Instituto Superior Técnico

Parallel and Distributed Computing

2nd Semester 2019/20

Recommender System

Catarina Amaro
nº 86938

Carlos Marques
nº 81323

Ricardo Espadinha
nº 84178

Group: 03

April 3, 2020

1 Introduction

This report intends to present our approach taken in the implementation of the PDC Project. Here you can find the general idea behind the way we structured our serial code, our strategy for the parallelized version, of the same code, using OpenMP and the comparison between the two.

2 Algorithm

The base algorithm consists in a simplified version of a recommendation system similar to the ones presented in many services nowadays. The approach is to match the previous activity of different users, and then suggest items that users with similar profiles have selected and given a high evaluation.

The algorithm was implemented as described in the Project Assignment with a little adjustment for the matrix A and B:

- The Matrix A, described in the project assignment, refers to a matrix where each entry corresponds to a User's rating for a given Item (matrix indexes). As specified in the assignment, for a large number of items and/or users, the majority of the matrix's entries will typically be 0 (not evaluated), which corresponds to a sparse matrix. Allocating this matrix, under normal conditions, not only would cause the program to allocate a lot of unnecessary memory, but it would also affect it's performance, since every time the program goes through all the entries, it would always be necessary to check whether the entry in question was valid or not. In our case, as we needed to run through all the items for each user and vice-versa, we used the "List of Lists" format. This consists of two arrays (whose indexes corresponds to the users and to the items) of pointers to structs that we called *entry*. The struct *entry* holds the information of the corresponding User / Item, its rating and two pointers to the next item evaluated by the same user and the next user who evaluated the same item. Image 1 is a schematic of this implementation (report attachment).

- Regarding matrix B, the algorithm only accesses the same non-zero entries in A, so there's only the need to update and store those entries, except for the final solution. Therefore, we also store the corresponding value of the entry of B (updated with each iteration of the matrix approximation) in the struct *entry*, described above. In the final solution, the program computes all the values of B into a normal matrix. It also sets the values of the entries that are already evaluated in A to zero, in order for them to not be considered for the final recommendation.

The result of both the serial and parallelized versions of our code are in agreement with all the output files provided by the teacher for comparison.

3 Parallel Implementation

3.1 Approach

The algorithm is based in an iterative method to reduce the sum of the squared errors between A and B, meaning that each iteration is dependent on the iteration before. For that reason a parallel region is initialized in each iteration to do the required calculations and waits for every calculation to be terminated before it continues. For each iteration there are 3 major time sinks that were parallelized, the calculation of L, R and the calculation of the elements of B present in A. These "for loops" were parallelized using the OpenMP directive *#pragma omp for*. The clause *firstprivate* was used for the variables **deriv** and **A_aux1** in the calculation of L and R because each thread will compute a column/line of L/R resetting the value of **deriv** to zero at the end of each element calculation and using **A_aux1** to run through the user/item lists. The clause *private* was used in the case of the calculation of B because the initialization was not required and each thread needed an auxiliary variable to run through the given list of user evaluations assigned to that thread. In the end of the method, all elements of B were calculated using *#pragma omp for* to parallelize it. Finally we used the same directive to parellelize the calculation of the item with highest rate for each user, assigning to each thread a specific user of B, going through the entries of that user, setting their B value to zero and then picking the highest value present in that line of B. The clause *private* was used to give to each thread their own copy of **sol_aux** and **A_aux1** to help go through the given list and matrix. The other clauses will be addressed in the following sections.

3.2 Decomposition

Considering that each iteration is given the previous iteration's L, R and lists of entries with the A value (static) and the updated B value, each thread will be assigned a line/column of L(t+1)/R(t+1) at a time and because there is no interdependence between elements of the matrices L and R nor dependency of L(t+1) and R(t+1), this can be done without extra precautions. This approach can be seen as *Input Data Decomposition* in the Foster method. Given the output of each iteration is the updated matrices L, R and B values, that are independent for each thread, we can call this decomposition *Output Data Decomposition.*

3.3 Synchronization

As stated before the calculation of L(t+1) and R(t+1) are independent from each other therefore we use the clause *nowait* in the first *#pragma omp for* so when a thread is finished with the calculations of L(t+1) it does not wait for the other threads to complete and goes on to start computing values of R(t+1). At the end of the calculation of R(t+1) there is an implicit barrier because the clause *nowait* is not used meaning all values of L(t+1) and R(t+1) are calculated before the matrices are updated and multiplication of B present in the entries takes place. There is also an implicit barrier at the end of the multiplication of the given B values which makes sure the new B values are calculated before the start of the next iteration.

3.4 Load Balancing

In the calculation of L(t+1) and R(t+1) the "for loops" are given the clause *schedule* with the argument *dynamic* because each thread calculates a given user in the case of L(t+1) and a given item in the case of R(t+1) and the given entries are not balanced with some users/items having more ratings than others. With the *dynamic* argument each thread is assigned a user/item and when it finishes it's calculation it requests the OpenMP runtime for a new user/item.

This logic also applies for the calculation of the B values present in the given entries and in the calculation of the item with highest rate for each user. In the case of the calculation of the entire B matrix and in the *malloc's*, because the workload is even between threads, we do not specify a *schedule* clause which defaults to *static* that divides the work evenly between the threads.

3.5 Performance

The Table 2 presented in the attachment only considers cases with more iterations and more data, because of the overhead. It only makes sense to parallelize an algorithm when it computes big data sets.

Given that the percentage of parallelized code went from 95,75% and 99,97%, for the instances inst30-40-10-2-10.in and instML100k.in respectively, according to Amdahl's law on equation 1 and considering $f = 0.0425$, $f = 0.0003$ and $p = 4$, we get a maximum *speed-up* of $S(f, p) = 3,548$ and $S(f, p) = 3,996$, again respectively.

$$S(p, f) = \frac{T_{serial}}{T_{parallel}} = \frac{1}{f + \frac{1-f}{p}} \tag{1}$$

On the Table 2 we can compare the theoretical and observed values for the program's *speed-up.* The difference between the two of them is due to the times of creating and destroying threads, allocation of workloads and the use of dynamic scheduling. This type of scheduling has an associated overhead related to the time spent by the threads requesting new tasks from OpenMP.

4 Conclusion

As observed, the *speed-ups* were relatively high, taking into account the fact that the nature of the algorithm doesn't allow a total parallelization. This happens because each matrix factorization iteration must be computed sequentially, therefore there is a big dependency on the amount of the data taken into consideration.

For small instances the time of the parallelization overcame the time of the actual algorithm causing an *overhead* of nearly 55% where on the larger ones the parallelization time had a much lower impact, having around 10% of overhead.

5 Attachment

Table 1: Performance of the algorithm for Serial and Parallel with different threads (performed on the lab computers)

Instance Number	Serial [s]	1 Thread [s]	2 Thread [s]	3 Thread [s]	4 Thread [s]
$inst0$	0.009	0.026	0.025	0.031	0.077
$inst1$	0.065	0.253	0.346	0.406	0.431
$inst2$	0.063	0.161	0.224	0.239	0.233
$inst30-40-10-2-10$	0.525	0.646	0.494	0.385	0.330
$inst1000-1000-100-2-30$	27.394	32.219	15.080	10.107	7.611
$inst200-10000-50-100-300$	41.707	42.344	24.042	16.116	12.538
$inst400-50000-30-200-500$	86.456	93.818	51.887	35.280	27.601
$inst500-500-20-2-100$	87.027	105.386	44.246	29.105	22.096
$inst600-10000-10-40-400$	123.057	127.434	68.434	46.615	37.657
$instML100k$	163.833	184.572	84.407	57.309	42.964
$instML1M$	354.294	380.040	184.020	121.205	91.408
$inst50000-5000-100-2-5$	376.268	416.925	209.837	142.102	107.780

Table 2: Comparison of *speed-up*, theoretical *speed-up* and efficiency

Instance	Thread 2			Thread 3			Thread 4		
Number	Speedup	Max Speedup	Efficiency	Speedup	Max Speedup	Efficiency	Speedup	Max Speedup	Efficiency
$inst0$	–	–	–	–	–	–	–	–	–
$inst1$	–	–	–	–	–	–	–	–	–
$inst2$	–	–	–	–	–	–	–	–	–
$inst30-40-10-2-10$	–	–	–	–	–	–	–	–	–
$inst1000-1000-100-2-30$	1.817	1.998	90.94%	2.710	2.993	90.54%	3.599	3.986	90.29%
$inst200-10000-50-100-300$	1.735	1.997	86.88%	2.588	2.991	86.53%	3.326	3.982	83.53%
$inst400-50000-30-200-500$	1.666	1.998	83.38%	2.451	2.993	81.89%	3.132	3.986	78.58%
$inst500-500-20-2-100$	1.967	1.998	98.45%	2.990	2.995	99.83%	3.938	3.990	98.70%
$inst600-10000-10-40-400$	1.798	1.999	89.95%	2.640	2.996	88.12%	3.268	3.993	81.84%
$instML100k$	1.941	1.999	97.10%	2.859	2.998	95.36%	3.813	3.996	95.42%
$instML1M$	1.925	2.000	96.25%	2.923	2.999	97.47%	3.876	3.999	96.92%
$inst50000-5000-100-2-5$	1.793	1.999	89.69%	2.648	2.998	88.33%	3.491	3.995	87.38%

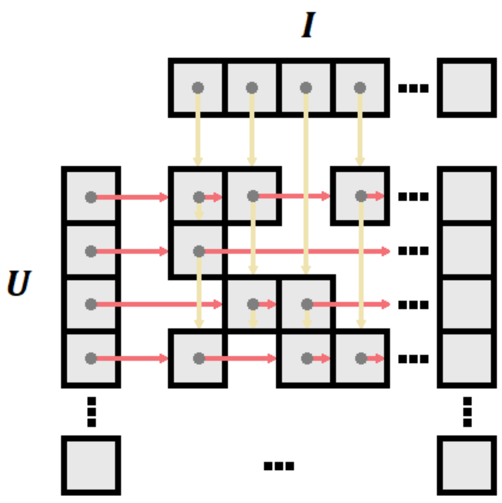

Figure 1: Schematics of the implementation of Sparse Matrix A ("List of Lists Format")