Recommender System Project Report – MPI

Parallel and Distributed Computing - 2019/2020

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

1 Approach for Parallelization

Our approach for this delivery was to split the matrices **by line** so that each processor would only have the lines from the *L* and *A* matrices it needed to multiply and calculate the matrix ***B*** in each iteration. For that purpose, we start by **splitting the non-zero positions of the matrix *A* by lines** and assigning a certain number of lines per processor, right away when processor 0 starts reading the input. After that, processor 0 also randomizes matrix ***L*** and sends its lines to the respective processor as soon as they are calculated, so it doesn't need to have the complete matrix ***L*** in memory. For the calculation, each processor keeps a copy of the full matrix ***R*** (and ***Rsum*****)**, but only updates the positions referenced by its non-zero elements. After that, an ***Allreduce*** is done so that all processors get the new values they need to calculate their new ***R*** matrix. Basically everything is done locally, the only communication done in each iteration of the loop is the sending of the matrix ***Rsum***.

Finally, each processor calculates its recommendations from its final matrix ***B***, sends them to processor 0 and this one prints the results.

2 Synchronization Concerns

The primary synchronization issues were when each processor was about to receive its data but the receive operation is blocking so no thread would start without receiving its data from processor 0.

Since we divided the problem using row wise decomposition we only needed to synchronize our transposed matrix ***R*** between each iteration. This was necessary because to calculate the new values each processor was responsible for entire rows and no other processor would need those values, but in the case of the matrix ***R*** it is different as we need all the columns in all the processors. To achieve this synchronization problem we used an ***Allreduce*** operation which reduces the entire matrix and stores the new calculated values in each processor.

(*After all the computations are done, to keep a correct output, processor 0 waits for each processor synchronously by order, meaning that he would print its results, then the results for the processor 1 and so on and so forth. This was done iteratively as we needed to print all the information by order otherwise the output would be wrong.*)

3 Decomposition and Load Balancing

As previously said, the non-zero values of matrix ***A*** were split by line amongst all processors, meaning each processor is responsible for *nUsers/p* lines of matrix ***A*** and consequently all the non-zero values belonging to them. This leads to each processor only keeping *nUsers/p* lines of matrix ***L*** (and their respective non-zero values of matrix ***A***) in memory, while still having to fully store matrix ***R***.

Since the non-zero values may not be evenly distributed between lines, some processors may end up with some more workload than others. Because of this, another solution to this problem was considered: dividing the non-zero values equally across all processors. This would raise another problem, since more than one processor could end up sharing the non-zero values of a single line, making it much harder to distribute matrix ***L*** between processors. For this reason, the first solution was the one adopted.

4 Performance Results

All the tests were performed in the lab machines running on the RNL's Cluster.

Instance	**Time (s)**						
	1 procs	**2 procs**	**4 procs**	**8 procs**	**16 procs**	**32 procs**	**64 procs**
inst500-500-10-2-10	84.77	46.75	23.38	23.22	31.66	37.70	37.27
inst1000-1000-100-2-30	26.64	14.90	8.23	17.70	14.64	17.28	18.97
inst50000-5000-100-2-5	-	148.80	72.38	77.35	58.16	57.55	57.67
instML100k	152.44	84.87	46.42	39.61	30.83	34.28	41.71
instML1M	171.40	95.02	49.68	30.41	21.93	17.23	9.76

Instance	Speedup // **Efficiency**					
	2 procs	**4 procs**	**8 procs**	**16 procs**	**32 procs**	**64 procs**
inst500-500-10-2-10	1.81//**0.91**	3.63//**0.91**	3.65//**0.46**	2.68//**0.17**	2.25//**0.07**	2.27//**0.04**
inst1000-1000-100-2-30	1.80//**0.90**	3.24//**0.81**	1.51//0.19	1.82//**0.11**	1.54//**0.05**	1.40//**0.02**
inst50000-5000-100-2-5	-	-	-	-	-	-
instML100k	1.80//**0.90**	3.28//**0.82**	3.85//**0.48**	4.94//**0.30**	4.45//**0.13**	3.65//**0.06**
instML1M	1.80//**0.90**	2.85//**0.71**	5.63//**0.70**	7.81//**0.49**	9.94//**0.31**	17.56//**0.27**

Regarding the instance inst50000-5000-100-2-5, the test for 1 processor runs out of memory due to the fact that the solution needs to save all matrices in memory. Therefore, we didn't calculate the speedup for that instance.

In most instances, as the number of processor *p* increases and the size of the instance stays constant, a certain point is reached where even though the speedup continues increasing on some instances, the level of efficiency cannot be maintained and decreases drastically (isoefficiency relation). This happens because the overhead of communication between the processors surpasses the benefits of parallelizing the computation.

The results regarding the even bigger instances were not presented since memory requirements were too big due to, according to our solution, all processors storing the complete matrix ***R***.