Recommender System

Phase 2 Project Report

Parallel and Distributed Computing

Group 42

May 20th, 2020

Instituto Superior Técnico

Introduction

As specified in the project statement, we modified our solution to support and take advantage of MPI. Among the implementation challenges, we had to support sending and receiving matrix blocks efficiently, minimize memory allocation and maximize useful computation between communications.

Implementation

We decided to divide the matrices L and R using row decomposition, so that matrices are sent and received in contiguous blocks of memory. To save on memory consumption, matrix B was represented as a vector with the coordinates and values of all entries in A, behaving as a sparse matrix until the full matrix needs to be calculated, at the end. Considering the calculations of ΔL, ΔR and entries of B during each iteration require columns of R or B, the two matrices are transposed during initialization, the latter being updated during each iteration. A custom MPI structure was created to send and receive entries of B and B^T.

Initialization

To improve performance during iterations, a lot of data is indexed at the beginning of the execution. The master thread calculates and sends the number of rows of L and R^T that should be processed by each thread, the entries in B and B^T belonging to the respective rows and the offsets of the data dealt in each thread. All threads participate in the calculations, including the master, getting an equal amount of rows to process. Vector B^T is obtained by copying B, sorting it by column and storing the indexes of the elements of B in B^T, so that future value updates don't require a new sorting operation.

Iteration

The iteration process is separated in two tasks, according to the data required: the calculations of ΔL and ΔR. At the beginning of an iteration, each thread calculates its attributed entries of B using a section of L and the complete R^T matrices. These values are sent asynchronously to the master thread, while the calculation of ΔL proceeds and eventually results in an updated L matrix section. The calculated sections are then merged into a single matrix on the master thread, who also updates B^T to prepare for the ΔR calculation.

The second task requires sending sections of R^T and B^T along with the full matrix L. Each thread only needs to update its section of R^T, leading to more evenly distributed workloads.

Output calculation

Finally, output calculation is also divided between threads. To accommodate the matrix B, auxiliar matrices used in the calculations of ΔL and ΔR as well as most indexing vectors are freed first. Each thread then allocates a section of B, calculates the product for all its entries, clears the coordinates present in the previous vector B, and sends a vector of results for the master thread to print sequentially.

Considered approaches

The solution we present is the result of improving over multiple iterations of its design. We started with a simple master-slave thread architecture that received the results at the end of each step, then merged operations with similar data requirements and added asynchronous data transfers, then redid all indexing structures from the previous delivery tailored for the MPI implementation. We believe we could have further improved the independence of threads, possibly communicating between each other instead of only with the master. However, this would add a big overhead on matrix slicing and merging, which is handled very efficiently by our solution.

Additionally, space allocation could be managed more dynamically. Our implementation allocates L and R^T matrices in full for every thread, despite only using sections of them depending on which matrix is being updated. We decided against allocating and freeing memory on every iteration to provide more stability and performance, especially considering the master thread is the one that allocates more memory and might exceed the hardware capabilities.

Results and Conclusion

		Number of threads						
		1	2	4	8	16	32	64
Tests	100ML-100k	2m49.8s	1m57.0s	1m10.9s	3m10.4s	11m06.8s	15m04.8s	17m51.3s
	600-10000	2m04.7s	1m15.2s	0m47.3s	5m5.6s	11m07.0s	22m18.7s	

The table shows the execution times in two large tests, *inst100ML-100k.in* and *inst600-10000-10-40-400.in*, by number of threads. The first test has a more dense matrix A that requires more operations per thread, but less data transfer than the second. Our solution performed well in both cases, better than both the sequential execution times provided and our own OpenMP results from the previous delivery, in all executions between 1 and 8 threads. However, from 16 to 64 threads, our solution suffers heavy time penalties from communications between different computers. We were also unable to run the largest tests provided, due to memory limitations. Overall, the results were still positive, and this project gave us a robust understanding of OpenMP and MPI parallelization.