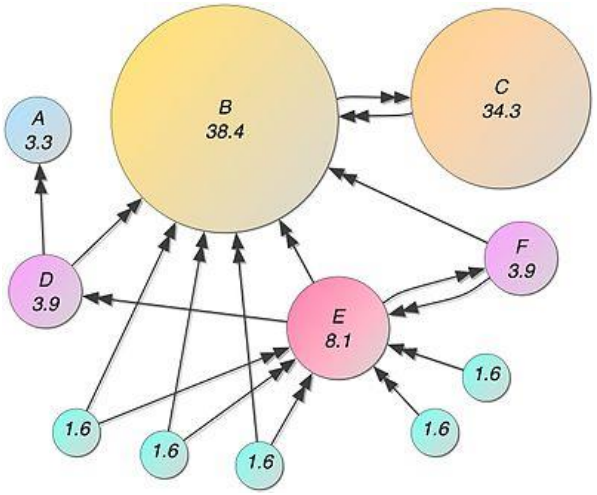
Introduction

With processing units seeing a max frequency near 5.0GHz, it is becoming more obvious that computing will head in a more parallel or concurrent direction. As seen with more recent CPU upgrades coming in with slight changes to frequency, but much bigger changes to concurrency and optimization of code using techniques such as branch optimization, are taking over. In order to scale with the increasing amount of data that is produced on a daily basis, more processing units are going to have to be used to in order to keep up with the amount of data crunching needed to understand systems and the people that use the systems.



In this paper we will be examining an approach to parallelize the PageRank problem. The PageRank algorithm attempts discover which website will be the top-ranking website by simulating users on the web. It does this by trying to simulate users clicking links, edges in the graph, and viewing pages, vertices in the graph. The algorithm manages to do this by starting at every node and then continuing a walk of size K randomly picking to either go to an outgoing edge or pick a random node in the graph to go to based up some dampening odds. In the implementation we will explore here, there was an added factor, going back from a node. This allows the extra simulation where the user hits the back button. They will only be able to return to the last visited node, just like hitting the back button on the browser. The odds of thing happening vs picking the outgoing edge is also a variable and allows for tweaking in the algorithm.

Problem Statement

The purpose of this paper is to define and implement the PageRank algorithm in a parallel manner that speeds up execution time by taking advantage of more available processor cores. The PageRank algorithm generally only requires a graph, a walk size K, and the dampening odds. In this specific variation of the algorithm, we will also need back dampening odds to compute the probability of simulating a user hitting the back button.

Inputs:

In this case the graph was presented as a CSV file that contained a list of directional edges and the nodes that are connected. The rest of the parameters, K, dampening odds, and back dampening odds were presented by the user in order for the application to run.

Outputs:

The main output of the algorithm in this specific case is the top 5 highest visited vertices from the graph.

Key Challenges in Parallelization

The biggest challenge for devising parallel approaches to PageRank is memory optimization and storage. In a shared memory space, this problem is quite simple to develop, assuming we have enough space to store everything. Techniques such as sparse arrays or other direct non memory-wasting techniques are going to have to be used to fix the maximum graph possible. There is also the big problem of concurrent and random reads across huge amounts of data leading to cache misses and other problems if not managed effectively, though this is out of the scope for this report. Trying to keep the efficiency of the increasing number of processors is also going to be a very important aspect of this.

When it comes to a distributed approach to the problem, there is going to have to be a ton of looking into ways of packing as much data as possible in a small space to reduce network delays, and other distributed memory solution problems. Along with this, there is going to be a huge problem when it comes to *what* data the processor needs to retrieve.

Proposed Approach(es)

There were a couple of experimental problems that needed to be sorted out before deciding what to code. One of which were the closure vertexes, or vertexes with no outgoing edges. I chose to approach this by always keeping track of previous nodes and allowing the graph to go back in the case of a *tails* flip. This is like clicking the back button on a browser to get the previous page you were on. I also allowed this behavior to happen any time that an outgoing edge was to be picked in the *tails* part of the algorithm. Another key idea to think about was graph storage and how to keep read times low. I decided to go for O(1) read and O(1) access to a random node. There were a couple ways of achieving this, though I went with an approach that could still be leaned up a little bit and takes extra memory and preprocessing. I chose to keep an *unordered_map*(O(1) readById/existsById) of nodes with a key being a node id and the

value being a pair which contained a vector(O(1) random access) for the outgoing edges as well as an *unordered_set*(O(1) readById) for the outgoing edges. I also kept a vector of all nodes(O(1) random access). The reason for not just doing an array that is of size maxNode-minNode was because of the potential for lots of missing nodes, though this was not the case within the dataset. This would have been more efficient memory-wise. The unordered_set could have been taken out without compromising performance.

The actual page rank algorithm for shared memory went like this:

```
Count = array of all nodes
totalVisited = number of total visited nodes
dampeningValue = selected Value
For every node in graph:
        SelectedNode = node;
        For 0 to walk size K:
                totalVisited++;
                count[selectedNode]++;
                if pick-random-number from 0-1 > dampeningValue
                        //Heads operation
                        add selected node to prevNodes
                        selectedNode = randomNodeInNetwork
                else
                        //tails operation
                        if random-number from 0-OutEdgesFromSelected+backDampening >= OutEdgesFromSelected
                                if no prevNodes
                                        selectedNode = randomNodeInNetwork
                                else
                                        selected node = top of prevNodes
                        else
                                add selectedNode to prevNodes
                                selectedNode = outgoingEdges[randomNumber]

return top 5 nodes
```

Due to all the access optimizations, the runtime of this is going to be O(KV) where K is the runs per node and V is the number of verities. Due to some of this preprocessing, we are storing each vertex once in the map, once in the vector, once in the counts array giving O(3V) or O(V) space. And each edge is being stored 2 times giving a space complexity of O(2E) or O(E). These combing to make a total space complexity of O(E+V).

For this to be solved with distributed memory, the approach would have to change a considerable amount. The biggest architecture change is going to have to be the way the graph is stored as all of it wont fit on

any single processor and information will have to be retrieved. What I believe to be a good way of dividing up the graph is to give a breadth first set number of nodes from the starting node and send each processor the first x(to be manipulated and tested) amount of nodes that are in the downstream of the current node as well as getting the top y (to be manipulated and tested) vertices with the highest number of incoming edges. Whenever a node gets a heads flip, or must pick a random node, if the current memory does not have the space to hold the new first x elements, then it will deallocate the breadth first search elements and retrieve the new ones. The main algorithm will still be applied as explained throughout the previous shared memory example with the only difference being retrieval of data changes.

Experimental Results and Discussion

All the tests below were performed on WSL 2, on Windows 10, with ubuntu 18.04.03, using libomp-dev verseion 5.0.1-1, gcc version 7.4.0, and c++ 11. The hardware it was run on were 32 GB of ram running at 2133MHz and an I9 9900k (8c-16t) running at around 4.8GHz during testing.

	Input Size (k)	Time(s)			
Threads		1	2	4	8
	64	13.5569	7.72504	3.96571	2.54958
	128	26.6929	14.9263	7.89967	4.51699
	512	108.46	61.3682	33.384	18.9882
	1024	211.632	121.84	62.5377	38.549
	2049	422.635	242.029	125.394	72.0837
	4096	858.606	479.561	247.051	142.878
	8192	1686.88	956.895	503.388	286.074
	16384	3374.64	1894.08	1000.86	566.52

Table 1-NotreDame, D=0.2

As we can observe from table 1 (looking down each column), increasing the input k size projects and almost perfect factor of 2 multiplication down showing that the amount of time complexity for k is

constant. Comparing across the rows, we can see that the time is not cutting directly in half showing that there are some problems with efficiency, which will be explored in more detail later.

	Input Size (k)	Speedup			
Threads		1	2	4	8
	64	1	1.75492942	3.41853035	5.317307
	128	1	1.78831325	3.37898925	5.909444
	512	1	1.76736486	3.24886173	5.711968
	1024	1	1.73696651	3.38407073	5.489948
	2049	1	1.74621636	3.37045632	5.863115
	4096	1	1.79039997	3.47542005	6.009365
	8192	1	1.76286844	3.35105326	5.896656
	16384	1	1.78167765	3.3717403	5.956789

Table 2-NotreDame, D=0.2

As you can see from table 2, speedups were relatively the same even with a changing K value, with the exception of 64 as the input for K. This is most likely due to the amount of threads being so close to the k value and it takes time to set up and distribute these threads. This is good to show that even with an increasing number of walks, the speedup is maintained again showing that increasing K increases it by a constant time.

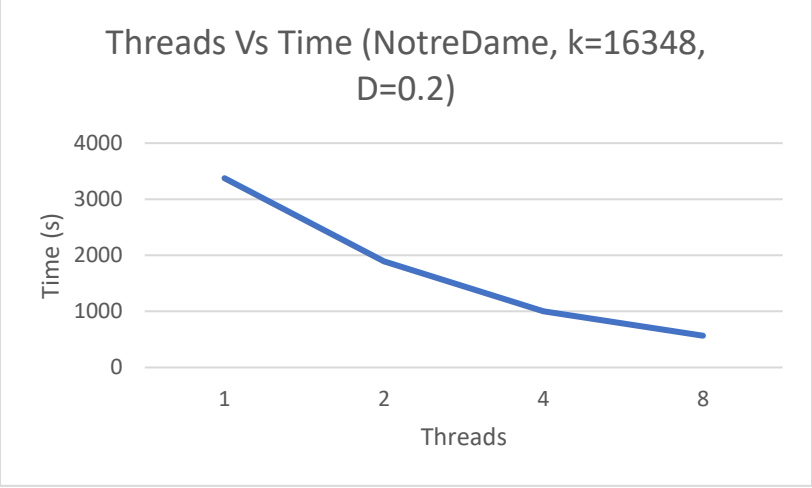


Figure 1

Figure 1 shows us that the time is decreasing at a decreasing rate and is going to bottom out at a certain point. This is a property known as diminishing returns, where the more processors you throw at the problem the less each of them will help as overhead closes in on the time.

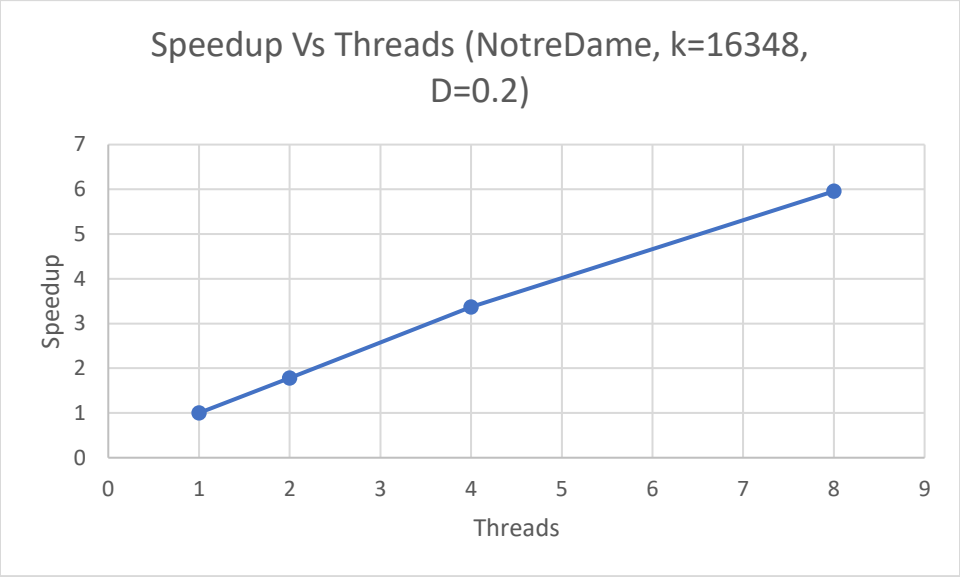


Figure 2

In Figure 2 we can see that the speedup is decreasing at a seeming near constant rate (Figure 3 also shows it more clearly). As we continue to push more cores and more threads at the problem our efficiency is going to go down as shown here and, in Figure 3.

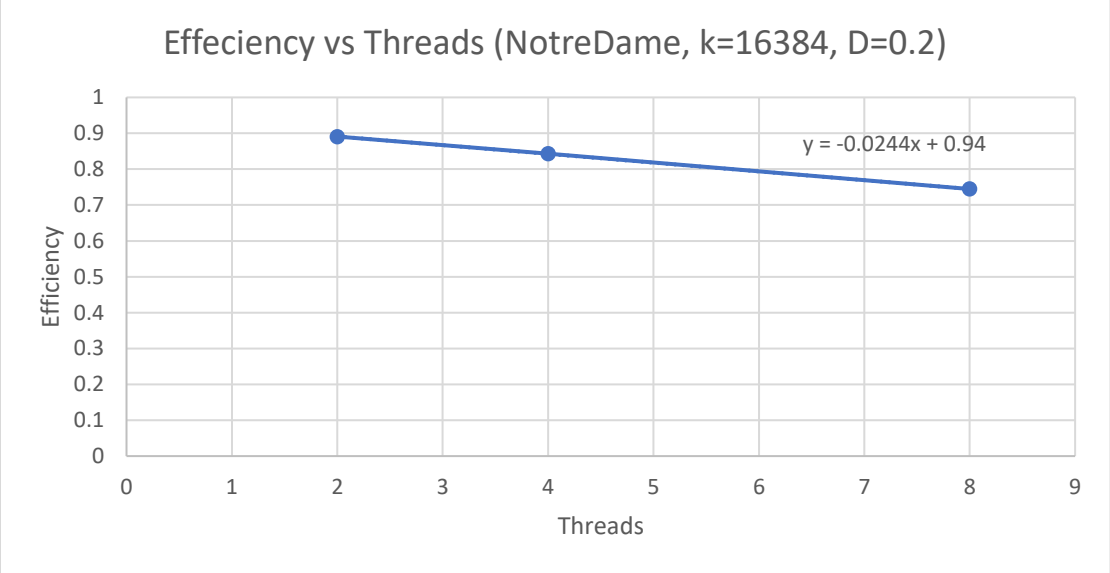


Figure 3

With the decrease of efficiency, it is apparent that the number of walks must keep increasing for the efficiency to keep growing. Though as we will see later, increasing the number of walks seems to have no affect on the top 5 results, after a certain size of k.

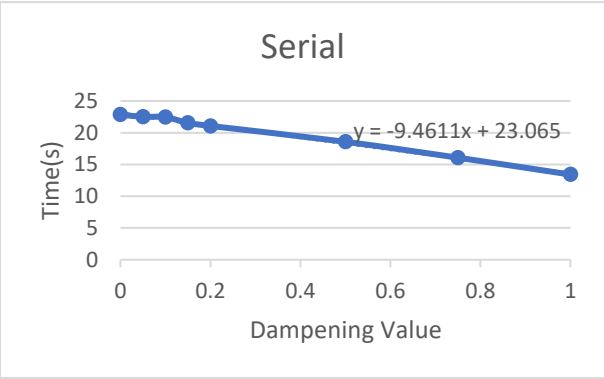


Figure 4-NotreDame (k=100)

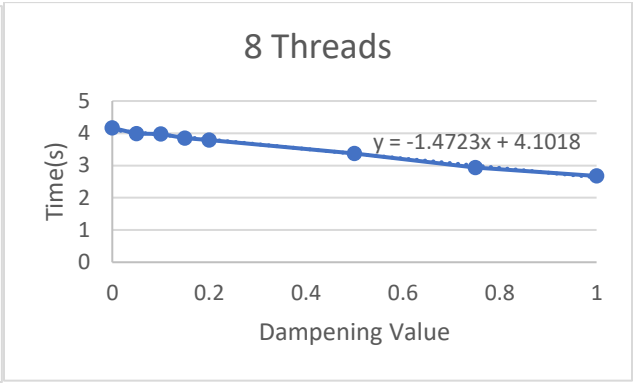


Figure 5-NotreDame (k=100)

As we can see from the different in times from both the serial and parallel runs, there was a time change as D was increased. This is likely due to the storage and usage of previous nodes making it take slightly longer to run with the changing odds of heads and tails. This is also when I discovered that 0.2 for the D value seemed to be the most stable, as there are some ups and downs in the beginning of the graph.

First 5 when changing D										
D Value	1st	PageRank %	2nd	PageRank %	3rd	PageRank %	4th	PageRank %	5th	PageRank %
0	32824	0.01410828	0	0.990139	124802	0.15777	258	0.123066	1	0.122096
0.05	32824	0.013486998	0	0.927225	124802	0.152664	258	0.114147	1	0.113908
0.1	32824	0.012668683	0	0.878131	124802	0.143426	1	0.1105	258	0.107123
0.15	32824	0.012111196	0	0.839575	124802	0.135592	258	0.112026	1	0.104382
0.2	32824	0.011324318	0	0.792391	124802	0.127327	258	0.100295	1	0.0979345
0.5	32824	0.007119547	0	0.496899	124802	0.0803308	258	0.0618799	1	0.0597707
0.75	32824	0.003604887	0	0.244173	124802	0.0413812	258	0.0312316	1	0.0296843
1	325722	6.10935E-06	325712	0.00061094	325711	0.000610939	325710	0.00061094	325703	0.000610939

Table 3- NotreDame (K=100)

As we can see from table 3, that as we increase k, we get a much weaker difference between how strong our top candidate is from the top. As we start to approach huge numbers of D (0.75=<) it almost becomes completely random. In the case of 1.0, we are always getting a random node and the distribution is almost completely random between them.

Conclusion

It is obvious from the high memory usage, diminishing efficiency, and slowing speedup, the algorithm can still be improved upon a ton. The results are impressive considering the scalability can be applied today using this algorithm and it has been extremely fun and challenging to work with. With an algorithm like this that needs so many threads, it could be a good use case for a GPU based model, or to be written in CUDA. I have been experimenting with this and have seen some impressive results although it was not ready to include statistically in the report, though I believe it would have been a fun and very insightful addition by adding a massive amount of processing units. By getting to do it with CUDA and without the ability to use C++ STL structures, I came up with more efficient data holding strategies, that were harder to implement but are much more efficient for the use case and would have implemented them in the shared memory version if I had time.