Noname manuscript No.
(will be inserted by the editor)

In-Memory, Distributed Content-Based Recommender System

Simon Dooms · Pieter Audenaert · Jan Fostier · Toon De Pessemier · Luc Martens

Received: date / Accepted: date

Abstract Burdened by their popularity, recommender systems increasingly take on larger datasets while they are expected to deliver high quality results within reasonable time. To meet these ever growing requirements, industrial recommender systems often turn to parallel hardware and distributed computing. While the MapReduce paradigm is generally accepted for massive parallel data processing, it often entails complex algorithm reorganization and suboptimal efficiency because mid-computation values are typically read from and written to hard disk. This work implements an in-memory, content-based recommendation algorithm and shows how it can be parallelized and efficiently distributed across many homogeneous machines in a distributed-memory environment. By focusing on data parallelism and carefully constructing the definition of work in the context of recommender systems, we are able to partition the complete calculation process into any number of independent and equally sized jobs. An empirically validated performance model is developed to predict parallel speedup and promises high efficiencies for realistic hardware configurations. For the MovieLens 10M dataset we note efficiency values up to 71% for a configuration of 200 computing nodes (8 cores per node).

Keywords recommender system · distributed · parallel · speedup

S. Dooms - T. De Pessemier - L. Martens
Wica, iMinds-Ghent University
G. Crommenlaan 8 box 201, 9050 Ghent, Belgium
E-mail: simon.dooms@intec.ugent.be
E-mail: toon.depessemier@intec.ugent.be
E-mail: luc.martens@intec.ugent.be

P. Audenaert - J. Fostier
IBCN, iMinds-Ghent University
G. Crommenlaan 8 box 201, 9050 Ghent, Belgium
E-mail: pieter.audenaert@intec.ugent.be
E-mail: jan.fostier@intec.ugent.be

1 Introduction

Recommender systems (RS) are systems designed to tackle the information overload problem by suggesting users relevant content. These systems typically learn from previous user behavior or use content information to drive their personalization engines. The popularity of RS is still growing as they are being deployed in very divergent domains like food [4], movies [32], events [13], music [35], books [27], travel [24], news [7], etc. At first, RS were often used with small datasets (e.g., the MusicFX system with 25 users and 91 items [26]) but since then datasets have massively expanded in size and nowadays a RS needs to be able to process datasets like the MovieLens 10M dataset[1] (10M ratings, 10k items and 72k users), the Netflix dataset[2] (100M ratings, 17k items and 480k users) or the Yahoo! Music dataset[3] (300M ratings, 600k items and 1M users) and they need to do it fast because users expect responsiveness and real-time behavior. However, because of these escalating dataset dimensions, very often online recommender systems are forced to implement a calculation process where recommendations are calculated in batch offline and are refreshed only on a daily basis.

Since sequential computers (or uniprocessors) are reaching the limits of maximum clock frequency, parallelism is often considered the solution [22] to cope with increasing dataset sizes and limited time constraints. A popular paradigm in the parallel data processing domain is MapReduce [23]. The MapReduce approach requires an algorithm to reformulate its logic in essentially two functions: *Map()* and *Reduce()*. An underlying framework (i.e., such as Hadoop) then takes care of the distribution of work across multiple computing nodes. Although work distribution is effortless, reformulating a given algorithm in the MapReduce mindset can be a tedious task and typically involves multiple MapReduce phases to be chained together. Since every single phase starts and ends with data access to and from disk, chained phases introduce disk access overhead which limits overall efficiency. Additional overhead is introduced by the setup of the distributed file system e.g. the Hadoop Distributed File System (HDFS) [10], where MapReduce programs depend upon. A well-known library that implements many scalable algorithms including some recommendation algorithms on the Hadoop framework is Mahout[4].

In this work, we show how a content-based recommendation algorithm can be parallelized and distributed over multiple computing machines without underlying MapReduce operations or file system restrictions. Because our approach executes completely in-memory (i.e., only RAM is used to store mid-computation values), disk access is reduced to a minimum and high efficiency values can be obtained. Although many more complex recommendation algorithms exist, neighborhood-based recommendation algorithms (such as the one in this work) are still relevant as they are often preferred in industrial use cases [31]. Such neighborhood-based algorithms introduce a neighborhood of similar items (or users) to act as guides towards interesting items (or users) in the recommendation process.

[1] http://www.grouplens.org/node/73
[2] http://www.netflixprize.com
[3] http://kddcup.yahoo.com/datasets.php
[4] http://mahout.apache.org

After the introduction of the MovieLens 10M dataset in Section 3, we detail in Section 4 an out-of-the-box content-based algorithm and show how it can be parallelized in an efficient way (Section 5). Section 6 tackles the issue of load imbalance and goes into more details regarding work definition and distribution in the context of recommender systems. In Section 7, we investigate performance by constructing an analytical speedup model which we empirically validate. We conclude our work with Section 8 in which we revise the strengths and limitations of our proposed parallel and distributed approach for content-based recommendation computations.

2 Related Work

2.1 Internal scalability

Related work that deals with the problem of scalable recommender systems, usually focuses on internal scalability of algorithms. Algorithms are tuned to focus on only the most relevant data and try to speed up the overall process by taking computational shortcuts (e.g., [29,30,33]). Especially for neighborhood-based methods a lot of scalability improvements exist. One of the most obvious is restricting the size of the neighborhood [3,18]. By processing only k (instead of *all*) neighbors, the recommendation process can finish faster. The value of k can however greatly influence the accuracy of the recommendations: k set too high may bring additional noise, while k set too low may reduce recommendation quality [20]. Herlocker et al. [17] suggest that for most real-world situations k set to something between 20 and 50 seems reasonable for the MovieLens dataset. In this work however, we do not focus on the qualitative aspect (i.e., accuracy) of our system but rather on optimizing its parallel performance. Therefore, in our algorithm we work with non-restricted neighborhoods to show the distributed system is able to handle this upper-bound situation.

2.2 External scalability

Rather than internal scalability, this work focuses on external scalability where we consider parallelism and extra hardware as the solution to our big data problem. Instead of changing the recommendation algorithm, the calculation of the algorithm is merely restructured and distributed over multiple computing instances. The final results calculated on a single node will be identical to the results calculated on a multi-node computing architecture. In general two types of parallelism can be defined: *data parallelism* and *functional parallelism.* In the case of data parallelism, multiple processors work on different parts of the data. Usually the same code is executed on all processors and therefore this scheme is sometimes referred to as SPMD (Single Program Multiple Data) [15]. Functional parallelism involves splitting computations into subtasks that can then be executed in parallel by different processors. In this case, the processors run different code on different data which is called MPMD (Multiple Program Multiple Data) [15].

In previous work [11], we employed *functional parallelism* by dividing the recommendation process into separate phases that could then again be decomposed into smaller chunks of work which could be mapped onto multiple worker nodes. While resulting processing times scaled linearly with applied hardware, efficiency was lost by the overhead introduced by chaining multiple phases together (each of which require reading and writing intermediate data results from and to permanent storage). In this work, we focus on *data parallelism* in order to simplify the parallel recommendation process, reduce the overhead of separate phases, and avoid load imbalance issues.

2.3 Distributed systems

In [16] and [34], a distributed collaborative filtering (DCF) algorithm DCFLA is introduced. The scalability of the system is guaranteed by distributing a user-profile management scheme using distributed hash table-based routing algorithms. The authors compared the performance of 4 CF approaches and showed how the scalability of their DCFLA approach surpasses that of the traditional CF algorithm. Performance results in terms of speedup or efficiency values however remained undiscussed.

When recommender systems are deployed in a distributed environment, research and industry often turn to MapReduce as underlying paradigm. In [36], a user-based collaborative filtering (UBCF) algorithm is implemented on Hadoop (which is an open-source MapReduce framework implementation). The authors illustrate that it is not easy to directly apply the MapReduce model to UBCF and show how it can be done by splitting up the logic in 3 phases. The results showed a linearly increasing speedup, for hardware configurations of up to 8 computing nodes. Schelter et al. [31] developed a MapReduce algorithm for the pairwise item comparison and top-N (i.e., recommend the best N items) recommendation problem. For the R2 - Yahoo! Music dataset they achieved a speedup value of about 4 using 20 computing machines (i.e., parallel efficiency of 20%). Jiang et al. [21] implemented an item-based collaborative filtering recommendation algorithm on the Hadoop platform. They chained 4 MapReduce phases and their parallel efficiency was about 90% using 8 computing nodes (using the MovieLens 10M dataset). While MapReduce has certainly proven its use for recommender systems [8, 9] in general, we believe the paradigm may sometimes lack the flexibility to take specific algorithm properties into account to reach high parallel efficiency values. In our work, we show that specifically for the content-based recommendation algorithm we can divide ‘work’ evenly and independently such that by tweaking certain parameters of the distribution process, a given hardware configuration is optimally engaged without inter-node communication and mid-computation disk access, thus paving the way towards higher parallel efficiency.

3 Dataset specification

Throughout this work, we employ the MovieLens 10M dataset to provide our experiments with input data. This dataset contains 10 million ratings and 95,580 tags applied to 10,681 movies by 71,567 users of the online movie recommender service MovieLens. Data is provided as three files: *ratings.dat* (252MB), *tags.dat* (3MB), and *movies.dat* (500KB). Interestingly, while every user in the MovieLens dataset has rated at least 20 movies, some users have rated considerably more. As shown in Fig. 1, 50% of all ratings originate from only 15% of all users. This unequal distribution of ratings can easily introduce load imbalance issues when carelessly distributing users across worker nodes (a challenge which we face in Section 6). For more detailed information about this dataset we refer to recommender systems literature (such as [28,6]).

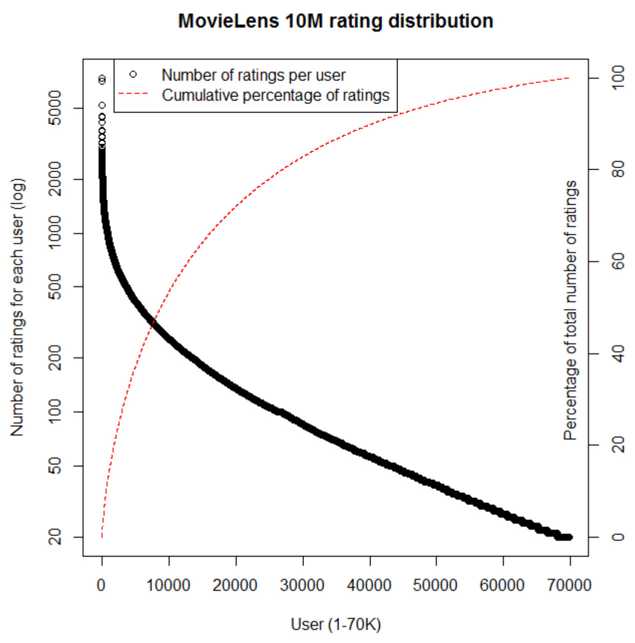


Fig. 1 The number of ratings per user for the MovieLens 10M dataset.

All described experiments were run on the Ghent University HPC cluster that we had at our disposal (see [11] for more details on the cluster infrastructure). This work was implemented in Python and our code can be found on the GitHub platform[5].

4 Parallel CB recommender

Here, we define the recommendation algorithm that will be used throughout this work. Since we focus mainly on optimally distributing and running the algorithm in parallel on a grid computing infrastructure, a default out-of-the-box content-based (CB) recommendation algorithm (as described in [20]) was used as starting point. The CB algorithm calculates the (Jaccard) similarity between items based on the item metadata (i.e., MovieLens files: *movies.dat* and *tags.dat*) and recommends new items to users based on these similarities. We use the MovieLens (10M)

[5] http://xxx

dataset as input and deploy the algorithm to predict the user ratings for unknown (*user*, *item*) pairs.

Calculating the recommendation value for a certain user and item requires the comparison of similar items previously rated by the user. Although the number of such processed similar items is usually limited in size to reduce the computational burden and minimize possible noise [20], we do not restrict the neighborhood size to illustrate the true scalability of our method. In fact, the only optimization that is incorporated in the algorithm is the temporary storage (caching) of calculated similarity values to prevent unnecessary recalculations. However, since these values can easily become too abundant to store (with limited RAM), they are cleared with every item iteration as is shown in the algorithm below.

```
Algorithm. Complete recommendation calculation
    parallel for item in items
        cached_item_similarities ← {empty}
        for user in users
            if user has not rated item
                calculate Rec(user, item)
            End if
        End for
    End parallel for

PROCEDURE - Rec(user, item) {Calculates the recommendation value for the (user, item) pair}
    vote ← 0
    weights ← 0
    for rated_item in items rated by user
        weight ← Simil(item, rated_item)
        vote ← vote + (weight × rated_item_rating)
        weights ← weights + weight
    End for
    Return vote / weights
End PROCEDURE

PROCEDURE - Simil(item_1, item_2) {Calculates the similarity value of the (item_1, item_2) pair}
    if similarity in cached_item_similarities
        Return similarity
    else
        top ← The number of metadata item attributes that item_1 and item_2 have in common (intersection)
        bottom ← The total number of metadata item attributes of item_1 and item_2 (union)
        similarity ← top / bottom
        store similarity in cached_item_similarities
        Return similarity
    End if
End PROCEDURE
```

The calculation of $Rec(user,\ item)$ thus requires the rating data of the user together with the item data of the item and any other rated items by the user. We note that although the algorithm used in this work is content-based, user ratings are still important as they are used in the weighted average formula of the target user in the recommendation calculation procedure. Because we want to distribute the calculation work over multiple computing nodes, both user data (i.e., ratings) and item data will have to be considered in the distribution process. Moreover, because both data types are taken into account, the distribution paradigm laid out in this work can be extended to fit collaborative filtering algorithms that focus on rating data only.

5 Parallel strategies

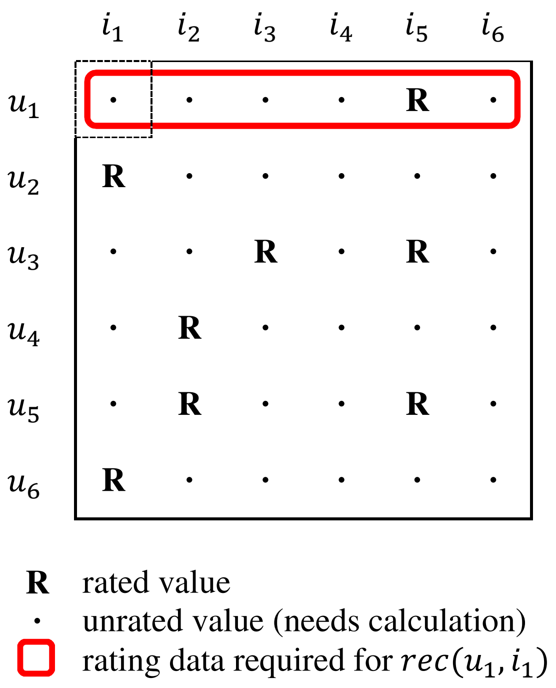
We want to split the work of a complete recommendation calculation into smaller pieces of work that require less input data and can be distributed over available worker nodes. In the context of a recommender system, input data consists of user data (usually ratings) and item data. The actual work consists of the calculation of the recommendation value (i.e., numeric value indicating the interest of the user) for every $(user,\ item)$ pair in the system. This is usually visualized as a user-item matrix as depicted in Fig. 2. Every dot in the matrix represents a recommendation value to be calculated. The value of some dots may already be known, as users may have rated some items (indicated by **R**s). These ratings are considered the perfect prediction of the interest of the user for that item.

The way in which the work (i.e., dots) is divided over available worker nodes will have an impact on the amount of input data needed for that node. In the following subsections we present three parallel strategies for dividing the work of a recommender system and their data related consequences.

5.1 Splitting in Userjobs

We could partition the user-item matrix in horizontal subsets to distribute the users that must be processed across the available worker nodes (Fig. 3). When these subsets are mapped onto worker nodes, every node must then calculate the recommendation value for each of these users in the subset and every item in the system.

Because the users are divided over different jobs (we refer to these as 'userjobs'), the user input data can be split accordingly into smaller subsets. Consequently, worker nodes will be able to work with smaller datasets which can help reduce RAM requirements. A rather technical downside of this division scheme is that with a bigger number of userjobs, fewer computed intermediate item similarity values can be re-used. For a $(user,\ item)$ pair, an item similarity value will be calculated between the item and all the items rated by the user. The more users are handled by a single worker node, the more of these intermediate values can be re-used. A large number of userjobs indicates a small number of users per job and so fewer recycling of intermediate similarity values.



i_1 i_2 i_3 i_4 i_5 i_6
u_1 u_2 u_3 u_4 u_5 u_6
R
R rated value
• unrated value (needs calculation)
rating data required for $rec(u_1, i_1)$

Fig. 2 The user-item matrix indicating the work related to a complete recommendation calculation of every (*user*, *item*) pair.

5.2 Splitting in Itemjobs

Alternatively, we could partition the user-item matrix into vertical subsets and distribute the items across the available nodes (Fig. 4). Every job (i.e., ‘itemjob’) must now process the recommendation value for each item of the subset and every user in the system.

The obvious advantage of this method is that since every user in the system is now matched with a subset of items by every worker node, the amount of redundant item similarity computations is reduced to zero. On the other hand, because all users are processed, all user input data (i.e., rating data) needs to be loaded by every itemjob which may turn out to be too big for the RAM of a single worker node.

5.3 Hybrid userjob, itemjob splitting

Since both splitting in userjobs and itemjobs have benefits and downsides, a meet-in-the-middle approach seems a good option. In this case, we split the grid of user and item pairs into disjoint subsets of users and items to be distributed across the available worker nodes. This approach allows partial recycling of similarity values while reducing the required user input data. The number of userjobs and itemjobs can be freely chosen and specifically tailored towards the available computing

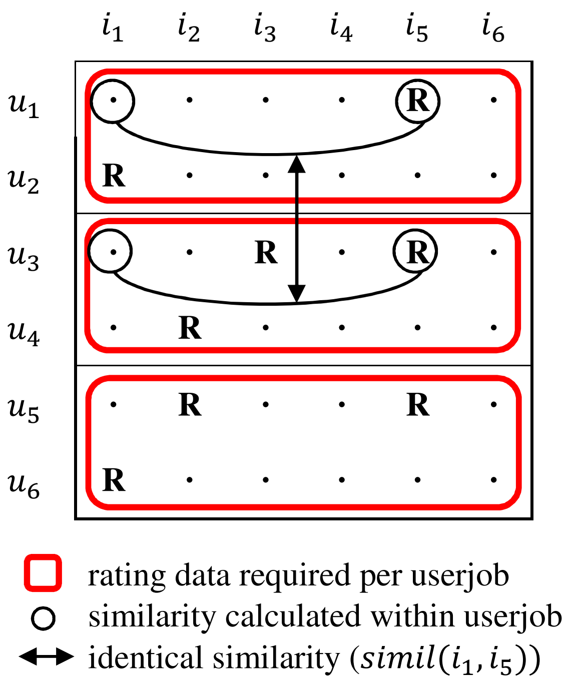

i_1 i_2 i_3 i_4 i_5 i_6
u_1 u_2 u_3 u_4 u_5 u_6
R
rating data required per userjob
similarity calculated within userjob
identical similarity ($simil(i_1, i_5)$)

Fig. 3 The user-item matrix split into a number of userjobs (every userjob processes all items).

hardware. In Section 7, we address the issue of choosing the right number of userjobs and itemjobs.

The hybrid userjob, itemjob approach introduces the most flexibility and can even be turned into one of the previous approaches by setting the value of the number of userjobs (or itemjobs) to '1'. Therefore, this hybrid data parallelism strategy was adopted by our recommender system.

6 Load Balancing

In this section we focus on load balancing and work distribution of our content-based recommendation algorithm. If work is not evenly distributed among available workers then load imbalance issues may arise. These occur when synchronization points are reached by some workers earlier than others [15] and so workers display significantly divergent walltimes (i.e., time to solution). Load imbalance greatly impacts the efficiency of algorithms because resources are underutilized while fast workers wait for slow workers to finish. If work is load balanced, it can be evenly divided over computing nodes without the need for extra inter-node communication (as would be the case in a master-slave scenario). This relaxes network constraints and allows for active engagement of all computing nodes.

For a system to be load balanced, work must be evenly distributed among its workers. Specifically in the context of recommender systems, this introduces two

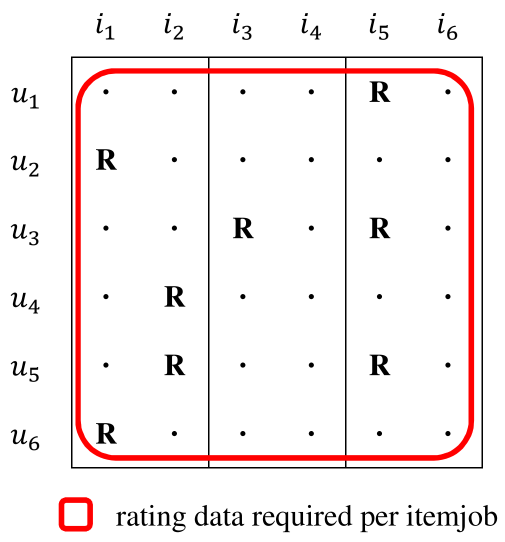

i_1 i_2 i_3 i_4 i_5 i_6
u_1 u_2 u_3 u_4 u_5 u_6
R
rating data required per itemjob

Fig. 4 The user-item matrix split into a number of itemjobs (every itemjob processes all users).

subproblems: How do we define work, and how can it be evenly distributed? In the next two subsections we elaborate on each of these problems.

6.1 The Definition of Work

A straightforward way of expressing the amount of work processed by a worker is by computation time. Longer computation times indicate more work has been done. However, before the recommendation calculations are performed, the actual computation time is unknown. What is known, are the total number of users, items, available ratings, etc. If one of these metrics shows a positive correlation with the calculation time, it can be used as a shorthand definition of work.

In Section 5, the notion of userjobs and itemjobs was introduced. Because of this distinction, we want to be able to define work in terms of both user-related metrics and item-related metrics. We devised two experiments focusing on these subsets of metrics.

6.1.1 Work in terms of Users

Intuitively, more processed users will result in longer computation times. The number of ratings that are provided by these users however may also be important. To avoid confusion, we note that when we refer to *ratings* we are referring to the user-provided ratings that are already available in the dataset (**R**s in Fig. 2).

The algorithm in Section 4 shows that to calculate *Rec(user, item)* all the ratings of that user will be taken into account. So a job processing users with a low number of ratings may finish faster than a job with many ratings per user. We

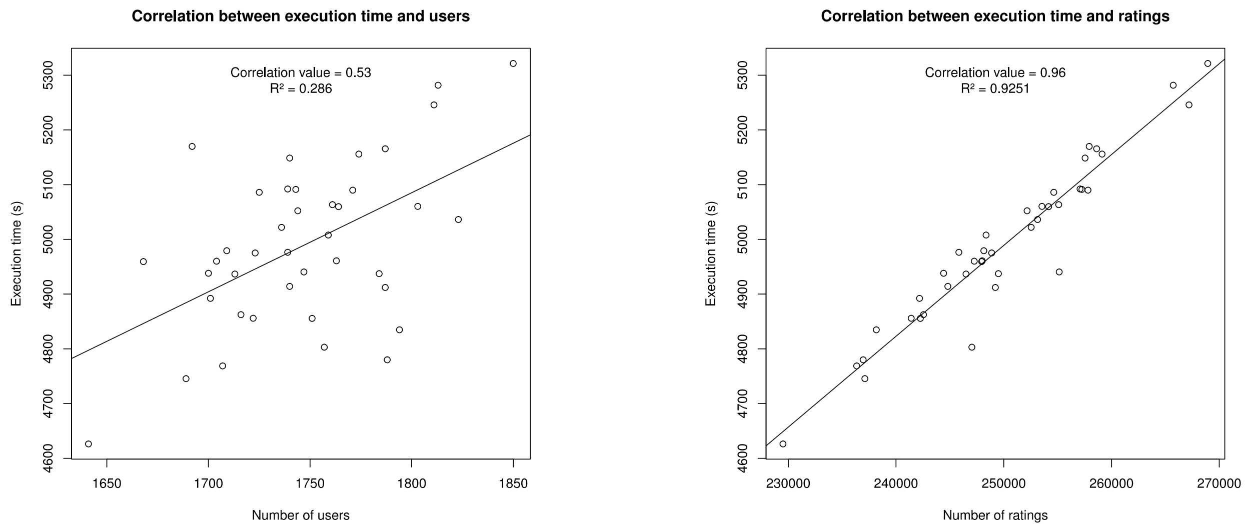


Fig. 5 Scatterplots indicating for a configuration of 40 worker nodes (i.e., 40 userjobs, 1 itemjob) the correlation of the number of users (left) and the number of given ratings (right) with the calculation time of each job.

define two recommendation metrics as possible candidates for a user-related work definition: *the number of users* and *the total number of ratings* provided by these users.

We measured the correlation and performed a simple regression analysis of these metrics with the actual resulting computation time for a configuration of 40 worker nodes (1 core per node). Each user was randomly assigned to one of these nodes and each node processed all of the items available in the MovieLens dataset (i.e., 40 userjobs, 1 itemjob). Since the set of items processed by each worker was the same, the influence of processed users (and therefore also ratings) on the computation time could be isolated. Fig. 5 shows two scatterplots indicating the computation time (of the 40 userjobs) in function of the number of users per job (left) or number of ratings (given by these users) per job (right).

While the number of users is not bad at predicting the execution time ($R^2 = 0.286$), the number of ratings is an almost perfect predictor ($R^2 = 0.9251$). We learn from this that in order to achieve a load balanced system we should take the number of available ratings processed by each worker node into account rather than the number of users.

To further illustrate this concept, we compared the calculation times when evenly distributing the number of users versus distributing the users in such a way that the total amount of ratings given by these users is (as good as) equal for each node. Fig 6 shows the results, and as expected, the distribution of users shows a random like calculation time pattern while the distribution of ratings results in a load balanced system.

Because work (or calculation time) is largely connected with the number of available ratings, we should strive towards an equal distribution in the number of

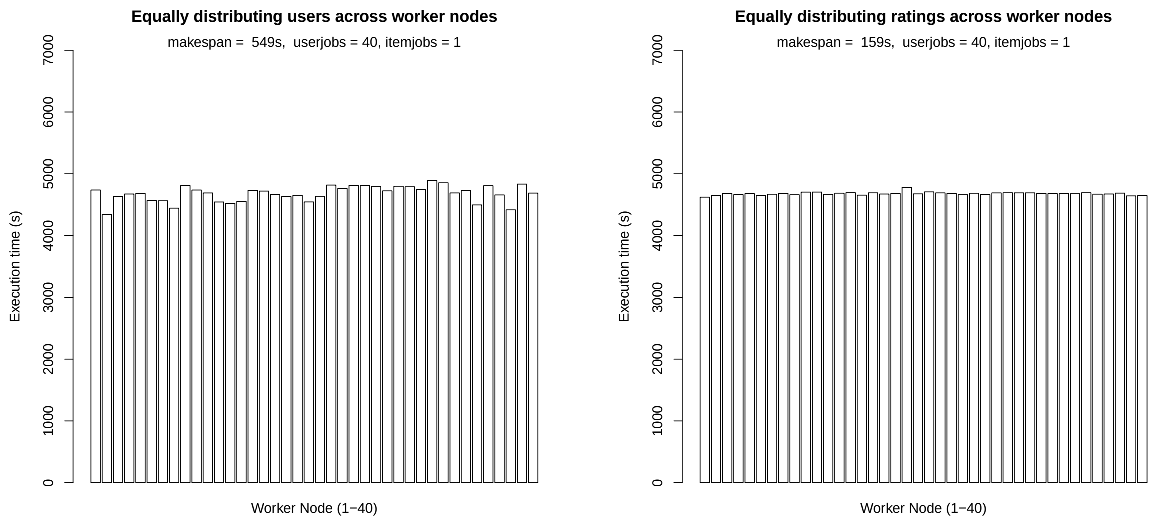


Fig. 6 The resulting calculation times when equally distributing the number of users across worker nodes (left) versus distributing the number of users such that their total number of given ratings are equal across worker nodes (right).

ratings among worker nodes. Equally distributing the users is not enough because of the large divergence in number of ratings per user for our dataset.

6.1.2 Work in terms of Items

To define work in terms of items the obviously available metric is *the number of items* that are processed by a worker node. However, with the algorithm presented in Section 4, the workload associated with an item may differ. To calculate the recommendation value $Rec(u,\ i)$ all the ratings of user u must be taken into account (**for** loop in *Rec(user, item)* procedure). We can express the true workload of item i by analyzing the number of times this *for* loop will be iterated on. This amount of iterations depends on the number of ratings of u and on the fact that u may or may not have already rated i. If u has rated i, $Rec(u,\ i)$ will not be calculated and the amount of iterations will be zero. Therefore, for a given set of users, the number of iterations for an item will be equal to the sum of the ratings of the users minus the sum of the ratings of the users that have rated i (because they will be skipped). We define a metric *item iterations* or $iter(i)$, as the total number of iterations that must be run for an item i to calculate the recommendation values $rec(u,\ i)$ for all users u.

$$\begin{aligned} iter(i) &:= |number\ of\ iterations\ for\ item\ i| \\ &:= |all\ ratings| - |skipped\ ratings| \\ &:= |all\ ratings| - \sum_{(\forall u | u\ rated\ i)} (|ratings\ of\ u|) \end{aligned}$$

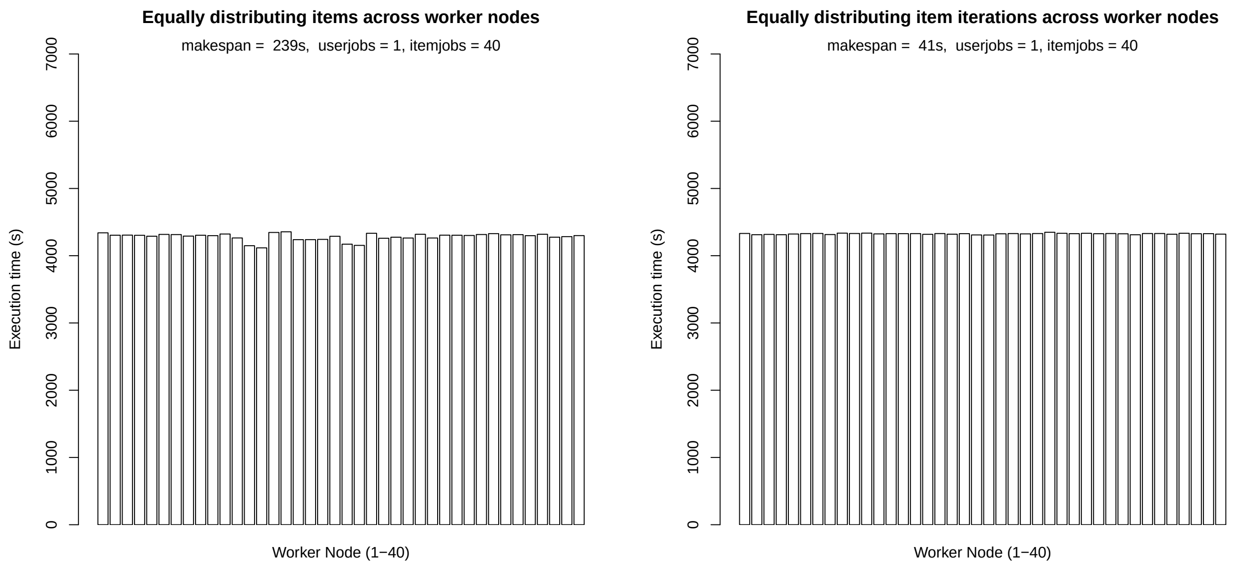


Fig. 7 Calculation times for equally distributing the number of items (left) versus equally distributing the number of item iterations (right).

Again we set up a simple regression experiment for the two item-related metrics (*number of items* versus *number of item iterations*) with the actual computation time. The items were randomly distributed over 40 worker nodes (1 core per node) and every node processed every user (i.e., 1 userjob, 40 itemjobs). When inspecting the predictive capabilities of our metrics, both *number of items* ($R^2 = 0.9342$) and *the number of item iterations* ($R^2 = 0.943$) were found to be very good predictors for the calculation time.

When work was actually divided equally according to these two metrics and results were compared, *the number of item iterations* proved to be slightly better in terms of load balance (Fig. 7).

To conclude, we note that to obtain a load balanced system we should strive to an equal distribution of both *number of ratings* and *number of item iterations* over the available worker nodes.

6.2 Work Distribution

In this subsection we detail how we distribute work (which we defined in the previous subsection) equally among worker nodes. Only if work is equally distributed, a load balanced situation can be achieved and therefore a higher parallel efficiency.

6.2.1 The partition problem

The problem of equally distributing work across homogeneous workers is not new, it is in fact a very well-studied problem in the theory of approximation algorithms [19], often referred to as 'makespan minimization'. In this problem setting, a number of jobs (with different estimated processing times) need to be scheduled across a number of (identical) worker nodes, such that the maximum time for any node to finish its work (i.e., the makespan) is minimized. In the context of our recommender system, we will need to partition users and items in subsets to be processed by worker nodes such that the resulting subsets show an equal amount of ratings and item iterations. To do this, we need to solve the makespan minimization problem, but since it is considered to be *NP-complete* [19], a fully polynomial algorithmic approach might not exist. However, countless approximation schemes have been proposed to tackle this problem (e.g., [5,14,25,1]).

The problem with optimization schemes such as Monte Carlo, genetic algorithms, etc. is that they often require a large number of iterations to converge to an acceptable solution. The runtime of the partitioning of work among worker nodes will however be of crucial importance to the final performance of the system. This partitioning will have to be executed sequentially and thus strongly limits the parallel efficiency. Moreover, we must make sure not to put more effort (i.e., time) into optimizing the partitioning than would be won by the improved load balancing. Since speed is so important, instead of applying more advanced optimization solutions we first employ a very simple O(n), greedy partitioning algorithm.

Algorithm. Work distribution

```
Let jobs be a list of jobs
Sort jobs according to estimated workload (high to low)
for each job
    assign job to worker node with currently lowest workload
End for
```

The accuracy of this solution depends on the specifics of the input data and the number of desired partitions. If all users and items are equal in terms of our definition of work then the algorithm will provide an optimal solution. On the other hand, if they are extremely divergent and lots of partitions are needed, the results (in terms of 'makespan minimization') may be poor.

While we found this solution to be sufficiently accurate (i.e., leading to sufficiently load balanced systems) for most of the hardware configurations we ran our algorithm with, some cases do require some extra attention. In the next subsection we show how the accuracy of this simple greedy algorithm can be heuristically improved.

6.2.2 Robin Hood extension

To improve the accuracy of our proposed greedy partitioning algorithm, we employ a method we refer to as the 'Robin Hood' extension. The idea is to iteratively take work from 'the rich' and give it to 'the poor' in order to balance out the wealth inequality. In this context, we define wealth as the workload of a worker node after initial partitioning. Since we defined 'work' in Section 6.1, we can compute this workload of a worker node by summing up either *the number of ratings* (when

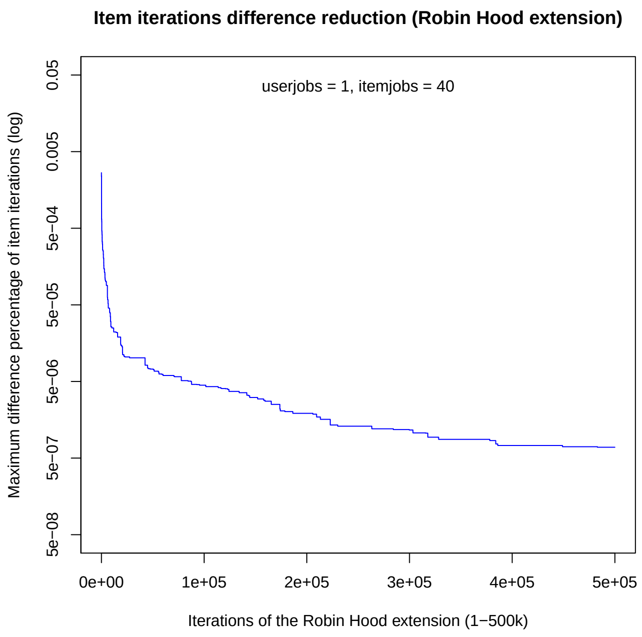


Fig. 8 The declination of the item iterations difference between worker nodes after each iteration of the Robin Hood extension (up to 500k iterations). Note that the y-axis is on a logarithmic scale and expressed as a percentage of the total number of item iterations.

dividing in userjobs) or *the number of item iterations* (when dividing in itemjobs) that need to be processed by the node. When the workload for every worker node is known, we select the richest (i.e., highest workload) worker node and the poorest (i.e., lowest workload) worker node. We then randomly pick a user (or item) from the rich node and add it to the users (or items) of the poor node. Doing so, we level out the maximum workload difference (i.e., makespan) associated with the worker nodes. This process can be repeated until a desired threshold of minimum makespan has been reached or a maximum number of iterations has been run.

As stated, this extension of the partition algorithm may only be needed in a few cases where the partitioning algorithm performs worse then a certain threshold of inequality. To illustrate the behavior of the Robin Hood extension, we executed it after partitioning the items of the MovieLens dataset over 40 itemjobs (and 1 userjob) with the simple greedy partitioning algorithm. We set the extension to run 500,000 iterations and we plotted the minimum makespan after every iteration (Fig. 8). As shown in the figure, the Robin Hood extension allows to rapidly reduce the makespan difference within only a few thousand iterations. While the makespan is reduced, work is more evenly divided over the available worker nodes and so it becomes more difficult to improve.

6.2.3 Dividing in Userjobs and Itemjobs

In the context of recommender systems, data parallelism is a very promising concept because it allows the processing of extremely large datasets on commodity PCs. More interestingly even, it introduces a degree of flexibility in the sense that data can be partitioned into pieces that maximize the utilization of worker resources (e.g., RAM) and therefore computational efficiency. We divide our data grid (users-items matrix) by first splitting the users into U chunks of users to be processed by an equal amount of userjobs, and then for each userjob splitting the items into another I itemjobs. The final number of jobs will consequently be the product of the number of userjobs and itemjobs.

$$total\ number\ of\ jobs\ = userjobs \times itemjobs$$

As detailed in Section 5, a worker node processing recommendation values for ($user$, $item$) pairs must be able to hold all the ratings and item data of these users and items in RAM. Therefore the largest resource requirement in the system is the RAM of a computing node. Since user feedback data is usually more abundant than item data (here ratio 70:1) we start by dividing the user data. These data are split into equally-sized but smaller parts (i.e., userjobs). By equally-sized we mean in fact equal in terms of the metrics we defined in Section 6.1. Since we are dealing with user data, the relevant metric here is *number of ratings.* We use the partition method as described in Section 6.2.1 to divide the users in a number of parts such that each part contains an equal number of ratings. The number of userjobs can be freely chosen, but as mentioned, this will drastically impact the worker nodes resource requirements.

Then we continue to split the input data in terms of items (i.e., itemjobs) for every userjob. As we defined *number of item iterations* as a good predictor for workload in terms of items, we divide the items in a number of parts such that each part contains about an equal number of iterations. Since the number of iterations depends on the ratings that must be processed, itemjob division must be carried out after the userjob division.

Since both the userjobs and itemjobs are divided in a way that optimizes load balance, running both processes sequentially will also lead to a load balanced system. A graphical overview of the distribution of work in terms of users and items can be found in Fig. 9.

While the number of userjobs and itemjobs can be freely chosen, the choice does also impact the final load (im)balance state of the system. When dividing a set of numbers into x subsets of numbers with an (as much as possible) equal sum, the value of x is of great importance. It is easier to split up the numbers equally over 2 subsets, than over 4 subsets. Since the employed partition algorithm is heuristic in nature, the solution it generates depends on the difficulty of the problem. To investigate the effect of the number of userjobs and itemjobs on the final load imbalance of the system we ran some benchmarks.

We ran a number of job division processes (without Robin Hood extension) with varying userjob and itemjob parameters and compared their resulting maximum item iterations difference (Fig. 10). The difference in terms of *number of ratings* was negligible (because there are 10 000 times less ratings than item iterations to divide) and is therefore not displayed.

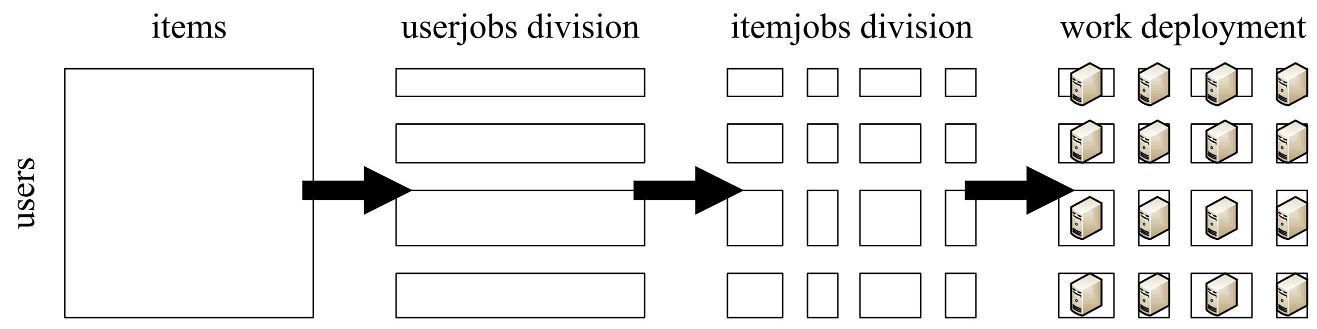


Fig. 9 Overview of the work distribution from input data in terms of users and items to the final mapping of the data on worker nodes.

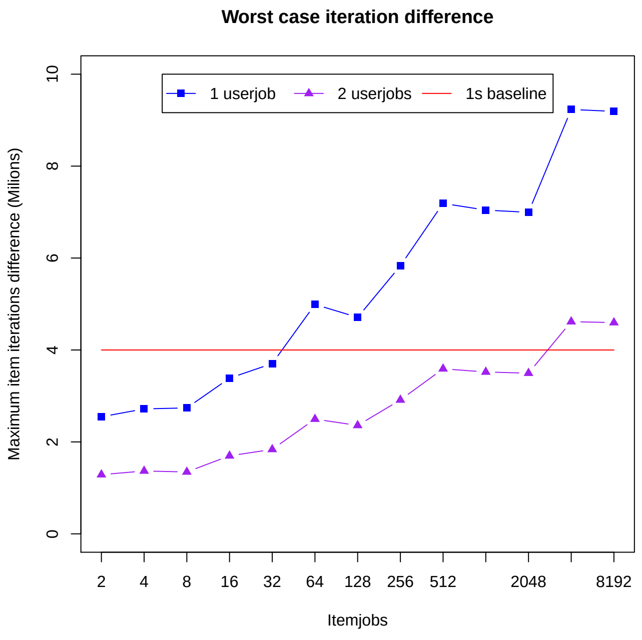


Fig. 10 The maximum iterations difference for a work division (without robin hood extension) with varying number of userjobs and itemjobs.

Fig. 10 shows the opposite effects of the number of userjobs and itemjobs on the resulting difference. A larger number of userjobs implies working with smaller subsets of ratings that consequently need to be divided further into itemjobs. Therefore, more userjobs implies a smaller item iterations difference while more itemjobs implies a bigger item iterations difference because of the increased complexity of the problem.

A difference of 4 million item iterations (indicated on the figure) in our system corresponded with 1s difference in computation time (which is only 0.002% of the total time). It is clear that for most configurations the item difference will be negligible with respect to the total calculation time. The results and benchmarks presented here, apply specifically to the MovieLens dataset. Other datasets might introduce other ratios of ratings, users and items and so these benchmarks would have to be repeated to determine the accuracy of partitioning. If the load imbalance caused by the partition algorithm turns out to be too big, the Robin Hood extension can be used.

We conclude this section by refocusing on the need for a load balanced system in order to achieve higher parallel efficiency. We defined 'work' in the context of a recommender system and correlated available metrics to resulting calculation times. Using these definitions of work we are able to evenly distribute the work across available worker nodes by applying a partitioning algorithm and optionally a heuristic improving extension. By defining userjobs and itemjobs, input data can be very flexibly divided into smaller chunks to optimize resource utilization while preventing load imbalance.

7 Performance Model

In this section we evaluate the performance of our proposed content-based recommendation algorithm. The final recommendations of the calculation process executed on a single machine will be identical to the final recommendation results of our distributed approach and so we do not consider accuracy (among other qualitative metrics) to be relevant in the evaluation of our system.

The performance of a parallel algorithm is not easily measured as it can be influenced by many things. As [15] suggests, performance may be limited by load imbalance issues, the amount of serialized parts of the concurrent execution, and communication overhead that may be introduced because of the increased amount of worker nodes. In the previous section, we succeeded in partitioning the complete recommendation problem into any predefined number of subtasks with equal complexity. These recommender subtasks can then be mapped onto worker nodes without introducing load imbalance issues. Our algorithm and parallel strategy are devised in such a way that jobs are executed independently and so no communication overhead is introduced by increasing the number of worker nodes. Serial code fragments, on the other hand, could not be avoided. That is, some code can only be executed by one processing node and therefore limits the performance of our system.

In previous sections, we mainly focused on the recommendation calculation part of our algorithm, but for a realistic performance model, we have to consider the complete recommendation process. This process also includes processing input data and running the work division as detailed in Section 6.2. To gain insight in the performance of the complete recommendation process we started by measuring the execution time of every part of the process. We define these parts in function of how well they can be parallelized: *serial*, $parallel_U$, $parallel_I$, or $parallel_C$. Parts that are *serial* can not be parallelized and have to be executed sequentially. Parts that are *parallel* on the other hand can be run in parallel on a number of worker nodes. We define three types of *parallel* to make a distinction between parts that are able to run in parallel on a number of userjobs ($parallel_U$), number of itemjobs

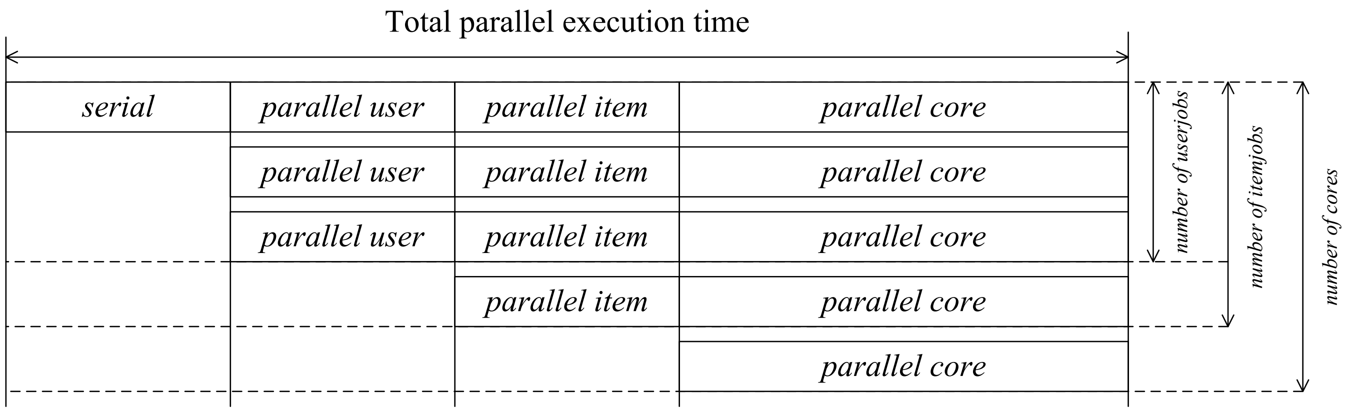


Fig. 11 A schematic view of the total execution time of the complete recommendation process in terms of how different parts of the algorithm can be parallelized.

Parallelizability	**Time (s)**	**Time (%)**
serial (s)	41	0.02339
$parallel_U$ (P_U)	29	0.01675
$parallel_I$ (P_I)	0.01	0.00001
$parallel_C$ (P_C)	174 936	99.95985
Total:	**175 006**	**100**

Table 1: The execution times for a complete recommendation calculation process with 1 worker node (1 userjob, 1 itemjob, 1 core).

($parallel_I$), or on all the jobs at the same time including multiple processing cores per job ($parallel_C$).

An example of what might be considered $parallel_U$ is the processing of user input data. After a certain node has read the input data from disk (which we do not take into account here), these data need to be parsed and stored into a memory structure. Every worker node needs to do this, but only for the subset of user input data that was divided by the userjob work division. Therefore we refer to this work as parallelizable in terms of userjobs.

We ran the complete recommendation process with a single worker node and one active processing core as a performance baseline. For this, we set the number of userjobs and itemjobs to '1' (so all users and items are processed by a single job). Table 1 shows the resulting execution times for each part in function of the way they can be parallelized. As expected, the recommendation calculation itself, which can be fully parallelized (P_C), accounts for the most processing time.

To gain some insights into the parallel performance of this algorithm, we calculate the speedup for a fixed-size problem (Amdahl's law [2]). In its simplest form, speedup S_p is defined by formula (3). If s is the amount of serial work and p the amount of parallel work, we define $T(1)$ as the execution time on 1 worker node and consequently $T(N)$ the execution time on N worker nodes.

$$T(1) = s + p \tag{1}$$

$$T(N) = s + \frac{p}{N} \tag{2}$$

$$S_p = \frac{T(1)}{T(N)} \tag{3}$$

We adapt this formula to introduce our notion of userjobs U, itemjobs I, and processing cores C. The total number of worker nodes in our context will be equal to $U \times I$ and every one of these worker nodes may be equipped with C processing cores. So we redefine the speedup in terms of U, I, and C.

$$S_p = \frac{T(1,1,1)}{T(U,I,C)} \tag{4}$$

To calculate the speedup of our system, we must know the baseline time $T(1,1,1)$ (which we measured in Table 1) and $T(U,I,C)$, which is the time for the system to complete the calculations with U userjobs, I itemjobs and C cores (per worker node). We define $T(U,I,C)$ as

$$T(U,I,C) = s + \frac{P_U}{U} + \frac{P_I}{I} + \frac{P_C}{U \times I \times C} \tag{5}$$

If we complete the baseline figures (as percentages) from Table 1 in equations (4) and (5), we can express the speedup model for our algorithm in terms of userjobs, itemjobs and cores:

$$S_p(U,I,C) = \frac{100}{.02339 + \frac{.01675}{U} + \frac{.00001}{I} + \frac{99.95985}{U \times I \times C}} \tag{6}$$

To validate this speedup model we compared the predicted values with empirically determined speedup values. For a number of (independent) variations of userjobs, itemjobs and cores, we ran the complete recommendation process and measured $T(U,I,C)$. Since also $T(1,1,1)$ is known, we were able to compute the actual speedup and compare it to the predictions of the model. Fig. 12 visualizes the model together with the actual speedup values for the different variations. When we perform a simple regression analysis, we find that our model has an $R^2 = 0.9982$. The maximum speedup error in the model was 9% (for $(U,I,C) = (2,2,6)$), the average error rate was 4%. We therefore consider our model to be valid with an average error rate below 5% and usable as predictor for the speedup value of our recommender system.

With this model we are now able to explore speedup values for any desired range of (U,I,C) settings. Fig. 13 shows the speedup values for a different number of cores with an equal amount of userjobs and itemjobs varying from 1 to 2048. It is interesting to see that the speedup value converges to a number between 4000 and 4500. This limitation is the consequence of the fraction of non-parallel code as predicted by Amdahl's Law [2,15] which states that

$$\lim_{N \to \infty} S_p(N) = \frac{1}{s} \ . \tag{7}$$

The execution time of the parallelizable parts of the code can be made infinitesimally small (for large values of worker nodes N), so that the resulting final

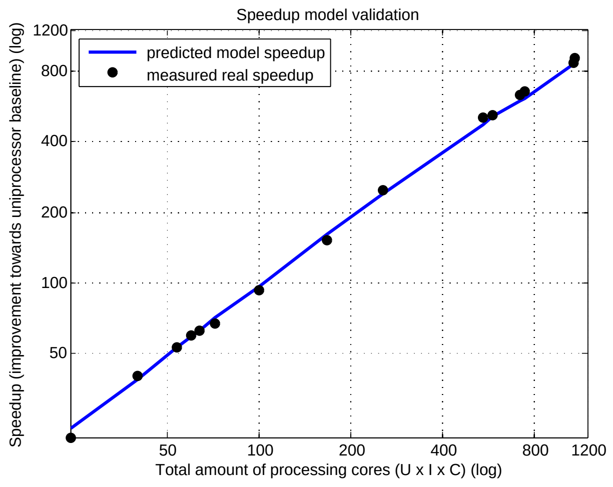


Fig. 12 Validation of the speedup model by comparing model-predicted speedup values with empirically determined speedup values. Note that both the x-axis and the y-axis are on a logarithmic scale.

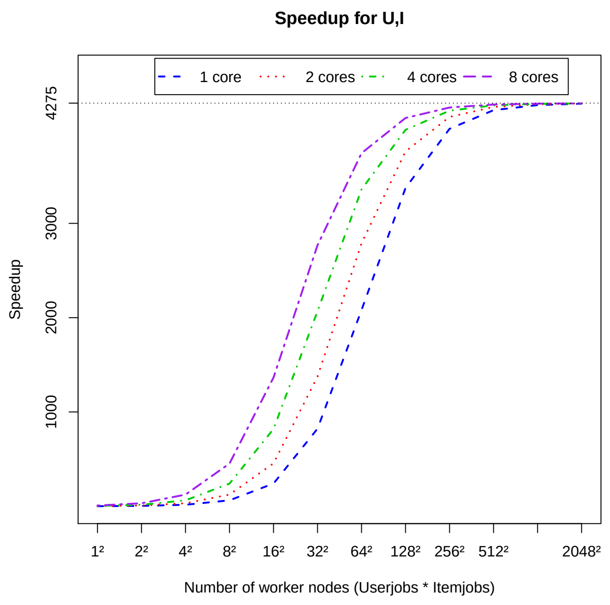


Fig. 13 Speedup as predicted by the speedup model for various (U, I) settings.

execution time consists almost completely of (and is therefore also limited by) the serial fraction of the code. If we calculate formula (7) with the known value of the serial fraction (Table 1), we find our speedup limitation (formula (9)).

$$\lim_{U,I,C\to\infty} S_p(U,I,C) = \frac{1}{s} \tag{8}$$

$$\lim_{U,I,C\to\infty} S_p(U,I,C) = \frac{100}{0.02339} = 4275 \tag{9}$$

Although the performance of our algorithm may be limited by a speedup value of 4275, this limit is only reached for very high values of userjobs and itemjobs (>512). If both the amount of userjobs and itemjobs is 512, the resulting number of jobs will be 512^2(=262 144). If every job is mapped on a worker node, this would require an unreasonably large cluster. Therefore we conclude that the scalability of our algorithm will be limited by the availability of computing hardware before its theoretical limit of speedup is reached.

We define parallel efficiency ϵ_p for our system in terms of userjobs, itemjobs, and cores as the following.

$$\epsilon_p(U,I,C) = \frac{S_p(U,I,C)}{U \times I \times C} \tag{10}$$

Using the speedup model as input, we are able to explore the scalability of our algorithm in terms of efficiency for any desired range of (U, I, C) settings. If we would run our recommender system on a cluster with 200 available worker nodes each with 8 processing cores (U, I, C equal to 10, 20, 8), we can expect a speedup value of 1142 which gives us a parallel efficiency of 71.4%. If we now compare our parallel efficiency values with results found in similar related work, such as 90% efficiency for a Hadoop-based item-based CF recommender with 8 nodes [21], we find that our solution easily achieves higher efficiency values (and so higher speedup values) for the same configuration (U, I, C equal to 4, 2, 1), namely 99.8%.

In Fig. 14 we plotted both speedup and efficiency (with four processing cores). Both graphs intersect at 50% for a number of worker nodes equal to $32^2(= 1024)$. The minimum parallel efficiency that should be maintained depends greatly on the availability of the hardware infrastructure and related costs.

It is clear that to come to an appropriate amount of userjobs, itemjobs and cores, a trade-off will have to be made between how fast the algorithm comes to a solution and how efficiently resources are used. Hager et al. [15] suggest the construction of a cost model and minimizing the product of walltime and infrastructure cost as a sensible balance. As was mentioned before, the available RAM of worker nodes may also be limited so that a certain minimum number of userjobs will be needed to meet hardware requirements. More userjobs implicates more fragmentation of the user input data and therefore reduced RAM requirements per worker node.

For general purposes, we suggest to set the number of used cores (C) to the number available in a single computing node and the number of userjobs (U) so that the associated RAM requirements match the available RAM. The number of itemjobs I can then be set such that the total number of jobs is a factor of the total number of computing nodes.

Our proposed algorithm combined with the parallel strategy described in this paper offers complete flexibility as to the number of subtasks the recommendation problem should be partitioned in, and can therefore easily be optimized for any given set of cost constraints or for any given hardware configuration.

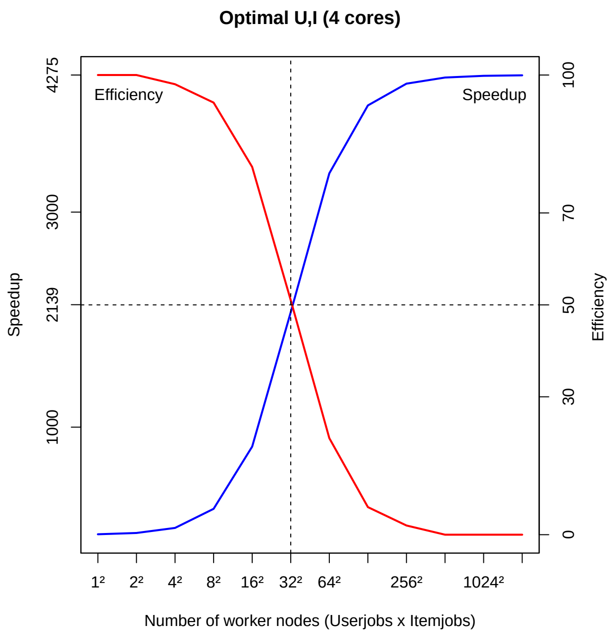


Fig. 14 Model speedup and efficiency for worker nodes with 4 cores and various (U, I) settings.

7.1 The performance on an other dataset

The true performance of the system will in the end be very dependent on the available hardware infrastructure, applied (U, I, C) settings and the dataset at hand. To gain more insight into the generalizability of our results, we did a small experiment with varying (U, I, C) settings on another dataset. We used a dataset borrowed from our previous research experiments on a cultural events website [13, 12]. From this website we collected 40,000 ratings, from 10,000 users on 40,000 events, which is considerably smaller than the MovieLens 10M dataset.

U	I	C	Time (s)	Speedup	Efficiency (%)
1	1	1	6691	1	100
2	2	1	2020	3.3	83
4	4	1	587	11.4	71
8	8	1	164	40.8	64
16	16	1	43	155.6	61
32	32	1	12	581.7	57
2	2	4	525	12.7	80
4	4	4	157	42.5	66
8	8	4	42	159.3	62
16	16	4	12	557.5	54
10	20	8	8	836.2	52

Table 2: Performance results of the system with a smaller dataset.

These results show that the efficiency of the performance of the recommender system on this dataset is slightly less than for the MovieLens dataset. For the calculation with 200 nodes, each having 8 cores, the efficiency drops from 71.4% (for MovieLens) to 52%. This is to be expected since the cultural dataset is much smaller (about 50 times). A smaller dataset indicates less work, while the typical overhead of reading data and executing the job division stays about the same, thus decreasing the overall efficiency.

These results however indicate an interesting transition from an offline recommendation scenario to a possible real-time recommender, without the need for a customized incremental recommendation algorithm. The full recommendation calculation on a single machine takes about two hours, which would force the recommender system to calculate recommendations in an offline batch scenario. However, if a computing infrastructure with 200 worker nodes (8 cores per node) is available then the calculation takes only 8 seconds which paves the way towards real-time (content-based) recommendation for an online recommender. So while our distributed recommender will show higher efficiency values for bigger datasets, even for small datasets the merits of increasing the recommendation calculation performance can be shown.

8 Conclusions and future work

In this work, we implemented an out-of-the-box content-based recommendation algorithm and showed how it could be efficiently parallelized and distributed across a distributed computing infrastructure. By focusing on data parallelism, the rating prediction recommendation task could be divided into equally sized and independent chunks of work which could then easily be mapped onto available worker nodes.

The total number of jobs can be expressed in terms of number of userjobs and itemjobs which introduces an increased level of flexibility. Setting the appropriate number of userjobs and itemjobs enables optimal use of RAM memory and allows to match the number of jobs to the number of available worker nodes. By carefully constructing the definition of 'work' in the context of recommender systems we were able to influence and improve the state of load imbalance between worker nodes. To actually divide the work (i.e., solve the makespan minimization problem) we introduced a simple greedy heuristic algorithm that showed an appropriate level of accuracy for most scenarios but could be supplemented with our proposed 'Robin Hood' extension if needed. Since the workload of chunks of work is load balanced, they can be evenly spread out across machines without the need for any inter node communication (as is the case for the MapReduce master-slave model).

An empirically validated performance model was built that allowed to model and predict the performance of our system in terms of parallel speedup and efficiency for any given configuration of userjobs, itemjobs and processing cores (U,I,C). Results, for the MovieLens 10M dataset, showed that although speedup converges to 4275 after a certain number of worker nodes are put to the task ($>512^2$), this limit is sufficient for realistic hardware configurations. With improved speedup the efficiency of our system decreases and so a cost model that takes both into account in the context of the available hardware infrastructure at hand will be needed. Thanks to its flexible design, the parallel and distributed recommendation algorithm presented in this work can be tailored to satisfy any set of cost constraints and thus makes optimal use of any given hardware configuration.

Our approach does not impose any file system requirements and can be run on any machine capable of running Python code. A disadvantage compared to a standard MapReduce setting is the lack of built-in fault tolerance. While MapReduce effortless handles corrupt or faulty computing nodes, we have to account for this situation manually and redistribute the work to other machines. Another downside is that the parallel approach presented in this work, has been specifically tailored to the content-based recommendation algorithm and therefore can not be applied to other algorithms without at least some minor modifications.

While we believe to have obtained higher parallel efficiency values than MapReduce solutions, the choice of which system to apply will in the end depend on the specific use case at hand.

In future work, we plan on evaluating and comparing our distributed recommendation approach with MapReduce implementations in a controlled environment on exactly the same hardware and for the same datasets. We are also currently working on parallelizing and distributing the computation of other collaborative filtering algorithms such as item-based collaborative filtering (IBCF).

Acknowledgements The described research activities were funded by a PhD grant to Simon Dooms of the Agency for Innovation by Science and Technology (IWT Vlaanderen). The computational resources (Stevin Supercomputer Infrastructure) and services used in this work were provided by Ghent University, the Hercules Foundation and the Flemish Government - department EWI.

References

1. Ahmadizar, F.: A new ant colony algorithm for makespan minimization in permutation flow shops. Computers & Industrial Engineering (2012)
2. Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proc. spring joint computer Conf., pp. 483–485. ACM (1967)
3. Anand, S.S., Mobasher, B.: Intelligent techniques for web personalization. In: Proc. Int. Conf. Intelligent Techniques for Web Personalization, pp. 1–36. Springer-Verlag (2003)
4. Berkovsky, S., Freyne, J.: Group-based recipe recommendations: analysis of data aggregation strategies. In: Proc. 4th ACM Conf. Recommender Systems, RecSys '10, pp. 111–118. ACM, New York, NY, USA (2010). DOI 10.1145/1864708.1864732. URL http://doi.acm.org/10.1145/1864708.1864732
5. Bilolikar, V., Jain, K., Sharma, M.: An annealed genetic algorithm for multi mode resource constrained project scheduling problem. Int. J. of Computer Applications **60**(1), 36–42 (2012)
6. Bobadilla, J., Serradilla, F., Bernal, J.: A new collaborative filtering metric that improves the behavior of recommender systems. Knowledge-Based Systems **23**(6), 520–528 (2010)
7. Chhabra, S., Resnick, P.: Cubethat: news article recommender. In: Proc. 6th ACM Conf. Recommender Systems, RecSys '12, pp. 295–296. ACM, New York, NY, USA (2012). DOI 10.1145/2365952.2366020. URL http://doi.acm.org/10.1145/2365952.2366020
8. Das, A., Datar, M., Garg, A., Rajaram, S.: Google news personalization: scalable online collaborative filtering. In: Proc. 16th Int. Conf. World Wide Web, pp. 271–280. ACM (2007)
9. De Pessemier, T., Vanhecke, K., Dooms, S., Martens, L.: Content-based recommendation algorithms on the hadoop mapreduce framework. In: Proc. 7th Int. Conf. Web Information Systems and Technologies. Ghent University, Department of Information technology (2011)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

11. Dooms, S., De Pessemier, T., Martens, L.: A file-based approach for recommender systems in high-performance computing environments. In: Proc. 22nd Int. workshop on database and expert systems applications, pp. 529–533. IEEE (2011). URL http://dx.doi.org/10.1109/DEXA.2011.3
12. Dooms, S., De Pessemier, T., Martens, L.: An online evaluation of explicit feedback mechanisms for recommender systems. In: Proc. 7th Int. Conf. Web Information Systems and Technologies, pp. 391–394 (2011)
13. Dooms, S., De Pessemier, T., Martens, L.: A user-centric evaluation of recommender algorithms for an event recommendation system. In: Workshop on Human Decision Making in Recommender Systems (Decisions@RecSys'11) and User-Centric Evaluation of Recommender Systems and Their Interfaces - 2 (UCERSTI 2) affiliated with 5th ACM Conf. Recommender Systems (RecSys 2011), pp. 67–73 (2011)
14. Gomez-Gasquet, P., Segura-Andres, R., Franco, D., Andres, C.: A makespan minimization in an m-stage flow shop lot streaming with sequence dependent setup times: Milp model and experimental approach. In: 6th Int. Conf. Industrial Engineering and Industrial Management, pp. 332–339 (2012)
15. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers, 1st edn. CRC Press, Inc., Boca Raton, FL, USA (2010)
16. Han, P., Xie, B., Yang, F., Shen, R.: A scalable p2p recommender system based on distributed collaborative filtering. Expert Systems with Applications **27**(2), 203 – 210 (2004). DOI 10.1016/j.eswa.2004.01.003. URL http://www.sciencedirect.com/science/article/pii/S0957417404000065
17. Herlocker, J., Konstan, J.A., Riedl, J.: An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. Information retrieval **5**(4), 287–310 (2002)
18. Herlocker, J.L., Konstan, J.A., Borchers, A., Riedl, J.: An algorithmic framework for performing collaborative filtering. In: Proc. 22nd Int. ACM SIGIR Conf. Research and development in information retrieval, pp. 230–237. ACM (1999)
19. Hochbaum, D.S., Shmoys, D.B.: Using dual approximation algorithms for scheduling problems theoretical and practical results. J. ACM **34**(1), 144–162 (1987). DOI 10.1145/7531.7535. URL http://doi.acm.org/10.1145/7531.7535
20. Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender systems: an introduction. Cambridge University Press (2010)
21. Jiang, J., Lu, J., Zhang, G., Long, G.: Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop. In: Services (SERVICES), 2011 IEEE World Congress on, pp. 490–497. IEEE (2011)
22. Keckler, S., Olukotun, K., Hofstee, H.: Multicore processors and systems. Springer (2009)
23. Lämmel, R.: Googles mapreduce programming modelrevisited. Science of Computer Programming **70**(1), 1–30 (2008)
24. Levi, A., Mokryn, O., Diot, C., Taft, N.: Finding a needle in a haystack of reviews: cold start context-based hotel recommender system. In: Proc. 6th ACM Conf. Recommender Systems, RecSys '12, pp. 115–122. ACM, New York, NY, USA (2012). DOI 10.1145/2365952.2365977. URL http://doi.acm.org/10.1145/2365952.2365977
25. Liu, M., Zheng, F., Wang, S., Xu, Y.: Approximation algorithms for parallel machine scheduling with linear deterioration. Theoretical Computer Science (2012)
26. McCarthy, J.F., Anagnost, T.D.: MusicFX: an arbiter of group preferences for computer supported collaborative workouts. In: Proc. ACM Conf. Computer supported cooperative work, CSCW '98, pp. 363–372. ACM, New York, NY, USA (1998). DOI 10.1145/289444.289511. URL http://doi.acm.org/10.1145/289444.289511
27. Pera, M.S., Ng, Y.K.: Personalized recommendations on books for k-12 readers. In: Proc. 5th ACM workshop on Research advances in large digital book repositories and complementary media, BooksOnline '12, pp. 11–12. ACM, New York, NY, USA (2012). DOI 10.1145/2390116.2390124. URL http://doi.acm.org/10.1145/2390116.2390124
28. Peralta, V.: Extraction and integration of movielens and imdb data. Tech. rep., Technical Report, Laboratoire PRiSM, Université de Versailles, France (2007)
29. Sarwar, B., Karypis, G., Konstan, J., Riedl, J.: Application of dimensionality reduction in recommender system-a case study. Tech. rep., DTIC Document (2000)
30. Sarwar, B., Karypis, G., Konstan, J., Riedl, J.: Incremental singular value decomposition algorithms for highly scalable recommender systems. In: 5th Int. Conf. Computer and Information Science, pp. 27–28. Citeseer (2002)

31. Schelter, S., Boden, C., Markl, V.: Scalable similarity-based neighborhood methods with mapreduce. In: Proc. 6th ACM Conf. on Recommender Systems, pp. 163–170. ACM (2012)
32. Symeonidis, P., Nanopoulos, A., Manolopoulos, Y.: Moviexplain: a recommender system with explanations. In: Proc. 3rd ACM Conf. Recommender Systems, RecSys '09, pp. 317–320. ACM, New York, NY, USA (2009). DOI 10.1145/1639714.1639777. URL http://doi.acm.org/10.1145/1639714.1639777
33. Takács, G., Pilászy, I., Németh, B., Tikk, D.: Scalable collaborative filtering approaches for large recommender systems. J. of Machine Learning Research **10**, 623–656 (2009)
34. Xie, B., Han, P., Yang, F., Shen, R.M., Zeng, H.J., Chen, Z.: Dcfla: A distributed collaborative-filtering neighbor-locating algorithm. Inf. Sci. **177**(6), 1349–1363 (2007). DOI 10.1016/j.ins.2006.09.005. URL http://dx.doi.org/10.1016/j.ins.2006.09.005
35. Yang, D., Chen, T., Zhang, W., Lu, Q., Yu, Y.: Local implicit feedback mining for music recommendation. In: Proc. 6th ACM Conf. Recommender Systems, RecSys '12, pp. 91–98. ACM, New York, NY, USA (2012). DOI 10.1145/2365952.2365973. URL http://doi.acm.org/10.1145/2365952.2365973
36. Zhao, Z., Shang, M.: User-based collaborative-filtering recommendation algorithms on hadoop. In: 3rd Int. Conf. Knowledge Discovery and Data Mining (WKDD'10), pp. 478–481. IEEE (2010)