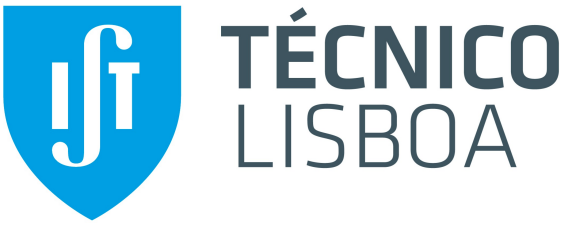

Instituto Superior Técnico

2019/2020, 2nd Semestre

Parallel and Distributed Computing

Recommmender System

Grupo 12:	*Number*
Gonçalo Valentim	84061
André Godinho	84006
Pedro Barbosa	86495

20/05/2020

1 Introduction

The second part of this assignment is to take the serial version of the program implemented in the first part and parallelize it using Message Passing Protocol (MPI). MPI is a message-passing protocol that facilitates communication between processes and is mainly used in distributed memory implementations, this approach differs from the OpenMP because OpenMP uses a shared memory system. The distributed memory implementation comes with the advantage of having better scalability and the disadvantage of requiring a lot of communication between processes.

2 Distributed Implementation

A distributed memory implementation enables the fragmentation of the problem amongst machines to reduce the amount of resources necessary for computation. In the recommender system the size of the problem depends on the number of non-zero entries of the input matrix and the total number of users, items and features, this is the matrices L and R and the structure used to store the non-zero entries of A, described in the first part of the project.

2.1 Decomposition

Regarding decomposition, since the matrix A is sparse, the best approach is to create partitions such that all the non zero entries are divided amongst all processes. To accomplish this, each non-zero entry is marked with it's respective process when it is being stored. To avoid replicating data, when the processes are being assigned it's assured that no two or more processes can have non-zeros from the same user.

In terms of memory allocation, the matrix L is line-split among all processes according to the non-zero elements each process has, assuring there is no data duplication. Furthermore, the array of non-zeros that was stored and marked in the master process is partitioned and split between all processes, where each process receives it's marked entries. After that, the entries not assigned to the master processes are freed to liberate resources for the matrices L and R not yet allocated.

Unfortunately, the matrix R is replicated among all process which presents complications when the number of items becomes too big, this could be solved by dividing the matrix R the same way L was divided but since instead of creating the matrix B using a normal multiplication of matrices, only the needed elements of B are calculated using the dot product of the respective row and column, the matrix R would still end up replicated among processes and this way excess computation is avoided.

2.2 Synchronization Concerns

The main concern regarding synchronization is to ensure that every process has the data it needs when it needs it and avoiding deadlocks. To increase performance the amount of communication between processes was reduced to the minimum necessary. For the processes to be able to do calculations independently, the data needs to be sent to all processes beforehand by the main process, this was secured by having the main process send all the data asynchronously and each process receiving it synchronously and also, before deleting all the excess data the main process makes sure all the other processes have received it to avoid deadlocks on the receiving end.

2.3 Load Balancing

The approach taken to balance the workload between processes was to try to equally split the amount of data between all tasks by dividing the total number of non-zero entries of A by the total number of processes and also splitting the rows of the L matrix equally between them. Achieving this goal and avoiding data replication was not possible because to ensure that there were no rows of the input matrix split between processes a slight unbalance was created. This unbalance is always dependent on the number of zeros in each line of the entry matrix but for the general case is not much significant.

3 Performance Analysis

To analyse the performance the recommender system was ran for multiple datasets and for various numbers of processes ranging, the results can be found on tables 1 and 2. From the MPI times it's noticeable that the execution time increases with the size of the input matrix as was expected, also, the number of items in the matrix has more impact in the performance since the matrix R is computed using the items and replicated among all processes.

The cost related to the communication between processes has more impact when the processes are running in different computers which is implied by the increase in execution time when the number of processes goes from 4 to 8. Also, in the inst400-50000-30-200-500 since the number of items is significantly bigger than the number of users the matrix R is big and the cost to send it to the other processes is not worth given the complexity of the problem which is why the time increases when more computers are included in the computation. On the other hand, for instances with a high amount of users the performance significantly increases when the number of processes is increased.

The two last instances are instances that can't be stored in a single computer because each one only has 8GB of RAM and the data needs over 10GB, for that reason, the program can't be ran with 4 processes or less. In this implementation, the data is distributed between the processes by spliting both L and the non.zeros of A which allows it to run the program for the bigger instances but, since the matrix R is allocated for all processes, the instance containing 1 million items can't be run as the matrix is too large to be stored in memory. To overcome this, a new implementation where the R matrix is also split between all processes should be considered.

However, for the hybrid version we were able to run with four processes because we managed to put all the different processes in four diferent machines.

Tabela 1: Execution time for MPI vesion.

Instance	MPI version						
	# processes						
	1	2	4	8	16	32	64
inst1000-1000-100-2-30.in	0m21.7s	0m12.3s	0m8.4s	0m25.1s	0m15.8s	0m16.3s	0m20.2s
inst400-50000-30-200-500.int	0m44.4s	0m29.5s	0m27s	1m56s	2m13s	2m29s	2m41s
inst-20000-10000-40-2-50.in	3m51s	3m1s	1m51s	1m55s	1m49s	1m47s	1m9s
inst50000-5000-100-2-5.in	3m15s	1m45s	1m6s	1m39s	1m3s	1m1s	1m6s
inst60000-2000-200-10-20.in	4m34s	2m36s	1m29s	0m55s	1m4s	0m31s	0m29s
instML100k.in	1m55s	0m50s	0m29s	0m31s	0m28s	0m27s	0m38s
instML1M.in	2m11s	1m8s	0m36s	0m25s	0m19s	0m15s	0m11s
inst1e6-100-700-1-3.in	x	x	x	4m36s	3m55s	3m11s	3m28s
inst1000-1e6-1000-1-3.in	x	x	x	x	x	x	x

Tabela 2: Execution time for MPI+OMP vesion.

Instance	MPI and OpenMP version						
	# processes						
	1	2	4	8	16	32	64
inst1000-1000-100-2-30.in	0m13.5s	0m15.7s	0m21.8s	0m28.6s	0m24.8s	1m41s	2m20s
inst400-50000-30-200-500.int	0m19s	1m12s	1m39s	´1m53s	2m08s	2m55s	3m11s
inst-20000-10000-40-2-50.in	2m35s	1m49s	1m40s	1m30s	1m29s	2m5s	1m41s
inst50000-5000-100-2-5.in	1m29s	1m14s	1m15s	1m21	0m59s	1m35s	6m58s
inst60000-2000-200-10-20.in	3m6s	1m18s	0m50s	0m34s	0m39s	0m19s	1m2s
instML100k.in	0m57s	0m41s	0m43s	0m34s	0m31s	1m03s	1m52s
instML1M.in	1m7s	0m40s	0m29s	0m20s	0m16s	1m15s	0m57s
inst1e6-100-700-1-3.in	x	x	2m57s	1m53s	1m45s	0m58s	0m54s
inst1000-1e6-1000-1-3.in	x	x	x	x	x	x	x