Efficient Parallel Access Patterns for Sparse Matrices

Kevin Waugh
kwaugh@andrew

Abstract

A notable characteristic of the scientific computing and machine learning problem domains is the large amount of data to be analyzed and manipulated. Here, carefully crafted parallel algorithms have the potential to make a massive impact on the size of a problem that is deemed tractable. Sparse matrices are a data structure frequently encountered within these domains. Typically, these sparse matrices have a non-uniform structure, which can make the task of designing efficient parallel algorithms for sparse matrices much more complex than for their dense counterparts. In this report, we consider multiple ways of parallelizing the most computationally intensive section of a typical sparse matrix algorithm — its inner loop. We compare the theoretical trade-offs between each of the approaches and empirically test performance on the Netflix Prize data set [4, 5], a large collaborative filtering task that seeks to learn movie preferences from a sparse collection of ratings.

1 Introduction

Large-scale scientific computing and machine learning have rapidly become popular research areas in the last decade. The advances in these areas have led to many practical applications that have dramatically impacted society in a generally beneficial way. For example, real time weather simulation, financial market analysis, web indexing and searching, voice and face recognition, product recommendation, intrusion and fraud detection, and control automation are tasks that computers were rarely used for or trusted with fifteen years ago. Though these applications would not be possible without the staggering advancements made by researchers in these areas, it is safe to say that the upward trend in processor performance and memory capacity has been pivotal to this progression. As of recent, the rate at which single processor computation is increasing is stagnating. Instead, computer architects have shifted focus towards multi-core designs to keep up with the ever growing computational demands. This paradigm shift will in require the efficient design of parallel algorithms to effectively make use of the newest computing resources.

A sparse matrix is a frequently reoccuring data structure in scientific computing and machine learning. Unlike a dense matrix, most of the entries in a sparse matrix are zero and are not explicitly stored in memory. For example, a web search engine might use a matrix of word frequency counts for a set of web pages. Here, a web page typically uses a small subset of the words in the dictionary, which leads to most of the matrix containing zeros. Similarly, a social networking graph, where each edge indicates some type of relationship, when represented as an adjacency matrix will likely be sparse. In these cases, a machine learning algorithm for creating a search index or inferring relationships must be equipped to handle this explicit sparse structure to gain traction on any reasonably sized data set.

Let us denote an m by n sparse matrix as a pair (A, S), where $(i, j) \in S$ is a non-zero entry in the matrix and $A(i, j)$ is its value. A reoccuring pattern in sparse matrix algorithms is, then, to repeatedly evaluate functions f and g on the non-zero entries to compute

$$u = \sum_{(i,j)\in S} f(i, j, A(i, j)) \text{ and } v = \sum_{(i,j)\in S} g(i, j, A(i, j)) \tag{1}$$

where $u \in \mathcal{R}^m$ and $v \in \mathcal{R}^n$ are row and column vectors respectively. This model encompasses a large class of algorithms including the vast array centered around matrix-vector products, which have been well studied and optimized [2, 6]. For example, solution techniques for linear systems including solving a positive definite system with the conjugate gradient method, a variety of matrix factorization algorithms including ones for computing eigenvector and singular value decompositions, and optimization algorithms like interior-point methods for linear and quadratic programming. Despite its simple structure, no one approach to parallelizing this family of algorithms is obviously best. Usually, determining how to effectively distribute this computation will depend on the non-uniform pattern of sparseness in the data in addition to the functions f and g. In this report, we will examine a variety of schemes for parallelizing algorithms in this family. We will compare and contrast these schemes empirically on both shared-memory machines and a chip multiprocessor as well as theoretically in terms of both computational and memory resource overhead in addition to cache performance.

This report is sectioned as follows. First, we will introduce the Netflix Prize data set and its collaborative filtering task, which we will use for our empirical study. Then, we will describe one possible approach for collaborative filtering, the pseudo-singular value decomposition learned using gradient descent, whose inner loop follows the noted sparse access pattern. Finally, we will investigate and contrast, both theoretically and empirically, a variety of ways to parallelize this approach.

2 Collaborative Filtering and the Netflix Prize Dataset

A collaborative filtering task is most easily described as a recommendation system. Take, for example, an online store selling a number of related items. The store might wish to infer the buying habits of a user to strategically advertise goods that are mostlikely to be purchased. Similarly, an online movie or music retailer might allow a user to rate music that they like and dislike. The system could then recommend similar music or movies. More formally, a collaborative filtering system takes as input a set of k user/item/rating tuples and gives as output a function that predicts any unobserved tuple. The quality of such a system is usually measured by considering the root mean squared error of the predictions on withheld test set.

Good collaborative filtering systems consider groups of users, or perhaps them all, at once when learning preferences. This is because typically the observation vector for a single user is sparsely populated and little can be inferred from the single user's ratings in isolation. That is, one typically assumes that, though there are many distinct users of the system, many of them share similar tastes. These similar tastes can be inferred by learning relationships between the users through their ratings. Thus, a sparse matrix of ratings is a natural way to represent the input data. Though here, the unrecorded entries in the matrix denote an unknown observation as opposed to a known zero.

In 2006, the online movie rental company, Netflix, posed a million dollar challenge to the machine learning community called the Netflix Prize [4]. Given an anonymized database of 100480507 ratings, on an integer scale from 1 to 5, from 480189 distinct Netflix users and 17770 movies, any team that could create a recommender system that outperformed Netflix's proprietary system, Cinematch, by 10% on a hidden test set would win the prize provided they published the inner workings of their system and allowed Netflix to employ its use. Just this October the prize was won. Since then, the training and test data sets have been donated to the scientific community and are stored at the UCI Machine Learning Repository [5]. This exchanged undoubtedly benefited both Netflix and the scientific community. That is, Netflix could not used the million dollars to otherwise increase the performance of their recommender system by such a large margin and the scientific community could not have otherwise gathered the data, a precious commodity in the machine learning community.

3 Pseudo Singular Value Decomposition

Given an m by n matrix A, there exists a unique factorization, $A = U\Sigma V^T$, called the singular value decomposition of A. Here, U is an m by m orthonormal matrix, Σ is a diagonal matrix of decreasing non-negative values and V is an n by n orthonormal matrix. The singular value decomposition of a matrix has many useful properties in scientific computing and machine learning, such as least-squares fitting, dimensionality reduction and, in our case, collaborative filtering tasks.

First, let us consider a matrix compression task. That is, we wish to write $A \approx UV^T$, where U is an n by k matrix, and V is an m by k matrix. Here, we wish to minimize the reconstruction error,

$$\min_{U,V} \|A - UV^T\| = \min_{U,V} \sum_{i,j} \|A(i,j) - u_i \cdot v_j\|_2 \tag{2}$$

which is the distance between the original matrix and its image after decompression under the Frobenius norm, an extension of Euclidean distance to matrices. It is known that, for a fixed k, the minimum reconstruction error is obtained by selecting the top k singular vectors of the singular value decomposition of A to populate U and V. This can be done, even on sparse matrices, using efficient iterative methods employing elementary matrix operations [3, 9].

At first glance, the singular value decomposition appears to immediately fit our collaborative filtering task. That is, one could select a sufficiently small k and solve for U and V using a singular value decomposition. To predict an unknown rating (i, j), one simply computes the dot product $u_i \cdot v_j$. Unfortunately, the unknown entries in our collaborative filtering task are *not* zero, and thus solution does not directly apply. That is, using this framework we seek a solution to

$$\min_{U,V} \sum_{(i,j)\in S} \|A(i,j) - u_i \cdot v_j\|_2 \tag{3}$$

which has become known as the pseudo singular value decomposition of A. Unfortunately, this problem can be non-convex depending on the sparse structure of of A and therefore it is likely no efficient methods exist for computing the best U and V.

Undeterred, we can still use this framework as the basis for a good recommender system by using gradient descent to find a local minima. Here, on each iteration one computes the gradients for each weight $u_{i,k}$ and $v_{j,k}$ as

$$\nabla u_{i,k} = -2 \sum_j (A(i,j) - u_i \cdot v_j) v_{j,k} \text{ and } \nabla v_{j,k} = -2 \sum_i (A(i,j) - u_i \cdot v_j) u_{i,k} \tag{4}$$

Once the gradient is computed, the weights are updated by stepping along the negative gradient by some small ε, called the learning rate. This process is repeated until a stop condition, such as achieving less than preselected decrease in the error, is satisfied. A slight extension to the noted procedure is to actually update the weights inplace while iterating through the ratings using

$$u_{i,k} \leftarrow u_{i,k} + \varepsilon(A(i,j) - u_i \cdot v_j) v_{j,k} \text{ and } v_{j,k} \leftarrow v_{j,k} + \varepsilon(A(i,j) - u_i \cdot v_j) v_{j,k} \tag{5}$$

If the ratings are sampled randomly, this procedure is known as stochastic gradient descent and, with high probability, will also converge to a local minima. In practice, even when the ratings are examined in a preset order the procedure moves towards the desired result. This approach was one of the approaches employed by the top teams competing for the Netflix Prize [1, 7, 8].

It is apparent that these gradient descent approaches follow the same access pattern as our family of algorithms – each entry is examined once per iteration and a row weight and a column weight are updated. Though, in the second, stochastic, approach the functions f and g depend on the updated weights when moving from entry to entry, as opposed to the previous iterations weights. We will see that this subtle difference has a not so subtle impact on performance depending on how procedure is parallelized in the next section.

4 Experimental Results

For our experiments we will consider a variety of parallel programs for computing a pseudo singular value decomposition of the Netflix Prize data set. Each of the programs runs for 100 iterations with $k = 1$. We note that any approach restricted to $k = 1$ can be used to solve for larger values of k by computing for each pair of vectors sequentially and adjusting the target ratings accordingly. We deemed 100 enough to accurately compare the running times across different methods as well as compute the speed ups.

All parallel programs were written using the `pthreads` threading library and initially tested on an Intel Core2 2.4 GHz quad core machine with 8 gigabytes of memory. Each particular processor has

its own 128 kilobyte L1 cache and shares a 2 megabyte L2 cache with one other processor. Some of the more promising approaches were also tested on the large shared-memory machines, pople and cobalt. These tests were not completed to a sufficient degree to be included in this report due to delays in jobs moving through the batch system[1], but in the future work conclusions sections we shall mention the preliminary trends and our expectations of the results.

4.1 Sequential Baseline, Movie Order

In order to draw accurate conclusions about the performance of the parallel programs, an optimized sequential baseline program was first developed. Multiple candidates for the best sequential program were examined, but the *movie order* sequential program reigned supreme. This particular program used the stochastic gradient descent update rule while examining ratings in movie-sorted order. By sorting by movie weights, the program could exploit spatial locality and minimize cache misses on at least the movie weights. By using movie order, as opposed to user order, the effect of fewer cache misses was more pronounced as the average number of ratings per movie is much higher than the average number of ratings per user. An additional advantage of movie order, as opposed to other orders, is that the movie need not be stored with each rating. Instead, only the end index of the movies need to be kept and the movie count can be incremented when passing this mark. That is, the memory requirement for this approach is $2m+n+2r$. The stochastic update rule was used in the baseline as it had better performance than the gradient rule. The additional work of stepping along and zero'ing the gradient were unnecessary overhead for the sequential program. The sequential movie order program completed 100 iterations (not including start up or clean up) on the quad core in 201 seconds, and on in 418 and 314 seconds on pople and cobalt respectively, as seen in Table 1, which contains the running time of all programs on the chip multiprocessor.

4.2 Movie Order

The straightforward parallelization for the movie order program is simply to introduce a number of spin locks, one for the iteration counter and one for each user. Each thread grabs a single movie at a time after locking the iteration lock and proceeds to update the weights for the movie and its associated users. Here, before updating the user weights, the thread must grab the associated user lock. Though there is almost no contention for the user locks as there are at most four threads competing for half a million locks, the overhead of acquiring the lock every iteration is very substantial and dominates the work done in the inner loop. Using this scheme, the four processor program ran 3.5 times slower than the optimized sequential program.

A number of other programs that acquired a single lock on each iteration of the inner loop were attempted, but all fared around equal to or worse to the parallel movie order program described above. For example, the user order program had higher contention for the iteration lock and exhibited worse spatial locality on the weights. Another interesting avenue that seemed to help, but only slightly, was grouping users with close identifiers together and having them share a lock. This reduced the number of user locks and allowed a processor to keep a lock in its cache between inner loop iterations. All the schemes using this approach have the memory footprint, $m+n+2r+l+x$, where l is the number of locks used and x is either m or n depending on if a movie or user order was used.

4.3 Local Copy, Movie Order

Next, two schemes partially employing the gradient computation rule were tested. The first, kept a local copy of the user gradient. It examined the ratings in movie order while updating its local copy of the user weight gradient and modifying the movie weights inplace. Here, only one lock was needed, the low contention lock that controlled access to the iteration counter. After completing a single pass through the ratings, each thread would sequentially update the user ratings based on their portion of the gradient. Even though this scheme removed the requirement of a costly lock acquisition from the inner loop, the added sequential work of size $m * P$ per iteration proved to be a substantial bottleneck. Also, with this method we noticed a decreasing trend in speed up as

[1]This is no one's fault but my own.

the number of processors increased. This implies that not only does the scale linearly in memory as more processors are added, but additional processors are unlikely to provide a performance increase.

4.4 Local Copy, User Order

When the previous method instead uses a local copy of the movie weights, as opposed to the user weights, we see a welcome change. Though the performance worsens from one processor to two, the performance of the three and four processor programs sees a relatively dramatic improvement. This approach is the first that "works" in the sense that the parallel version is faster than the optimized sequential version and its performance was the highest of all parallel programs with a speed up of close to 1.7 on four processors. Before explaining the performance increase, we should note that when this program iterates over the ratings in a user order, a processor grabs a batch of users, as opposed to a single one as in the movie case, to decrease contention for the iteration counter lock. This measure is important in the user case, where it makes little difference in the movie case, as the number of users is much greater than the number of movies. Like the previous attempt, this program has a sequential section of code at the end of each iteration. Fortunately, since the number of movies is much smaller than the number of users, this sequential work in this case is a smaller proportion of the running time and as such does not have as dramatic an effect on the speed up.

4.5 Two Pass

The two pass parallelization scheme is dead simple, almost lock free, and in theory scales well with the number of processors. Thus, we believe has the highest potential for increased performance when scaling to large numbers of processors on shared-memory machines. Here, the program first passes over the data in movie order and updates the movie weights using the stochastic gradient update. After, it passes over the ratings in user order and updates the user weights, again using the stochastic gradient update. This method requires a lock only for the iteration counter. The memory cost of this scheme is higher than the others as it must keep both a movie and user indexing scheme, but this requirement does not grow as the number of processors increases. Similarly, the program must pass over the ratings twice per iteration, as opposed to once, but this computation cost is also fixed regardless of the number of processors. That is, as processors grow the main point of contention is simply the iteration lock. By using a smart static ordering on the separate passes, this method could be made lock free in addition to its constant overhead. In the chip multiprocessor setting, this method performed well, but not quite to the level of the local copy in user order.

4.6 Greedy Static Order

The greedy static ordering method was the most complicated method that we attempted. Prior to the program running, a greedy scheduling algorithm would each user and each movie to a processor. This induces a set of ratings that the processor must examine to update the weights. To generate this static ordering, we first considered assigning the movies to processors. Iteratively, the processor with the least load was assigned the largest unassigned movie. After movies were assigned, each user was given a per processor score on how much work it would require to complete. That is, if the movie and user shared a rating, that particular rating was considered to take no time to update. Again, the lowest load processor was assigned its highest work user. Surprisingly, this greedy scheme showed very good load balancing up to four processors. The difference between the maximum number of ratings assigned and the minimum was about 5%. Here, though we are updating both the user and the movie weights at the same time, because only one processor actually modifies its assigned weights, a lock is not required in the inner loop. That is, cache coherency nicely keeps everything in the places that it needs to be. Also, since the order is statically assigned, no iteration lock is required. This greedy static ordering has the same worst-case computational and memory complexity as the two pass algorithm, up to twice the memory and twice the computation, but in practice it appeared that these memory and passes over the ratings were closer to 1.5 times. The main disadvantage of this scheme compared to the two pass scheme is that it will likely have more cache invalidations as both user and movie ratings are updated at the same time.

Program	**Running time (s)**			
	$P = 1$	$P = 2$	$P = 3$	$P = 4$
sequential	201.141			
movie order		944.747	749.371	708.655
local copy, movie order		292.011	294.706	301.553
local copy, user order		227.206	139.767	120.083
greedy static		198.024	172.761	154.275
two pass		261.325	165.268	149.251

Table 1: Running times for all programs on the Intel Core2 quad core machine

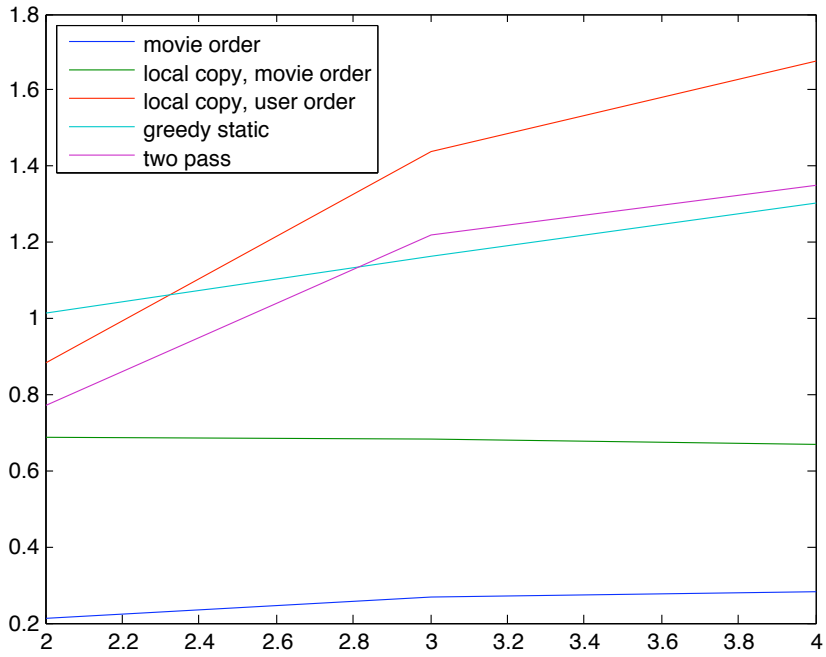

Figure 1: Speed up for all programs on the Intel Core2 quad core machine

5 Future Work

The most promising direction for future work is the combine a static ordering with the two pass method. Here, we get the benefits of a lock free method with constant computational and and memory overhead that should scale well as the number of processors increases. By updating the movies and users in two passes, we reduce the effect of invalidating other processors caches at the cost of being required to visit each rating twice. This is less of a concern, though, because as the number of processors increases, it is expected that effective load balancing will force each rating to be visited twice. Though the greedy load balancer appears to work well for small numbers of processors on this data set, it is not clear how its quality will scale as the number of processors increases.

If the greedy load balancer does not scale, another potential load balancing scheme would be to consider the following 0/1 integer programming problem. We introduce a variables of the form $x_{i,p}$ and $y_{j,q}$ to denote that movie i is assigned to processor p and user j is assigned to processor q respectively. Then, we minimize an additional variable C, which is the difference between the maximum load on a processor and the minimum load on a processor. The global solution to this 0/1 program is an optimal way to distribute the ratings assuming a constant number of cache misses and invalidations. Here, the 0/1 integer program is NP-hard to compute, so instead we can relax the integer constraints to be linear and solve the relaxed linear program to determine the load balancing. Here, one can consider the variable $x_{i,p}$ to be the probably that movie i is assigned to processor p. Though we have not proven it, we believe it is likely that this will lead to a good (constant) approximation to the load balancing problem. Hopefully, the quality of this approach would scale better than the greedy algorithm.

6 Conclusion

In this report, we introduced a variety of schemes for parallelizing a family of iterative sparse matrix algorithms. These schemes were tested using gradient descent to compute a pseudo singular value decomposition, one such algorithm in this family, on the large and sparse Netflix Prize data set. We found that, unsurprisingly, a well optimized sequential program that is designed to effectively exploit spacial locality and avoid cache misses is hard to beat by trivial parallelization schemes. In particular, schemes that involved excessive locks (even uncontested), or additional sequential portions, did not perform as well as the sequential version and did not show trends of scaling to large numbers of processors. The simplest lock free scheme, the two pass approach, which pays a large constant overhead penalty upfront, but scales well appears to be the front runner for any parallelization scheme. One could potentially investigate more sophisticated dynamic or static load balancing schemes within this two pass framework, but our findings suggest that the row and column weights should either be updated in separate passes or on a local copy.

References

[1] R. M. Bell and Y. Koren. Improved neighborhood-based collaborative filtering. In *Proceedings of the KDD Cup and Workshop*, 2007.

[2] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Math Software*, 28(2):135–151, 2002.

[3] J. J. Leader. *Numberical Analysis and Scientific Computation*. Addison-Wesley, 2005.

[4] Netflix, Inc. Netflix Prize. `http://www.netflixprize.com`.

[5] Netflix, Inc. Netflix Prize Data Set. `http://archive.ics.uci.edu/ml/datasets/Netflix+Prize`.

[6] D. R. O'Hallaron. Spark98: Sparse Matrix Kernels for Shared Memory and Message Passing Systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, Oct. 1997.

[7] A. Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of the KDD Cup and Workshop*, 2007.

[8] T. Raiko, A. Ilin, and J. Karhunen. Principal component analysis for large scale problems with lots of missing values. In *Proceedings of the European Conference on Machine Learning*, 2007.

[9] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997.