Università Ca' Foscari Venezia

ACS1 - Learning with massive datas

Brilli Matteo

Report - Assignment 1

The present report describes the genesis process, results, and shortcomings of my C++ implementation of the Page Rank algorithm, following what was requested by the assignment.

I began by storing the adjacency matrix of the graph under analysis using a CSC format [1]. With this choice it's possible to directly scan the edgelists which contains the informations about the graph, without any sort of costly preprocessing. Thus, my program simply scans the input file line by line, and keeps track, for each coloumn of the matrix, of the indices of the non-zero elements. Practically, I defined a matrix struct which stores the number of nodes (n) and edges (NNZ) of the graph plus four vectors:

- one with the indices of of the non-zero elements in each coloumn (length: NNZ);
- one which keeps track of the number of non-zero elements before each coloumn (length: n+1);
- one with the weights of each node, so the sum of the elements in the corresponding coloumn (length: n);
- one which registers the position of the null coloumns (variable length, n in the worst case).

This struct also implements a few methods to print the informations of the matrix and to access the elements.

In the sequential version, each page rank iteration initializes a vector of length n and then does the following:

1. calculates the contribution of the dangling ends, using the informations we have on the positions of the empty coloumns. This gives us a value that must then be added to every element of the output vector;
2. performs the matrix-vector multiplication. This is done scanning the matrix by coloumns (so, for example, if in the third coloumn only the second and fifth element are non-zero, each of them would be multiplied by the third element of the input vector, divided by two, and then added, respectively, to the third and fifth element of the output vector);
3. calculates the norm of the difference between the result and the input vector. As a stopping criteria, I decided to consider 'norm $\leq 1e-6$'.

I tested the program with various input files, making sure to compile with the flag -O3 for increased efficiency. The times reported in this analysis, however, will be the ones calculated using the biggest dataset I could find, that is "soc-LiveJournal1.txt", composed of 4847571 nodes and 68993773 edges. Over a hundred runs, the program takes on average ($12.0612 \pm$

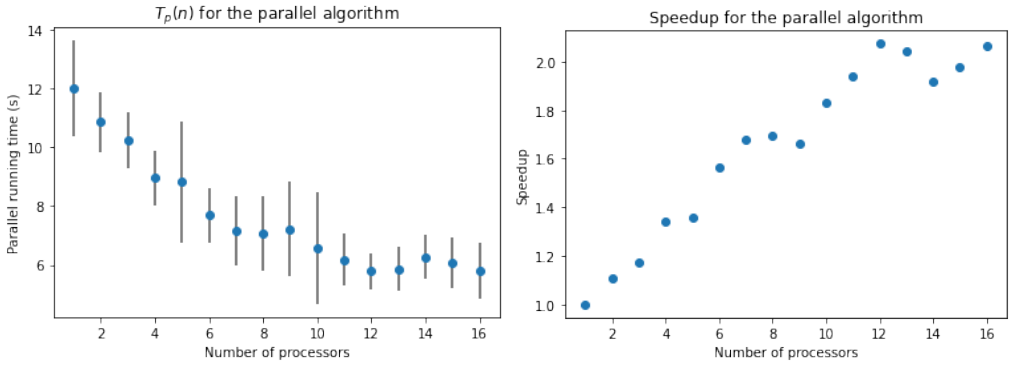
[1] https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_column_(CSC_or_CCS)

0.851823) s to execute the Page Rank algorithm. As a reference, this is already faster than simply reading the file, which takes around twenty seconds.

While trying to parallelize this program I encountered some issues. Due to the CSC format, simply splitting the for cycle that computes the matrix multiplication is not very efficient. It is infact analogous to splitting the matrix by coloumns, so there are race conditions on the output, since all threads may need to write on the same memory locations. Working on a machine with 16 processors I got the same running time independent of the number of processors p used, thus obtaining no speedup.

To solve this I decided to split the matrix by rows. This was impractical to do once the matrix was already initialized, so instead of one big $n \times n$ matrix, I now store p $m \times n$ smaller rectangular matrices, where m is determined so that the n rows are equally split among all p processors. Then, each processors can perform the calculations relative to one of these matrices.

The results of this choices are reported in the following figure. On the left we can see the times measured as a function of p, while on the right the speedup calculated with respect to the sequential algorithm (all times are averaged over 100 runs).



As we can see, the total speedup is sublinear, and not very significant. I think this can be explained through a couple of considerations.

First, the part I managed to parallelize is not the whole page rank algorithm, i.e. not all the three steps analized above, but only the second one, that is, the matrix-vector multiplication. This is because the contribute of the dangling ends can be simply calculated once at the beginning of the iteration and then added to every element of the output vector, while to calculate the norm we need the multiplication to be fully done. Still, given enough time, I think it could be possible to incorporate these two steps in the parallel computation, thus reducing even further the running time of the algorithm.

Secondly, load imbalancing needs to be taken into account. As an example, with the dataset I used, the busiest thread had 17653674 non-zero elements in the input, while the less busy had only 659121 (around 25 times less), and the others were pretty evenly spread between these two values. With time, this issue can probably be solved quite effectively. Infact, I know the number of non-zero elements in each coloumn, so it would be straightforward enough to, for example, divide each $m \times n$ matrix into blocks, each with the same number of non-zero elements.