Recommender System

Project Report

Parallel and Distributed Computing

Group 42

April 3rd, 2020

Instituto Superior Técnico

Implementation

Matrix representation

We represented the sparse matrix A using a coordinate representation (COO) vector, meaning each entry is stored as a structure with row, column and value. Matrices L, R and B are stored through an array of arrays. Memory is allocated at the beginning of the program (once the input file is read) for each matrix and freed at the end, to ensure contiguous regions of memory are used for entries in the same row.
Additionally, we have two indexing vectors where we store all entries in A arranged by column and row, at the beginning of the program, saving time looking for entries when calculating the deltas. These auxiliary vectors also keep the order of the matrix entries, maximizing cache locality.

Iteration

In each iteration, for each element in the A vector, we calculate the deltas and update L and R accordingly. To calculate the deltas, we use the coordinates of the element to find all other values of A in the same column or row (given to us by the indexing vectors). Each iteration is O($nA * nF * \left(\frac{nA}{nU} + \frac{nA}{nI}\right)$), which tends to be well below O(n^3) for sparse matrices.
All updated values are stored in auxiliary matrices, L_{aux} and R_{aux}, which have their pointers swapped with L and R at the end of the iteration, effectively acting as L^{t+1} and R^{t+1} without the need of copying the results.
Finally, the updated values of L and R are used to calculate a new matrix B. This operation has a O(n^3) cost. To minimize cache misses in accessing R by column, we transpose the matrix O(n^2) and use R^T for calculating the product.

Considered approaches

CSR (compressed sparse row) was originally proposed for sparse matrix representation, but having the row coordinate facilitates indexing and iteration. Transposing matrices L and B was also considered in order to minimize cache misses in each iteration, but the time penalty of transposing outweighed the benefits.
As for the matrix multiplication, we believe it could be improved using a divide-and-conquer algorithm. We attempted to replicate Strassen's algorithm which has less complexity, but actually took more time to terminate (possibly due to excessive jumps in recursion and less cache optimization).

Parallel optimization

We further optimized our implementation for parallel execution using OpenMP directives. The code inside each iteration is parallelized. We avoided parallelizing smaller portions of code in order to maximize the work done by each thread. For smaller matrices, parallel

execution is disabled to avoid excessive overheads. According to ompP, our solution has a parallel coverage of 99.37% of the execution, and the static schedule used in the main for loop provides the best load balancing from our tests (only 3.13% imbalance).

Results

Test	Serial	Parallel (1 thread)	Parallel (2 threads)	Parallel (4 threads)	Parallel (8 threads)
inst0	0.023s	0.054s	0.054s	0.054s	0.054s
inst1	0.386s	1.342s	1.346s	1.466s	1.356s
inst2	0.430s	1.261s	1.254s	1.187s	1.349s
inst30	8.147s	8.213s	4.687s	2.682s	3.767s
inst1000	19m49.546s	20m35.969s	9m47.259s	4m57.258s	4m56.043s

The results fall short when compared to the execution times provided. However, great speedups can be noticed when comparing the execution times of the last two tests in 1, 2 and 4 threads, which confirms the parallel coverage obtained. There doesn't seem to be any improvement running our solution with 8 threads, likely caused by the memory accesses acting as a bottleneck.