J Supercomput (2013) 66:1418–1437
DOI 10.1007/s11227-012-0830-6

Semi-sparse algorithm based on multi-layer optimization for recommender system

Hu Guan · Huakang Li · Cheng-Zhong Xu · Minyi Guo

Published online: 13 October 2012
© Springer Science+Business Media New York 2012

Abstract Similarity among vectors is basic knowledge required to carry out recommendation and classification in recommender systems, which support personalized recommendation during online interactions. In this paper, we propose a Semi-sparse Algorithm based on Multi-layer Optimization to speed up the Pearson Correlation Coefficient, which is conventionally used in obtaining similarity among sparse vectors. In accelerating the batch of similarity-comparisons within one thread, the semi-sparse algorithm spares out over-reduplicated accesses and judgements on the selected sparse vector by making this vector dense locally. Moreover, a reduce-vector is proposed to restrict using locks on critical resources in the thread-pool, which is wrapped with Pthreads on a multi-core node to improve parallelism. Furthermore, among processes in our framework, a shared zip file is read to cut down messages within the Message Passing Interface package. Evaluation shows that the optimized multi-layer framework achieves a brilliant speedup on three benchmarks, Netflix, MovieLens and MovieLen1600.

Keywords Semi-sparse algorithm · Reduce vector · Thread pool · Message passing interface · Pearson correlation coefficient

H. Guan · H. Li · M. Guo (✉)
Department of Computer Science & Engineering, Shanghai Jiao Tong University, Dongchuan 800, Shanghai, P.R. China
e-mail: guo-my@cs.sjtu.edu.cn

H. Guan
e-mail: guanhu@sjtu.edu.cn

C.-Z. Xu
Department of Electrical and Computer Engineering, Wayne State University, 5050 Anthony Wayne Dr., Detroit, MI, USA
e-mail: czxu@wayne.edu

1 Introduction

Collaborative Filtering (CF) algorithms based on similarity [22] have been successfully adopted by some commercial systems, including Amazon [14], TiVo [16] and Netflix [13]. CF relies on the past user behavior to give the recommendation. Given enough ratings per item, CF becomes the preferred and accurate technique [12]. However, a recommender system has to manage a massive, growing and changing data set because of the explosive growth of the World Wide Web. One challenge of recommender systems is to improve the scalability and speedup of the collaborative filtering algorithms.

A conventional CF algorithm is used to search for neighbors among a large population of potential candidates. A fast but basic CF algorithm, such as Pearson Correlation Coefficient [9] (PCC), can help to afford a frequent but massive update on the growing data set for a recommender system. If the PCC algorithm [5, 26] can be accelerated, many CF algorithms will benefit. As complained about in a recent survey [22], an item-based PCC [14, 20] (IPCC) is expensive in time complexity. A semi-sparse algorithm [25] can accelerate a series of comparisons among sparse vectors by making the selected sparse vector into a dense vector locally. However, to date it has only been used with machine learning algorithms in text-classification scenarios.

In this paper, we provide a multi-layer optimization to accelerate a recommender system. It has three optimizing methods at different levels to achieve a large overall speedup. For message transfer among different nodes, we make full use of the sparse data required by PCC and cut down the overhead by reading a shared zip file. For the basic-level thread on a node, a semi-sparse algorithm is suggested to speed up sparse vector similarity-comparison, which is the fundamental overhead in a recommender system. Moreover, in a node based on a multi-core architecture, we wrap trivial Pthreads into a stable thread-pool using a reduce-vector to reduce the chance of synchronizing threads with mutex-locks on those critical resources.

Experimental results on Netflix and MovieLens give convincing evidence that the original IPCC is significantly accelerated in an online recommendation. Specially, in Netflix, the recommender system based on IPCC takes only 1272 seconds to handle 4,974,870 recommendations for 21,280 test users on an 8-node cluster, including system initialization. For a scale of rating matrix $R_{425,596\times17,770}$, this system completes one recommendation for an active user within 0.06 seconds on average. We accelerated the original IPCC with a speedup $32\times$ using the framework. Furthermore, compared with the conventional sparse algorithm [4], a grid analysis was also done for the semi-sparse algorithm to illustrate the rationale of speedup.

The remainder of the paper is organized as follows. In the next section, the basic notions and notation will be described, and related work will be summarized. In Sect. 3, we elaborate the design based on motivations. After an introduction to the experimental setting in Sect. 4, Sect. 5 evaluates the speed of recommendation. Lastly, Sect. 6 concludes our work and discusses future research directions.

2 Background and related work

In a recommender system, given an active user for whom we have limited information (some given rated items $I_{(a)}$), a Pearson Correlation Coefficient based on z-score (mean 0, standard deviation) [9] has been a basic role of finding similar neighbors in a rated matrix $R_{m\times n}$, as stated in Eq. (1):

$$\mathrm{sim}(a,u)=\frac{\sum_{i\in I_{(a)}\cap I_{(u)}}(r_{a,i}-\overline{r_a})(r_{u,i}-\overline{r_u})}{\sqrt{\sum_{i\in I_{(a)}\cap I_{(u)}}(r_{a,i}-\overline{r_a})^2}\sqrt{\sum_{i\in I_{(a)}\cap I_{(u)}}(r_{u,i}-\overline{r_u})^2}}, \tag{1}$$

where $\mathrm{sim}(a,u)$ is a similarity metric in the common set $I_{(a)}\cap I_{(u)}$ between the active user a and a candidate u; $r_{a,i}$ is the given rated item i for the active user a; and $\overline{r_u}$ is the average rating for the user u. For an item-based PCC, the similarity of between the items i and j can be stated as:

$$\mathrm{sim}(i_l,i_j)=\frac{\sum_{u\in U_{(i_l)}\cap U_{(i_j)}}(r_{u,l}-\overline{r_l})(r_{u,j}-\overline{r_j})}{\sqrt{\sum_{u\in U_{(i_l)}\cap U_{(i_j)}}(r_{u,l}-\overline{r_l})^2}\sqrt{\sum_{u\in U_{(i_l)}\cap U_{(i_j)}}(r_{u,j}-\overline{r_j})^2}}, \tag{2}$$

where $\mathrm{sim}(i,j)$ is the similarity in the comment set $U_{(i)}\cap U_{(j)}$ between the items i and j; $\overline{r_i}$ is the average rating for the item i in the training set; and $\overline{r_j}$ is the average rating for all users on the item j in the training set.

As illustrated in the above equations, PCC has a space-complexity $\mathcal{S}(mn)$ [5, 26] and a time-complexity $\mathcal{O}(mn^2)$. PCC based on a distributed platform has good scalability because of the low overhead on space. Moreover, PCC has an eminent parallelism because the main overhead on computation is similarity comparison, and the good parallelism on similarity comparison has been generally validated by the Map-Reduce methods [19]. However, as shown in a recent survey [22], an item-based PCC (IPCC) is still expensive in time complexity.

In the conventional implementation of similarity-comparison based on Eq. (2), a traditional sparse algorithm [4] will be used to carry out the comparison.

Because it does the fewest multiplications between two sparse vectors, the traditional Algorithm 1 is conventionally viewed as efficient based on the compact structure. We observe that elements in an active user's items i_a and i_u will be visited and judged once. In the worst case, to finish the comparison for two sparse vectors, there are $|I_{(a)}|+|I_{(u)}|$ times of access and judgement. Due to many judgements, embedded judgments in the computing process, and the irregular structure of sparse matrix operations [8], the traditional Algorithm 1 based on the compact structure seems not to be easily accelerated by GPU methods [6, 23] or CUDA methods [24].

Given $R_{m\times n}$, to find the similar neighbors $S_{(a)}$, in the worst case, there should be $\sum_{u=1}^{m}(|I_{(a)}|+|I_{(u)}|)=m|I_{(a)}|+\rho$ times of access and judgement, where $\rho=\sum_{u=1}^{m}|I_{(u)}|$ is the number of elements in the training matrix $R_{m\times n}$. Thus, elements in $I_{(a)}$ will be traversed $m-1$ times repeatedly. This could be an important reason why IPCC is still viewed as a time-consuming method. If the repeated traverse and judgement can be removed, similarity comparison can be accelerated.

We take the semi-sparse [25] algorithm used in text categorization and explore it in a CF algorithm. It can avoid the repeated traverse and judgement in similarity

Algorithm 1 Traditional dot product on two sparse vectors.

Require: index[a], val[a], index[u], val[u]

```
float sim = 0.0;
for (i = 0, j = 0) to (i < |I_(a)| && j < |I_(u)|) do
    if index[a][i] == index[u][j] then
        sim = sim + val[a][i] * val[u][j];
        j = j + 1;
        i = i + 1;
    else
        if index[a][i] > index[u][j] then
            j = j + 1;
        else
            i = i + 1;
        end if
    end if
end for
return sim
```

comparison, though validity is only checked in the training process of the text classification. In a training process of classifying tasks, if the training process can be accelerated by making a sparse selected vector into a dense vector in the similarity comparison, why cannot we get a kind of acceleration in the recommendation, by making the selected vector $I_{(a)}$ into a dense vector as demanded in the semi-sparse algorithm? Because in a CF algorithm, a selected vector $I_{(a)}$ will also be compared with other vectors in a large training set $R_{m\times n}$. This motivation encourages us to try a semi-sparse algorithm in the recommendation domain.

3 Design

Following the above analysis, we revamp the traditional sparse algorithm into semi-sparse algorithm, and then, multi-layer optimizations are designed to get a speedup on a cluster. The rationale of semi-sparse's acceleration compared to the traditional sparse algorithm will be presented first.

3.1 Semi-sparse algorithm for PCC

In the traditional sparse algorithm [4], elements in an active user's items i_a and i_u will be visited and judged once. So, in the worst case, to finish the comparison for two sparse vectors, there are $|I_{(a)}| + |I_{(u)}|$ times of access and judgement. Given a rating matrix $R_{m\times n}$, to find similar neighbors $S_{(a)}$, in the worst case, there are $\sum_{u=1}^{m}(|I_{(a)}| + |I_{(u)}|) = m|I_{(a)}| + \rho$ times of access and judgement, where $\rho = \sum_{u=1}^{m}|I_{(u)}|$ is the number of elements in the training matrix $R_{m\times n}$. Thus, elements in $I_{(a)}$ will be traversed and judged $m-1$ times repeatedly. However, for a sparse matrix $R_{m\times n}$, semi-sparse algorithm can change the time complexity $\mathcal{O}(m|I_{(a)}| + \rho)$ into $\mathcal{O}(|I_{(a)}| + n + \rho) \approx \mathcal{O}(\rho)$ in finding similar neighbors $S(a)$.

Algorithm 2 Dot product on two sparse vectors in semi-sparse algorithm.

Require: index, val, n, m, index$a$, val$a$

```
    //prepare a workplace vector for a batch of dot product
 1: float * workplace = new float[n];
 2: memset(workplace, 0, n * size of(float));
    //put the active user's data onto the workplace
 3: for (i = 0) to |I_(a)| step 1 do
 4:     workplace[indexa[i]] = vala[i];
 5: end for
 6: float sum = 0.0;
    //get all comparison in similarity vector
 7: for (i = 0) to m step 1 do
 8:     sum = 0.0;
 9:     for (j = 0) to |I_(u)| step 1 do
10:         sum = sum + workplace[index[j]] * val[j];
11:     end for
12:     similarity[i] = sum;
13: end for
14: return similarity
```

In Algorithm 2, the main characteristic is making the selected sparse vector $I_{(a)}$ into a dense vector workspace locally, while other sparse vectors are kept sparse. In fact, semi-sparse uses an extra dense-workspace vector to obtain an acceleration on a batch of doc products.

Although an extra vector workspace and a batch number m are needed, comparing with the traditional sparse algorithm, excessive duplication of access and judgement can be avoided in finding similar neighbors among a large number of candidates. Thus, to find similar neighbors $S(a)$ for an active user a, the time complexity for semi-sparse is $\mathcal{O}(|I_{(a)}| + n + \rho) \approx \mathcal{O}(\rho)$, because $(|I_{(a)}| + n) \ll \rho$. So, compared with the traditional sparse algorithm, the speedup for the semi-sparse algorithm will be $1 + \frac{m|I_{(a)}|}{\rho}$. Given a very small m or n, semi-sparse will not be economic because of the required vector workspace in Algorithm 2.

The traditional sparse algorithm can be implemented effectively on map-reduce framework, while the semi-sparse algorithm can be parallelized easily by the following simple thread-pool.

3.2 Thread-pool for semi-sparse

The thread-pool we designed manages a homogeneous pool of worker threads, reduces the thread creation and teardown overhead, and limits the resource consumption as the number of restricted threads. This leads to improved stability for a fast recommender system.

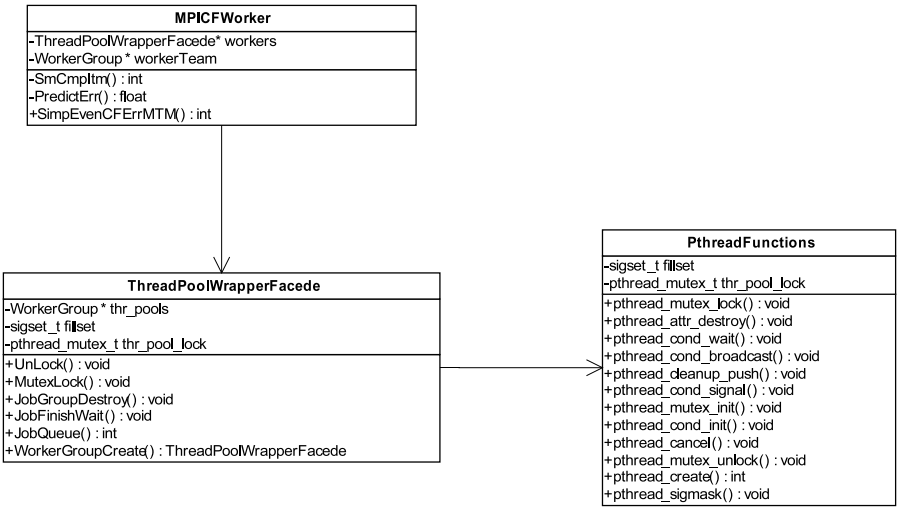

Fig. 1 Design the special thread-pool with WrapperFacede pattern

3.2.1 Design and implementation

Generally, the WrapperFacade pattern [21] is used in the design of a simple thread-pool. This implementation is based on Pthread, and the class-diagram is given in Fig. 1.

As demonstrated in Fig. 1, the schedule policy for those threads is discarded for brevity while locks and barriers are implemented in the thread-pool based on Pthread. The whole thread-pool is wrapped into one class ThreadPoolWrapperFacade, which can be composited by other classes more easily with a singleton pattern [11] in implementation, and the interfaces are listed in Table 1.

3.2.2 Semi-sparse algorithm based on thread-pool

The data flow shows the brevity in the framework, as illustrated in Fig. 2. Compared with a standard map-reduce process for a multi-core or multi-processor system [19], the map stage is merged into the splitting process, while the multi-reduce process is finished in a special reduce-vector.

To reduce the cost of communication, a dense reduce-vector is designed to store the temporary result returned from a thread. For example, in MovieLens, an active user a will be compared with 63,709 candidates, and then, a reduce-vector $\text{sim}_{1\times 63,709}$ will be applied for storing the similarity sim_i between users u_i and u_a. When a worker/thread worker[i] finishes a comparison for u_i and u_a, this similarity will be set on the position sim_i for this shared reduce-vector sim. Moreover, to finish 63,709 similarity comparisons for an active user u_a, a worker/thread will be assigned a series of comparison according to the user's identifier u_i. In providing a series of comparisons for a thread/worker, the semi-sparse algorithm is superior to the traditional sparse algorithm.

Table 1 ThreadPoolWrapperFacade

Methods	Descriptions
WorkerGroupCreate	Create a work team for different jobs
Param: int min_threads	Minimum stable threads in this work team
Param: int max_threads	Maximum threads can be applied
Param: int linger	Time for desolate a idle thread
Param: pthread_attr_t *attr	A thread's attribute
JobQueue	Assign a FIFO queued jobs to the work team
Param: WorkerGroup *pool	The target work team
Param: void *(*func)(void *)	The job to be assigned
Param: void *arg	The parameter for this job
JobFinishWait	A barrier synchronize for those threads in queue
Param: WorkerGroup *pool	The target work team
JobGroupDestroy	Tear down those threads in queue
Param: WorkerGroup *pool	The target work team
MutexLock	Add a lock for the critical source for a thread
UnLock	Unlock the critical source for other threads

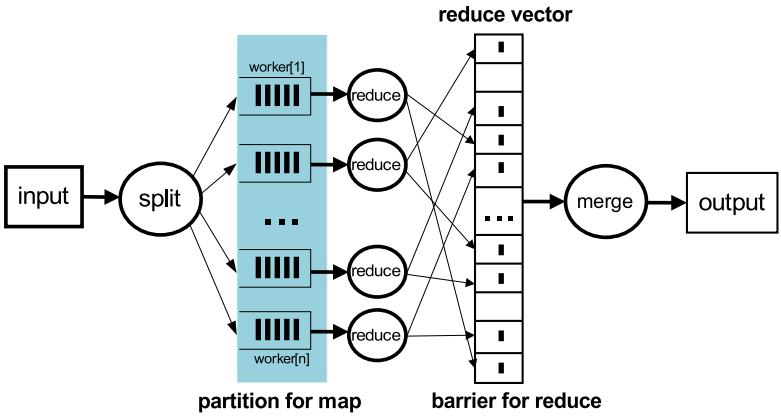

Fig. 2 Data flow in IPCC based on thread-pool

Algorithm 3 demonstrates the above ideas. First, we evenly divide the whole comparison for u_a into NumOfCore parts, and then, each core will accomplish the assigned comparison according to the core's ID.

Then, for every worker/thread based on ThreadPoolWrapperFacade, a semi-sparse algorithm is used to get the last comparison. The barrier JobFinishWait(workerTeam) is used for the work team workerTeam for the subsequent merging of the reduce-vector.

As illustrated in Algorithm 4, with a semi-sparse algorithm, a thread/worker takes charge of parts of the comparison according to the ID of the core in the recommender system. After this worker finish a comparison between an active user u_a (put on the

Algorithm 3 An example of ThreadPoolWrapperFacade.

```
Require: workers //an object of ThreadPoolWrapperFacade
Require: workerTeam //generated by WorkerGroup
    //CFSimObj is a wrapped struct for the argument of thread-pool
 1: CFSimObj * ssim * [NumOfCore];
    //get all comparisons for an active user in Multi-core system
 2: for (i = 0) to NumOfCore step 1 do
 3:    ssim[i] = new CFSimObj(i, sim, workplace, index, val);
       //sim is the reduce-vector
       //workplace is the dense vector initialized with an active user
       //index is the index matrix for a sparse rating matrix R
       //val is the value matrix for a sparse rating matrix R
 4:    workers->JobQueue(workerTeam, getSm, ssim);
       //getSm is a function interface for a thread
 5: end for
 6: workers->JobFinishWait(workerTeam);
 7: delete[ ] ssim;
```

Algorithm 4 Semi-sparse based on ThreadPoolWrapperFacade.

```
Require: workplace, *arg, NumOfCore
    // *arg is the argument of the function getSm
    // getSm is defined as void * getSm(void * arg)
 1: CFSimObj * cf = ((CFSimObj*) arg);
 2: float sum = 0.0;
    //get the core of ID
 3: int CoreID = cf->IDOfCore;
    //get the number of comparisons for an active user
 4: for (i = 0) to cf->index->row step 1 do
 5:    if (i % NumOfCore)! = CoreID then
 6:       continue;
          //a worker only takes the charge of comparison in its parts
 7:    end if
 8:    sum = 0.0;
 9:    for (j = 0) to sizeof(cf->index[i]) step 1 do
10:       sum = sum + workplace[cf->index[i][j]] * cf->val[i][j];
11:    end for
12:    cf->sim[i] = sum;
       //similarity is set on ith element in reduce-vector sim
13: end for
```

workplace vector) and u_i, the similarity sum is set to the ith element in the reduce-vector sim.

3.3 Concurrency and locality management

For a commercial recommender system for many online users, the paradigm we offer is based on the MPI library [1, 17]. To lessen the number of message among processes for a data-intensive recommendation, each node will have one copy of the sparse rating matrix $R_{m\times n}$. PCC is a CF algorithm based on a sparse rating matrix $R_{m\times n}$. A sparse rating matrix R can have good scalability because the rating information is very sparse. Thus, PCC has good scalability for a recommender system. For example, in the Netflix data set, the rated movies are less than 2 % of the sparse matrix $R_{480,189\times 17,770}$, and the memory used is less than 2 GB with a sparse format. However, the sparse matrix R still occupies memory as the scale increases, so one node has only one copy of the zipped matrix R in the design. As for the parallelism in a node, PCC based on the thread-pool is used for carrying out the prediction for each user.

Algorithm 5 IPCC based on MPI with an embedded ThreadPoolWrapperFacade.

Require: $R_{m\times n}$, $T_{l\times n}$, givenLength, process_id
//$T_{l\times n}$ is a set of testing or online users
//assign the predicting tasks among nodes in cluster
1: **for** ($i = 0$) to l step 1 **do**
2: **if** (i % process_id) == me **then**
3: SmCmpItm(index, val, givenlength, sim, ...);
//me is the process ID in MPI
//find similar candidates $S_{(a)}$ for an active user
4: getErr(errorElements, ...);
//get the error elements to predict the error
5: predict(sim, errorElements, ...);
//predict a $r_{a,i}$ for the active user with IPCC
6: getStatistic(sim, errorElements, ...);
//get the statistical information for MAE or RMSE
7: **end if**
8: **end for**
9: MPI_Barrier(MPI_COMM_WORLD);
//send statistical information to root process (process_id = 0)
10: **if** me > 0 **then**
11: MPI_Send(stat, ...);
//stat is an array for storing different statistical information
12: **else**
13: **for** ($j = 1$) to process_num step 1 **do**
14: merge(stat);
//merge statistic information from other processes
15: MPI_Recv(stat, ...);
16: **end for**
17: **end if**
18: MPI_Barrier(MPI_COMM_WORLD);

Table 2 Multi-core systems used in evaluation

Items	Centrino	Xeon	Athlon
CPU type	Intel	Intel	AMD
Core Num	2	8	4
L1 size	16 k	16 k	16 k
L2 size	3 M	1 M	1 M
Clock Freq. (MHz)	2.9	2.0	2.6
Memory	4.0 G	8.0 G	2.0 G

Algorithm 5 gives a global view of PCC based on MPI. In this parallel design, the cost of communication among processes is limited to merging the statistical information, and there are no locks except the barrier for the collective communications in different stages.

In the next section, we will discuss how to appraise these approaches for a recommender system.

4 Methodology

This section describes the experimental methodology we used to appraise our semi-sparse algorithm in the recommendation domain. Our system would work without modifications on an architecture that supports the MPI library with Pthreads.

4.1 Multi-core systems

To evaluate a semi-sparse algorithm based on the specific thread-pool, we ran our recommender system on the three multi-core systems described in Table 2. Different multi-core systems allow us to evaluate whether the framework can deliver its fast speed for a recommender system.

4.2 Distributed systems

MPICH2[1] provides a standard for low-level inter-processor communication in a cluster. The cluster we used has eight nodes equipped with Intel Xeon 2.0 GHz CPU and 8 GB memory. In this cluster, based on Network Information Services (NIS) [2], the Network File System (NFS) [15] gives each node a copy of the rated matrix R, obtained by reading a zip file with zcat [10] directly.

4.3 Applications

In the recommendation application, we selected IPCC as a test semi-sparse implementation since IPCC was viewed as a time-consuming algorithm for recommendation. However, three drastically different benchmarks allow us to evaluate whether

[1] http://www.mcs.anl.gov/research/projects/mpich2/.

a semi-sparse recommender system can deliver on its promise: a semi-sparse algorithm for collaborative filtering can achieve a considerable speedup compared with the state-of-the-art sparse algorithm, while a scalable framework for semi-sparse is very fast in handling large-scale recommendation for online users in a recommender system.

In classification, SVMTorch within a Gaussian kernel [4] is generally time-consuming. SVMTorch can also benefit from a semi-sparse algorithm because, in the Vector Space Model (VSM), similarity for different sparse vectors is generally used in different classifiers.

4.4 Data set

The three classical benchmarks for recommendation applications were used to evaluate the performance of an item-based CF.

MovieLens (MLens). MovieLens [2] contains 95,580 tags applied to 10,681 movies by 71,567 users of the online movie recommender service MovieLens.

Movie1600 (M1600). MovieLens1600 was collected through the MovieLens web site (movielens.umn.edu) during the seven-month period from September 19th, 1997 to April 22nd, 1998.

Netflix. Netflix[3] was collected by Netflix in a period of approximately 7 years.

Furthermore, in classification, a skewed data set and a balanced data set are used to validate the efficiency of semi-sparse in classifying tasks.

WebKB. The WebKB data set[4] contains web pages gathered from university computer science departments. The pages are divided into seven categories: student, faculty, staff, course, project, department and other. After the parsing process, there are 8203 pages (7031 training and 1172 testing) left in the corpus. For the "Title","H1", and "URL" parts in the page, we give a weight five times more than for those in "Body". We kept 28,473 unigram terms that occurred at least once in the training set.

20-newsgroup. This data set from 20 Usenet newsgroups[5] consists of 19,997 text messages (about one thousand text messages per category), and approximately 4 % of the articles are cross-posted. The stop words list has 823 words, and we kept words that occurred at least once and messages that had at least one term. Altogether, there are 19,899 messages (13,272 training and 6637 testing) left in the corpus. We only keep "Subject", "Keywords", and "Content". The total number of unigram terms is 31,138. Words in "Subject" and "Keywords" are given a weight five times more than those in "Contents".

4.5 Parameter settings

In these 5-star-rating benchmarks, those users who rated at least 21 movies will be kept in R, and the test users (active users or newcomers) were selected out evenly

[2] http://www.grouplens.org/node/73.

[3] http://www.netflixprize.com.

[4] http://www.cs.cmu.edu/afs/cs/project/theo-20/www/data.

[5] http://kdd.ics.uci.edu/databases/20newsgroups.

Table 3 Different scale of benchmarks

Names	#Ratings	Scale	Train	Test
Movie1600	100,000	$R_{943\times1682}$	865	45
MovieLens	10,000,054	$R_{71567\times10681}$	63,709	3320
Netflix	100,480,507	$R_{480,189\times17,770}$	404,316	21,280

from a filtered R (there is one testing user in every 20 users), as summarized in Table 3. An active user is a new user or a newcomer for a recommender system: this is a cold start problem in model-based CF algorithms [7, 22, 26]. The given items for an active user $I_{(a)}$ were selected randomly from this testing user's item-list, while the size of $I_{(a)}$ was set to 10 in all evaluation on speed.

For the above two classification corpora, we used the tokenizer tool provided in the Trinity College sample. Stemming and word clustering were not applied. Original document vectors were constructed with TF-IDF score. In the training process, we varied the training error value from default 0.01 to 0.1.

5 Evaluation

In this section, the following points will be presented:

- Appraise the effectiveness of a semi-sparse algorithm for IPCC by comparing with the traditional sparse algorithm and validate semi-sparse's effectiveness in an SVMTorch classifier.
- Illustrate that the semi-sparse method with a thread-pool is more effective than Pthread.
- Demonstrate that the semi-sparse method using MPI with a thread-pool can give a good speedup and scalability.

All speedups are calculated with respect to the sequential code for recommendation.

5.1 Performance overview

Given $|I_{(a)}| = 10$, Table 4 presents the speedup (higher is better) of IPCC as we scale the rating matrix $R_{m\times n}$ on the cluster. A speedup is calculated compared to the sequential code based on the traditional sparse algorithm.

Results in Table 4 show that a recommender system based on semi-sparse provides a good speedup in all benchmarks. For example in Netflix, on a scale of rating matrix $R_{425,596\times17,770}$, it costs only 1272 seconds to give 4,974,870 recommendations for 21,280 testing users. The speedup in Netflix is 32.7, and this included the elapsed CPU time on initialization for loading $R_{m\times n}$ into memory. Semi-sparse is more effective for a large scale data set, because it has a relatively low speedup in MovieLen1600 though the speed of recommendation is extremely fast.

The following sections will give a detailed explanation for this brilliant performance.

Table 4 Overview of speedup

Items	MovieLen1600	MovieLens	Netflix
$R_{m\times n}$	6041 × 3706	63,709 × 10,681	425,596 × 17,770
#user	604	3320	21,280
#recommend	48,635	466,976	4,974,870
time-cost(s)	3	24	1272
speedup	2.7	22.3	32.7

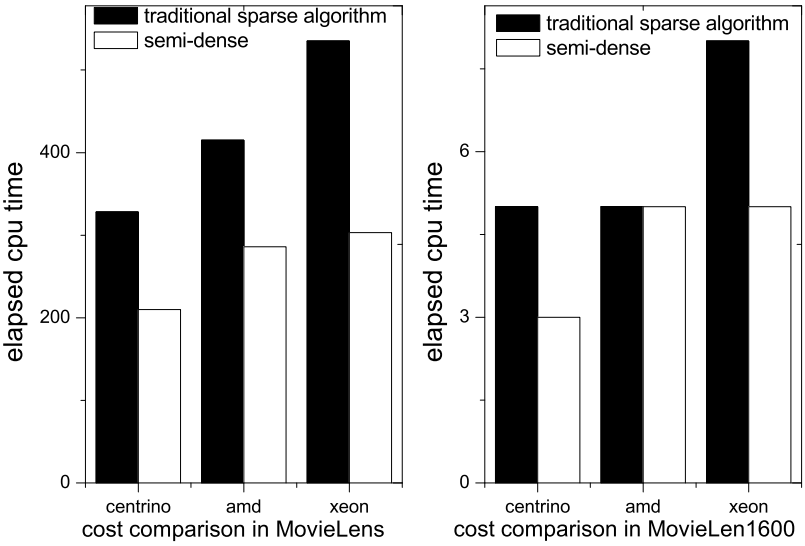

Fig. 3 Comparison between traditional sparse algorithm and semi-sparse algorithm

5.2 Semi-sparse algorithm

Compared with the traditional sparse algorithm, the semi-sparse algorithm we provided can deliver a faster speed. This section will appraise this promise.

5.2.1 Effectiveness in collaborative filtering

Figure 3 illustrates semi-sparse's speedup compared with a typical sparse algorithm. IPCC was run on a single thread with a different sparse algorithm. The result from three types of CPU shows that the semi-sparse algorithm is much faster than the traditional sparse algorithm. The reason lies in that, on the selected vector in the batch of multiple processes, the semi-sparse algorithm spares out an excess of judgement and traverse.

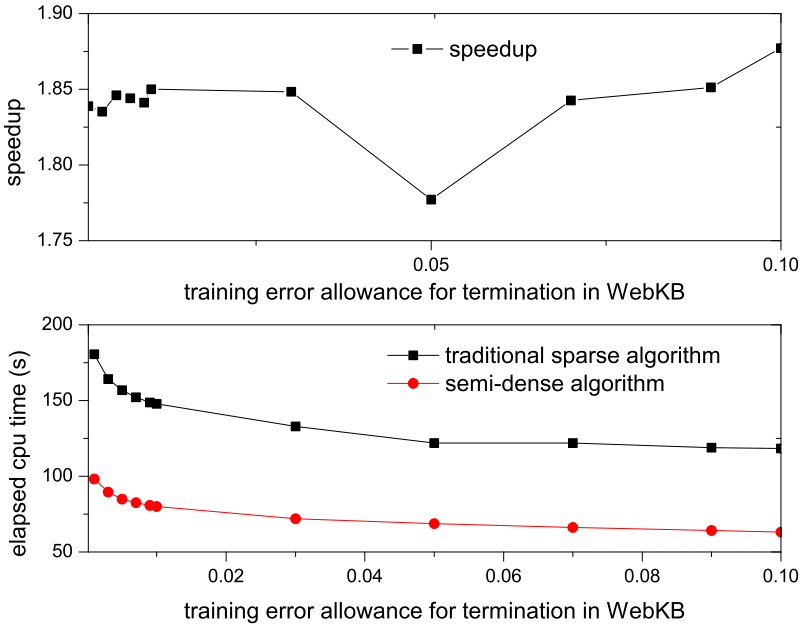

Fig. 4 Comparison on training time between traditional sparse algorithm and semi-sparse algorithm with different training error in WebKB

5.2.2 Effectiveness in classification

We select the SVMTorch classifier as an illustration to demonstrate that a semi-sparse algorithm can boost a classifier's performance in the training process, as illustrated in Fig. 4. The modified SVMTorch based on semi-sparse was implemented with C++ and run on a personal computer with 2.33 GHz Pentium CPU and 2 GB memory in Windows XP.

In SVMTorch's training process, the most time-consuming step should be getting the training model in Gaussian kernel space. The elapsed CPU time in this process is extracted out in Fig. 4 to illustrate that, though there is fluctuation on speedup, a semi-sparse algorithm can boost this process dramatically.

In 20-newsgroup, Fig. 5 shows that a semi-sparse algorithm has a smaller speedup than in WebKB, but a semi-sparse algorithm also accelerates the training process significantly.

Furthermore, the training process based on a semi-sparse algorithm will keep the original SVMTorch's accuracy for classifying tasks. For example, SVMTorch based on semi-sparse has the same accuracy in WebKB or 20-newsgroup as the original SVMTorch. While, in PSVM [3], ICF will give the training process a reduced accuracy for the classifying tasks if the dimension is small.

5.3 Thread-pool

In this section, we show that a semi-sparse algorithm can be readily integrated into a thread-pool, which is an important contribution to the brilliant performance.

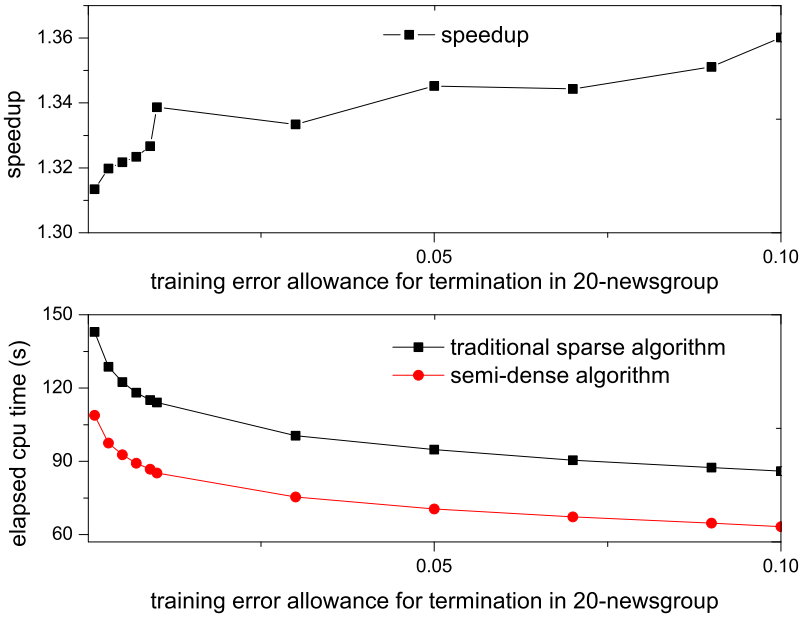

Fig. 5 Comparison on training time between traditional sparse algorithm and semi-sparse algorithm with different training error in 20-newsgroup

The following speedups are calculated based on the semi-sparse algorithm, not the traditional sparse algorithm. Although these values are extracted from different multi-core platforms in MovieLens benchmarks, the tendency in Fig. 6 is quite clear: more cores, higher speedup.

We also compared the thread-pool with the traditional Pthread library. The result in Fig. 7 demonstrates that it is not efficient to use the special thread-pool when the number of threads is smaller than three, because the traditional Pthread library for multi-threaded programs works really well. However, if there are over three threads in a recommender system, the thread-pool can beat the traditional Pthread because the overhead for thread creation and teardown is reduced by the Thread Pool pattern [18].

5.4 Scalability

Our framework for a recommender system has good scalability in handling large data sets. MPI is used in assigning tasks on distributed nodes, while the thread-pool is embedded into every task on a node. A speedup is computed based on the sequential semi-sparse algorithm on IPCC.

In the MovieLens benchmark, Fig. 8 illustrates: speedup increases with more nodes, while the rate of growth of speedup decreases. Each node keeps a copy of $R_{m\times n}$ in the design, so there is a stable cost (8s) on initialization for reading the raw data. The cost of initialization becomes more and more obvious compared with the declining overhead on recommendation. Therefore, the speedup grows less quickly as more nodes are added.

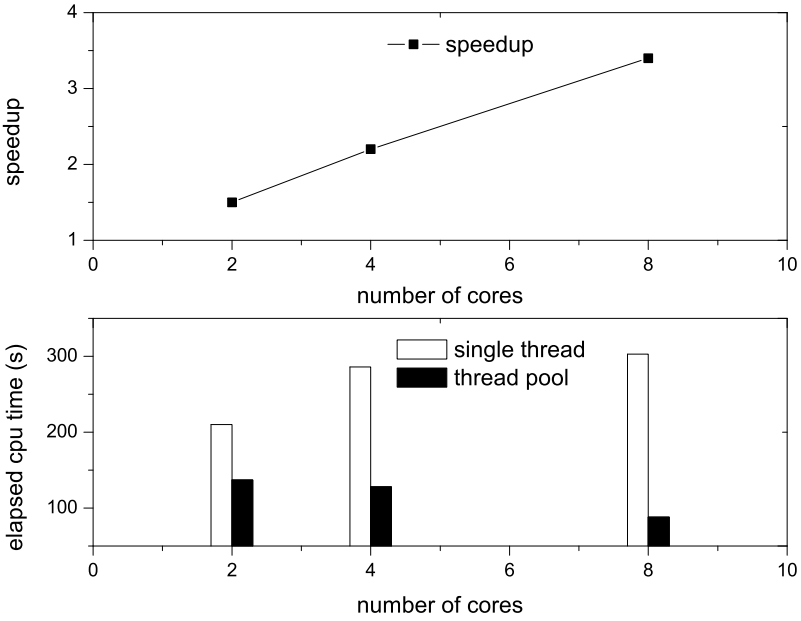

Fig. 6 Speedup for the thread-pool based on semi-sparse algorithm

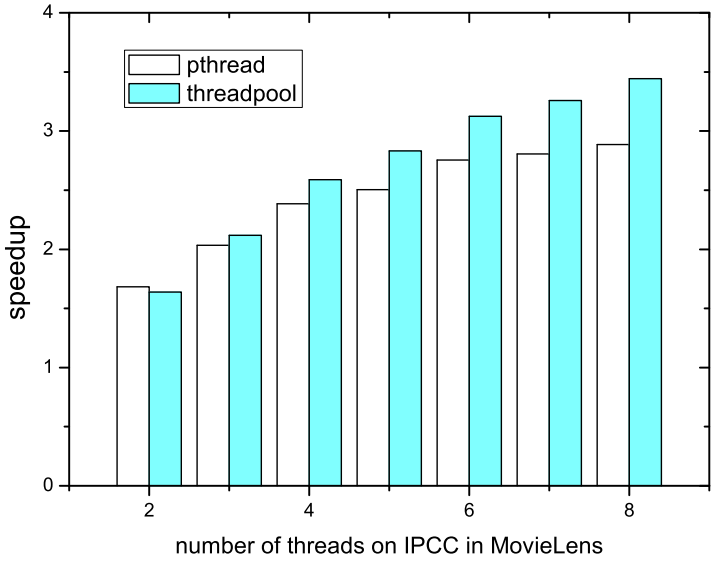

Fig. 7 Comparison on speedup between thread-pool and Pthread on MovieLens benchmark

Compared with Fig. 8, the line of speedup in Fig. 9 is steeper than in the MovieLens benchmark, and this means semi-sparse has better scalability on a larger benchmark. Compared to MovieLens, the initialization in Netflix occupies a relatively

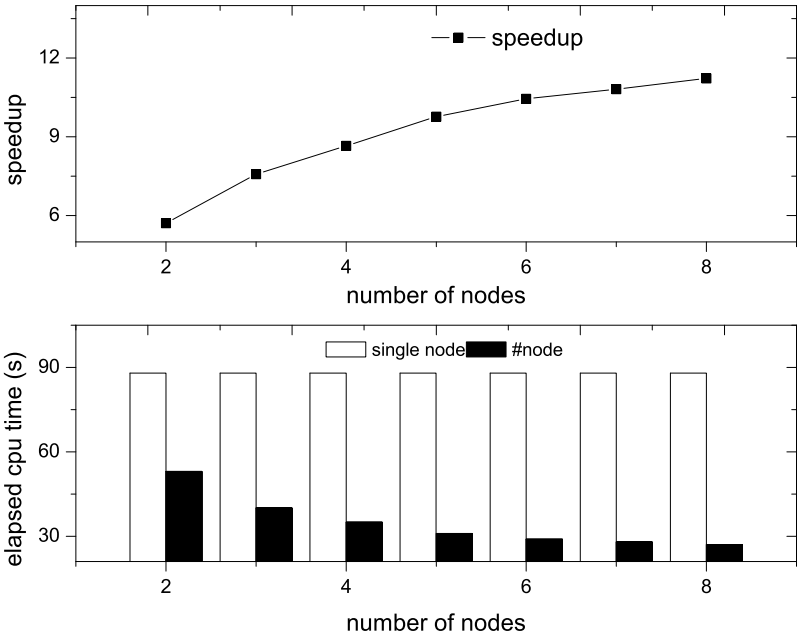

Fig. 8 Scalability on MovieLens benchmark

smaller share in the whole process. Thus, it is the intrinsic parallelism in IPCC and the design that brings the recommender system good scalability.

5.5 Multi-layer optimization on parallelism

In this section, we will give an overview of why the recommender system we offer can deliver brilliant performance.

In the high-level design for parallelism on IPCC, to reduce the overhead on passing messages among nodes, only the necessary partial result-array is permitted to be passed in the design with MPI. In the lightweight parallelism on the thread-pool, to avoid lock synchronization among threads as much as possible, the reduce-vector is suggested in the node-level parallelism. As for a basic-level thread, a semi-sparse algorithm is used to accelerate the similarity comparison by multiplying a selected vector with a batch of sparse vectors.

These optimizations are represented with the results in the following Table 5. Although the traditional sparse algorithm for IPCC is used in prediction, results in Table 5 demonstrate that, in the recommending process for 21,280 testing users, a semi-sparse algorithm can save 16,896 seconds at least, ThreadPoolWrapperFacade with semi-sparse can obtain a 5× speedup, and MPI with a thread-pool can achieve 32× speedup on 8 nodes.

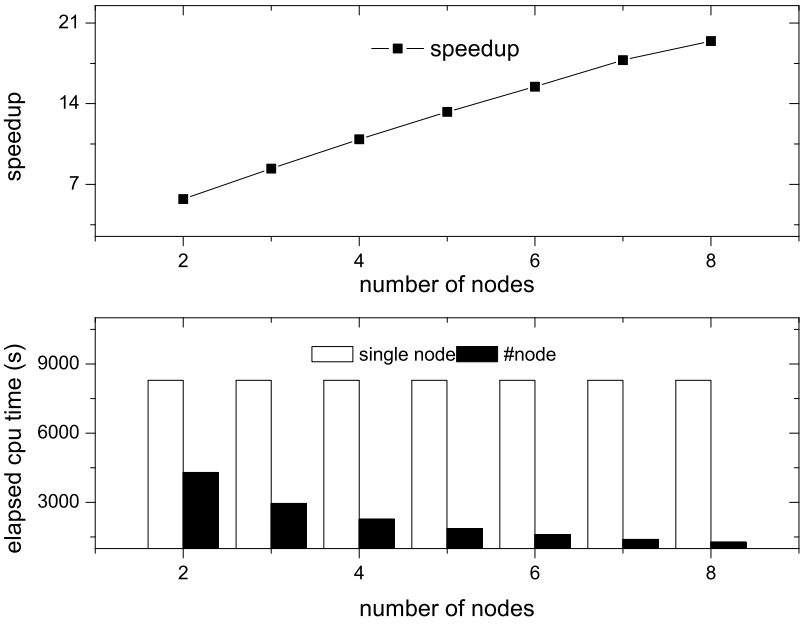

Fig. 9 Scalability on Netflix benchmark

Table 5 Overview of optimization on parallelism in Netflix benchmark

Items	Method	Time-cost (s)	speedup
Sequential IPCC	Traditional sparse	41,609	1.0
Sequential IPCC	Semi-sparse	24,713	1.7
Parallel IPCC	ThreadPool	8293	5.0
Parallel IPCC	MPI with pool	1272	32.7

6 Conclusion

We propose a multi-layer framework with a semi-sparse algorithm to speed the common collaborative filtering algorithms for recommender systems. To reduce the overhead on passing messages among nodes, we keep an original data copy on each node because the data set for PCC is very sparse, and only the result-array is passed among nodes. In a basic-level thread, we use a semi-sparse algorithm to boost the similarity-comparison. The effectiveness of semi-sparse is also validated in the SVMTorch classifier. To reduce the chance of synchronizing threads with mutex-locks, the reduce-vector is used for the node-level parallelism. This strategy is convenient for a parallel implementation.

Results from experiments on Netflix, MovieLens1600 and MovieLens demonstrate that our system achieves a significant speedup compared with the traditional sparse algorithm. This framework can be used to speed up a cluster algorithm based on sparse vectors and so on.

Acknowledgements We would like to thank the anonymous reviewers for their insightful comments on this paper. This work was supported in part by NSFC 61003012, 863 Program of China 2011AA01A202, and Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), China. The views expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. Berka T, Kollias G, Hagenauer H, Vajteršic M, Grama A (2011) Concurrent programming constructs for parallel mpi applications. J Supercomput 1–22
2. Carns P, Ligon W III, Ross R, Thakur R (2000) PVFS: a parallel file system for Linux clusters. In: Proceedings of the 4th annual Linux showcase & conference-volume 4, p 28. USENIX Association
3. Chang E, Zhu K, Wang H, Bai H, Li J, Qiu Z, Cui H (2007) Psvm: parallelizing support vector machines on distributed computers. Adv Neural Inf Process Syst 20:16
4. Collobert R, Bengio S (2001) SVMTorch: support vector machines for large-scale regression problems. J Mach Learn Res 1:160
5. Deshpande M, Karypis G (2004) Item-based top-n recommendation algorithms. ACM Trans Inf Syst (TOIS) 22(1):143–177
6. Fatone L, Giacinti M, Mariani F, Recchioni M, Zirilli F (2012) Parallel option pricing on GPU: barrier options and realized variance options. J Supercomput 1–22
7. Gantner Z, Drumond L, Freudenthaler C, Rendle S, Schmidt-Thieme L (2010) Learning attribute-to-feature mappings for cold-start recommendations. In: 2010 IEEE international conference on data mining. IEEE Press, New York, pp 176–185
8. González-Domínguez J, García-López Ó, Taboada G, Martín M, Touriño J (2012) Performance evaluation of sparse matrix products in UPC. J Supercomput 1–10
9. Herlocker J, Konstan J, Borchers A, Riedl J (1999) An algorithmic framework for performing collaborative filtering. In: Proceedings of the 22nd annual international ACM SIGIR conference on research and development in information retrieval. ACM, New York, pp 230–237
10. Hutchins R, Hemmady S (1996) How to write Awk and Perl scripts to enable your EDA tools to work together. In: Proceedings of the 33rd annual design automation conference. ACM, New York, pp 409–414
11. Jahnke J, Zundorf A. Rewriting poor design patterns by good design patterns. In: Proc ESEC/FSE, vol 97. Citeseer, pp 1841–1897
12. Koren Y (2010) Collaborative filtering with temporal dynamics. Commun ACM 53(4):89–97
13. Koren Y, Bell R, Volinsky C (2009) Matrix factorization techniques for recommender systems. Computer 42(8):30–37
14. Linden G, Smith B, York J (2003) Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Comput 7(1):76–80
15. Lombard P, Denneulin Y (2002) NFSP: a distributed NFS server for clusters of workstations. In: Parallel and distributed processing symposium. Proceedings international, IPDPS 2002, abstracts and CD-ROM. IEEE Press, New York, pp 35–40
16. Miller B, Albert I, Lam S, Konstan J, Riedl J (2003) MovieLens unplugged: experiences with an occasionally connected recommender system. In: Proceedings of the 8th international conference on intelligent user interfaces. ACM, New York, pp 263–266
17. Pacheco P (1998) A user's guide to MPI. University of San Francisco
18. Pyarali I, Spivak M, Cytron R, Schmidt D (2001) Evaluating and optimizing thread pool strategies for real-time CORBA. In: Proceedings of the ACM SIGPLAN workshop on languages, compilers and tools for embedded systems. ACM, New York, p 222
19. Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C (2007) Evaluating MapReduce for multi-core and multiprocessor systems. In: Proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture. Citeseer, pp 13–24
20. Sarwar B, Karypis G, Konstan J, Reidl J (2001) Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th international conference on world wide web. ACM, New York, pp 285–295
21. Schmidt D (1999) Wrapper Facade-A structural pattern for encapsulating functions within classes. In: C++ report. Citeseer

22. Su X, Khoshgoftaar T (2009) A survey of collaborative filtering techniques. Adv Artif Intell 2009:2
23. Vazquez F, Ortega G, Fernandez J, Garzon E (2010) Improving the performance of the sparse matrix vector product with GPUS. In: 2010 IEEE 10th international conference on computer and information technology (CIT). IEEE Press, New York, pp 1146–1151
24. Wang Z, Xu X, Zhao W, Zhang Y, He S (2010) Optimizing sparse matrix-vector multiplication on CUDA. In: 2010 2nd international conference on education technology and computer (ICETC), vol 4. IEEE Press, New York, pp V4–109
25. Yang X, Guan H, Tang F, You I, Guo M, Shen Y (2011) Improvements on sequential minimal optimization algorithm for support vector machine based on semi-sparse algorithm. In: Innovative mobile and Internet services in ubiquitous computing, international conference, pp 192–199. http://doi.ieeecomputersociety.org/10.1109/IMIS.2011.128
26. Zhen Y, Li W, Yeung D (2009) TagiCoFi: tag informed collaborative filtering. In: Proceedings of the third ACM conference on recommender systems. ACM, New York, pp 69–76