PageRank Implementation

Gianluca C.
Gaia O.
Roberta P.

Contents

1 Introduction

PageRank is a *link analysis algorithm* used by Google Search to rank websites. Named after Larry Page, one of the founders of Google, PageRank produces an estimate of how important a certain website is by counting the number and quality of links directed to that page, with the assumption that the more important a website is the more likely it is to receive links from other pages. PageRank is based on the democratic nature of the Web since its vast link structure is used as an indicator of the quality of a page. The simple number of votes, however, is not enough to produce the estimate: PageRank also analyzes the page that casts the vote and, depending on the importance of such page, assigns a higher or lower weight to its votes.

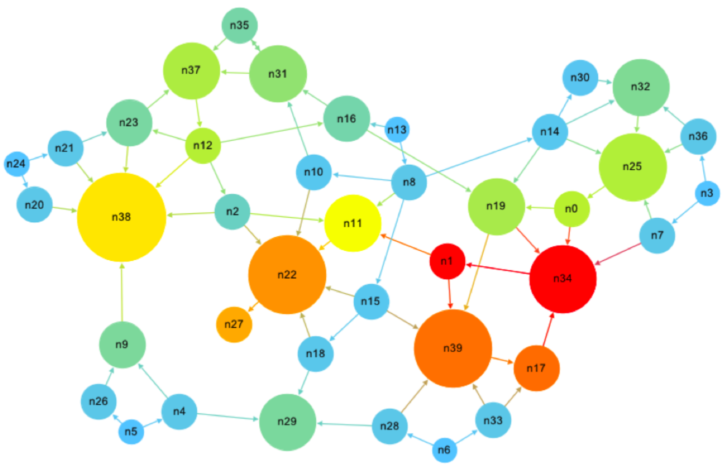

Figure 1: PageRank: ranking web pages using the democratic nature of the Web.

PageRank is computed iteratively; while at the very beginning (time t) all n pages have the same score $1/n$, as time passes (time $t+1$) the i-th page has instead the following PageRank value:

$$p_{i,t+1} = \frac{1-d}{n} + d \sum_{j=1}^{n} A_{ij} P_{j,t}$$

In the previous formula d represents a *damping factor* usually set to a value between 0 and 1 (the implementation uses $d = 0.85$); A_{ij} represents instead the

(i, j)-th element of the *state transition probability matrix*, an incidence matrix with a stochastic structure:

$$A = \begin{bmatrix} A_{11} & & ... & A_{1n} \\ A_{21} & ... & ... & ... \\ ... & ... & ... & ... \\ A_{n1} & ... & ... & A_{nn} \end{bmatrix}$$

More specifically the element A_{ij} represents the transition probability that the user in page i will navigate to page j, and is computed as:

$$A_{ij} = \begin{cases} 1/O_j & \text{if } (j, i) \in E \\ 1/n & \text{if } (j, i) \notin E \text{ and } O_j = 0 \\ 0 & \text{otherwise} \end{cases}$$

Notice that O_j is the number of links – called *outlinks* – from page j to other pages, assuming the user clicks links in the i-th page randomly and uniformly without using the `back` button of the browser or typing an URL; pages with no outlinks are indeed possible, and are called *dangling pages.* Since a page is typically linked to a limited set of other pages the matrix A, albeit very large, is usually sparse: for this reason we can store it using the so-called *Compressed Sparse Row* representation. The implementation of PageRank also accesses the adjacency matrix A using the `mmap` library, which maps files to memory regions, which are then accessed without using the RAM.

2 PageRank implementation

The PageRank algorithm has been implemented sequentially in series of steps, which are described below.

2.1 Step One

In the first step of the project we are tasked with the **sequential implementation of PageRank**. Once the implementation is completed, we need to track its execution times for several input sets, which are freely available at the *Stanford Large Network Dataset Collection.*

2.1.1 Data structures

Several **data structures** have been used to complete this step:

- *Input adjacency matrix*, a matrix derived from a graph that reflects the hyperlink structure of the web and a user-centered model of web-surfing behaviours. The matrix is transposed in the code, becoming the second data structure discussed below;

- *Transposed matrix*, a stochastic matrix of a Markov chain. The matrix element A_{ij} represents the transition probability that the user in state i – corresponding to page i – will move to state j. The matrix must satisfy the following condition:

$$\sum_{j=1}^{n} A_{ij} = 1$$

- *Output link vector*, a vector initially set to 0 and eventually containing the number of out-links of each node of the graph.

- *Probability vector*, a vector representing the initial probability distribution that a surfer is at each state. Initially it's an arbitrarily guessed vector in which each cell contains one divided by the number of nodes.

2.1.2 Power Iteration Method

The PageRank of a page is considered to be the steady-state probability that the surfer is visiting a particular page after a large number of click-throughs. The algorithm we use to implement PageRank is the *Power Iteration Method.* The termination condition is defined by the case in which two consecutive iterations of the algorithm produce two almost identical p-vectors. At the end of the algorithm we obtain the final PageRank values.

2.1.3 Theoretical issues and solutions

In the real web context, to perform the PageRank analysis correctly we need to check that the adjacency matrix A is *stochastic*,*irreducible* and *aperiodic.* To obtain a matrix that satisfies the first condition we need to add a complete set of outgoing links from each page with no out-links (the *dangling pages* described before) to all the pages on the Web. In our code this is done right before the beginning of the *power iteration* method. To obtain a matrix coherent with the second and third properties, instead, we need to add a link from each page to every page and give each link a small transition probability controlled by a parameter d. Such parameter is called the *damping factor*, which is set to 0.85 as per the PageRank paper.

2.2 Step Two

In the second step of the project we need to re-implement PageRank without transposing the matrix A, which still needs to be traversed in row-major order. A instead must be stored in a sparse-compressed form called *compressed sparse row* (CSR). This strategy is useful for two reasons:

- as the dimension of the adjacency matrix grows, the sparseness of matrix A also increases: a sparse representation is therefore extremely convenient.
- working with "complete" matrices of large domains is unfeasible in practice: the memory requirements would be too high and the PageRank algorithm would not work.

We can also map the file to a memory region and then access it via pointers. Finally, for an even better implementation, it is also possible to `mmap` specific partitions of the file, and share the files between more threads.

2.2.1 Data Structure

We use three vectors for the implementation of the compressed A matrix:

- *`val`*, which stores the values of the non-zero elements of the matrix (in this case the only possible value in each cell is 1);
- *`col_ind`*, which stores the column indexes of the elements in the val vector;
- *`row_ptr`*, which is recursively defined as:

```
row_ptr[0] = 0$
IA[i] = IA[i-1] + nr. non-zero elements on the $(i-1)-th row
in the original matrix
```

If all the elements of the n-th row of the original matrix are zero, $row\_ptr[n] = row\_ptr[n+1]$.

2.2.2 Modified algorithm

The *stochastization* of the matrix is handled in a different way with respect to the previous step. In fact, during the first step, stochastization was handled by adding a complete set of outgoing links from each page with no out-links to all the pages on the web; however, if we know which are the dangling nodes, we can avoid filling zero-rows with values $1/n$. This is the reason why in the second step of the project we only store the values different from zero without adding "artificial" outgoing links. Furthermore, we don't need to transpose the matrix. The power iteration method is performed as before, except for the fact that, to update the probability vector, we use the *val* vector instead of the transposed matrix of the first step.

2.2.3 Mmap

Our implementation also uses the `mmap` library to map the file to a memory region in order to access it via pointers, just as we would access ordinary variables and objects. In this way, the two arrays of the compressed representation ***col_in*** and ***row_ptr*** are written on external files.

- *row_ptr*:

```
int *row_ptr, *col_ind;
fdr = open("./row_ptr_map", O_RDONLY);
row_ptr = (int *) mmap(0, row_ptr_size * sizeof(int),
   PROT_READ, MAP_SHARED, fdr, 0);
if (row_ptr == MAP_FAILED) {
   close(fdr);
   printf("Error mmapping the file");
   exit(1);
 } close(fdr);
```

- *col_ind*:

```
fdc = open("./col_ind_map", O_RDONLY);
col_ind = (int *) mmap(0, col_ind_size * sizeof(int),
   PROT_READ, MAP_SHARED, fdc, 0);
if (col_ind == MAP_FAILED) {
   close(fdc);
   printf("Error mmapping the file");
   exit(1);
 } close(fdc);
```

The two files are unmapped as soon as we stop using them. To sum up, thanks to `mmap` it is possible to manage any matrix regardless of its size, without using physical RAM. Both the second and the third step are implemented with and without this mapping, in order to compare the results.

2.3 Step Three

The third step represents the core of the project: we need to parallelize the optimized code of the second step using either SIMD SSE, shared-memory or message passing parallelization. Once this is completed, we need to:

- measure *speedup* and *efficiency* as a function of the processors and cores exploited by the implementation for a couple of different data sets.
- measure how the execution time changes when the problem size is increased, without changing the number of processors and cores employed. This is done by considering different data sets, with different amounts of data.

We implemented the parallelization using a shared memory system with POSIX threads as follows:

```
#pragma omp parallel for schedule(static) if(parallel)
        num_threads(numthreads)
```

Notice we can insert in the beginning of the code the number of threads and the desired granularity. In the tests, we have experimented with several combinations of threads and granularities to obtain the best possible results; we have also tried changing the flag from `static` to `dynamic` in order to test more combinations of the execution.

3 Results

To analyze the behaviour of our implementation we have mainly used a notebook with an Intel Quad Core i5-2400s, 2.5GHz and with Mac OsX Yosemite as the operative system. As for the graphs, we have selected the following data sets with different magnitudes:

Graph name	**Nr. of Nodes**	**Nr. of Edges**
Notre-Dame	325'729	1'497'134
BerkStan	685'230	7'600'595
Stanford	281'903	2'312'497

Our implementation is written in C and uses POSIX threads to parallelize the PageRank algorithm. The actual parallelization is performed over a `for` loop that must be computed in parallel. Experimenting with the number of threads we have noticed that, as expected, using less threads than the number of cores of the CPU is not useful, since some cores remain idle. Using too many threads, instead, increases the communication cost of the scheduling. As for the granularity of the decomposition, we have performed several tests to get the best possible value for each data set. In fact, only by assigning a correct portion of computation to each thread the parallel computation can improve its performance. Studying the results we can observe that the speedup is not extremely high: the computational times are acceptable even when using the sequential version of PageRank. We believe that this is caused by the very convenient implementation of the CSR storage representation for the matrix A. Here follow the best results without `mmap`:

Graph	**Serial**	**Parallel**				
		Time	Granularity	Scheduling	Speedup	Efficiency
Notre-Dame	1.253s	0.613s	1/16	Static	2.04	0.51
BerkStan	3.002s	2.753s	1/16	Static	1.09	0.28
Stanford	0.938s	0.883s	1/8	Dynamic	1.06	0.27

Here follow the best results with *mmap*:

Graph	**Serial**	**Parallel**				
		Time	Granularity	Scheduling	Speedup	Efficiency
Notre-Dame	1.486s	0.805s	1/8	Static	1.85	0.46
BerkStan	3.916s	3.140s	1/16	Static	1.25	0.31
Stanford	1.320s	1.110s	1/8	Dynamic	1.19	0.30

To complete our set of tests we have also tried to execute our code on a notebook with a different CPU – Intel Dual Core I5-5257U 2.7GHz with hyperthreading – and most importantly a SSD M.2; the focus of this set of tests was to understand if the version implemented with *mmap* was bound to perform worse even with a different speed of access to the hard disk. We obtain these results without *mmap*:

Graph	**Serial**	**Parallel**				
		Time	Granularity	Scheduling	Speedup	Efficiency
Notre-Dame	1.036s	0.446s	1/8	Static	2.32	0.58
BerkStan	2.121s	2.006s	1/16	Static	1.06	0.27
Stanford	0.701s	0.609s	1/8	Dynamic	1.15	0.29

Here follow the best results with *mmap*:

Graph	**Serial**	**Parallel**				
		Time	Granularity	Scheduling	Speedup	Efficiency
Notre-Dame	1.199s	0.571s	1/8	Static	2.10	0.53
BerkStan	2.747s	2.004s	1/16	Static	1.37	0.34
Stanford	0.889s	0.850s	1/8	Dynamic	1.05	0.26

As we can see the results are overall better, with smaller execution times. We include some plots to visually describe the results of our tests:

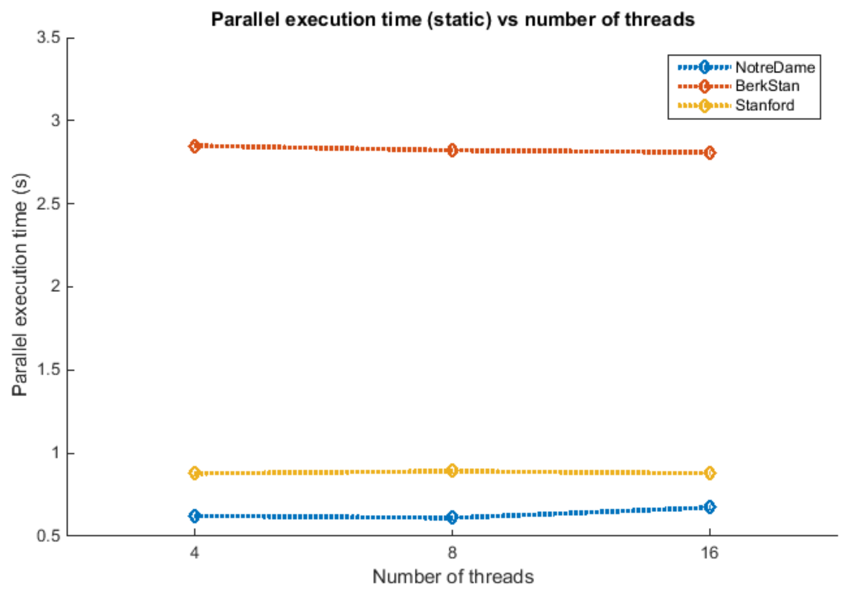

Figure 2: Different results influenced by the number of threads (4, 8, 16). As we can see it appears that with 8 threads we obtain the overall best results.

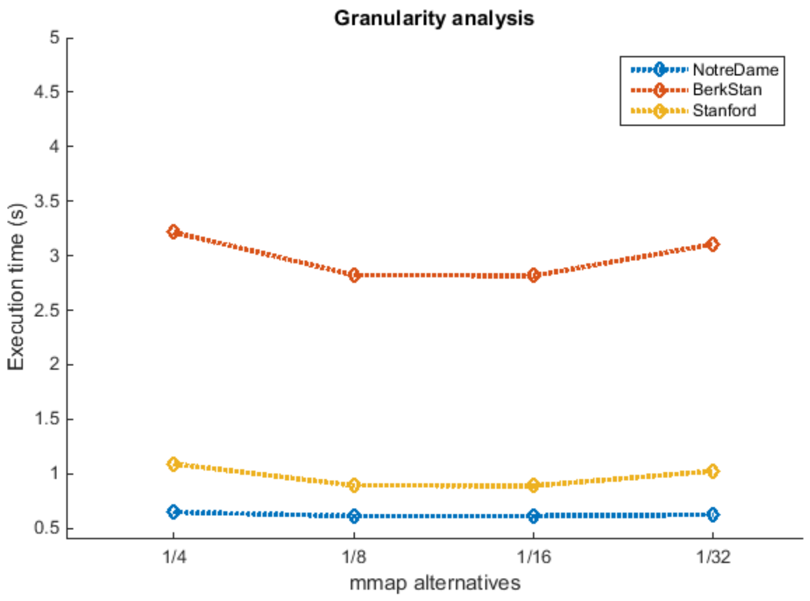

Figure 3: Different results influenced by the differences in granularity. Best results are obtained with 1/8 and 1/16, with variances depending on the data set.

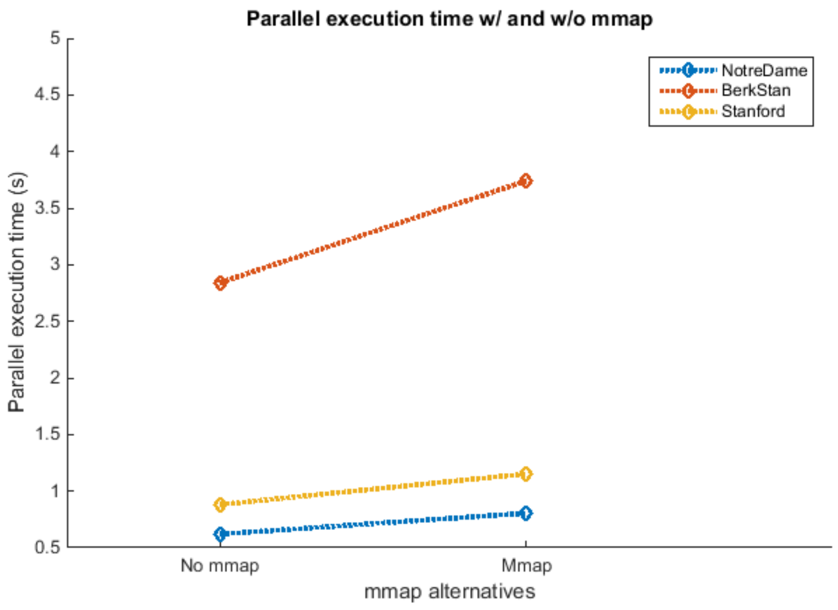

Figure 4: Different results influenced by mmap. As we can see, `mmap` causes some overhead when accessing the physical memory.

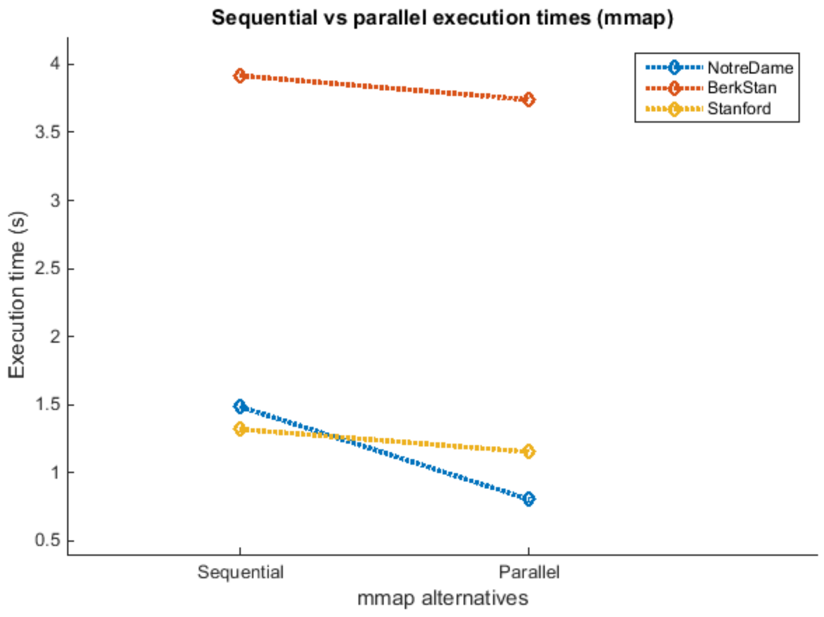

Figure 5: Comparison between execution times of the sequential and parallel implementations of PageRank, using `mmap`.

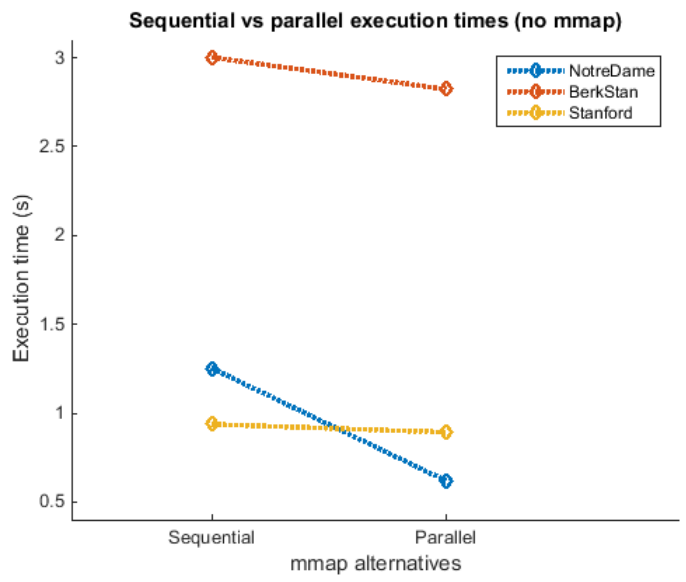

Figure 6: Comparison between execution times of the sequential and parallel implementations of PageRank, without using `mmap`.

References

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.

[S. Brin and L. Page, 1998] . S. Brin and L. Page (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 33:107-117