Parallel Books Recommendation System

Department of Physics and Computer Science

Wilfrid Laurier University

CP631 - Advanced Parallel Programming

By

ID: 245813460

ID: 245830320

Parallel Books Recommendation System

Introduction

The objective of the project is to build a recommendation system that suggests items tailored to user preferences. Such systems are common in applications we use today. For example, Netflix's recommendation system combines algorithms with human insights to provide personalized content suggestions, such as movies, for each user. In our case, we chose to work on a book recommendation system. These systems are always considered time-consuming.

The main problem with the system is that it is time-consuming. For instance, to recommend a book to a user, we first have to calculate how similar the user is to all other users. Then we identify the most similar user and compare their books list to filter out those the original user has already read. This process takes significant time, and in such systems or application, time is crucial, whether a user clicks a button and waits for recommendations or the system runs in the background to update homepage preferences. Therefore, implementing parallelization techniques would be helpful and would streamline the process more efficiently.

To achieve our goal efficiently, our plan is to explore and experiment with multiple techniques for parallelization. Specifically, we will employ approaches using Apache Spark with machine learning capabilities, as well as Spark implementations without machine learning. Additionally, we will also explore using traditional parallel computing paradigms such as MPI and OpenMP. Our exploration will include a comparison between the processing times. We will implement the methods using two programming languages: C++ and Python. by doing this, we will enable a comparison and and understanding of performance differences and advantages inherent in each approach.

Dataset

To build the system, we will use the Book-Crossing dataset, it is a well-known benchmark in building recommendation systems. The dataset contains three files:

- **Books**: includes details such as title, genre, author, ISBN, and publication year. The file contains 271379 records.
- **Users**: contains user-related information, including user ID and age. The file contains 278859 records.
- **Ratings**: records the ratings that users have assigned to each book. The file contains 1149780 records.

Methodology

In this section, we will describe the methods we used in order to build the recommendation system.

Collaborative Filtering (CF)

Collaborative filtering is a method used to build recommendation systems by suggesting items to users based on the preferences of similar users or items. It begins by identifying users or items with similar behavior or patterns, and uses these patterns to predict what a user will like. Collaborative filtering can be user-based—where recommendations are made according to the preferences of similar users—or item-based—where recommendations rely on the similarity of items to those the user has liked.

In our system, we implement a hybrid collaborative filtering approach which combines both user-based and item-based similarities to generate personalized book recommendations for a target user. We use a parameter α to balance the contribution of each method, as shown in Equation 1. In the com:

$$\text{HybridRecommendations} = \alpha\,\text{item\_based} \;+\; (1-\alpha)\,\text{user\_based}. \tag{1}$$

We set $\alpha = 0.5$ for equal contributions from each type of collaborative filtering. To build this system, we employed two implementations: one using a hybrid MPI/OpenMP approach and another using a Spark transformations.

1. Collaborative Filtering using MPI and OpenMP

In this approach, we implement a hybrid MPI and OpenMP implementation, where we combine the distributed memory parallelism and shared-memory concurrency as follows: MPI (Distributed memory) to partition the user–rating dataset across multiple processes (ranks). OpenMP (Shared-memory) to use multithreading on each process for compute-intensive loops. The approach steps are as follows:

- **Data Loading**: Each process reads the data file, and only retains partition of the records based on the user ID. This distributes the data evenly, with zero communication at load time.
- **Mean Centering**: Ratings are centered using the mean per user to take account for individual user ratings bias.
- **Computing Norms**: We need to calculate norms of ratings in order to calculate similarities. User norms and item norms are computed at the same time using OpenMP parallel loops. Each thread independently calculates norms for different users or items. The results are merged using critical sections to prevent data races.
- **Computing Similarity**: As we will implement a hybrid user-based and item-based approach, we need to calculate cosine similarities between users (user-user) and items (item-item). To efficiently parallelize these calculations, we employ threads. Each OpenMP thread keeps its local similarity calculations to avoid frequent synchronization overhead. When local computations are finished, we merge results into a global similarity map using short critical sections.
- **Top-K Neighbors**: After similarities are calculated, we keep only Top-$K = 50$ most similar neighbors for each user and item. This step reduces memory and computation in the subsequent stages.
- **Hybrid rating predictions:** To calculate the recommendations for a we merge the user-based and item-based predictions into a hybrid score. As the data is distributed across processes, recommendations are computed only by the process

responsible for the target user. Further computations parallelization is introduced by the threads within the process to aggregate scores.

- **Recommendations:** Items are ranked based on their predicted ratings and displayed to the user.

2. Collaborative Filtering Using Spark Transformations

The parallelization in this method is introduced by using parallel computation through Resilient Distributed Datasets. The data will be partitioned or split across multiple executors, which enables parallel processing. The cosine similarity between user-user and user-item are calculated using distributed transformations such as reduceByKey, or map. Top-$K = 50$ most similar neighbors for each user and item only are kept. The norm values for users and items are calculated and distributed to executors using broadcast to reduce network overhead and redundant computations.

Machine Learning

We used the ALS (Alternating Least Squares) model of PySpark. In ALS, the user-item matrix is decomposed into two lower-dimensional matrices: 1) User Matrix and 2) Item matrix. The dot product of the user and item vectors in the latent space approximates the original rating. The lookup algorithm is:

- The code creates a new DataFrame for the user.
- Filter for Target User: The code filters the DataFrame to find the row(s) where the user matches the target user (matching user ID).
- Extract User Index: If matched, the code extracts the userIndex value from the filtered DataFrame.
- This lookup code maps the User-ID (identifier: userIndex) to a numerical userIndex that can be used by the ALS model to generate recommendations

The benefit of ALS method is that it can process very large datasets without stressing the computational units. For proof, our source code (of ALS) has a simple Python

implementation of the K-nearest Neighbours, which causes session crash due to high memory consumption. So, it only works properly with a fraction of the dataset (20%-25%). Then we used that same dataset, but full, in ALS to train without any crash. And we generated recommendations for users within a minute.

Improving the ALS Further

The ALS approach was more feasible. However, we wanted to improve it further. We thought of applying parallelism in the lookup algorithm to make the code faster. So far, we have item vectors. Then we also collected item factors from the trained ALS model. Next, we split the items into partitions for parallel processing. After partitioning, the code sends the user's factor vector to all workers (Broadcast). The idea is to parallelize the dot product computation by partitioning the item vectors. This method reduced the recommendation generation time for the user. Also, the code is modular, so the number of partitions can be changed. The outcomes of this approach is discussed in the result section.

User ID	Serial Time (s)	Parallel Time (s)
11676	6.743	3.372
35859	1.888	1.758
16795	2.784	1.856
60244	1.92	2.514

Table 1

Time comparison between the serial and the parallel hybrid collaborative filtering algorithm using C++.

Results

In this section we will compare the results of the implemented methods using Spark (not the ALS approach) and MPI/OpenMP parallelization.

Hybrid MPI/OpenMP (CF)

when we experimented using the method to recommend items for a target user and compared it to the serial implementation we found out that the number of neighbours a target user has effect the performance. For example when a user has a lot of other user who rated same books or at least one book the calculations will take more time but when using the parallel implementation it effectively outperforms the serial code, however some cases when the target user has very few neighbors the serial code outperforms the parallel. Figure 1 illustrates the processing time of the serial and parallel algorithms based on the number of neighbours of the target user. As the number of neighbours increases, the serial processing time also increases, while the parallel method outperforms it. Additionally, Table 1 shows the processing times for several target users using both the serial and parallel methods with (p=4), showing that the parallel method outperforms the serial implementation in some cases. Moreover, we experimented with increasing the number of processes, and the results indicated that doing so introduces additional overhead, which leads to longer processing times.

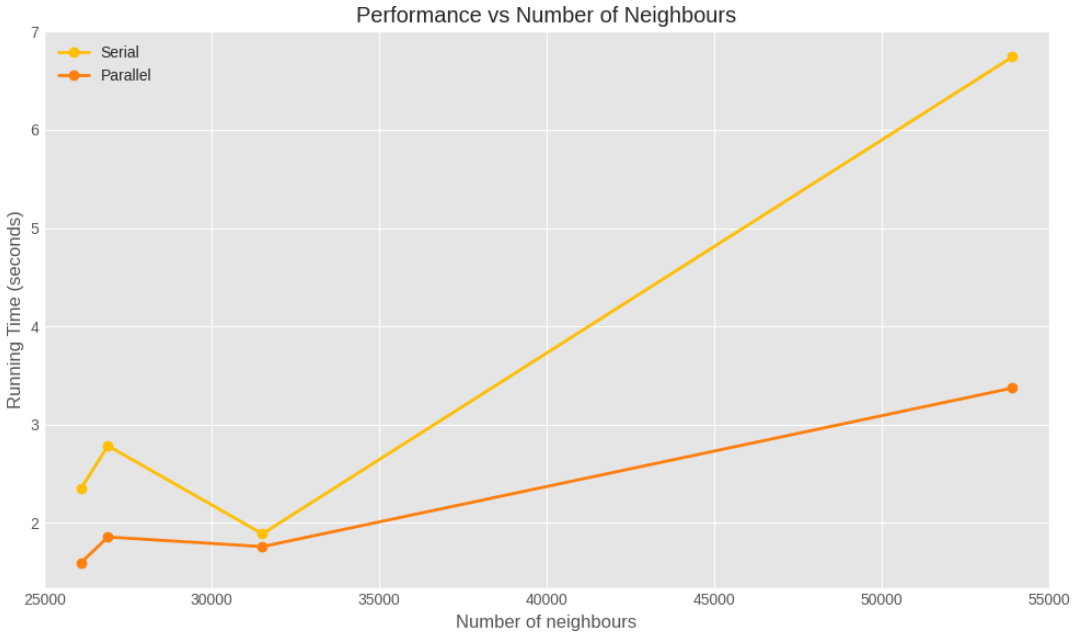

Figure 1. Performance of the serial and the parallel algorithms given the number of neighbours of the target user.

Target User ID	Processing Time
11676	3min 28s
35859	1min 41s
269566	1min 19s
52584	1min 24s

Table 2

Processing time of the Spark Transformation Collaborative Filtering approach with different target users.

Spark Transformations (CF)

Using this method, the algorithm finish processing slower than the MPI/OpenMP implementation in C++. Every operation in Spark such as map, flatMap, or reduceByKey is translated first to the JVM, then back into a Python object for lambda expressions, and then back into the JVM. The serialization–deserialization steps generate more overhead, specifically when performing many small operations. On the other hand, the MPI/OpenMP code runs in native memory with pointers and registers, resulting in $\hat{=} 0$ dispatch overhead. Additionally, joining or grouping in introduce a distributed shuffle, in C++ the data is retained in local structures vectors or hash maps and communicates the minimal dot-product via MPI, avoiding heavy disk I/O. Table2 shows the results for different target users, demonstrating the slower performance.

Performance of the Improved ALS

Despite the efficiency, we noticed some things while exploring with different partition numbers. After a certain value, partitioning more means more overhead and thus results in slower computation. The reason is with the dataset size. If the dataset is extremely large, it will benefit from more partitions. But for our dataset (200K+ books info (rows)), 2-10 partitions yield the minimum time. The Serialized ALS approach takes around 45-55 seconds to generate recommendations. The lookup time for parallel ALS based on the partition number is presented in 3.

Partition	Lookup-Time (seconds)
2	∼21
5	∼23
10	∼24
20	∼33
50	∼52
100	∼80

Table 3

Lookup Times depending on the Partitions using ALS.

Moreover, the recommendations generated using both ALS approaches are the same, proving that the partition is working properly despite the task split.

Conclusion

In conclusion, in this project, we explored different approaches to create a parallel books recommendation system using the Book-Crossing Dataset. Each approach produced different results, and we observed that as the number of neighbours increases for a target user, the lookup processing time also increases. The results showed that using only Spark transformations and RDDs to build collaborative filtering recommendation system is not the best approach and achieved the slowest results. On the other hand, when we used Spark with an ALS model, it achieved better performance and lower processing times; however, as the number of partitions increased, the processing time also increased due to overheads. Finally, using MPI and OpenMP together achieved good performance and showed lower processing times.