Parallelized Preconditioned Model Building Algorithm for Matrix Factorization

Kamer Kaya, Ş. İlker Birbil, M. Kaan Öztürk, and Amir Gohari

Faculty of Engineering and Natural Sciences
Sabancı University
İstanbul, Turkey
{kaya, sibirbil, mkozturk, amir}@sabanciuniv.edu

Abstract. Matrix factorization is a common task underlying several machine learning applications such as recommender systems, topic modeling, or compressed sensing. Given a large and possibly sparse matrix A, we seek two smaller matrices W and H such that their product is as close to A as possible. The objective is minimizing the sum of square errors in the approximation. Typically such problems involve hundreds of thousands of unknowns, so an optimizer must be exceptionally efficient. In this study, a new algorithm, Preconditioned Model Building is adapted to factorize matrices composed of movie ratings in the MovieLens data sets with 1, 10, and 20 million entries. We present experiments that compare the sequential MATLAB implementation of the PMB algorithm with other algorithms in the `minFunc` package. We also employ a lock-free sparse matrix factorization algorithm and provide a scalable shared-memory parallel implementation. We show that (a) the optimization performance of the PMB algorithm is comparable to the best algorithms in common use, and (b) the computational performance can be significantly increased with parallelization.

Keywords: Preconditioned model building, matrix factorization, multicore parallelism.

1 Introduction

We investigate the performance of a novel optimization algorithm on the matrix factorization problem. The classic matrix factorization problem involves approximating a given matrix A as the product of two unknown matrices W and H:

$$A \approx WH, \tag{1}$$

where $A \in \mathbb{R}^{m\times n}$, $W \in \mathbb{R}^{m\times r}$, $H \in \mathbb{R}^{r\times n}$, with r a given integer (the *rank* of the factorization). In typical applications, A is sparse, and r is much smaller than either m or n. However, the resulting factor matrices W and H can be dense. The associated optimization problem is the minimization of the sum of squares of errors in the approximation

$$\min_{W,H} \sum_{i,j\in S} \left(A_{ij} - \sum_{k=1}^{r} W_{ik} H_{kj} \right)^2 , \tag{2}$$

where the outer sum is over the set S of (i, j) pairs where A_{ij} is known (nonzero). If a good approximation with a small r can be found, the factors can be used to represent the original data in a more compressed form, with less redundancies.

One application of matrix factorization is in the field of recommendation systems, particularly content-based filtering. As a concrete example, suppose that each row of A corresponds to a particular user, each column to a particular movie, and the matrix element A_{ij} is a numeric value representing the rating given by user i to movie j. This matrix is very sparse, because most of the users have rated only a small fraction of all available movies. Furthermore, the data have redundancies, because the ratings given by users with common tastes and interests are likely to be correlated. After the original matrix A is factorized into factor matrices with relatively small rank r, we can multiply them back to obtain a full matrix A^*. The entries in A^* will then be estimates for the missing values in A. In other words, we can estimate whether a user would give a high ranking to a given movie, and display it as a recommendation to the user.

Intuitively, matrix factorization can be seen as discovering some hidden variables in the data. For example, the hidden dimensions can be movie genres, movies with a strong female character, movies that appeal to an adolescent audience, etc [6]. If the input matrix comprises e-mails and the words in them, such as the now-public Enron e-mail data set, the hidden dimensions turn out to be topics like professional football, California blackout, and Enron downfall [1].

The power of matrix factorization as a recommender system is demonstrated in the Netflix Prize challenge. In this challenge, many different algorithms were compared with each other to see which one would improve the recommendation accuracy by more than 10%. The first algorithm that crossed this mark was based on matrix factorization [6].

Although other methods such as Principal Component Analysis or Latent Semantic Analysis can also be applied to that end, matrix factorization has the advantage that it does not regard empty matrix entries as zero values. The optimization problem considers only the sum of squares over existing values. This property reduces the error of the approximation [5].

Another application of matrix factorization is data compression, or representing the data in a low-dimensional subspace. Assume again that each row of A represents ratings of users. Then, from $A \approx WH$ it follows that the i-th row of A can be written as a linear combination of the rows of H, with coefficients taken from the i-th row of W:

$$A_{i,:} = \sum_{k=1}^{r} W_{ik} H_{k,:} \tag{3}$$

where the notation $A_{i,:}$ indicates the i-th row of matrix A. We can then interpret W_{ik} as a measure of user i's interest in movies that have property k. Similarly, we can interpret H_{kj} as a measure of how much of k is carried by the movie j.

Due to its ability to compress information, matrix factorization can also be used for unsupervised classification problems. To this end, the preferred variety is *nonnegative matrix factorization*, where both the data matrix A and the factor matrices W, H are constrained to have only nonnegative entries. With nonnegativity, the linear combination (3) gives a recipe for constructing $A_{i,:}$ by adding ingredients $H_{k,:}$ in amounts of W_{ik}. Because no subtraction is involved, we can interpret the results in a more intuitive way [7]. In this study, we only consider unconstrained optimization, therefore nonnegative matrix factorization is beyond our scope at the moment. We solve only the *classic* matrix factorization problem, where entries can be negative real numbers.

For the experiments, we factorize user-movie rating matrices, provided by the MovieLens database. To minimize the objective function (2), we use the Preconditioned Model Building (PMB) method that we describe in Section 2.

We have developed a MATLAB implementation of the algorithm with no parallelization. We first factorize the MovieLens 1M rating matrix with PMB, as well as with other established optimization methods in the `minFunc` package. In Section 2.1 we show that the performance of PMB on this problem is comparable to the best ones that are in widespread use, and better than some others.

2 Preconditioned Model Building

In our recent work, we have proposed a new method that could be used as an alternative to line search procedure in unconstrained optimization algorithms [9,10]. From this perspective, the proposed method is another globalization mechanism that aids algorithms to converge from remote points to a local minimizer. The main idea of the proposed method is to build a series of quadratic model functions using trial points around the current iterate. With each trial point, the simpler quadratic model function is minimized and the next trial point is set to the location of the attained minimum. If this minimum point provides a sufficient decrease in the original objective function according to the Armijo condition, then it is accepted as the new step to move to the next iteration. Otherwise a new model is built around the incumbent trial point. As we construct a new quadratic model at each trial point, we aptly refer to this approach as model building (MB) algorithm in this paper.

At iteration k, MB takes an initial vector and uses it as the first trial point, s_k. To guarantee the convergence of the algorithm, this initial vector should be gradient related providing a sufficient descent. Let us formalize this discussion. Consider the unconstrained optimization problem of the form

$$\min_{x \in \mathbb{R}^n} f(x),$$

where $f : \mathbb{R}^n \mapsto \mathbb{R}$ is the objective function. Let x_k denote the point at iteration k. To obtain the next iterate x_{k+1}, the MB algorithm requires the initial vector s_k to satisfy the following two conditions:

$$\begin{aligned} m_0\|\nabla f(x_k)\| \leq \|s_k\| \leq M_0\|\nabla f(x_k)\|, \\ -\mu_0\|\nabla f(x_k)\|^2 \leq s_k^\top \nabla f(x_k) \leq 0 \end{aligned} \tag{4}$$

for some $m_0, M_0, \mu_0 \in (0, \infty)$. If we simply set $s_k = -\nabla f(x_k)$, then both conditions are satisfied with $m_0 = M_0 = \mu_0 = 1$. This choice of s_k is, in fact, used when MB is first introduced by Öztoprak and Birbil [10].

Another way of setting the initial vector for the MB algorithm is to use a positive definite matrix. That is, we can set $s_k = -H_k \nabla f(x_k)$ and use this direction as an input to the MB algorithm. Since H_k is positive definite, the conditions in (4) are satisfied by taking m_0 as the minimum eigenvalue of H_k, and M_0 along with μ_0 as the maximum eigenvalue of H_k. Algorithm 1 shows explicitly the steps of our implementation, where the first trial step is determined after a preconditioner is computed (line 4). The model building steps are given between line 14 and line 21. The original algorithm in [10] takes $\eta \in (0, 1)$ as an input of the algorithm. In our implementation, we have observed that adjusting this parameter dynamically as shown in line 14 improves the performance.

Algorithm 1: Preconditioned Model Building

```
1  Input: $x_0$; $\rho = 10^{-4}$; $k = 0$
2  $f_k = f(x_k)$; $g_k = \nabla f(x_k)$;
3  while $x_k$ is not a stationary point do
4      Compute the preconditioner $H_k$;
5      $s_k = -H_k g_k$;
6      for $t = 0, 1, 2, \cdots$ do
7          $x_k^t = x_k + s_k$; $f_k^t = f(x_k^t)$; $g_k^t = \nabla f(x_k^t)$;
8          $v_6 = s_k^\top g_k$; $\Delta f = f_k - f_k^t$;
9          if $\Delta f \geq -\rho v_6$ then
10             $x_{k+1} = x_k^t$, $f_{k+1} = f_k^t$, $g_{k+1} = g_k^t$;
11             $k = k + 1$;
12             break;
13         end
14         $v_0 = s_k^\top g_k^t$; $\eta_1 = \frac{|\Delta f|}{v_6}$; $\eta_2 = \frac{|\Delta f|}{v_0}$; $\eta = \frac{\min(\eta_1, \eta_2)}{\eta_1 + \eta_2}$;
15         $y = g_k^t - g_k$; $v_1 = y^\top s_k$; $v_2 = s_k^\top s_k$;
16         $v_3 = y^\top y$; $v_4 = y^\top g_k$; $v_5 = g_k^\top g_k$;
17         $\sigma = \frac{1}{2}(\sqrt{v_2}(\sqrt{v_3} + \frac{1}{\eta}\sqrt{v_5}) - v_1)$;
18         $\theta = (v_1 + 2\sigma)^2 - v_2 v_3$;
19         $c_g = -v_2/(2\sigma)$; $c_s = \frac{c_g}{\theta}(-(v_1 + 2\sigma)v_4 + v_3 v_6)$;
20         $c_y = \frac{c_g}{\theta}(-(v_1 + 2\sigma)v_6 + v_2 v_4)$;
21         $s_k = c_g g_k + c_s s_k + c_y y$;
22     end
23 end
```

The introduction of such a positive definite matrix H_k is also known as preconditioning. The advantage of preconditioning in the optimization context is to incorporate second order information into the step evaluation [2]. Quasi-Newton

Dataset	**#users**	**#movies**	**#ratings**	**density**
1M	6,040	3,952	1,000,209	0.042
10M	71,567	10,681	10,000,054	0.013
20M	138,493	27,278	20,000,263	0.005

Table 1: Metadata of MovieLens data sets.

methods obtain this information by making use of the gradient information collected in the previous iterations. The most famous one among the quasi-Newton methods is the limited BFGS method (L-BFGS) method [8]. In this current work, we have also used L-BFGS update mechanism for estimating our preconditioning matrices. Thus, we refer to the resulting procedure as Preconditioned Model Building (PMB) algorithm.

2.1 A first comparison with other optimizers

The MovieLens data [4] This is a public data set containing a large number of ratings of movies by individuals. The data is collected from the `movielens.org` web site, maintained by the GroupLens research group at the Univ. of Minnesota. Although the full data set keeps growing in time, stable data sets are available for benchmarking purposes. These are referred to as 1M, 10M, and 20M datasets. The names refer to the number of ratings contained in each data set.

Table 1 lists the number of users, number of movies and number of ratings in each data set. Each set contains users who have rated at least 20 movies. Ratings are integers between 1 and 5.

Numerical comparison We solve the optimization problem (2) with PMB, as well as several other optimization functions commonly used in literature. We see that PMB does not have a significant handicap when compared against the other accepted methods.

We factorize the matrix of MovieLens 1M data set with several popular optimization algorithms, along with PMB. All of these factorizations are performed using MATLAB R2015b. For all methods except PMB, we have used `minFunc` package [11]. The codes for generating these results are available in the accompanying GitHub repository[1] for those who wish to replicate our results.[2]

Each algorithm is initialized with random matrix entries. Each entry is sampled from the uniform distribution $U(1,5)/\sqrt{r}$, where r is the rank of the factorization (set to 50), so that the resulting matrix product has entries mostly between 1 and 5. Algorithms are stopped after 500 iterations. The maximum number of function calls and the maximum number of iterations are both set to 1000. Every algorithm run is repeated 50 times with randomized initial points, and 95% confidence intervals for the mean values are estimated using bootstrapping.

[1] https://github.com/sibirbil/PMBSolve

[2] The PMB results in this section are obtained with the MATLAB implementation, which is not parallelized and thus different from the results given in Section 4.

Method	**Mean final RMSE**	**95% confidence interval**
Barzilai and Borwein	0.6436	(0.6096, 0.6855)
Cyclic Steepest Descent	0.5894	(0.5871, 0.5919)
Hessian-Free Newton	0.5561	(0.5544, 0.5581)
Conjugate Gradient (CG)	0.5558	(0.5548, 0.5568)
Scaled CG	0.5391	(0.5385, 0.5398)
PMB	0.5148	(0.5138, 0.5160)
Preconditioned CG	0.5020	(0.5002, 0.5038)
Limited memory BFGS	0.4954	(0.4944, 0.4965)

Table 2: Comparison of PMB with other optimizers for the 1M dataset with factorization rank 50, averaged over 50 runs.

The resulting RMS error values and gradient norm for each algorithm is shown on Table 2. We see that PMB is one of the most successful methods to solve this large matrix factorization problem.

3 Parallelization of PMB-based Matrix Factorization

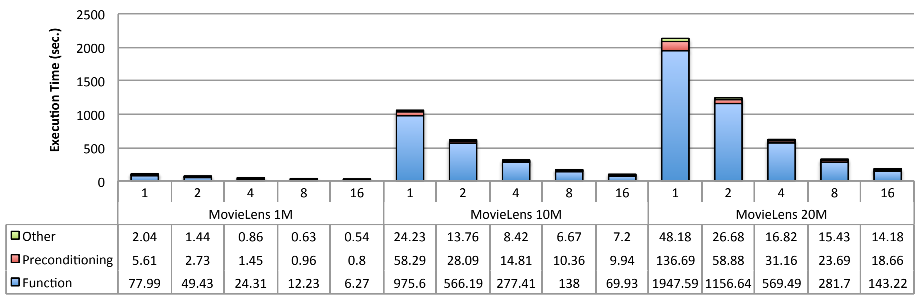
The PMB engine is implemented by using templates in `C++11`. Various optimization problems, e.g., matrix factorization as in this study, can be solved with the engine once the appropriate function/gradient computation source code is integrated. Moreover, this integration does not need a modification on the engine and a separate source file is sufficient.

The only time consuming part of the engine is the preconditioning; however, for the matrix factorization problem, preconditioning is only responsible for the 5% of the execution time. The rest is spent to the function and gradient computations for sparse factorization. Hence, in this work, we mainly focus on the function and gradient computations since they form the main bottleneck. The computations in the engine, mostly dot products, are also parallelized in a straightforward manner whlie optimizing the data reuse and memory accesses as much as possible. The execution time of the PMB for the matrix factorization problem dissected into three parts is given in Figure 1. As the figure shows, the factorization-specific functions is responsible for most of the execution time.

3.1 Computational tasks for sparse matrix factorization

Given a sparse matrix A with τ entries, there are three tasks at each iteration:

1. Computing the error Δ_{ij} for known A_{ij} entries, i.e.,
$$\Delta_{ij} = (W_{i,:} \cdot H_{:,j}) - A_{ij} \text{ for all } (i.j) \in S \tag{5}$$
where $\cdot$ is the dot product operator and $W_{i,:}$ and $H_{:,j}$ are row and column vectors corresponding to the i-th row of W and j-th column of H, respectively. This task simultaneously computes the overall function value $\sum_{(i,j)\in S} \Delta_{ij}^2$.
2. Computing the gradient entries for W; let $Z1$ be the matrix containing these entries. Then
$$Z1_{i,:} = \sum_{(i,j)\in S} \Delta_{ij} H_{:,j}^T. \tag{6}$$



	MovieLens 1M					MovieLens 10M					MovieLens 20M				
	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
🟩 Other	2.04	1.44	0.86	0.63	0.54	24.23	13.76	8.42	6.67	7.2	48.18	26.68	16.82	15.43	14.18
🟥 Preconditioning	5.61	2.73	1.45	0.96	0.8	58.29	28.09	14.81	10.36	9.94	136.69	58.88	31.16	23.69	18.66
🟦 Function	77.99	49.43	24.31	12.23	6.27	975.6	566.19	277.41	138	69.93	1947.59	1156.64	569.49	281.7	143.22

Fig. 1: The execution time of PMB dissected into three parts for bottleneck detection: The most time consuming part, *Function*, computes the error for each A_{ij} and updates the factor matrices accordingly. The next part handles the *Preconditioning* stage. The *Other* parts of PMB, i.e., memory allocations, transfers etc., are considered as a third part for completeness.

3. Computing the gradient entries for H; let $Z2$ be the matrix containing these entries. Then

$$Z2_{:,j} = \sum_{(i,j)\in S} \Delta_{ij} W_{i,:}^{T}. \tag{7}$$

For all the tasks, the time complexity is $\mathcal{O}(r \times \tau)$ where r is the factorization rank.

3.2 Storing the sparse matrix and auxiliary data in memory

We start by mentioning the common data structures for the implementation of the algorithms. For matrix factorization, the pattern and the numerical values of a sparse matrix is stored in both the compressed row storage (CRS) or compressed column storage (CCS) formats. These are well known storage formats for sparse matrices (see, e.g., Section 2.7 of Duff et al. [3]). Consider an $m \times n$ sparse matrix A with τ nonzeros. In CRS, the pattern of A is stored in three arrays:

- `colids`$[1, \ldots, \tau]$ stores the column index of each entry.
- `vals`$[1, \ldots, \tau]$ stores the corresponding numerical value of each entry. The column ids and values in a row are stored consecutively; and
- `ptrs`$[1, \ldots, m+1]$ stores the location of the first entry of each row in array `colids` where `ptrs`$[m+1] = \tau + 1$. In particular, the column indices of the entries in row i are stored in `colids`$[$`ptrs`$[i], \ldots,$ `ptrs`$[i+1]-1]$. Similarly the values in the i-th row are stored in `values`$[$`ptrs`$[i], \ldots,$ `ptrs`$[i+1]-1]$

The CCS of a matrix A is the CRS of its transpose and vice-versa. In CCS, there are two pattern arrays `(ccs)_rowids` and `(ccs)_ptrs`, with functions similar to the first and the third arrays just described above. However, we will not need a `(ccs)_vals` array separately for the CCS format. Both formats are necessary for our parallel implementation.

The auxiliary sparse matrix Δ has τ entries and the same sparsity pattern of A. Hence, the same CRS/CCS pattern arrays can be used. We also use an extra `delta` array of size τ to store the Δ_{ij} values in CRS format. When a column-wise access to Δ is required (that will be necessary to avoid race conditions), we will utilize a static, precomputed `ccs_trans` array that translates the CCS-location to a CRS-location. The usage of this array will be described in more detail later.

The matrices W, $Z1$, H and $Z2$ are all dense and the first two and the last two contain $m \times r$ and $n \times r$ entries, respectively. To optimize the spatial locality of reference for the accesses to these matrices, we use the row-major layout for W and $Z1$ and the column-major layout for H and $Z2$.

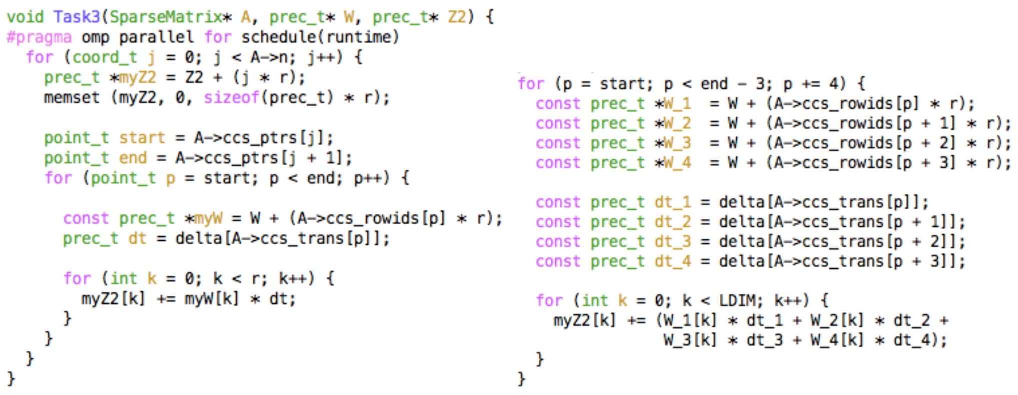
3.3 Efficient and lock-free parallel implementation of the tasks

The memory accesses for the sparse factorization problem can deteriorate the performance if they are not handled carefully. As mentioned above, our first task computes the Δ matrix which is used by the later tasks; if the implementation uses barriers in between the tasks, there will be (at least) τ memory accesses to the `delta` array by the second task. This overhead can be avoided when the first two tasks are integrated; in this version, each Δ_{ij} computed by (5) is immediately used by (6). This Δ_{ij} value is then stored in the corresponding entry of the `delta` array to be used later by the third task to compute (7).

The first task can be parallelized in two different ways: in the fine-grain approach, each Δ_{ij} computation can be assigned to a different thread, and in the coarse-grain approach, the values in $\Delta_{i,:}$ are assigned to the same thread. Although the former increases the degree of concurrency and eases load balancing, the latter is more appropriate for the integration of the first two tasks. As it can be seen by (6), a gradient entry $Z1_{ij}$ is modified for each entry in $\Delta_{i,:}$. Hence, when the fine-grain approach is taken and two threads independently compute and use $\Delta_{i,j'}$ and $\Delta_{i,j''}$, the entry $Z1_{ij}$ needs to be updated by both of these threads. To avoid such race conditions, expensive synchronization mechanisms are required. However, a lock-free implementation is possible when each row $\Delta_{i,:}$ (and hence $Z1_{i,:}$) is assigned to only a single thread. Since we access the elements of A and Δ in a row-wise manner, the CRS pattern and value arrays are used for this implementation.

A similar analysis of (7) implies that a lock-free parallel implementation of the third task is possible if the updates on each column of $Z2$ are solely assigned to a single thread. However, this requires an efficient access to the columns of Δ. Since the array `delta` is organized via CRS, the entries in a column of Δ are not consecutively stored in memory. On the other hand, with a CCS-to-CRS translator, one can access to these non-consecutive locations one after another. In our implementation, we use a helper array `ccs_trans` of size τ to convert the CCS-locations to CRS-locations and access the correct Δ_{ij} values in the same column. The lock-free implementation of this task is given in Figure 2a.

With pinpoint analysis, we identified the main bottleneck of the lock-free code in Figure 2a as the memory updates in the innermost loop, which is expected since one needs to perform two data loads (from `myW` and `myZ2`) and a store (to



```
void Task3(SparseMatrix* A, prec_t* W, prec_t* Z2) {
#pragma omp parallel for schedule(runtime)
  for (coord_t j = 0; j < A->n; j++) {
    prec_t *myZ2 = Z2 + (j * r);
    memset (myZ2, 0, sizeof(prec_t) * r);

    point_t start = A->ccs_ptrs[j];
    point_t end = A->ccs_ptrs[j + 1];
    for (point_t p = start; p < end; p++) {

      const prec_t *myW = W + (A->ccs_rowids[p] * r);
      prec_t dt = delta[A->ccs_trans[p]];

      for (int k = 0; k < r; k++) {
        myZ2[k] += myW[k] * dt;
      }
    }
  }
}
```

(a) Lock-free parallelization

```
for (p = start; p < end - 3; p += 4) {
  const prec_t *W_1  = W + (A->ccs_rowids[p] * r);
  const prec_t *W_2  = W + (A->ccs_rowids[p + 1] * r);
  const prec_t *W_3  = W + (A->ccs_rowids[p + 2] * r);
  const prec_t *W_4  = W + (A->ccs_rowids[p + 3] * r);

  const prec_t dt_1 = delta[A->ccs_trans[p]];
  const prec_t dt_2 = delta[A->ccs_trans[p + 1]];
  const prec_t dt_3 = delta[A->ccs_trans[p + 2]];
  const prec_t dt_4 = delta[A->ccs_trans[p + 3]];

  for (int k = 0; k < LDIM; k++) {
    myZ2[k] += (W_1[k] * dt_1 + W_2[k] * dt_2 +
                W_3[k] * dt_3 + W_4[k] * dt_4);
  }
}
```

(b) With loop unrolling

Fig. 2: The lock-free parallelization of the third task is given on the left. The unrolled form of its middle loop which performs four iterations at once is given on the right. For simplicity, only the first part of the loop is given and the part that completes the remaining $|\Delta_{:,j}| \bmod 4$ iterations is omitted.

myZ2) for each update. To reduce the accesses to/from myZ2, we unroll the middle loop and process multiple Δ and W values in the same line. In this way, we reduce the number of accesses to myZ2 by at most $4\times$. The loop-unrolled version of the third task is given in Figure 2b. A similar loop-unrolling mechanism is applied to the integrated implementation of the first and second tasks, but a detailed explanation is omitted in the paper due to space limitations.

4 Experimental Results

All the simulation experiments in this section are performed on a single machine running on 64 bit CentOS 6.5 equipped with 384GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz, where each socket has 15 cores (30 in total). Each core has a 32kB L1 and a 256kB L2 cache, and each socket has a 30MB L3 cache. All the codes are compiled with gcc 4.9.2 with the -O3 optimization flag enabled. For parallelization, we used OpenMP with (dynamic, 16) scheduling policy. For each datapoint in the figures and tables, we perform five experiments and presented the average.

We first investigate the impact of loop-unrolling. From now on, the integrated Δ and $Z1$ computation will be denoted as dtZ1. Similarly, we will use Z2 to denote the third task of Section 3.1. Figure 3 and 4 show the execution times of the lock-free implementation and its loop-optimized version for dtZ1 and Z2, respectively. As the figures show, unrolling the loop and perform four iterations at once significantly improves the performance of both tasks.[3]

[3] We repeated this experiment by performing eight iterations at once but no further improvement is observed.

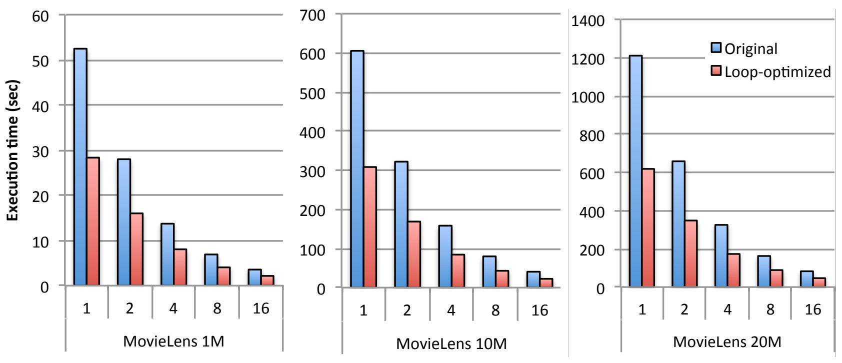


Fig. 3: The impact of loop-unrolling on the integrated `dtZ1` computation for all three datasets and $1, 2, 4, 8$ and 16 threads.

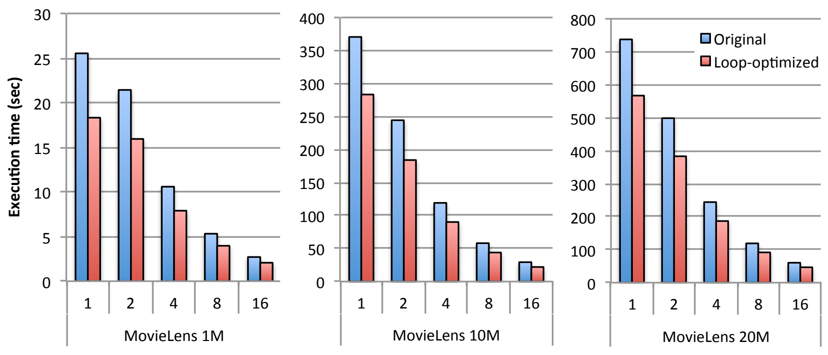
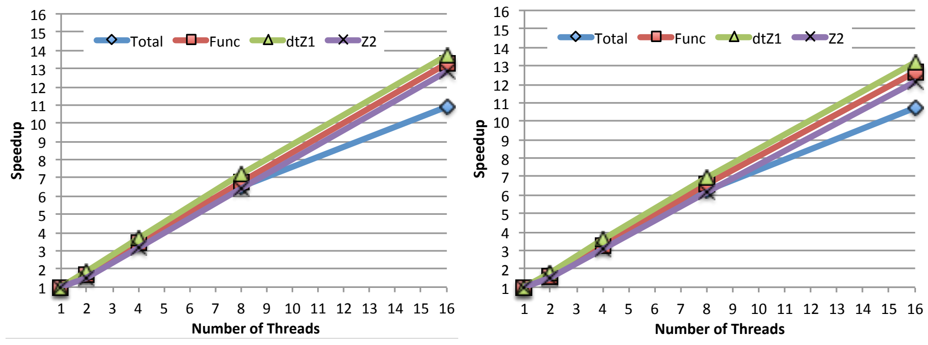


Fig. 4: The impact of loop-unrolling on the `Z2` computation for all three datasets and $1, 2, 4, 8$ and 16 threads.

The individual speedups of `dtZ1`, `Z2`, as well the overall speedup of the whole matrix factorization process, are given in Figure 5 for MovieLens 10M and 20M datasets. As the figures show, with 16 threads, the speedup for the combined `dtZ1`, `Z2` (`Func`) is around 13× whereas the overall speedup is around 11×. The overall speedup is smaller since except `dtZ1` and `Z2`, PMB performs only vector dot products which is a memory-bounded task infamous about its bad scalability.

We also experimented with a single-precision PMB implementation to see its impact on the performance. As expected, the performance is significantly improved; the performance is 1.36×, 1.52×, and 1.90× better for `dtZ1`, `Z2`, and preconditioning, respectively, compared to the double-precision variant. Although this improvement comes with a possible reduction on the accuracy, this is not the case for the datasets as the following experiment shows.

Using this implementation, we factorize each of the 1M, 10M, and 20M rating matrices with factorization ranks 20 and 100. Each factorization is repeated 50



(a) MovieLens 10M (b) MovieLens 20M

Fig. 5: Individual speedups with MovieLens 10M (left) and 20M (right) datasets and $1, 2, 4, 8$ and 16 threads for `dtZ1` and `Z2`. The charts also show their combined speedup (`Func`), and the overall speedup of whole execution.

		Double precision			**Single precision**			**Improvement**		
Dataset	**#threads**	**dtZ1**	**Z2**	**Pre.**	**dtZ1**	**Z2**	**Pre.**	**dtZ1**	**Z2**	**Pre.**
1M	1	28.3	18.3	5.6	22.7	12.5	2.9	1.25	1.46	1.93
	16	2.2	2.1	0.8	1.6	1.4	0.4	1.38	1.50	2.00
10M	1	308.2	283.3	57.7	230.3	174.0	33.1	1.34	1.63	1.74
	16	22.5	22.1	9.9	16.1	15.1	4.8	1.40	1.46	2.06
20M	1	618.0	567.1	132.1	462.0	359.4	70	1.34	1.58	1.89
	16	46.9	46.7	19.3	32.4	31.2	10.6	1.45	1.50	1.82
							Average improvement	1.36	1.52	1.90

Table 3: The execution times (in secs.) for `dtZ1`, `Z2`, and the Preconditioning phase of the loop-unrolled version when double and single precision arithmetic and data representation is used for single and 16-thread version.

times with randomized initial conditions. The final RMSE values are then found by averaging, and confidence intervals are determined by bootstrap resampling. The algorithm stops when the number of iterations reaches 500 or when the absolute value of the largest element of the function gradient drops below 10^{-5}. Table 4 displays the results for this experiment.

We see that using single-precision version of PMB does not make a significant difference in the final RMSE value for matrix factorization, compared to the double-precision version. However, in every case, the single-precision version runs faster by a factor of $1.5\times$–$2.0\times$. Therefore, in this particular problem, single-precision arithmetic can be preferred.

5 Conclusions

Preconditioned Model Building algorithm is a powerful optimizer that combines local model-building iterations with second-order information. Our results show that for the matrix factorization problem, the performance of the PMB algorithm is comparable to the best algorithms in general use.

Dataset	Rank	Precision	Mean RMSE	95% confidence interval
1M	20	single	0.681926	(0.681265, 0.682619)
		double	0.682071	(0.681296, 0.682815)
	100	single	0.335744	(0.332367, 0.339220)
		double	0.338259	(0.334817, 0.341658)
10M	20	single	0.682580	(0.682190, 0.683038)
		double	0.683356	(0.682781, 0.683952)
	100	single	0.529536	(0.469352, 0.592387)
		double	0.528092	(0.468423, 0.590662)
20M	20	single	0.670611	(0.670027, 0.671292)
		double	0.679019	(0.669434, 0.697753)
	100	single	0.673502	(0.593685, 0.751497)
		double	0.667708	(0.589608, 0.745404)

Table 4: Final RMSE results from the factorization of MovieLens matrices with factorization rank 20/100 and single/double precision.

Since it is cheap, the algorithm spends most of the execution time for the evaluation of the error and in the update of factor matrices. These computations are similar to the traditional sparse-matrix computations, therefore we can go around this bottleneck with appropriate parallelization techniques. Indeed, the PMB algorithm can be parallelized very well. The experiments show that there is little overhead thanks to the lock-free parallelization, and the speedup with 16 threads is about 11. Hence, the algorithm can be promising for large-scale optimization problems.

References

1. M. W. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, Sept. 2007.
2. D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.
3. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., New York, NY, USA, 1986.
4. F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, Dec. 2015.
5. A. Hernando, J. Bobadilla, and F. Ortega. A non negative matrix factorization for collaborative filtering recommender systems based on a bayesian probabilistic model. *Knowledge-Based Systems*, 97:188–202, 2016.
6. Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37, Aug. 2009.
7. D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, Oct. 1999.
8. D. C. Liu and J. Nocedal. On the limited-memory BFGS method for large scale optimization. *Mathematical Programming*, (45):503–528, 1989.
9. F. Öztoprak. *Parallel Algorithms for Nonlinear Optimization*. PhD thesis, Sabancı University, 2011.
10. F. Öztoprak and S. I. Birbil. An alternative globalization strategy for unconstrained optimization. *arXiv preprint, arXiv:1705.05158*, 2017. To appear in Optimization.
11. M. Schmidt. minFunc: unconstrained differentiable multivariate optimization in matlab. http://www.cs.ubc.ca/~schmidtm/Software/minFunc.html. Accessed: 2017-03-22.