# Report for numerical problem

## Description of functions

### Global variables

All three functions in returns void. The data needed created by a function and are needed by another function are stored as global variables. The parameters to each function are the inputs variables that the user sends when starting the program.

### void read_graph_from_file( char filename[] )

This function loads the file into an array. First, second and fourth line of the file are thrown away. The node and edge count in line three are stored in the global variables called nodes and edges. The rest of the file are loaded into a 2D-array while the two arrays called row_nodes_occurrence and column_nodes_occurrence that has the length of nodes that counts the number of elements in each row and nodes. The 2D-array are called CCS and have the dimensions nodes*3. The first and second column record fromnode and endnode. Third column record how many occurrences there has been in the current row so far.

Once the file is loaded into the 2d-array the code will traverse through the array and transfer the data into a condensed row storage. The variables names are row_ptr, col_idx and values.

After this the function will create a global variable called dwp that contain all the dangling webpages. This is done by transferring all the indices in column_nodes_occurrence that have the value zero to a value in dwp. Dwp_size is the the length of the dwp array.

### void PageRank_iterations(double d, double e)

This function starts with declaring variables and arrays, some of these are initialized. The function will then enter the parallel region that will start with identifying the number of threads assigned and the number and initiate arrays based on this number. $X^0$ will also be initiated here. Each thread will then enter a while-loop that they will only exit when all the values of $x^k$-$x^{k-1}$ are lower than the convergence threshold value.

The iterative procedure will happen inside the while-loop. It will start by using a pragma omp single where $x^k$ will become $x^{k-1}$ and the sum of all dangling websites will be computed. After that the function will enter a pragma omp for where the matrix multiplication and addition will happen. The workload will be divided equally between all the threads. At the end of the while-loop there will be pragma omp single where the max difference between $x^k$ and $x^{k-1}$ will be found.

### void top_n_webpages(int n)

After declaring variables, the function will enter parallel region where each thread will find top n webpages. Each thread will have their own two arrays of length n that contains the index of a value and the value itself. There will also be a pointer that points to the lowest value in the array. The threads will only traverse the final $x^k$ only once and when the thread comes across a value that is higher than the lowest value in the top n array it will replace the lowest value with the new value.

Then it will search the top n array for the new lowest value and continue traversing the array when this is done.

## Hardware

I used a school computer with Linux to test the code.
The CPU name is intel core i5-6500 CPU, 3,2 GHz. This CPU has four cores and each core has one thread. It also has 8GB of memory.
The gcc version is 4.8.5.

## Time measurements

To measure the time the iteration takes I have chosen to use an Openmp function called omp_get_wtime, this also means that stopwatch start after the threads have been created. All the threads get their time measured, but since the first five decimals are always the same, I will assume they take the same amount of time.
The damping is set to 0,99 and epsilon are set to $10^{-7}$.
The time measurement is the average of five measurements.

| Cores | Time | Improvement |
|---|---|---|
| 1 | 4,684 | 0% |
| 2 | 2,860 | 39% |
| 3 | 2,280 | 52% |
| 4 | 1,900 | 60% |

## Discussion

The maximum achievable speedup due to parallelization would be T/N where T is time and N are the number of cores. The maximum speed in this instance would be 1,171 seconds. This is not possible because of some reasons. The first reason is that not all the code inside the parallel region are run in parallel. There is serialized code in the beginning and the end of the parallel region. This is code that can only be done once in each iteration such as swapping the pointers of $x^k$ and $x^{k-1}$.

Another reason is that the parallel code has barriers or implicit barriers that will turn all threads except the slowest idle until the slowest thread can catch up. This will lead to a slower code and one should always have the lowest number of barriers as possible.

Other reasons for not achieving ideal computing speed would be overhead. Loops and if-sentences come with overhead that one can't get rid of. In addition, if an if-sentence are inside a loop it will cause branching that can cause the compiler optimization to fail.

The last reason that prevent ideal computing speed would be the bandwidth and latency between the ram and CPU. The bandwidth and latency can't keep up with the increased speed of the CPU. This means that the CPU has to wait for the ram to supply it with data that it can process.