PageRank: A Parallel Approach

Manos Chatzakis
csd4238@csd.uoc.gr

Computer Science Department
University Of Crete

March 2021

1. Introduction

This report is part of the first assignment of cs342 - Parallel Programming course of Computer Science Department, University of Crete. The main point is to present a method to parallel PageRank algorithm efficiently, and the rest of the report is organized as follows: Section 1 is the introduction. Section 2 describes the input graph. Section 3 analyzes the parallelism method. Section 5 shows the measurements. Section 6 is the conclusion.

2. Input Graph

2.1 Structure

The input is converted into a PageRank graph:

```
typedef struct graph_t{
    long size;
    double init_score;
    node_t *nodes;
} graph_t;
```

```
typedef struct node_t{
    long id;
    double score, score_add;
    long outlinks_num,
    long inclinks_num;
    link_t *inclinks_head;
} node_t;
```

```
typedef struct link_t{
    long from_node_index;
    struct link_t *next;
} link_t;
```

2.2 Graph Division

For the work division, every thread handled a set of nodes, determined from the equation:

$$sets = \frac{graph.size}{threads}$$

The above method requires less work spend on the data parsing, but it does not always result to the best work division. Another approach could try to divide the nodes in a way that nodes with high work demand (etc. many outlinks or inclinks) to different threads, but it would result in high overhead to manipulate the dataset.

3. Algorithm and Approach

3.1 Formula

Our approach calculates the PageRank of each node using the simplified formula:

$$PR_n(u) = \begin{cases} \sum_{k \in B_u} DF \frac{PR(k)}{L(k)} + PR(u), & \text{if } OutDegree(u) = 0 \\ \sum_{k \in B_u} DF \frac{PR(k)}{L(k)} + (1 - DF) \times PR(u), & \text{if } OutDegree(u) > 0 \end{cases} \quad (1)$$

$$\Rightarrow$$

$$PR(u)_n = PR(u) + \sum_{k \in B_u} DF \frac{PR(k)}{L(k)} - \sum_{k \in T_u} DF \frac{PR(u)}{L(u)} \quad (2)$$

PR(x): PageRank of node x in the current iteration, PRn(x) : PageRank of node x for the next iteration, L(x): The number of outgoing links of node x, Bu: Set containing the nodes that point node u, Tu: Set containing the nodes that u points at, DF: Dumping factor, e.g. the percentage that every node offers from its own PageRank. In this approach, this value is considered 0.85, thus every node offers 85% of his PageRank in every iteration.

The above formula implies that the PageRank of a node in the next iteration is the sum of all the scores given from the neighbors, incremented with the previous score minus the score that this node gave to it's neighbours throughout the iteration. Note that using this equation means that the total PageRank score of the graph remains constant.

3.2 Parallelism

The parallelism method relies on the observation that PRn(u) refers to the PageRank value that will be used in the next iteration. Thus, this implementation saves the offered PageRank score by each neighbour to a new field of the node struct, and adds it to the PageRank score when the calculations of the corresponding iteration ends. This method does not face data race, as only the thread assigned the node has read/write access to the temporary score field:

```
curr = nodes[i].inclinks_head;
while (curr != NULL){
    from_index = curr->from_node_index;
    nodes[i].score_add += nodes[from_index].score * D_FACTOR / nodes[from_index].outlinks_num;
    curr = curr->next;
}
```

An important part for this implementation is the use of barriers, to ensure that the score update for each node and the start of the new iteration for each thread will be done concurrently.

```
pthread_barrier_wait(&bar);
for (i = data->from; i < data->to; i++){
    score = nodes[i].score_add;
    if (nodes[i].outlinks_num != 0)
        nodes[i].score = score + nodes[i].score * (1 - D_FACTOR);
    else
        nodes[i].score = score + nodes[i].score;
    nodes[i].score_add = 0.0;
}
pthread_barrier_wait(&bar);
```

4. Measurements

In this section the different approaches are measured and the metrics are presented. The values below were extracted using 3 SNAP datasets (Facebook, Gnutella, Enron) on a local machine (8 cores,16 GB RAM), using the buildin posix command:

```
clock_gettime(CLOCK_MONOTONIC, &start);
pagerank();
clock_gettime(CLOCK_MONOTONIC, &finish);
```

The time is measured in seconds.

- Facebook Dataset.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Dev.	Average Dev.	Speedup*
1	0.038759	0.037749	0.039297	0.04045	0.039666	0.03918	0.00101	0.00074	-
2	0.021782	0.02137	0.02144	0.021435	0.022365	0.2168	0.00042	0.00032	1.807x
3	0.015493	0.01498	0.015683	0.015355	0.015551	0.01541	0.00027	0.0002	2.542x
4	0.015658	0.015022	0.014813	0.014786	0.014616	0.01498	0.00041	0.00029	2.615x

- Gnutella Dataset.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Dev.	Average Dev.	Speedup*
1	0.056732	0.055963	0.059056	0.057208	0.06052	0.0579	0.00186	0.00151	-
2	0.038491	0.037579	0.036734	0.038122	0.038693	0.03792	0.00079	0.00061	1.526x
3	0.027344	0.028106	0.028176	0.027397	0.027783	0.02776	0.00039	0.00031	2.085x
4	0.023385	0.022163	0.022034	0.025295	0.023941	0.2336	0.000135	0.00101	2.478x

- The text in the entries may be of any length.

Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Dev.	Average Dev.	Speedup*
1	0.642836	0.649105	0.617986	0.666572	0.667904	0.64888	0.02041	0.01478	-
2	0.590239	0.588571	0.588146	0.569202	0.583949	0.58402	0.0086	0.00596	1.111x
3	0.513008	0.513008	0.515894	0.511074	0.579348	0.52647	0.02961	0.02115	1.232x
4	0.482256	0.484116	0.479822	0.54109	0.492112	0.49588	0.02569	0.01808	1.308x

Measurements on a CSD machine are also provided. (*): Speedup was calculated using the formula:

$$SpeedUp = \frac{t_{seq}}{t_{par}} = \frac{thread1_{average}}{threadN_{average}} = S \Rightarrow SpeedUp = Sx$$

$$Assuming : t_{seq} = thread1_{average}$$

5. Conclusion

The approach presented seems to improve the running time of PageRank, especially when the multi-threaded program runs on 2 or 3 threads. Thus, PageRank can be very efficient, even when its used for vast graphs.