# Parallel and Distributed Computing - Recommender System


## Abstract

*We describe the developed solution for an OpenMP implementation of a recommender system. This report outlines the serial implementation as well as the approach used for parallelization and the decomposition, synchronization and load balancing concerns.*


## 1. Introduction

The purpose of this project is to simulate a recommender system which, given a set of items rated highly by a user and those with similar profiles, suggests new items for each user to try out. For this, we use an iterative method called *matrix factorization*.

In section 2, we start by describing the structures used in the program and the phases of the algorithm to achieve the desired output. In section 3, we present the proposed parallel solution alongside some idiosyncrasies noted during development. Finally, we test both versions against one another and compare empirical results on varying metrics.

## 2. Overview

### 2.1. Structures

- *mat2d*: represents any matrix. It has 3 components: *number of rows*, *number of columns* and a *values* array.
- *non-zero-entry*: represents any non-zero entry of the initial matrix. It has 3 components: the *row* and *column*, which situate the entry, and its *value*.
- *non-zero-entry array*: represents the initial matrix *A* by storing all non-zero entries. We opted for this representation as it holds all necessary information and is contiguous in memory. This array is accessed exclusively for **reads**.

### 2.2. Phases

#### 2.2.1 Initialization

Read from the input file and allocate the initial copies of matrices *L* and *R*, as well as matrix *B*. Transpose *R* so as to properly index it during the next phase and improve cache locality.

#### 2.2.2 Matrix Factorization

Allocate the stable versions for *L* and *R*, *L-stable* and *R-stable*, respectively. For all iterations, iterate through each *non-zero-entry* $(i, j)$ of the array, compute the dot product of row $i$ of *L-stable* with row $j$ of *R-stable* and with it part of the summation which is added to every entry $(i, k)$ of *L* and $(j, k)$ of *R*, $\forall k : 0 < k < numberOfFeatures$.
Compute matrix $B = L \times R$.

#### 2.2.3 Termination

Print the program output by iterating though *B* and the *non-zero-entries array* simultaneously. Free all remaining resources.

## 3. OpenMP Version

### 3.1. Parallel section and parallel for-loop

We decided on a single parallel section in the program. Additionally, the loop which iterates through the *non-zero-entry array* was parallelized, with each thread taking a chunk of the non-zero entries and computing part of matrices *L* and *R*. The final matrix product was also trivially parallelized.

### 3.2. Parallel `memcpy`

Given that we copy two matrices each iteration, and although `memcpy` is fairly efficient as is, this needed to be addressed. To this end, we assign a stride per thread where $stride = \lfloor matrixSize \div threadCount \rfloor$.
Finally, to account for uneven distributions, the last thread of the process will copy all leftover bytes.

### 3.3. Synchronization

There are three parts in the program which need synchronization:

1. The aforementioned matrix copies, as without the copied values, matrices *L-stable* and *R-stable* would not be up to date for that iteration.
2. Access to the unstable matrices when updating the values
3. Part of the manual reductions implemented, discussed in a following paragraph.

### 3.4. Load Balancing

Upon experimenting, we decided to leave the scheduling as *static*. There were small imbalances but *dynamic* had too big an overhead to compensate and *guided* didn't show any comparable improvements. Furthermore, the nature of the problem doesn't lend itself to any patterns and since we only distributed entries from the *non-zero-entry array* among threads, execution time should be similar between them, excessive synchronization notwithstanding.

### 3.5. Reductions

In phase 2, the writes on the entries of matrices *L* and *R* needed to be atomic to prevent race conditions. In order to improve on this, we created two manual reductions, one for reducing *L* and the other for reducing *R*, in each iteration of the cycle. However, we never run both at the same time.

Intuition states that when splitting the iterations between threads and the *non-zero-entries array* is ordered by *rows* (*columns*), most threads will have access to mostly different *rows* (*columns*) to work with and so the big contention point happens when writing to each individual *column* (*row*). In other words, reducing only the matrix whose accesses would cause more conflicts while keeping the accesses to the other *atomic* not only made the most sense, it also proved more efficient for all test cases.

We took advantage of this by reducing *R* when $numberUsers \geq numberItems$ and ordering by columns and reducing *L* otherwise.

To accomplish this we allocated a new *reduction-array* with $size = threadCount$, where each thread writes to their own matrix. This array is created at the beginning of phase 2 and each entry is zeroed out (by `memset`-*ing* everything to *0* in each iteration.

Moreover, we decided on manual reductions over built-in custom reductions as these allowed us to parallelize the aggregation of the results of the partial matrices in the final matrix by splitting the load between threads in groups of rows. To guarantee the right results we needed another barrier at this end of this loop.

## 4. Evaluation

The tests were performed on a machine running *Manjaro Linux* with *16GB of DDR4 RAM at 3200MHz 16-16-16-36* and an *AMD Ryzen 1700X 8 core / 16 thread CPU at 3.8 GHz*. The program was compiled with *GCC 9.3* without optimizations.

Times were measured using `clock_gettime()` with `CLOCK_MONOTONIC`. For memory usage, *valgrind*'s *massif* tool was used.

Two parallel implementations were compared against the *serial* version

- *atomic (A)*: naive approach using *atomic* directives
- *reduction (R)*: refined version using our custom matrix reduction mechanisms

### 4.1. Speedup

As shown in the graphs in the Appendix, both approaches provide us with speedups - with reductions either achieving or getting close to achieving ideal speedups.

Our implementation scales close to linearly up until 8 threads - the number of physical cores in the machine. Beyond this, scaling deteriorates as SMT is engaged, which is to be expected, and despite 12 threads sometimes showing performance loss for certain instances, it brought about considerable improvements when testing with all 16 threads.

Instances where speedup is greater than the ideal are, coincidentally, instances where we order by columns. The likely scenario is that the data is then distributed in favor of the way we iterate over it.

Running the OpenMP implementation for very small matrices or with a single thread is generally not worth it, indicated by the slowdowns in Table 2, due to the thread initialization overhead overshadowing any performance gains.

### 4.2. Memory usage

| Implementation | Peak memory usage (Bytes) |
|---|---|
| Serial | 32 824 |
| Atomic (16 threads) | 33 312 |
| Reduction (8 threads) | 51 144 |
| Reduction (16 threads) | 74 568 |

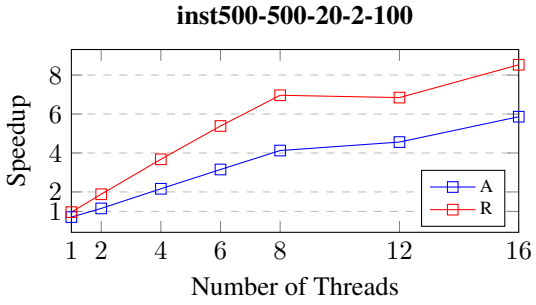
Table 1: Peak memory usage of *inst30-40-10-2-10*

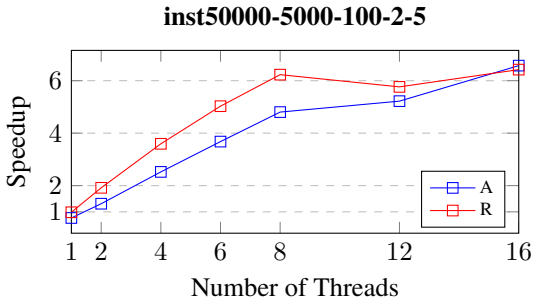
As expected, using a reduction increases memory usage compared to both *atomic* and *serial* implementations due to needing additional matrices to calculate partial values. It also scales with the number of threads because the amount of auxiliary matrices corresponds to the number of threads.

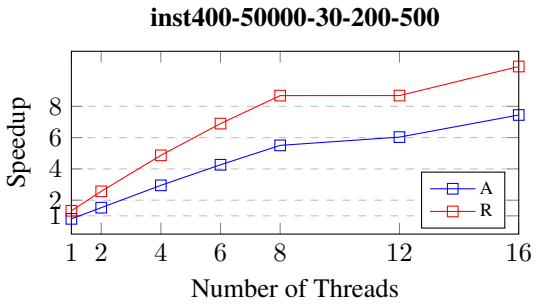
## 5. Conclusion

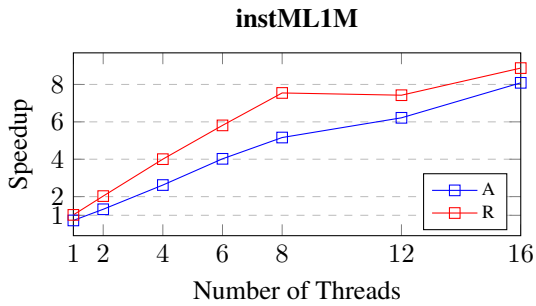
Overall, our solution achieved very good results, even going beyond the ideal speedup for certain instances and when ran for any number of threads less or equal to the number of physical cores.

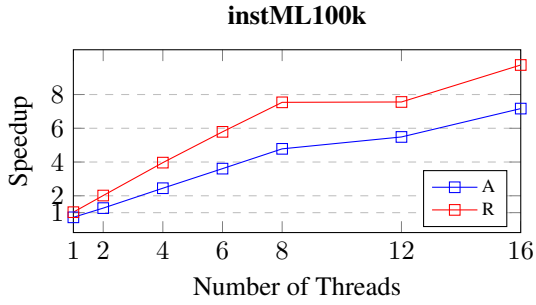
# Appendix

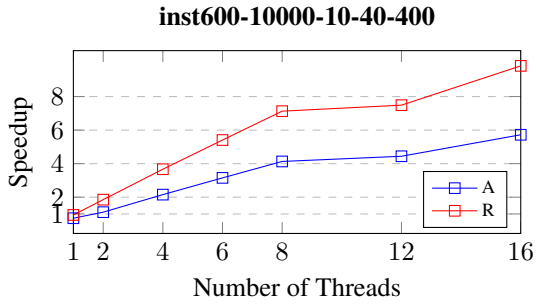



| Instance | Exec. Time (s) | | | | | Speedup w/ **4 Threads** |
|---|---|---|---|---|---|---|
| | Serial | OMP-1 | OMP-2 | OMP-4 | OMP-8 | |
| inst0 | 0.001 | 0.006 | 0.008 | 0.015 | 0.018 | - |
| inst1000-1000-100-2-30 | 18.123 | 18.172 | 9.434 | 5.022 | 2.922 | 3.609 |
| inst1 | 0.063 | 0.106 | 0.242 | 0.294 | 0.374 | - |
| inst200-10000-50-100-300 | 24.711 | 22.661 | 11.433 | 5.820 | 3.043 | 4.246 |
| inst2 | 0.060 | 0.086 | 0.119 | 0.151 | 0.202 | - |
| inst30-40-10-2-10 | 0.421 | 0.497 | 0.325 | 0.228 | 0.224 | 1.879 |
| inst400-50000-30-200-50 | 35.813 | 27.084 | 13.962 | 7.367 | 4.129 | 4.861 |
| inst50000-5000-100-2-5 | 156.984 | 159.688 | 81.943 | 43.698 | 25.208 | 3.570 |
| inst500-500-20-2-100 | 57.798 | 60.307 | 30.910 | 15.822 | 8.354 | 3.653 |
| inst600-10000-10-40-400 | 83.490 | 88.823 | 44.955 | 22.774 | 11.717 | 3.666 |
| instML100k | 104.930 | 102.058 | 52.098 | 26.467 | 13.922 | 3.965 |
| instML1M | 125.201 | 122.862 | 61.969 | 31.691 | 16.656 | 3.951 |

Table 2: Execution times for the serial and OpenMP (w/ varying number of threads) implementations