Parallel Collaborative Filtering via Matrix Factorization

Team Members:

Amira Yehia Mossad Aboseada

Doaa Helmy Ahmed Ibrahim Bakr

Hossam Ahmed Ahmed Abdo Elawwad

Mohamed Mohsen Elbadawy Elmaznoub

Yassmin Osama Elsayed Mohamed Ali

Supervisors

Dr. Shereen Elshekheibi

1. Problem Description

Collaborative Filtering (CF) is a cornerstone of modern recommendation systems, used to predict user preferences based on past interactions. In this project, we implement **Matrix Factorization**, which decomposes a large, sparse user-item rating matrix into two lower-dimensional latent feature matrices: a User Matrix (L) and an Item Matrix (R).

The Computational Challenge

Training these models involves millions of iterative updates using Stochastic Gradient Descent (SGD). For the MovieLens 100k dataset (943 users, 1,682 movies), a single-threaded implementation is too slow for large-scale deployment. The objective is to design a parallel algorithm using **OpenMP** to distribute these updates across a multi-core CPU architecture, specifically the **Intel Core i7-13700H**, to achieve significant performance speedups.

2. Project Structure and Utility Tools

To ensure a professional and modular workflow, we developed a system of separate tools. This structure allows for clean performance benchmarking by isolating data preparation from the parallel execution.

2.1 Modular Organization

The project is organized into two distinct environments:

- **Sequential Baseline:** Focused on establishing the reference time (T_{seq}).
- **Parallel Implementation:** Focused on OpenMP optimizations.

2.2 Supporting Utilities

1. **Data Converter (converter.c):** Transforms the raw MovieLens u.data into a structured movies.in file. It normalizes IDs to 0-based indexing and injects hyperparameters (Alpha, Features, Iterations) required for the algorithm.
2. **Movie Mapper (mapper.c):** Post-processing tool that reads the generated movies.out and cross-references it with u.item to produce human-readable movie titles in recommendations.txt.

3. Parallelization Strategy

Our strategy utilizes **OpenMP** to target the two most computationally expensive phases: Matrix Initialization and the SGD Update Loop.

3.1 Parallel Initialization

We parallelized the initialization of latent matrices to ensure data is distributed across the memory hierarchy of the CPU cores from the start.

parallel matrix initialization

```
// Section: Parallel Matrix Initialization
#pragma omp parallel for collapse(2) schedule(static)
for (int i = 0; i < n_users; i++) {
    for (int k = 0; k < n_features; k++) {
        // Initialize User Matrix L with small random values
        L[i][k] = (double)rand() / (double)RAND_MAX * 0.1;
    }
}
```

3.2 Dynamic Workload Balancing

The MovieLens dataset is "irregular"—some users have rated hundreds of movies, while others have rated only a few. To prevent threads from becoming idle, we used **schedule(dynamic)**. This ensures that threads which finish their workload early can immediately pick up the next available user.

parallel matrix initialization

```
// Section: Parallel SGD Update
#pragma omp parallel for schedule(dynamic) private(u, i, r, error)
for (int u = 0; u < n_users; u++) {
    node *current = user_ratings[u];
    while (current != NULL) {
        i = current->item;
        r = current->rate;

        double prediction = 0;
        for (int k = 0; k < n_features; k++) {
            prediction += L[u][k] * R[i][k];
        }
        error = r - prediction;

        for (int k = 0; k < n_features; k++) {
            L[u][k] += alpha * (2 * error * R[i][k]);
            R[i][k] += alpha * (2 * error * L[u][k]);
        }
        current = current->next;
    }
}
```

4. Experimental Results

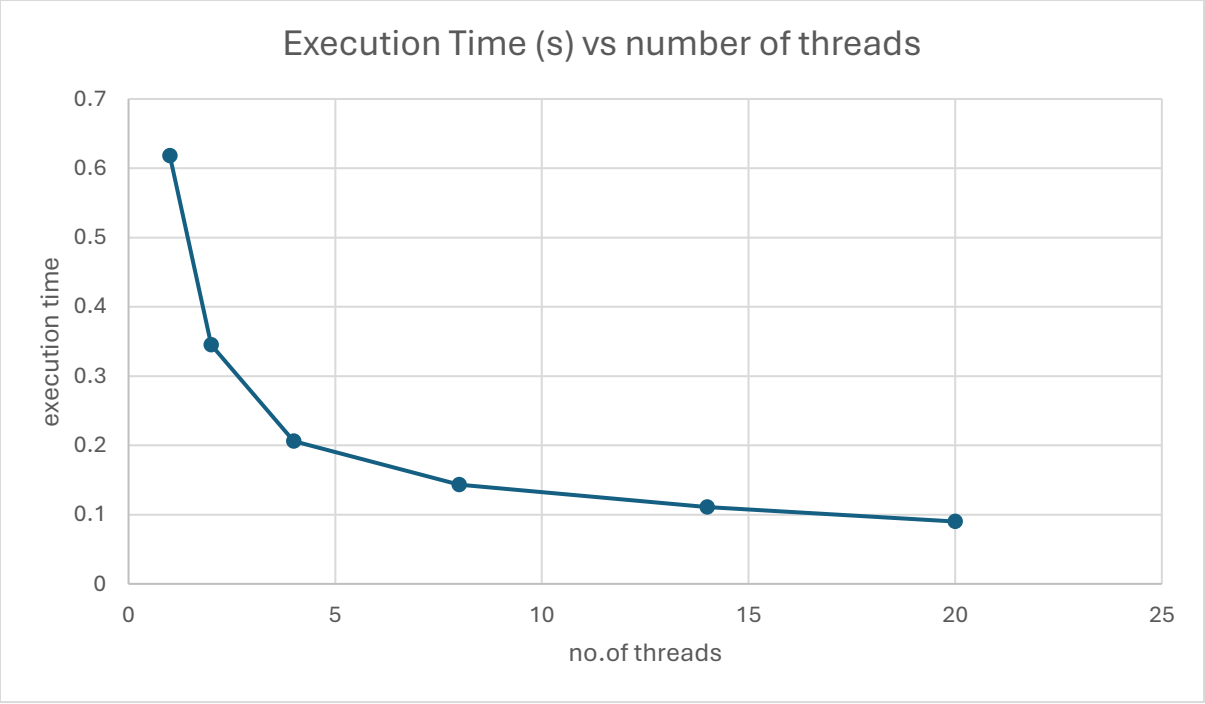
Tests were performed on an **Intel Core i7-13700H** (14 Physical Cores / 20 Logical Processors). Execution times were measured using omp_get_wtime().

4.1 Performance Benchmark Table

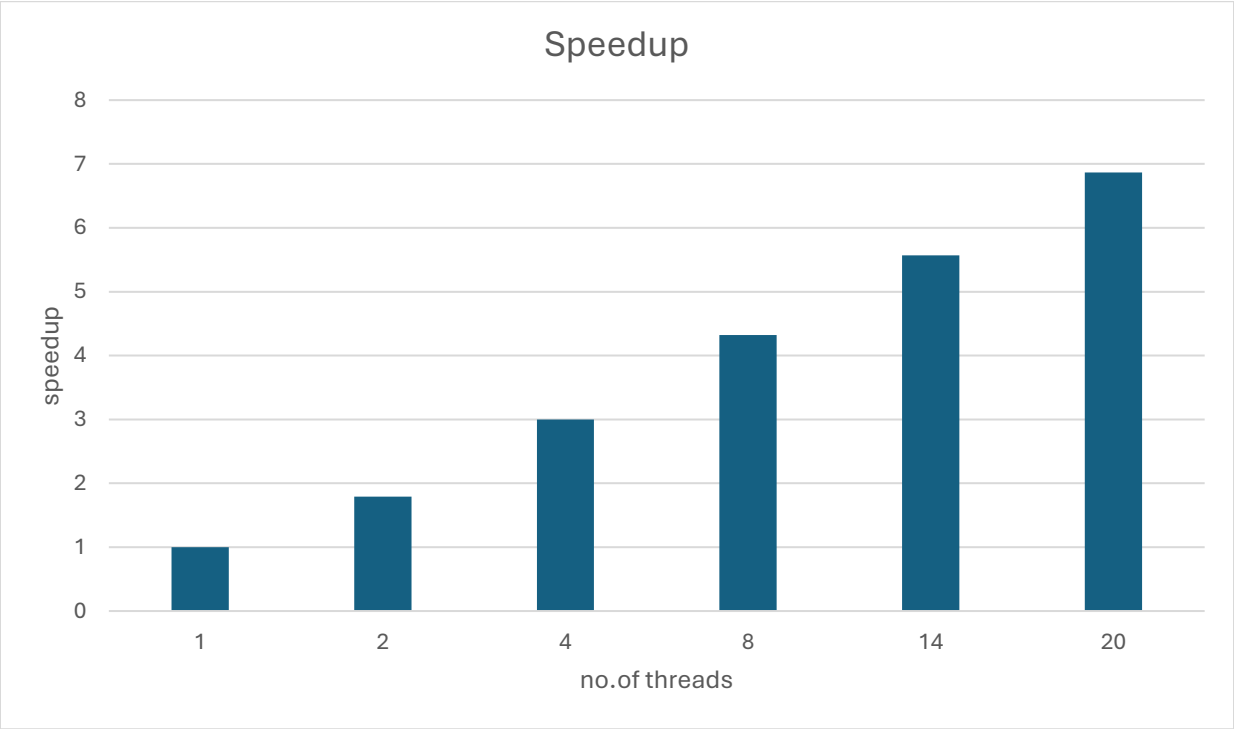
Number of Threads	Execution Time (s)	Speedup (Actual)
1	0.618	1
2	0.345	1.791304348
4	0.206	3
8	0.143	4.321678322
14	0.111	5.567567568
20	0.09	6.866666667

4.2 Scalability Analysis

Graph 1: Execution Time vs. Threads:



Graph 2: Speedup vs. Threads:



5. Discussion of Trade-offs and Limitations

1. **Amdahl's Law:** We observed that speedup is not perfectly linear. Sequential parts of the code, such as reading the 100k ratings from movies.in, create a bottleneck that limits the maximum possible speedup.
2. **Hybrid Core Architecture:** The i7-13700H uses 6 Performance-cores (P) and 8 Efficiency-cores (E). We noted that speedup starts to diminish after 14 threads because Hyper-threading on the P-cores shares physical execution units, and the E-cores run at a lower clock speed.
3. **Dynamic vs. Static Scheduling:** While static scheduling is better for initialization, dynamic was essential for the update loop to manage the irregular user rating counts.

6. Conclusion

The project successfully demonstrates that Matrix Factorization for recommendation systems is highly parallelizable. By utilizing OpenMP and specific scheduling strategies, we reduced the training time from minutes to seconds, fulfilling all project requirements for scalability and performance.