Parallel Page Rank Estimator

December 9, 2019

Abstract

In this paper we look into an analysis and implementation of a parallel page rank estimator. We discuss different algorithms and ways of creating a parallel page rank algorithm, in two forms; that of a multi-threaded format, which we will focus on, and that of a parallel processor format. We discuss certain key challenges to parallelism that are must be encountered and overcome in creating an algorithm. In our two proposed approaches, we discuss high level details such as space and runtime complexity, as well as different data structures to use. Finally, we implement an openMP approach, and discuss experimental results such as runtime, speedup, and efficiency, over varying forms of input data as well as thread counts.

1 Introduction

The PageRank problem presents a challenge in computer science in terms of parallelization. What is the PageRank problem? It is ”a method for computing a ranking for every web page based on the graph of the web.”[1] In this problem, we will be simulating web pages with numbers rather than actually running it with web pages. Why parallelization however? While our simulated sets are large, they are not as large as you might expect in a real world application. For example, upon searching the term ”Page Rank” in Google, right underneath the search bar you will see an estimation of the number of results and the time that it took to find them. In this case it's estimated at about 1.5 billion results in about 0.6 seconds. How is it determined which results to show first? Through which result has the most traffic when given that specific search. To do this it uses a PageRank algorithm. However, it would take massive amounts of computing power to glean through all the data that Google has, however, this is where parallelization comes into play. By running a Page Rank Algorithm in parallel, we are able to cut down the time that it takes to run given massive sets of data.

2 Problem Statement

The **input** to the problem consists of:

1. an integer N, the number of trials to run. This can be substituted with the number of vertices.
2. an integer K, the length each walk through the graph should be.
3. an element D, the damping factor to use in choosing nodes for each walk.
4. a graph G, represented in the form of an adjacency list with n elements (v_i, u_i) where v, u are nodes in the graph.

The **output** to the problem consists of:

1. elements x_1, x_2, x_3, x_4, x_5 where $x_i = (v_i, pr_{est})$, pr_{est} is the PageRank estimate for v_i.
2. T, the time taken to calculate the PageRank estimates.

Given these inputs, for N number of trials, we will perform a random walk of length K, keeping track of how many times each node is visited. The PageRank estimate for a given vertex v will then be the number of times it was visited, divided by the total visits across all the vertices.

3 Key Challenges in Parallelization

Parallelism poses a challenge for this problem in conversion of the sequential algorithm to parallel. It is trivial to create an algorithm to sequentially find the Page Rank estimate, but it will not be efficient due to how much data is involved in estimating it. Some of the main limitations of parallelizing the Page Rank algorithm are described below.

3.1 Partitioning

The task must be partitioned correctly. By this I mean the algorithm must be separated out to appropriate subsections that are each able to be ran in parallel. Whether these subsections are independent or not, they must be synchronized in their output to avoid race conditions and deadlock, where two processors, or threads, depend on values generated in the other. To solve the synchronization in part, each iteration of any loop must be independent of the previous iteration.

3.2 Sharing Data

Data, such as the input and output data, must be synchronized between processors, or threads, and balanced so that the load on a processor, or thread, does not exceed the max that it can handle. In the event that the load is not balanced between processors, the parallel algorithm will not run well compared to the sequential.

3.3 Synchronization

The output data must be synchronized across all the processors. For example, synchronization plays a part in dependencies in the use of a for loop. To be parallelizable, each iteration of a loop must be independent of other iterations.

4 Proposed Approach(es)

The high level approach to the Page Rank algorithm is stated below:

1. Traverse N paths of length K through graph G, keeping count of times a vertex is visited
2. Estimate Page Rank by the fraction of times a vertice was visited

4.1 Key Idea(s) for openMP)

In thinking about ways to parallelize the problem, I decided to first write the algorithm in serial. My main question at the very beginning was how I wanted to contain the data. I started out with researching on different data structures. Knowing that I would have to continually access different vertices in the graph, I knew that a structure with a fast access time would be optimal. As well, considering that we are not required

to include data input or output in the parallel run-time, insertion time does not matter. In my first attempt at this problem, I considered using the STL unordered_map structure to hold my adjacency list, as I figured it would have a quick access time while being able to easily perform insertion. As well, I decided to use one to contain the vertices and their visit counts. Having figured out how to contain my data, I proposed to run the actual PageRank portion in two **for** loops, as seen below in **Algorithm 1**, where V represents the vertices, E represents the edges, C represents vertices and their visit counts, N is the number of trials, and K, D are user inputs.

Upon initial testing as currently stated however, I discovered that it was taking much longer than expected, and thus propose to change from unordered_map structures, to dynamically allocated arrays and vectors. This would allow me to have guaranteed $O(1)$ access time, as well as fast insertion, with the caveat that the number of edges and nodes are known beforehand, in order to allow for the dynamically allocated arrays to be initialized.

Algorithm 1 PageRank Algorithm

1: **procedure** (V, E, C, N, K, D) ▷ Traverse N paths of length K
2: **for** i ← 0 to N **do**
3: $current\_vertex \leftarrow V_i$
4: **for** j ← 0 to K **do**
5: $C[i]++$ ▷ Update the number of visits at current vertice
6: $prob_i = rand()$
7: **if** $prob_i > D$ **and** $\exists edge \in E_i$ **then** ▷ There is a one-hop neighbor
8: $target = rand()\ \%\ (number\ of\ vertices\ connected\ to\ V_i)$
9: **else** ▷ Move to a random vertice
10: $target = rand()\ \%\ (total\ number\ of\ vertices)$
11: $current\_vertex = target$

Having this as a base serial algorithm, I needed to decide how to parallelize it. I decided upon research into the $\#pragma\ omp\ parallel\ for$ call to share the data structures V, E, C, N, K, D, while keeping $i, j, current\_vector$ private among all the threads. As well I had to handle how the value of the visits would be changing across all threads. Besides this, I also decided to handle the randomness by calling the functions $drand48\_r$, $lrand48\_r$, $and\ srand48_r$ to obtain random values, as $rand()$ is not thread safe.

In terms of time as well as space complexity, the proposed algorithm, with the $O(1)$ access time guaranteed by the array, needs just to be analyzed in terms of parallel complexity. In terms of space complexity, as we are given the data structures, rather than creating them inside our loop, and again only need to hold small values such as integers, we can determine that we have an $O(1)$ space complexity. Time complexity is a completely different problem however. Serially, time complexity would be $O(N * K)$, however, this is changed by the parallelism of the algorithm. To add in parallelism, I would imagine that the runtime would be $O(N * K)/(\#threads)$

4.2 Key Idea(s) for MPI

While not having too much time for this, my thoughts on an MPI version of the Page Rank algorithm run similar to this. The high level approach to the Page Rank algorithm is stated below:

- Initialize counter for vectors
- One processor P should handle updates and counter, the others should have the graph
- Each processor not handling updates should send to P an updated vertex and ask for the target
- P should update vertices and check for convergence in vertex ranking among all processors
- When N trials have been finished, finalize

This implementation would actually be reasonably close to the openMP implementation, in that there is a parent thread that handles calling of the parallel threads. Instead of threads however, with MPI we would have a parent processor that handles sending out vertices and receiving updated vertices from child processes.

5 Experimental Results and Discussion

Before starting the discussion and presentation of results, here are the assumptions I am going off of: that the input file has the number of vertices and edges in the format "# Nodes: ? Edges: ?", as well, for the chart where D is fixed, I omitted runtime for $K = 10,000$ to allow for a clearer picture of what is going on. It is however included in the Speedup and Efficiency charts. All the data from the charts comes from running the web−Google_sorted.txt file. As well, the runtimes and charts are all made with the parallel processing time. I would have added the total time, but ran out of time and space.

5.1 Setup:

All tests were ran on a single node in the Pleiades Cluster. Taken from Piazza, it is described as this: "Pleiades is a 64-core cluster. The cluster contains 8 nodes, each with 8 Intel Xeon E5410 cores @ 2.33GHz, and with 4 GB of shared RAM. The compute nodes are labelled n000[0-7]. The MPI implementation used is OpenMPI." The code must be compiled as follows: "g++ −o page_rank -fopenmp test.cpp −std=gnu++11". The goal of this project was to estimate the Page Rank of vertices in a graph. To do so, I ran for a set amount of trials, N, where in this case N was determined to be the number of vertices in the graph. For each trial, I traversed a graph G in a path of length K. The way I determined which vertices to go to in the path was by generating a random value between 0 and 1, and determining if it was greater than the user−input damping factor D. If so, I chose a random target vertex from the adjacent vertices. If it was not, I chose a random vertex to traverse to from all the vertices in G.

5.2 Runtime, Speedup, and Efficiency

Runtimes, Speedup, and Efficiency are displayed in **Figure 1** and **Figure 2**. However, we will mainly be discussing the charts where K is fluctuating. This is because the charts where D is fluctuating are almost equivalent in their data, so it's harder to see large transitions. The charts where K fluctuates should be able to be sufficiently analyzed in terms of runtime, speedup, and efficiency such that the others are not

needed. Below, I have displayed the runtime with K changing along with thread count, while D is fixed at 0.10.

Runtime (D = 0.1)	**Thread**	**Thread**	**Thread**	**Thread**
K	1	2	4	8
10	1.78988	1.0239	0.555648	0.298814
100	13.1642	6.57846	3.32776	1.69464
1000	133.454	68.0531	35.4364	18.4724
10000	1334.93	751.78	392.277	215.514

The overall runtimes gathered are reasonable. The overall trend on the data is that of faster completion time given more threads, and an increase in runtime with an increase of path length K. In fact, it shows a trend in speedup proportional to the number of threads used. Finally, notice chart (b) of **Figure 1** for runtime. This chart was plotted with D and thread count fixed for varying values of K. Here, it is much easier to see how much faster running the algorithm is with 8 threads.

To test a precision in time, I set K fixed to 1000, and varied D along with the number of threads used. The table of runtime values are shown below.

Runtime (K = 1k)	**Thread**	**Thread**	**Thread**	**Thread**
D	1	2	4	8
0.25	141.947	79.2359	47.5063	31.6816
0.2	141.353	78.8339	47.3856	31.5979
0.15	141.424	79.2467	47.5737	31.8967
0.1	141.757	79.0015	47.3565	31.7453

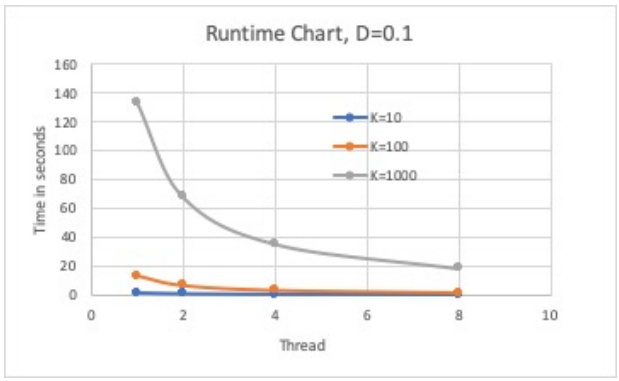
As can be seen, the runtimes with a fluctuating damping factor D are almost equivalent across threads. This is to expectations, as the damping factor does not have a large impact on runtime.

In **Figure 1**, we can see the speedup chart, and recognize that it has a bit of variation, in that the Speedup for $K = 10,000$ is actually less than the Speedup for several other values. This could be due to small variations in random targets, as there is not a very large difference. However, upon looking at the Efficiency chart, we can see that there is a decrease in efficiency in certain values of K. Again, there is not a huge difference, but there is still one there. Both charts actually show a trend in displaying $K = 100$ as an optimal path length, where its both the most efficient, as well as has the most difference in speedup. To really see a much larger difference, I would want to run the program with $K > 10,000$ to determine if efficiency, and speedup, is still dropping.

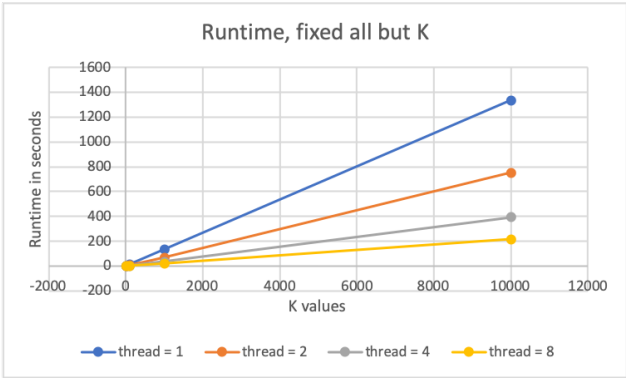
In the table below, we can also see the top 5 results as given by running $K = 10,000$ and $D = 0.10$

Thread	**Vertex:PR**	**Vertex:PR**	**Vertex:PR**	**Vertex:PR**	**Vertex:PR**
1	551829: 0.00116983	407610: 0.00107657	504140: 0.00101651	163075: 0.000980796	914474: 0.000952189
2	551829: 0.00122655	597621: 0.000946516	163075: 0.00088982	504140: 0.000871921	32163: 0.000833057
4	551829: 0.0010474	605856: 0.00094313	597621: 0.00091142	32163: 0.000900016	537039: 0.000858187
8	537039: 0.000933031	605856: 0.00092558	486980: 0.00092163	41909: 0.000885901	551829: 0.000855864

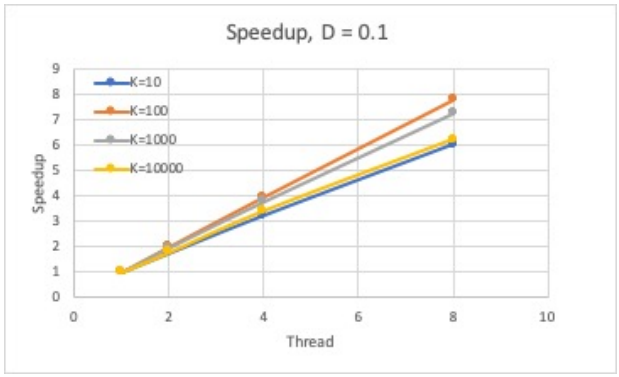
As it is, it is clear to see a natural convergence in the values. While there are outliers, the values per thread generally match in the nodes that are the top ranked. As well, I have included in **Figure 3** a bigger picture of an earlier run of the algorithm. Notice the values are slightly different and may not converge as much between threads. I am not sure why this happened but upon running the code restricted to one node, the values stopped converging as much.



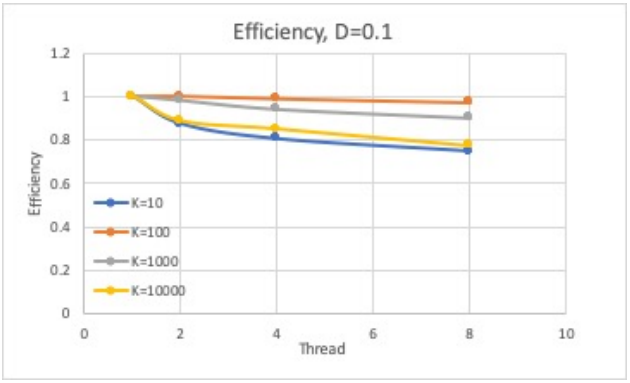
(a) Runtime.



(b) Runtime, K changing.

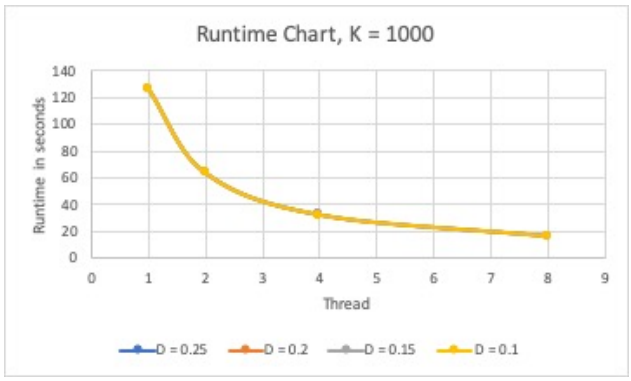


(c) Speedup.

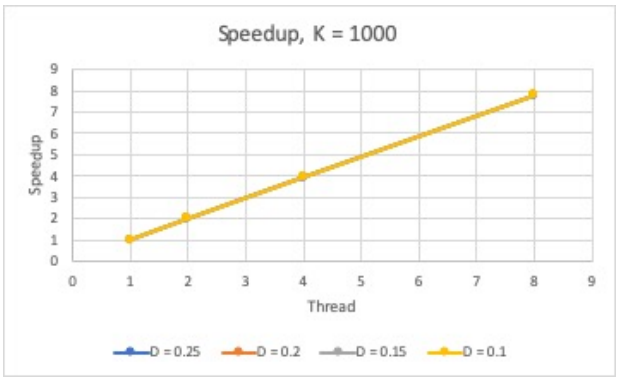


(d) Efficiency.

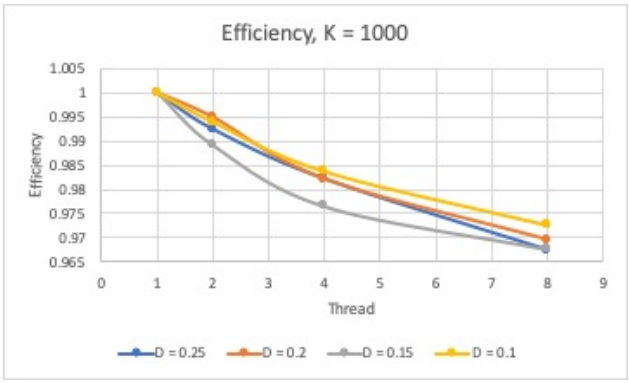
Figure 1: D is fixed at 0.1, with input of web−Google_sorted.txt



(a) Runtime.



(b) Speedup.



(c) Efficiency.

Figure 2: K is fixed at 1000, with input of web−Google_sorted.txt

K=1000,D=0.1	Threads			
Node:PR	1	2	4	8
1	297518:0.00587001	785830:0.00257903	41909:0.0015403	163075:0.00124319
2	529269:0.00579734	911951:0.00257718	173976:0.0015134	41909:0.00123519
3	370818:0.00573794	895519:0.0025615	911951:0.00145026	597621:0.00119131
4	497975:0.0055729	322721:0.00224936	895519:0.00144677	691633:0.001025
5	74583:0.00449217	228680:0.00213419	909301:0.00140879	537039:0.00102127

K=10000,D=0.1	Threads			
Node:PR	1	2	4	8
1	486980:0.00141602	486980:0.00117168	41909:0.00116761	41909:0.0011184
2	213432:0.00135211	213432:0.00104627	163075:0.000988817	163075:0.000924507
3	41909:0.00108274	504140:0.00101514	213432:0.000938234	537039:0.00090366
4	551829:0.000945443	32163:0.00100402	504140:0.000922718	384666:0.000861711
5	614831:0.000886031	163075:0.000953614	384666:0.000913735	551829:0.000845429

K=1000,D=0.15	Threads			
Node:PR	1	2	4	8
1	366385: 0.0188602	747106: 0.0107135	747106: 0.00822525	747106: 0.0099412
2	496502: 0.017406	24576: 0.0106809	24576: 0.00818707	24576: 0.00990243
3	490634: 0.0156752	544138: 0.0102696	544138: 0.00665992	544138: 0.00778964
4	447994: 0.0146946	868185: 0.0100067	370344: 0.00634205	370344: 0.00771804
5	495538: 0.0144647	904346: 0.00999496	86480: 0.00452128	914474: 0.00428673

K=1000,D=0.1	Threads			
Node:PR	1	2	4	8
1	544138: 0.0111872	366385: 0.00942474	287007: 0.00844812	747106: 0.0071422
2	747106: 0.0111249	496502: 0.00871853	165617: 0.00558829	24576: 0.00707708
3	24576: 0.0111144	603104: 0.00847791	819579: 0.00558578	495538: 0.00539422
4	370344: 0.00902428	799639: 0.00806088	747106: 0.0055391	370344: 0.00538476
5	366385: 0.00840668	490634: 0.00787117	24576: 0.00547105	544138: 0.00492522

Figure 3: Top Five Results from Google with Page Rank.

6 Acknowledgments

I would like to acknowledge Benjamin Hellwig and Tyler Scheffler. I would also like to acknowledge Ryan Neisses, although I did not actually discuss much in the way of approaching this problem with him.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd. ”The PageRank citation ranking: Bringing order to the web”, Stanford InfoLab, p2, 1999.