contains almost 4 million nonzero values with a fill factor of about 0.1%, making it the smaller of the two data sets. This data set was used for initial scaling tests.

The second data set is the well-known Netflix Prize data set, kindly made available by Netflix in 2006 [9]. The Netflix data set contains around 100 million user ratings of movies from the early 2000s. This user-item matrix is rectangular, with 480,189 users rating 17,770 movies, and its fill factor is approximately 1%. This data set was used to further test the scalability of the recommender, since it is about 25 times larger than the TVTropes data set, as well as the accuracy of the recommender, because it is well known and often used for accuracy tests.

B. Efficiency

All the following experiments were run on nodes of a cluster with dual Intel E5-2690v4 processors (28 cores total) and 128 GB of memory per node. PETSc version 3.8.4 was used, and compiled with debugging turned off and –O3 optimization. The timing information was collected using PETSc's built-in logging stages, and includes separate times for computation and communication, to show the increase in overhead for adding more cores and/or nodes. I/O and data initialization times are not shown because they are generally negligible compared to the total run time (usually less than 5%, often less than 1%), and they are constant, compared to the compute and communication portions of the algorithm.

1) Timing Comparisons: Each timing test was done with 350 factors and 20 iterations, unless stated otherwise. In practice, the optimal number of factors can vary per data set, although more factors will of course give a better fit, at the risk of overfitting. We chose 350 because it comes close to using up all the available memory per node with the Netflix data set, and does in fact sometimes cause nodes to run out of memory when all cores are used. Koren et al.'s implementation usually converged after about 10 iterations [1], however since this implementation has none of their accuracy improvements, it can take longer to converge, hence these tests being run for 20 iterations.

The training times of iALS on the TVTropes and Netflix data sets are shown in Figs. 2 and 3, respectively. As can be seen, the benefits of using additional cores diminish more quickly for the smaller TVTropes data set, while the Netflix data set continues to scale well until the node runs out of memory. The communication overhead for the Netflix data is also significantly higher than the overhead on the TVTropes data, although this is only because the Netflix data set is so much larger, and the overheads are roughly constant across higher numbers of cores for each data set.

As well as the version described in Sections III and IV, whose run times are displayed in Figs. 2 and 3, two other versions of iALS, one based closely on Koren et al.'s original algorithm, and another with only the optimization described in III-A, were implemented with the intention of comparing them with the fully optimized version. However, even after giving both these versions twice the amount of time allotted

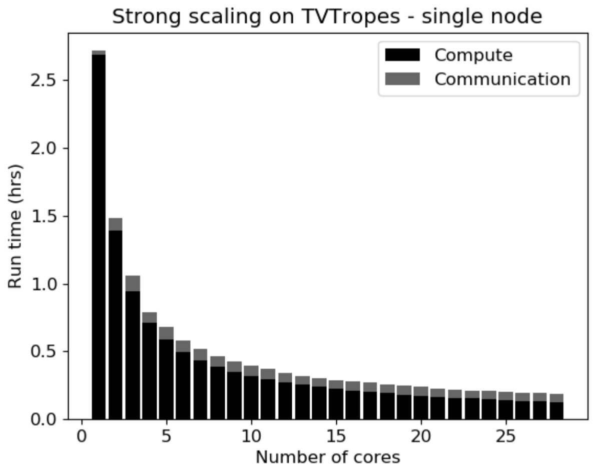

Fig. 2. Training times on the TVTropes data using various numbers of cores on a single node.

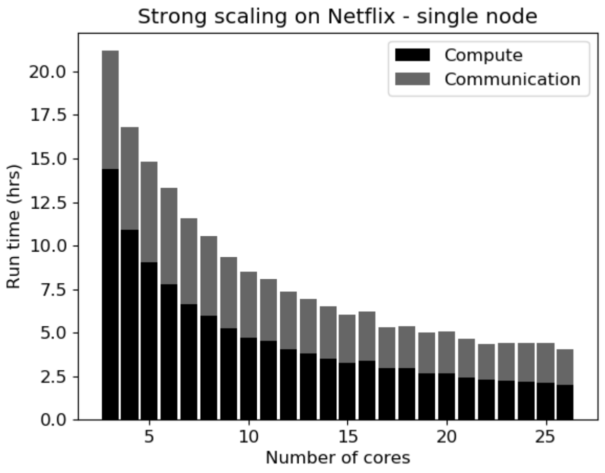

Fig. 3. Training times on the Netflix data using various numbers of cores on a single node. Times are not available for 1 or 2 cores because these tests exceeded the time limit on the cluster. Times are also not available for 27 or 28 cores because these tests required more memory than was on a single node.

to the final version, each of them only completed at most five iterations on the smaller data set before timing out. Runs done on a small number of cores barely finished a single iteration. These non-optimized versions are so slow, they are rendered useless for any practical application, while the fully optimized version can complete at least 20 iterations on the Netflix data set with many factors in a couple hours, given a sufficient number of cores.

2) Scalability: As the original authors mention in [10], iALS is slower than other matrix factorization algorithms, unless it is run in parallel. Therefore, in addition to testing scalability across the cores of one node, scalability across multiple nodes, connected by an EDR InfiniBand network, was also investigated, and the results are shown in Figs. 4 and

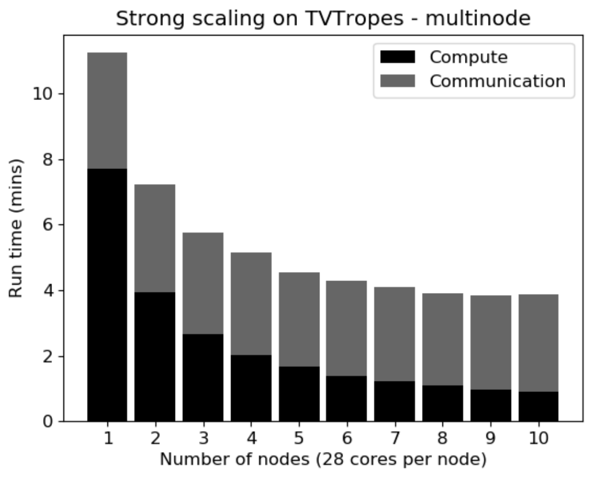

Fig. 4. Scalability across nodes on the TVTropes data. Note that the vertical scale is in minutes, not hours.

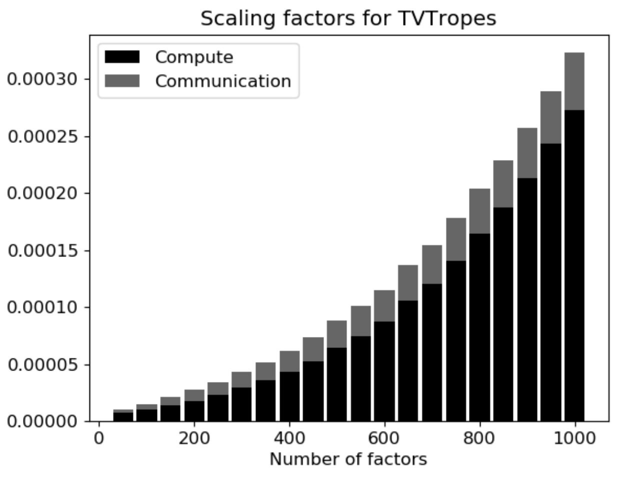

Fig. 6. Scaling across factors on the TVTropes data. Each run was done on a single node using all 28 cores.

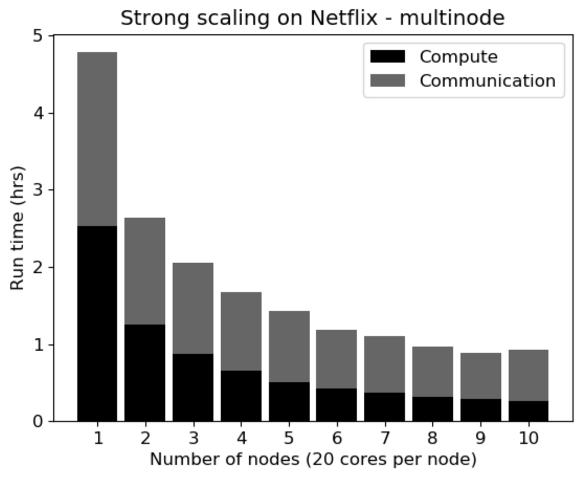

Fig. 5. Scalability across nodes on the Netflix data. Only 20 of the 28 cores available on each node were used to avoid exceeding the memory available per node.

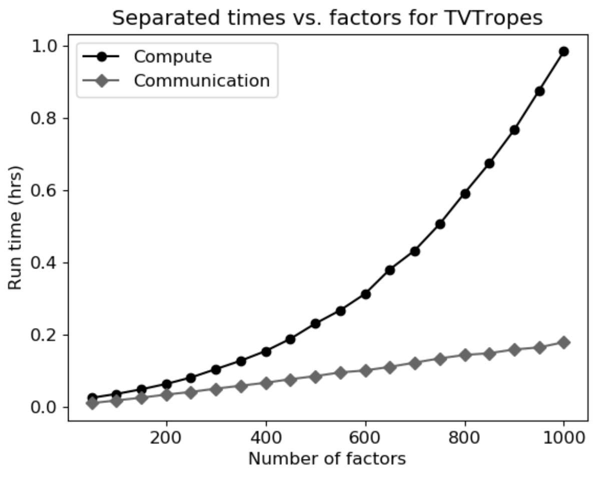

Fig. 7. Compute and communication times for scaling across factors on the TVTropes data shown separately.

5. Such a high degree of parallelism is possible with iALS that gains can still be seen for both data sets when running on 200 or more cores. However, there is a point for each data set where the increasing communication overhead from additional nodes outweighs the decreasing computation time. For the smaller TVTropes data set, this appears to be at around eight nodes; for the Netflix data set, nine nodes. Theoretically, iALS could continue to scale until each row of the item and user factors matrices is recalculated on its own dedicated core, but these results imply that would be inadvisable unless communication speeds improved drastically.

Another important aspect of scaling for iALS is how the algorithm scales as the number of factors is increased, since more factors will generally give more accurate results. The results of running the algorithm on varied numbers of factors for 20 iterations on all 28 cores on a single node are shown in Fig. 6. The same results are shown in Fig. 7 with compute and communication times separated to more easily see how each component of the run time scales across factors. Communication scales linearly, since the same number of messages are sent, but the message size grows with the number of factors that must be sent. However, the computation time scales with the cube of the number of factors, since the underlying matrix multiplication and linear solver algorithms have an $O(n^3)$ asymptotic complexity.[2] Thus the whole algorithm scales with the number of factors cubed.

C. Accuracy

The Netflix Prize data set is well known in the machine learning community, and many papers have been published de-

[2]Matrix multiplication algorithms with complexity less than $O(n^3)$ exist, but for our purposes, estimating the complexity of matrix multiplication as $O(n^3)$ is sufficient.

scribing algorithms designed to recommend movies to Netflix users. Netflix chose to use the root-mean-square error (RMSE) as their metric for how accurate a recommendation system is, so it provides a good baseline for comparison with other recommender systems. In the original challenge, the RMSE to beat was calculated over a smaller, held-out test data set with the Cinematch algorithm, with a minimum RMSE of 0.9525. Netflix provided a probe data set for contestants to test their algorithms against, and Cinematch achieved an RMSE of 0.9474 on the probe set [9]. The RMSE in this paper is calculated over the same probe set after the recommender was trained on the entire data set with the probe set removed.

The lowest RMSE observed for this implementation of iALS was 1.8501 after training the recommender with 1000 factors for over 100 iterations. It must be noted that no attempts were made to improve the accuracy of this recommender above that of the base algorithm. Therefore, this RMSE and the convergence rate could certainly be improved by adding the same accuracy optimizations Koren et al. did, some of which are described in Sec. VI. Using a scientific library for the linear algebra primitives does not imply a trade-off between accuracy and speed or ease of implementation.

VI. Related Work

When Netflix announced their challenge in 2006, with a reward of $1,000,000 to the winning team, there was naturally a large increase in interest in collaborative filtering methods, and a great deal of research has been done in the area. Popular methods include gradient descent [11], singular value decomposition [12], nearest neighbors [13], [14], and, of course, alternating least squares. The two research groups with the most in common with this paper, however, are those led by Koren and Takács.

Koren et al., as mentioned earlier, originally created the algorithm implemented in this paper, but most of their work was on improving the models used, by incorporating new dimensions, like user and item biases, temporal components, and demographic information, rather than optimizing the base matrix factorization algorithm. Their work was geared towards improving accuracy, not speed. Adding additional parameters, such as biases, can improve results by accounting for other underlying causes of similarities (or differences) in the data. For example, some users tend to give higher or lower ratings than others. When a user who almost always gives a movie five out of five stars gives a movie only four stars, that is more significant than a user whose average rating is 3.8 stars giving that movie four stars, and the model should be able to understand and incorporate that into its results. Bias terms can normalize all users' ratings to be on the same "scale," removing these personal differences [10].

Takács et al. have also done a great deal of work on optimizing collaborative filtering recommendation systems, however their work has been on other forms of matrix factorization, primarily stochastic gradient descent (SGD). The basic SGD matrix factorization algorithm works similarly to iALS, but recalculates each user and item factors row by finding the gradient of the error for each estimated rating and updating the rows in the direction opposite the gradient. Like Koren et al., they also refine the algorithm by adding additional parameters to the update step, such as individual learning rates and regularization terms for each rating, to account for some users' tendencies to give higher or lower than average ratings. Takács et al. have also researched ensemble methods that use multiple different algorithms and make recommendations which are a combination of the individual results. Often, ensemble methods give better results than a single algorithm. A secondary method they considered is nearest-neighbor based algorithms, which find similar sets of users (or items) and make recommendations based on the preferences of the group [11], [15].

With respect to the mathematics of the algorithm, the matrix-matrix multiplication reduction has a similar effect to, and was inspired by, the graph-coloring matrix multiplication algorithm described by McCourt, Smith, and Zhang in [16]. Both algorithms aim to turn a sparse matrix multiplication into a dense one by removing zero entries, however they differ in how the zeros are removed. The algorithm in this paper seeks to remove entire columns and rows along the inner dimensions of the multiplication, while McCourt et al.'s algorithm tries to find columns whose non-zeroes can be merged by mapping the matrix columns onto a graph and then coloring the graph. This coloring will find sets of matrix columns that have no non-zeroes in the same rows, so that those columns can be combined to make the matrix multiplication more dense. Since those columns are on the outer dimensions of the multiplication, the true result must be recovered afterwards by reconstructing the original non-zero pattern. The algorithm in this paper does not require any such reconstruction; the result is the final product.

VII. Conclusions

This paper described an optimized, highly scalable implementation of iALS using PETSc. The original algorithm as presented by Koren et al. was optimized by directly using preferences instead of calculating confidences and reducing the large, sparse Y^TC^uYmultiplication to a small, dense multiplication. iALS is easily parallelized by dividing the recalculation of the user and item factors matrices between processors. This implementation scales well, but after a point the communication overhead overshadows any decreases in compute time from additional cores. While further improvements could be made to the accuracy and convergence rate, the low time per iteration means this approach is promising.

Acknowledgments

Many thanks to Professors Norris and Malony in the Computer and Information Sciences department at the University of Oregon for their support and advice during this project, and to Erin McCarthy for being a wonderful office-mate and proofreader. The author would also like to thank the administrators at TVTropes.org for allowing their website to be scraped, and Netflix for continuing to provide the Netflix Prize data set.

This work benefited from access to the University of Oregon high performance computer, Talapas.

References

[1] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *2008 Eighth IEEE International Conference on Data Mining*, Dec. 2008, pp. 263–272.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.9, 2018.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[4] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, 2011, New Computational Methods and Software Tools.

[5] ——, "Parallel distributed computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124 – 1139, 2011, New Computational Methods and Software Tools.

[6] L. Dalcín, R. Paz, and M. Storti, "MPI for Python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108 – 1115, 2005.

[7] L. Dalcín, R. Paz, M. Storti, and J. D'Elía, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655 – 662, 2008.

[8] N. Developers, "Numpy reference," Apr. 2018, https://docs.scipy.org/doc/numpy-1.14.1/reference/.

[9] Netflix, "Netflix Prize," Oct. 2006, https://www.netflixprize.com.

[10] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, pp. 30–37, Aug. 2009.

[11] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Matrix factorization and neighbor based algorithms for the Netflix Prize problem," in *Proceedings of the 2008 ACM Conference on Recommender Systems*, ser. RecSys '08. New York, NY, USA: ACM, 2008, pp. 267–274.

[12] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering," in *Proceedings of KDD Cup and Workshop*, vol. 2007, 2007, pp. 5–8.

[13] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01. New York, NY, USA: ACM, 2001, pp. 285–295.

[14] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins, "Eigentaste: A constant time collaborative filtering algorithm," *Information Retrieval*, vol. 4, no. 2, pp. 133–151, Jul. 2001.

[15] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Scalable collaborative filtering approaches for large recommender systems," *Journal of Machine Learning Research*, vol. 10, pp. 623–656, Mar. 2009.

[16] M. McCourt, B. Smith, and H. Zhang, "Efficient sparse matrix-matrix products using colorings," *SIAM Journal on Matrix Analysis and Applications*, 2013.

[17] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Investigation of various matrix factorization methods for large recommender systems," in *2008 IEEE International Conference on Data Mining Workshops*, Dec. 2008, pp. 553–562.