LEARNING WITH MASSIVE DATA
Multi-threaded implementation of PageRank

Serena Zanon 887050

Data Structures

To solve the Page Rank problem, two data structures were chosen:

- Page Rank Matrix Data Structure [PRM]
  The Page Rank Matrix is a squared matrix whose dimension depends on the number of nodes composing the graph. In particular, the value of each cell is assigned based on the following rules:

$$M[i,j] = \begin{cases} \frac{1}{o(j)} \text{ if } j \to i \\ \frac{1}{n} \text{ if o(j) = 0} \\ 0 \text{ otherwise} \end{cases}$$

  Where by *o(j)* is intended the number of outgoing links and by *n* is intended the number of nodes of the graph.
- Column Wise Data Structure [CW]
  This data structure is given by a vector of vectors of integers.
  - The external vector contains as much elements as the number of the nodes of the given graph: the i^{th} element corresponds to the i^{th} node.
  - The i^{th} internal vector contains as much elements as the number of its outgoing edges and these elements are represented by the IDs of the nodes to which the links of the i^{th} node arrive.

  The advantage of this data structure is that it does not save any zero-entry.

Algorithms

Since two data structures were created, two versions of the algorithm were implemented. Before explaining how they work, let's identify what they have in common:

- They don't return anything and both accept in input the exact same number of parameters: the data structure that describes the given graph, two vectors which respectively represent the vector needed for the computation and the result vector, two floats used to balance and guarantee convergence, the number of threads required to execute the function and a vector of double that will contain the execution time of the threads;
- In order to compile the parallel version, the library OpenMP was adopted: thanks to this library was possible to change the number of threads needed to compute the code and to capture the time required by each thread during the parallelization;
- Both algorithms are within a while loop that loops for 50 times. After each iteration the current result vector becomes the computational vector for the next execution.

The algorithm for the PRM computes the product between the matrix and the first given vector, saving the result in the second given vector. This product is done by two nested for loops over the dimension of the matrix. At the end of these loops, another for loop scan all the cells of the result vector in order to apply the two constant floats to guarantee convergence.
The algorithm for CW Data Structure follows the same idea of the first algorithm and the same value assignment. The core of the function is the first for loop:

- It scans all the elements of the external vector and checks if they contain an internal vector;
- If the i^{th} element contains a vector then a for loop over it is done. By considering that the reference index of this last loop is j, in order to compute the product the values $\frac{1}{i^{th}vector.size}$ and the i^{th} value of the computational vector are multiplied and their result is added to the value assumed by the j^{th} cell of the resulting vector;
- If the i^{th} element does not contain a vector, this means that the i^{th} node is a dead end. In this case, the product is between $\frac{1}{n}$ (where *n* is the number of nodes) and the i^{th} value of the computational vector is added to a support variable.

Finally, the last for loop touches all the result vector's cells and does two things: it adds the value contained in the support variable for the dead end nodes and applies the two constant floats to guarantee convergence.

Analysis

In the following tables, there are the construction time required by the Data Structures to be created and the space needed to store them. Almost all datasets were created artificially except the *small_sparse* which was taken by the given site (The real name is *p2p-Gnutella25*). The datasets' names are slightly modified for report space reasons.

Dataset	adHoc_025	adHoc_05	adHoc_075	adHoc_1
CT PRM	6 713 ms	13 572 ms	20 646 ms	22 740 ms
SR PRM	196 MB	196 MB	196 MB	196 MB
CT CW	6 588 ms	13 897 ms	20 387 ms	22 583 ms
SR CW	49 MB	98 MB	147 MB	196 MB

Dataset	small_sparse	small_dense	medium_sparse	medium_dense	big_sparse	big_dense
CT CW	41 ms	437 ms	429 ms	2 011 ms	1 377 ms	17 152 ms
SR CW	763 kB	3 MB	5 MB	12 MB	17 MB	76 MB

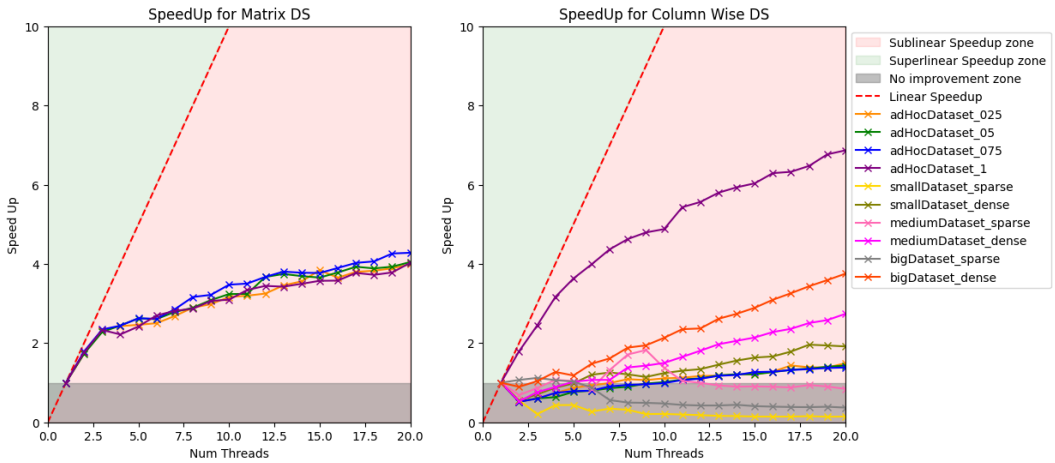
Where by *CT* is intended Construction Time and by *SR* is intended Space Required.
As it can be noticed, not all datasets were tested on both algorithms. The datasets that were tested on the PRM algorithm have all the same number of nodes with an increasing density. That's why the space required doesn't change but the time increases (just to mention, *adHoc_1* has density equal to 1 while *adHoc_025* has density equal to 0.25). Due to memory issues, it has been decided to not test any other dataset on this algorithm. On the other hand, the CW algorithm was tested with the just mentioned datasets and new six ones: these last are divided in *small*, *medium* and *big* based on the number of nodes and *sparse* and *dense* based on the number of edges. As it can be seen by the data inside the tables, the CW Data Structure is more efficient in space and time construction since it saves only the non-zero entries.

In the following tables, there are some information about the execution time of the threads during the computation. These times were only counted on the first for loop, since its body has more instructions than the last one.

Dataset	adHoc_025	adHoc_05	adHoc_075	adHoc_1
Min Gap Time PRM	24 ms	34 ms	21 ms	20 ms
Max Gap Time PRM	36 ms	58 ms	28 ms	33 ms
Min Gap Time CW	171 ms	327 ms	351 ms	160 ms
Max Gap Time CW	248 ms	496 ms	597 ms	231 ms

Dataset	small_sparse	small_dense	medium_sparse	medium_dense	big_sparse	big_dense
Min Gap Time CW	2 ms	6 ms	5 ms	16 ms	18 ms	148 ms
Max Gap Time CW	3 ms	7 ms	6 ms	18 ms	35 ms	222 ms

The *Min Gap Time PRM* and the *Max Gap Time PRM* represent respectively the minimum and the maximum time required by a thread to compute the PRM algorithm and the same applies to *Min Gap Time CW* and *Max Gap Time CW* on the CW algorithm. It can be noticed that the minimum and maximum duration of a thread fluctuates a lot. This could be caused by load imbalance between threads. A possible way to alleviate this problem could be assigning an amount of equal work among all threads. These data were retrieved thanks to this procedure: for each dataset and for each possible number of threads, the minimum and the maximum thread duration is identified thanks to OpenMP utilities. Another step was necessary: for each dataset, it has been identified the greatest gap among all the computations done with a different number of threads. To be clear, the duration of the first single thread was not taken into consideration since that specific computation is equivalent to the sequential computation.



As mentioned before, there is load imbalance. This can be noticed even with the different Speed Up achieved: all of them are in the *Sublinear Speed Up zone* or even in the *Zero Improvement zone*. This could be due to the fact that there is no information about when a thread will start. Let's notice that if the dataset is small then the Speed Up has a really bad trend, since the overhead caused by the creation of the thread might exceed the overhead caused by the actual computation.

DISCLAIMER: some datasets are not included in the project folder since they were too big to be uploaded on Moodle.