IN3200
Exam 1

April 2020

1 Full matrix

The webgraphs are read from files line by line and the relevant lines are split to find the number of webpages (N), the number of total links (N_l) and the links themselves (ToNodeId FromNodeId). This is done in a similar way for the full matrix and the CRS, with minor differences.

For the matrix version (2d array) all elements are first set to be '0' ($\mathcal{O}(N^2)$), before the relevant elements are simply set to '1' as each line is read from the file ($\mathcal{O}(N_l)$). Note that it is also necessary to control that we have no self links (node1 != node2) and that both nodes always have a value smaller than the total number of web pages, i.e. we don't link to web pages not included in the graph. This is simply controlled with a three statement if-test.

In order to find the number of mutual links we loop through each row of the array and count the number of elements (e_i) on the row. The total number of pairs (p) is

$$p = \sum_{i=0}^{N} \frac{e_i(e_i - 1)}{2}, \tag{1}$$

and the number of pairs (p_e) for each element present on the row is

$$p_e = e_i - 1. \tag{2}$$

Thus, the total number of pairs an element is a part of is simply $\sum p_e$.

After counting the elements on the row and adding to the number of pairs, the number of pairs for each element present on the row is added to. This means that we need to keep track of both the total number of pairs and the number of pairs an element is a part of when parallelising (i.e. each element in the array num_ involvements). Additionally we need to keep the number of elements on a row as a private to each thread. Parallelising is done with OpenMP by including the line

```
#pragma omp parallel private(elements) reduction(+:pairs)
reduction(+:num_involvements[:N])
```

just before the for-loop begins.

Note that we need two separate nested for loops inside a common outer loop when dealing with a full matrix; one to count the number of elements and one to count the number of pairs each element on the row is part of.

2 Compressed row storage

For CRS the nodes are read from the file in a similar fashion, but stored as pairs (on the index) in separate arrays before the column and row indices are stored ($\mathcal{O}(N \times N_l)$). The number of links that are not self-links or illegal links are also counted. The cols and rows arrays are set in a nested for loop with a counter that is incremented for each element that is added to the column array. This counter will be the index inserted into the rows array after the nested for loop is ran through, and it is the index of the cols array.

As I mention below this solution is very slow on the Notre Dame set. An alternative, and probably faster approach is to first read the file once and increment the counter, and then reading the file again, using the counter to index the rows and cols arrays in the correct manner. I was a bit slow to test my code on the large set and realized this a bit too late to start changing my approach.

In particular the loop that fills in the rows and cols arrays. This is probably because in my solution we have to access several arrays and retrieve data for each iteration.

Counting the number of pairs and elements is very simple with CRS since the number of elements on row i is simply the difference between element i+1 and i of the row_ptr array. Thus, finding the number of pairs on the row and adding to the sum is simple.

However, it is still necessary to loop through the individual elements on each row to count their number of involvements.

OpenMP is included in the same manner as it was before (same private and reductions).

3 OpenMP and time

Implementing OpenMP in the way I have chosen allows the user to choose wether they want to run parallelised (by including -fopenmp when compiling) or not.

For some reason I was unable to make the clock() function to work on my computer, and it allways returns 0. I have made it work in weekly exercises before, but just can't get it to work here. Therefore I have been unable to produce timing of the various methods. And potential speedup (or slowdown) from implementing OpenMP. I realize that this is a huge drawback, but I will discuss a bit around what I would expect and what I have seen from others results.

On a large dataset I would expect a speedup from parallelization, but on smaller datasets the overhead from implementing OpenMP would outweigh the speedup. This is what I have noted from the results from others and the discussions on piazza as well. The datasets are simply too small. I might expect some speedup when counting the number of mutual links in the Notre Dame set while using the full matrix version, but the matrix is too large so it can't be tested.

Even without being able to get time measurements I can see that my implementation of read_graph_from_file2 is quite slow.

4 Top webpages

The n top webpages are found by finding the index of the n largest values in the num_involvements array. To do this we need to keep track of the previously largest element and wether or not it is already represented in an array containing the indices of large elements. This means that we need a triply nested loop. One per n elements we want to find, one per web page and one to check if the element already has been found.

Note: I made a last minute change to the code to account for the fact that we need to remove mutual links and links between webpages out of the scope of the webgraph. This change unfortunately caused a segmentation fault when I run it on the Notredame set and counts an extremely wrong number for the number of pairs present for webpage 8 in the example file from the exercise set (when parallelised) even though the total number of pairs and the count for the other pairs is correct. The rows and cols arrays look the way they should as well. I've been trying to get the code back to the way it was, but for whatever reason I can not do it (although it looks the same to me).