Implementation and comparison of Parallel and Sequential PageRank algorithm

Umaraj Potla
Master's in Computer Science
Georgia State University
upotla1@student.gsu.edu

Aditya Bhamidipati
Master's in Computer Science
Georgia State University
vbhamidipati1@student.gsu.edu

Purna Sai Pushkal Kalipindi
Master's in Computer Science
Georgia State University
pkalipindi1@student.gsu.edu

***Abstract*— PageRank is a fundamental graph algorithm to evaluate the importance of vertices in a graph. In this project, we are implementing the efficient parallel PageRank design based on an edge centric scatter-gather model algorithm. The vertices are first partitioned into non-overlapping vertex sets in a way that the data of each vertex set fit in the cache, then sort the outgoing edges of each vertex set based on the destination vertices to minimize random memory writes. In previous related works, authors have used various large scale real-life and synthetic datasets for evaluation. We plan to implement the parallel algorithm in CUDA and compare it with sequential PageRank implementation in C on the same dataset to compare the performance of sequential and parallel approaches of the algorithm.**

I. INTRODUCTION

A. Overview

Graph analytics plays a critical role in many applications, such as genome analysis, cybersecurity, and social networks. However, it is challenging to achieve high-performance large-scale graph analytics. To improve memory performance, we examined the prior in improving the graph layout or reordering the computation to increase locality. The PageRank algorithm was developed to determine the popularity of webpages in order to assist web search algorithms.

B. Scope

We implement the fast graph partitioning approach, which improves cache performance and eliminates random memory accesses to the DRAM. We assume that the input graph is stored in text format. Assuming each vertex set has m vertices, the graph is partitioned into the upper limit of V/m partitions; the i-th partition maintains the vertex set that includes the vertices whose indices are from im to (i+1)m-1 ($0 \leq i < \left\lceil \frac{|V|}{m} \right\rceil$).

All the edges whose origin vertices are in the vertex set of the partition are stored in the edge list; the message list stores all the messages whose destination vertices are in the vertex set of the partition. The data of each vertex is uniform in size, and hence, the memory requirement of each vertex set is identical. Edge lists and message lists can be different in size; the memory requirement of each edge list is dependent on the number of edges whose source vertices are in the corresponding vertex set; the memory requirement of each message list depends on the number of edges whose destination vertices are in the corresponding vertex set.

Assuming the multi-core platform has p cores, we divide all the computations of k partitions into p chunks and use the dynamic scheduling in CUDA to execute each chunk on one of the p cores. All the cores run the computations of distinct partitions in parallel. When a core is executing the scatter phase or gathering phase of a partition, the core can access the vertex data from its private cache, rather than from the memory or the private caches of the other cores. We compare the performance of our optimized algorithm implementation against the PageRank Base Sequential algorithm consisting of a sequential single-threaded PageRank implementation.

II. BACKGROUND

PageRank is a connection examination calculation created by Google fellow benefactors Sergei Brin and Larry Page [Brin and Page 1998, Page et al 1999]. The algorithm was initially applied to position Web pages for a catchphrase look. The algorithm gauges each Web page's relative significance by allocating a numerical rating from the most critical to the least significant page to list of pages available.

The PageRank algorithm dissects the topology of a chart speaking to joins among Web pages and yields a likelihood conveyance used to speak to the probability that an individual arbitrarily tapping on connections will show up at any specific page.

Search engines run a unique program to assess the PageRank's of websites. The program assigns a particular score to the websites, which is an indication of the importance of the page. The overall idea is: A webpage is marked as important if it is pointed towards by other important webpages. For better understanding, we assume that each hyperlink is a recommendation; thus, a webpage that has more in-links or recommendations must be valuable. If the recommendation is from another relevant webpage, then it is considered to be more valuable than recommendations from less essential webpages. Therefore, an important page is defined as the page, which (1) has many in-links, (2) has in-links from other essential pages, or (3) both.

From that assumption, It computes the PageRank value of each vertex, which indicates the likelihood that the vertex will be reached by. A higher PageRank value corresponds to more importance. When the input graph is static (i.e., does not change over time), the PageRank algorithm traverses the entire graph iteratively. In each iteration, each vertex v updates its PageRank value based on Equation (1), where d is a damping factor (0.85 is used usually); PR(i) is the rank of page i, O(j) is the number of the outlinks of page j (the number of links on page j). L(i) indicates the set of pages that link to the page i.

Equation 1:

$$PR(i) = d * \sum_{j \in L(i)} \frac{PR(j)}{|O(j)|} + \frac{(1-d)}{n}$$

III. PROBLEM STATEMENT

The PageRank of a node in a graph is indicates the importance of a node. For instance, a web graph, where nodes are webpages/websites and edges are crosslinks (incoming or outgoing; directional) between two webpages/websites. In this scenario, the PageRank of a node is the corresponding importance of that webpage on the internet (higher the better).

The PageRank description in the analytical approach.

Input: Given a directed graph G(V, E) with n vertices (aka. "nodes") and m edges. Vertices represent webpages. Edge (u,v) means there is a hyperlink from webpage u to webpage v (and hence this edge is directed from u to v).

Output: Produces the PageRank of every vertex in the graph.

IV. INPUT DESCRIPTION

The input for the program is a Google Web graph where Nodes represent websites/webpages and directed edges represent hyperlinks between websites/webpages. The data was released by Google in 2002 as a part of Google Programming Contest is used in this experiment. It has 875713 Nodes and 5105039 edges. An average cluster coefficient of 0.5143 (a clustering coefficient is defined as a measure of the degree to know which nodes in a graph tend to cluster together.). The input is a .txt file where every line represents a link between source and destination. A sample snapshot of the dataset is as follows:

```
# Directed graph (each unordered pair of nodes is saved once): web-Google.txt
# Webgraph from the Google programming contest, 2002
# Nodes: 875713 Edges: 5105039
# FromNodeId    ToNodeId
0       11342
0       824020
0       867923
0       891835
11342   0
11342   27469
11342   38716
11342   309564
11342   322178
11342   387543
11342   427436
11342   538214
11342   638706
11342   645018
11342   835220
11342   856657
11342   867923
11342   891835
```

Fig 1: Snapshot of the dataset – web-Google.txt

We used multiple subsets of real and synthetic graph datasets to evaluate the performance of Sequential and Optimized Parallel algorithms. The graph sizes vary from as low as 10 to 7500 vertices and 43 to 5.1 million edges.

V. SEQUENTIAL APPROACH

In the sequential approach, we implement iterative PageRank algorithm, which will be explained in detail in Power Iteration Method for Page Rank in the Parallel approach. The entire algorithm runs on a single CPU, which consists of steps (1) listing all the webpages from and analyze the data to find the links between them. (2) Calculate the maximum number of steps to update the distribution. (3) Iterate through page rank algorithm for every vertex in the vertex list. The distribution field keeps track of the probability of each page. Initially, all pages have equal probability: 1 / num_pages. In each step, updates redistribution of each page's probability among the pages it links to. old_distribution saves the distribution of the previous step for use during the step update and convergence check. However, all these operations are done on a single code, and aggregate data is stored in local memory, which makes it difficult to run all graph data and do the read and write operation quickly.

VI. PARALLEL APPROACH

A. Introduction to GPU and CUDA

Graphics Processing Unit (GPU) is commonly used to quicken endearing graphics pipelines, which furnish a high computational power with minimal effort. The principle contrast between CPU and GPU processing power (see Figure 2) is that a CPU is intended for a single assignment. In contrast, a GPU is intended for the Single Instruction Multiple Data (SIMD) model. Therefore, GPU endeavors to expand the number of tasks that can be acted in equal, and the latest GPUs can comprise hundreds to thousands of cores while the quantity of cores in like manner CPUs has not arrived at eight cores yet.

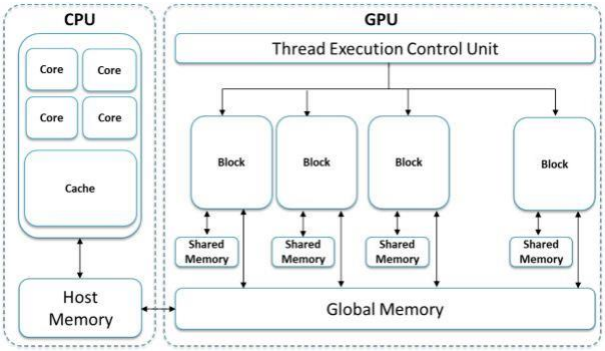


Fig 2: Comparison of CPU and GPU architecture

Today, GPUs are utilized as hardware accelerators agents for some non-graphics applications since NVIDIA presented the Compute Unified Device Architecture (CUDA) for applications, for example, matrix multiplication, framework augmentation, databases, or logical calculation. CUDA uncovered the enormous string parallelism and the quick between processor correspondence by means of the shared memory.

Likewise, it utilizes a C/C++ like interface for designers to program a GPU gadget for broadly useful calculation. A GPU gadget is commonly made out of a lot of graphics processors running from 2 to 32 in some amazing graphics cards, which have three kinds of memory (global, shared, and local). A GPU is made out of several cores that contain a memory shared by the strings of a similar square. As to parallelism, CUDA utilizes a high number of threads in a core and guarantees an intercommunication of centers by means of shared memory. All threads access to the global memory on reading/compose tasks, yet the local memory can be accessed by only its thread, while the shared memory is accessed by threads of a similar block.

B. Power Iteration Method for Page Rank

Many graph problems can be processed based on the edge-centric scatter-gather model. In this model, the computation is iterative; each iteration consists of a scattering phase followed by a gathering phase. In the scattering phase, each edge produces a message, which carries the data of the source vertex of the edge, and It is used to update the destination vertex of the edge. In the gather phase, all the messages produced in the previous scatter phase are traversed to update the corresponding destination vertices.

For Web graph containing n nodes, typically, a matrix M of n*n is designed to save the relationships between nodes. As for each element in M_{n*n}, $m_{ij}=1$, shows that the web page i has link to web page j. while $m_{ij}=0$, shows that the web page i has no link to web page j. But in actually web structure graph, n is huge, so the expression and preservation of M is difficult.

The essence of PageRank algorithm is using the power iteration method to solve the eigenvectors, which is corresponding with the eigenvalues of the state matrix.

The power iteration method merit is that the algorithm is simple and easy computer realization.

The PageRank computation can be represented in matrix linear form as follows:

Equation 2:

$$PR(i) = d \cdot \mathrm{A}^{\mathsf{T}} \cdot PR(j) + \frac{(1-d)}{n} 1$$

where d is the damping factor, n is the number of pages, and matrix A^{T} representing the transposed matrix of the transition matrix A.

The web graph is a graph designed by the nodes representing websites and directed edges to represent corresponding out-links and in-links. It is stored in a file and it is read into an n × n adjacency matrix A^T where the entries a_{ij} are equal to 0 if there is a no link from page i to page j and equal to 1 if there is a link from page i to page j. The main problem is to load the data from the file to the matrix A^T which is a very hard task due to its size: it may contain millions to billions of web pages and hyperlinks which will be impossible to load into RAM.

The original algorithm proposed in equation (1) is not designed for iterative algorithms as it is in not linear form. The updated one can be used easily for iteration. Therefore, it produces the better results.

Iterative equations:

$$PR(i)^0 = 1/n$$
$$PR(i)^1 = d \cdot A^{\mathsf{T}} \cdot PR(j)^0 + (1-d)1/n$$
$$PR(i)^2 = d \cdot A^{\mathsf{T}} \cdot PR(j)^1 + (1-d)1/n$$
$$\dots$$
$$PR(i)^t = d \cdot A^{\mathsf{T}} \cdot PR(j)^{t-1} + (1-d)1/n$$
$$\dots$$
$$PR(i)^{\infty} \to r$$

In order to understand our representation of the web graph, we give the following example. We suppose that each web page is represented by an ordinal number, and we let n = 5 be the number of pages in the web graph. The web graph gives rise to a matrix A and PR^{∞} is the matrix obtained from equation (2). More precisely, we obtain

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$PR^0_{i\epsilon(0,5)} = \begin{bmatrix} 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \end{bmatrix} \qquad PR^{\infty}_{i\epsilon(0,5)} = \begin{bmatrix} 0.1556 \\ 0.1622 \\ 0.2312 \\ 0.2955 \\ 0.1556 \end{bmatrix}$$

Where A represents the adjacent matrix of nodes, PR^0 shows the initial page ranks of nodes obtained by equation (2), and PR^{∞} represents final page ranks of nodes. Our parallel model uses the matrix representation of the PageRank algorithm in order to transform all treatments into elementary matrix operations. Each operation will be implemented in a CUDA kernel (GPU function) using threads in one or two dimensions. These kernels will be executed within a loop until the convergence condition is achieved. This condition is also executed on GPU in order to improve the performance of our proposed model.

First, we initialize the matrix A from the data file using the previous storage data format, the damping factor d, the number of pages n. Then, the data are copied to GPU. In GPU, we divide the computation steps into parallel operations computed by threads where the computation of the matrix A is done on GPU while each thread calculates one element of the matrix A, and where the computation of the matrix M is done on GPU kernel (equation (2)), while each thread calculates one element of matrix M. The goal is to reduce the sequential loop from complexity O(n) into CPU to the complexity O(1) into GPU. This is due to the simultaneous execution of n operations within GPU. Hence there are two key points which help in optimization - utilization of CUDA blocks and threads and the efficient memory coalescing.

Applied vertex smoothing onto the dataset as the vertex number is largely higher than the allowed integer bounds. Vertex smoothing is a process of adjusting the vertex numbers in such a way that they fall within the specified range. Here the vertex smoothing has been applied for the complete dataset of web-Google.txt, having more than 5M lines establishing the relation between 0.8M vertices. Since the integer ranges do not allow to take up for the enormous numbers, vertex smoothing is a must. We have applied vertex smoothing to obtain the vertex ranges for 1000, 3000, 5000, and 10000 vertices to maintaining the edges to be of the

same number. We analyze the CUDA performance for the determination of page ranks on these large datasets.

VII. RESULTS AND COMPARISON

In this section, we sketch our experimental results of parallel and sequential approach and comparison between the results. The algorithm is verified in terms of execution time on various datasets.

The performance of our parallel implementation for PageRank computing evaluated by the various real-time datasets. Some of the descriptions of the datasets used in our experiment are shown in Table 1.

Vertices	Edges	Dataset
10	22	mini-Web-Google.txt
43	66	Small-Web-Google.txt
3000	10,000	3000-10kweb-Google.txt
3000	25,000	3000-25kweb-Google.txt
3000	50,000	3000-50kweb-Google.txt
5000	25000	5000-25kweb-Google.txt
5000	50000	5000-50kweb-Google.txt
7500	1000	7500-web-Google.txt
7500	10,000	7500-10kweb-Google.txt
7500	25000	7500-25kweb-Google.txt
7500	50000	7500-50kweb-Google.txt

Table 1: Different Datasets Description

In this aspect, different data sets from mini-Web-Google, small-Web_Google, cleaner-web-Google.txt, 3000-web-Google, to 7500-50000-web-Google chosen from google. The algorithm implemented on the same data sets in both parallel and sequential approaches, and the calculated results will compare to verify execution time and the efficiency of the parallel method than the sequential method.

The numerical results of PageRanks obtained from the execution of parallel and sequential approaches are same which shows the accuracy in comparison results. The sample Pageranks results of dataset mini-Web-Google.txt which has 10 vertices with 22 edges are shown in Table 2.

Vertices	PageRank
1	0.149300
2	0.077607
3	0.075437
4	0.147130
5	0.141059
6	0.149300
7	0.061577
8	0.061577
9	0.061577
10	0.075437

Table 2: PageRank's of mini-Web-Google dataset

A. Sequential Results

The summarized results of sequential execution with the number of vertices, edges and total execution times of the sequential algorithm is shown in Figure 3. As we can see the execution time of different datasets used in our experiment. Sequential PageRank algorithm takes a minimum of 9 seconds to execute a graph of 10 vertices and 22 edges and up to 60 seconds on an average to execute a larger graph with 7,500 vertices and 25,000 edges. We observed that the sequential algorithm execution crashed with segmentation fault on executing even bigger datasets with 5000 or 7500 vertices and 50,000 edges or higher as shown in Table 3. Another observation from our experiments, is that irrespective of the number of edges, the sequential algorithm implementation took almost same execution times for fixed number of vertices in the graph.

Figure 3: Results of Sequential Execution on datasets

Vertices	Edges	Execution time
10	22	9 seconds
43	66	42 seconds
43	100	42 seconds
3000	1000	52 seconds
5000	1000	54 seconds
7500	1000	57 seconds
3000	10,000	53 seconds
5000	10,000	58 seconds
7500	10,000	59 seconds
3000	25,000	53 seconds
3000	50,000	53 seconds
5000	25000	58 seconds
5000	50000	Segmentation Fault
7500	50000	Segmentation Fault

Table 3: Results of Sequential Execution

B. Parallel Results

The results of parallel execution are shown in the Figure 4. As we can see that as the number of vertices increase, the execution time of the page rank algorithm also increases (direct Proportion). Parallel Pagerank algorithm with CUDA implementation took just 1 second or even lesser for a graph of 10 vertices and 22 edges and up to 2 minutes for larger graphs like 7500 vertices with 50000 edges. Using vertex smoothing, the complete dataset could run with a maximum of 5000 vertices, and around 5 million edges, executed in approximately 4500 seconds, which is nearly 15 minutes over an hour of execution. Also, when we look at the Table 4, we see that there is not much difference in execution time when number of iterations are taken as parameter for same set of vertices (the reason for this behavior is because of power iteration method. Using that, we converge to the result quick and the values remain constant for subsequent iterations after convergence).

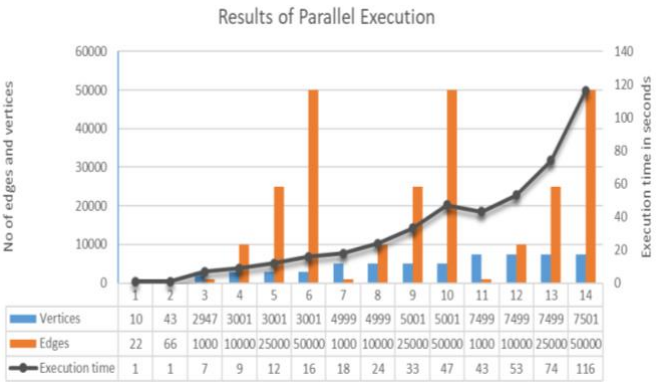


Fig 4: Results of Parallel Execution on datasets

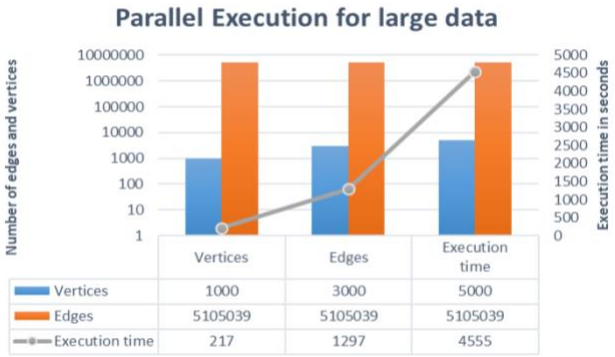


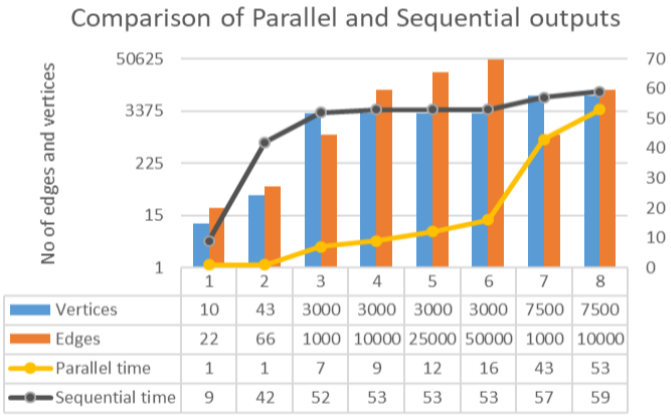
Fig 5: Results of Parallel Execution on large datasets

Vertices	Edges	Execution time	
		25 iterations	5 iterations
10	22	1 second	<1 second
43	66	1 second	1 second
2947	1000	7 seconds	6 seconds
3001	10000	9 seconds	9 seconds
3001	25000	12 seconds	12 seconds
3001	50000	16 seconds	16 seconds
4999	1000	18 seconds	18 seconds
4999	10000	24 seconds	23 seconds
5001	25000	33 seconds	34 seconds
5001	50000	47 seconds	46 seconds
7499	1000	43 seconds	42 seconds
7499	10000	53 seconds	52 seconds
7499	25000	74 seconds	72 seconds
7501	50000	116 seconds	116 seconds
1000	5105039	217 seconds	217 seconds
3000	5105039	1297 seconds	1280 seconds
5000	5105039	4555 seconds	4488 seconds

Table 4: Results of Parallel Execution

C. *Results Comparison*

In comparison between the GPU (parallel) and the CPU (sequential) implementation of the algorithm. We present the results of our experiments in terms of execution time and compare the parallel implementation with sequential implementation. Figure 6 illustrates the performance comparison of the baseline sequential algorithm with the optimized parallel algorithm. The sequential algorithm completes the execution of data of 10 vertices with 22 edges in 5 seconds where parallel algorithm completes in only 1 second as shown in Figure 6. Likewise, for data of 3000 vertices with 25000 edges the parallel algorithm took only 12 seconds while sequential took 53 seconds which is nearly 4 to 5 times than parallel. Therefore, the optimized parallel algorithm improves the execution time by 4 to 5 times in small datasets and 2 to 3 percent in large dataset when compared to the sequential algorithm. The improvement in execution time is because: (1) the partitions are accessed in a regular manner with accesses to the vertices and edge in each partition sequential in nature and (2) the data layout optimization reduces the number of random accesses while writing the messages into memory during the scatter phase. Therefore, in comparison with the sequential, the parallel algorithm significantly reduces random accesses leading to higher sustained memory bandwidth and lower execution time.



	1	2	3	4	5	6	7	8
Vertices	10	43	3000	3000	3000	3000	7500	7500
Edges	22	66	1000	10000	25000	50000	1000	10000
Parallel time	1	1	7	9	12	16	43	53
Sequential time	9	42	52	53	53	53	57	59

Fig 6: Comparison between sequential and parallel results

VIII. CONCLUSION

In this project, we successfully implemented and compared sequential and an optimal parallel Page Ranking algorithm, by efficiently utilizing the GPU architectural benefits. The parallel algorithm has been implemented with the CUDA language which gives access to massive thread parallelism and uses a high number of threads in core and assures an intercommunication of cores via shared memory. As observed in results of our experiments, clearly, the parallel approach increases the performance in terms of execution time taken to compute PageRank of a different datasets compared to the sequential version.

IX. REFERENCES

[1] Shijie Zhou1, Kartik Lakhotia1, Shreyas G. Singapura1, Hanqing Zeng1, Rajgopal Kannan2, Viktor K. Prasanna1, James Fox3, Euna Kim3, Oded Green3, David A. Bader3, 2017. Design and Implementation of Parallel PageRank on Multicore Platforms.

[2] Mauro Bisson, Everett Phillips and Massimiliano Fatica, 2016. A CUDA implementation of the pagerank pipeline benchmark

[3] [Brin & Page 1998] Brin, S.; Page, L. (1998). "The anatomy of a largescale hypertextual Web search engine", Computer Networks and ISDN Systems 30: 107–117. doi:10.1016/S0169-7552(98)00110-X . ISSN 0169-7552

[4] [Page et al 1999] Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank citation ranking: bringing order to the Web.

[5] https://snap.stanford.edu/data/web-Google.

[6] https://github.com/yuhc/web-dataset.