Instituto Superior Técnico

2019/2020, 2nd Semestre

Parallel and Distributed Computing

Project - First Part

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

01/04/2020

1 Serial Implementation

The serial version consists of the following steps: read input data, allocate memory for the auxilliar matrices, initialize these matrices, apply matrix factorization, write output data and free allocated memory.

1.1 Structures

One of the main concerns was the fact that the matrix A is sparse. Thus, to optimize the matrix factorization it was stored an **array of** the **structure** *non zero* which has the row, column and value of each non-zero element of A. Also, since the matrix factorization also uses the values of the matrix B in the correspondent positions, these values were stored. With this **array of structure** it will be possible to access the non-zero positions directly, avoiding running the entire matrix.

1.2 Optimization

First of all, to decrease the number of cache misses during calculations (such as inner products, when calculating different B_{ij}) matrix R was transposed. Secondly, instead of copying matrices L and R to their stable copy, two auxilliar pointers were swapped among them before applying matrix factorization. Moreover, one pointer points to the matrix that will store the next iteration and another one that points to the current one. Thirdly, by using the **array of structure** and the matrix R being transposed it was possible to double loop during matrix factorization efficiently, accessing directly non zero positions and with less cache misses. Finally, we used the **array of structure** to generate output data without allocating memory for the matrices A and B.

2 Parallel Implementation

The approach used to parallelize the serial version was to identify all task performed by the serial version and determine the dependency between the task and the best form of decomposition. From the tasks performed by the serial version only the ones that allocate the memory to store the matrices for the calculation of the output can be executed in parallel but given the overhead, by managing the threads, and the low complexity of each task the trade off is not worth.

2.1 Decomposition

Even if the matrices can't be allocated at the same time, the fact that, in these matrices, each row is an array, presents the perfect conditions to use data decomposition to parallelize operations that cycle through the matrices while assuring cache coherence since all array positions are stored sequentially in memory. Not only the matrices and the structure that stores the input non-zero information, but also the A_{ij} and B_{ij} coefficients is an array too, and can be decomposed to parallelize the loops that iterate through it.

2.2 Synchronization Concerns

One of the main concerns regarding the synchronization was on the function Recalculate Matrix, where the program needed to assure that when accessing the variables L and R, no more than one thread would write on the same position in memory, the approach to this problem was using the **pragma atomic** .

2.3 Load Balancing

The primary concern regarding the load balancing is to equally distribute the workload between all threads and minimize the amount of overhead. The loops that go through the matrices L and R iterate by row first and by column second and, since the matrix R is transposed, the number of columns is the same for both matrices by parallelizing the loop in the outer loop of the cycles is possible to achieve equal load distribution by guaranteeing the all threads iterate over an array of the same size. As for the cycles that iterate over the non-zeros structure, in each iteration, each thread is given a row from a matrix to though and since the number of columns is the same so is the workload.

2.4 Performance Analysis

After running all the test and analyzing the results present in the table 1, there is a clear decrease in the run time when comparing the Serial Version to the Parallel Version as expected. In a perfect instance, the full parallelization of the serial version would take $\frac{Serial\ Time}{Number\ of\ threads}$ time to run, but since the parallel version is not 100% parallel and there is delay associated with the overhead of managing the threads, this perfect time was not to be expected, only a decrease in the execution time.

Tabela 1: Execution times.

Instance	Serial Version	Parallel Version			
		1 thread	2 threads	4 threads	8 threads
inst30-40-10-2-10	506ms	859ms	557ms	398ms	1115ms
inst500-500-20-2-100	1m00s	1m59s	0m57s	0m29s	0m35s
inst1000-1000-100-2-30	0m18s	0m34s	0m17s	0m09s	0m10s
inst200-10000-50-100-300	0m30s	1m16s	0m24s	0m12s	0m14s
inst400-50000-30-200-500	0m47s	1m17	0m32s	0m17s	0m18s
inst600-10000-10-40-400	1m46s	2m35s	1m24s	0m44s	0m49s
inst50000-5000-100-2-5	2m59s	4m38s	2m31s	1m25s	1m33s
instML1M	2m05s	4m11s	1m52s	1m00s	1m01s
instML100k	1m47s	3m11s	1m38s	0m50s	0m55s