IN3200 - PARTIAL EXAM
COMPUTING PAGERANK

ERIK JOHANNES HUSOM

JUNE 15, 2019

Contents

1. Introduction

The goal of this project was to implement the PageRank algorithm, which is a part of Google's search engine. Input is taken from a file containing a *web graph*, which has information about all the links (*edges*) between a set of webpages (*nodes*). This web graph is then used to calculate a score for each page, which indicate the importance of the page based on how many and which webpages that are linked to it, either by outbound or inbound links.

The main focus of the project was to produce effective code, partly by parallelizing the PageRank algorithm using OpenMP, which is an application programming interface for shared memory multiprocessing programming. I tested the code on three large web graphs from the Stanford Large Network Dataset Collection[1], which are called *web-NotreDame*, *web-Stanford* and *web-BerkStan*. The last of these web graphs is the largest one, with 685,230 nodes and 7,600,595 edges, and with such a large amount of data that needs to be processed there is also need for highly effective code.

This report contains information about how the program works, and an analysis of the time usage for different parameters and number of parallel threads.

2. Program structure

2.1. **Functions.** In this section I will explain the basic content of the code. The source code is split into two files. The file called `PE_main.c` contains the `main` function, that sets up the necessary arrays and variables and calles the functions that compute the PageRank score. This file also contains code to measure time usage for all the functions. Info about compilation and execution is found in the README-file of the project.

The file called `PE_functions.c` contains in total five functions, and the details of these are explained below. Some of the algorithms and loops are somewhat hard to explain in consice terms, but comments are also provided in the source code.

2.1.1. *`read_graph_from_file()`.* This function's task is to read all the data from a file, and then set up a *hyperlink matrix* based on the data. The matrix is of size $N \times N$, where N is the number of webpages in the data set, and the matrix elements are defined as

$$a_{ij} = \begin{cases} \frac{1}{L(j)}, & \text{if there's an inbound link from webpage no. } j, \ (i \neq j), \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

where $L(j)$ is the number of outbound links from webpage no. j. Self-linkage should not affect the PageRank score, so any links from a webpage to itself is removed. This results in a sparse matrix, where most elements are zero, and we use the Compressed Row Storage (CRS) format to store this matrix. This format consist of the following arrays:

- `val`: Contains the non-zero values of the matrix.
- `col_idx`: Contains the column index of the corresponding element of `val`.
- `row_ptr`: Contains the index where a new row starts in the array `val`.

This function starts with reading in the number of nodes and edges in the web graph, which is used to allocate arrays of the correct size. An example of the accepted format of the web graph is as follows:

```
# Directed graph (each unordered pair of nodes is saved once): 8-webpages.txt
# Just an example
# Nodes: 4 Edges: 7
# FromNodeId    ToNodeId
0       1
0       2
1       3
2       3
2       1
3       1
3       2
```

All the edges/links in the file are then read through, with an if-test to remove potential self-links. At the same time the program counts how many outbound and inbound links each node has. This can then be used to find `row_ptr`, since the inbound link count corresponds to how many values there are in each row in the matrix. The array `row_ptr` is then found by summing up this element count for each row.

Another necessary part of the PageRank algorithm (explained below) is to find any dangling webpages, that is webpages with no outbound links. This is simply done by examining the outbound link count, and see which page indices that has noe outbound links.

The most complex part of this function is finding the elements of `col_idx`. We cannot assume that the edges in the web graph are sorted in any way, so either we have to sort the data, or we have to use an algorithm that does not depend on sorting (or only minimally depend on sorting). Sorting arrays up to sizes of 7 million elements (in the case of the *web-BerkStan* data set) can be relatively expensive, and therefore we want to avoid that. The algorithm I came up with iterates through the edges as they are presented in the web graph, and because we already have produced the array `row_ptr`, we know the minimum and maximum index of where a column number is to be placed in `col_idx` based on what row number it has. In this loop the values of `col_idx` will be placed in the correct domain (i.e. the correct row), but not necessarily in

the correct order within that domain. This is then solved by sorting the column indeces within each domain. This algorithm is not easily explained in just a few lines, but may become clearer when looking directly at the source code and following the data flow. The minimal use of sorting ensures that even the hyperlink matrix of the *web-BerkStan* data set is set up in a few seconds (see accurate timing information in the next section).

The last step is finding `val`, which is easily done by using the outbound link count for each column.

In this function I have included some comment sections that are marked with `DEBUG`, where it is possible to get printouts of certain arrays. This is done to make debugging easy by checking the values that are produced by the code.

2.1.2. *`PageRank_iterations()`.* The goal of this function is to produce a vector $\mathbf{x}$ containing the PageRank scores of all the webpages. And initial guess

$$\mathbf{x}^0 = \frac{1}{N}\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix} \tag{2}$$

is used, N being the number of webpages, and then the following formula is used:

$$\mathbf{x}^k = \frac{(1-d+d\cdot W^{k-1})}{N}\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix} + d\cdot\mathbf{A}\mathbf{x}^{k-1}, \tag{3}$$

where $\mathbf{A}$ is the hyperlink matrix and d is a damping constant that usually is set to 0.85. W^{k-1} is the sum of the PageRank scores, from the previous iteration, of all the dangling webpages. Equation 3 is iterated through until the scores converge. The stopping criterion I used in my code is

$$\sum_{j=1}^{N}\left(|x_j^k - x_j^{k-1}|\right) < \epsilon, \tag{4}$$

where ϵ is a given threshold. The default threshold value in my code is set to 1×10^{-10}, because the difference between converged PageRank scores usually is of magnitude 1×10^{-4} (see example of results in the next section). With this convergence threshold the results should be of good enough numerical precision. The reason for choosing to sum up the differences between all elements from on iteration to the next, is that it is a simpler implentation (with minimal if-tests) compared to finding the maximum difference of all the elements in the array.

The PageRank equation is implented inside a while-loop that checks the stopping criterion for each iteration. In order to maximize the efficiency of this function I have also used OpenMP to parallelize the code. Number of threads has to be given as a command line argument. Loops with few iterations are not worth parallelizing, because OpenMP overhead leads to bad performance. There are usually several ways to parallelize a certain code block, and I chose to use a rather straight-forward method, by calling `#pragma omp parallel for` on the matrix-vector-multiplication (MVM) in the PageRank algorithm (and also when intializing the result vector). This lets OpenMP divide the labour automatically on the number of threads that has been assigned. The function also counts the number of iterations that are needed for the result to converge within the given threshold.

TABLE 3.1. Size of the data sets used in this project.

Data set	Number of nodes	Number of edges
web-NotreDame	325,729	1,497,134
web-Stanford	281,903	2,312,497
web-BerkStan	685,230	7,600,595

2.1.3. *`top_n_webpages()`.* This function takes (among other things) a number n as input argument, and provides a list of the n highest ranked webpages, together with their PageRank scores and their indeces. It is of course possible to do this by sorting the complete array of PageRank scores from highest to lowest, and then printing out the first n of those, but considering that the number n typically will be much smaller than the number of webpages (which for our test data sets was several hundred thousand), it is almost always unnecessary to sort the whole array. Therefore I wanted to avoid having to sort the large array of scores, because this is relatively expensive in terms of processing power. Instead, the function takes a copy of the PageRank score array, iterates through the array and finds the highest score, prints out the rank, index and score of that page, and then sets that element to zero. The algorithm is set up as follows:

```
n = number of top webpages to print;
max = 0;
for n webpages do
    for all elements in PageRank score array do
        if element is bigger than max then
            set max to value of element;
            store index of element;
        end
    end
    print the page index and score of the maximum element;
    set maximum element to zero;
    reset max to zero;
end
```

This is a quick and easy way of making sure that in the next iteration, the algorithm will find the second highest score, and at the same time avoiding potential problems with multiple pages having the same score (this is usually only a problem for very small data sets).

2.1.4. *`sort()` and `swap()`.* A part of the method used to set up the CRS matrix depended on a sorting algorithm, which consists of the functions `sort()` and `swap()`. These two functions were taken from the first exercise set of the subject IN3200 at the University of Oslo[2].

3. Results and time measurements

In this section I will present time measurements with different parameters. The hardware used is the following:

- Processor: Intel Core M-5Y71 CPU @ 1.20GHz × 4
- Memory: 8 GB DDR3L SDRAM @ 1600 MHz - PC3L-12800

The program was run on Ubuntu 18.04 with the C compiler `gcc`, version 7.3.0.

To better understand the performance differences, I have listed the size of the data sets in table 3.1.

3.1. **Time measurements of parallelized function.** The time measurements for the function `PageRank_iterations()` are shown in figure 3.1 for all three data sets. The data sets are processed with varying number of threads for the parallelized part of the code, and each execution is run 5 times to obtain an average. In the graph, all measurements are plotted to give an impression of the spread of the data, and the mean is plotted with a line and a cross. The difference is most notable with the *web-BerkStan* dataset, and the precise mean run time for this dataset are also presented in table 3.2.

For all the datasets, the most drastic improvement happens when going from 1 to 2 threads. For *web-BerkStan* and *web-Stanford*, the run time actually increases when going from 2 to 3 threads, and the same happens for *web-NotreDame* when going from 3 to 4 threads. It is possible that the performance is decreased by heat generation in the CPU, since the measurements were obtained by executing the program several consecutive times. Each test was only done 5 times, which also means that it could be coincidential that the run times happened to be slightly slower for a larger number of threads, even though one would expect at least some performance gain when increasing the thread number. Lastly, it is also possible that my implementation of the parallelized code, and the use of OpenMP, was not done in an optimal way. I chose a rather simple way to use OpenMP, and it is of course a possibility that mistakes in the source code leads to suboptimal performance.

In any case, we can see that for the largest data set, *web-BerkStan*, there is a clear trend of higher performance when increasing the number of threads. The run time is almost cut in half when going from 1 to 4 threads.

TABLE 3.2. Mean time measurements for the function `PageRank_iterations()` on the *web-BerkStan* dataset, with different number of threads. Measurements are averaged over 5 runs per thread number.

Number of threads	Mean run time [s]
1	8.4848562
2	5.0602186
3	5.1053918
4	4.4328932

3.2. **Time measurements of serial functions.** The time measurements results for the functions `read_file_from_graph()` and `top_n_webpages()` are presented in table 3.3. As we can see, the run times are pretty similar for *web-NotreDame* and *web-Stanford*. The former has a larger number of nodes, but the latter has a larger number of edges. The function `top_n_webpages()` iterates through an array that has one element per node in the data set, which means that this function should use longer time for the *web-NotreDame* data set, and this is indeed the case. The processing time of the function `read_file_from_graph()` depends more on the number of edges, which also explains why the run time is longer for the *web-Stanford* data set. The last data set, *web-BerkStan*, has a significantly larger amount of both edges and nodes, and therefore has the longest processing time for these two functions.

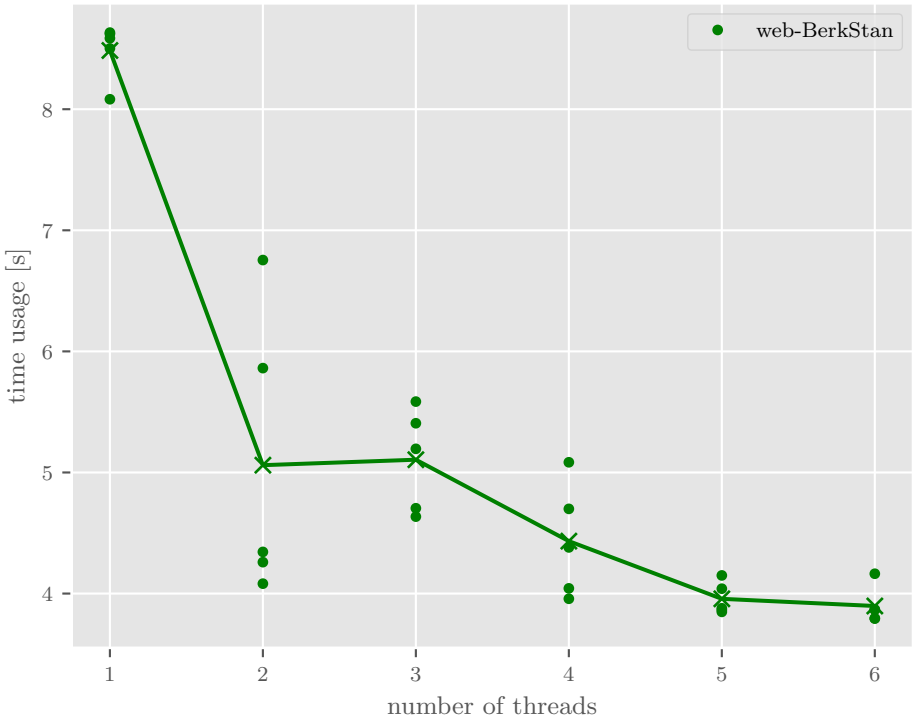

FIGURE 3.1. Run time for the three different data sets, for different number of threads. Measurements are represented as dots, and the mean as crosses with line.

TABLE 3.3. Time measurements for the two serial functions for three different data sets. The measurements are averaged over 5 runs per function per data set.

Data set	Run time `read_file_from_graph()` [s]	Run time `top_n_webpages()` [s]
web-NotreDame	0.40107	0.0081102
web-Stanford	0.67739	0.0069156
web-BerkStan	3.15396	0.0212098

3.3. **Results of `top_n_webpages()`.** For easy comparison with other programs, I have provided the results from my source code on the *web-NotreDame* data set, executed with the default parameters, in table 3.4.

TABLE 3.4. The 8 webpages with the highest calculated PageRank score, using the source code in this project on the *web-NotreDame* data set.

Rank	Page index	PageRank score
1.	1963	0.0056267674
2.	0	0.0054045118
3.	124802	0.0033262290
4.	12129	0.0028570560
5.	191267	0.0027487313
6.	32830	0.0027318341
7.	3451	0.0025899846
8.	83606	0.0024596915

REFERENCES

[1] Stanford University. Stanford large network dataset collection. `https://snap.stanford.edu/data/index.html#web`.

[2] University of Oslo. In3200 exercise set 1. `https://www.uio.no/studier/emner/matnat/ifi/IN3200/v19/teaching-material/exercises-in3200-week01.pdf`, 2019.