Recommender System Project Report – OpenMP

Parallel and Distributed Computing - 2019/2020

Group 05:
86447 - João Palet
88080 - Miguel Francisco
97077 - José Silva

1 Optimizations

The following optimizations were implemented:

- Since in matrix multiplications the second matrix is usually accessed by column, that is, not in contiguous memory positions, the matrix ***R*** is kept **transposed** (as matrix ***RT***) in memory in order to be accessed by line. This allows to **reduce cache misses**, by taking advantage of the **principle of cache locality**.

- Matrix ***A*** can be very **sparse** so instead of keeping *nUsers * nItems* in memory, only the positions of non-zero elements and their respective values are kept. This helps saving memory and time when accessing matrix ***A***.

- When multiplying matrix ***L*** by matrix ***R*** to obtain matrix ***B***, in every iteration of the loop, only the lines on matrix ***L*** and columns on matrix ***R*** (lines on ***RT***) respecting to every non-zero position of ***A*** are multiplied, since those will be the positions needed for matrix ***B*** to compute the squared errors.

2 Approach for Parallelization

In the serial version of the program, in every iteration of the loop matrices ***L*** and ***R*** are multiplied as explained in the previous point and only after that the update is done.

Taking into account that creating and destroying threads multiple times can create an overhead delaying the execution of the program, a single parallel section is created before the iterations start, only ending after all iterations are completed. Inside the update function, two approaches were considered:

- Using a *#pragma omp atomic* when writing on the matrices ***L*** and ***R***, eliminating the data race to these memory positions. This raises a problem, since many threads could be trying to access the same positions of the matrixes, making it an overhead in the execution.

- Having an array of matrices with size *numThreads* for each of the matrixes ***L*** and ***R***, where each thread only accesses its part of the array (i.e. thread 0 would only access the matrix in position 0). This eliminates the data race when updating the private matrices, but leaves the aggregation of all the values calculated by each thread to be later joined together in ***L*** and ***R***.

Although the first option uses less memory and performed a bit better on smaller instances (benefiting from the fact that the non-zero values are ordered by line, which minimizes the collisions when writing on matrix ***L***), the second option was the one adopted for its better scalability power.

3 Synchronization Concerns

One of the synchronization concerns was that no thread could start iteration *n+1* of the loop before all the others had finished iteration *n*. This is guaranteed by the clause *#pragma omp for* at the end of the update function, since it has an implicit barrier. Another concern was that no thread would start updating the matrixes before the positions needed of matrix ***B*** were all calculated. Once again, this is guaranteed by the implicit barrier of the *#pragma omp for* in the function that calculates ***B*** in every iteration of the loop. When no synchronization was needed, like between randomly filling matrix ***L*** and ***R***, the clause *nowait* was used.

4 Decomposition and Load Balancing

Since only the non-zero values of the matrix ***A*** were stored, when parallelizing the loop iterating over them, each thread gets around the same amount of non-zero positions to work with. This leads to an even distribution of workload. This way, no need was found to change the scheduling algorithm. In the case matrix ***A*** would be stored with all its rows and columns, there could be certain cases where some rows would have more non-zero values than others, leading to a worse distribution of work. In that case, it would make sense to use dynamic or guided scheduling.

5 Performance Results

All the tests were performed in a laptop with an Intel Core i7 6700HQ CPU @ 2.60GHz running Windows 10 with 4 cores and 8 logical processors, running Windows 10.

Instance	Time (s)				Speedup (for 8 threads)
	1 thread	2 threads	4 threads	8 threads	
inst500-500-10-2-10	108.73	58.58	39.98	38.51	2.82
inst600-10000-10-40-400	189.96	100.42	76.6	71.42	2.66
inst1000-1000-100-2-30	35.74	19.24	13.96	14.49	2.47
instML100k	185.26	99.51	69.33	53.46	3.46
instML1M	228.94	114.91	73.98	53.25	4.3

Both the results and the speedups obtained match the expectations. It would be unreal to expect a speedup of 8, when using 8 threads. We can also see that the increasing speedup confirms the scalability of our solution, since the overheads of creating and killing threads gets less relevant as the instance size increases.