Optimizing PageRank Algorithm

CSC 485C/586C - Final Report

Kevin(Zhe) Chen
Yiming Sun
Xiangwen Zheng
April 21st, 2020

Abstract

As part of fulfilment to our course, we are to boost the speed of the naive PageRank algorithm with the central processing unit (CPU) and graphic processing unit (GPU). On a modern Intel 6-core workstation with a Nvidia Tesla V100 GPU, we have achieved 4.5x speed up (algorithm iteration time, excluding I/O and preprocessing; same as below) over our optimised single-core baseline using OpenMP multi-thread programming, and over 80x speed up over the baseline mode for the GPGPU CUDA programming. We observe that with multi-thread parallelization, the performance speed up remains similar as the size of the dataset (number of nodes) grows, but the performance gain keeps boosting on GPGPU parallelization.

Introduction

The rapid development of the internet over the last 20 years has changed people's lifestyle as it has become the most essential good among all the human needs that people browse and exchange content using the internet around the globe. The increase of the internet speed has caused massive data to be generated at each second. To make the browsing more productive, Google has come up with one unique graph ranking algorithm, namely PageRank algorithm, which helps their search engine deliver the most efficient information.

Given a huge set of webpage data, it can be challenging for one to retrieve the webpage with the desired media contents. If we treat such web query problems as a simple graph problem, the task would be to find the centrality of the graph network according to one's interests and deliver the best fit one with the largest centrality score. PageRank is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages [1]. Also, as Google has described, PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. PageRank is a classical graph ranking algorithm that could not only be applied to website search engines, but also applicable on many other applications such as to evaluate the "importance" of a person in a social network, or to evaluate the weight of a package in software dependency networks.

The reason that most drives us to dive into this problem is the popularity and impact on today's browsers and search engines. Once the number of pages increases massively, the more time and memory will be taken. Our purpose is to optimize it as much as possible.

Algorithm Description

Original Formula

In this project report, we will be focusing on the simplified version of the PageRank algorithm. Figure 1 represents the relationships between several web pages using a simple directed graph with nodes being the web pages and edges being the citation relationships. In the given graph, node A is pointing to node B and node C, node B is pointing to node D, node C is pointing to node A, node B, and node D. In a practical context, the content of nodes will contain sufficient information about the pages, which will not be considered in our case due to a simplified introduction.

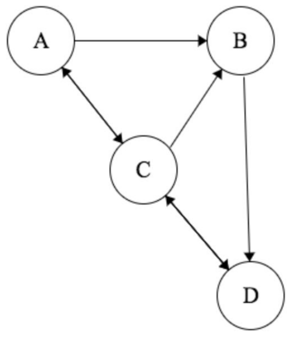


Figure. 1 simple graph example

A naive formula that was consistently used in PageRank algorithm:

$$PR(A) = \frac{PR(T1)}{C(T1)} + \ldots + \frac{PR(Tn)}{C(Tn)}$$

where PR(A) indicates the PageRank score of node A, T1...Tn are the set of nodes that directly point to node A, C(A) refers to the number of outgoing pointers from node A.

At the initial state, the page score will be 1/n for every page node, where n is the number of the page nodes. The score indicates the rank of the page thus gives importance and priority to each page node. In the next iteration, we need to consider how many nodes are pointing to the current node. For node A, there is only pointed by one node which is node C. The PageRank score of node C from the last iteration is $\frac{1}{4}$, and node C has three outgoing pointers. The PageRank score of node A will be calculated as:

$$PR(A) = \frac{1/4}{3} = \frac{1}{12}$$

Using the same methodology, scores after the second iteration for nodes A, B, C, D are $\frac{1}{12}$, $\frac{2.5}{12}$, $\frac{4.5}{12}$, $\frac{4}{12}$ respectively. The number of iterations varies depending on the problem's context and practical situation. When the PageRank scores do not change significantly overall, the iteration can be stopped and finished. The process should stop under a normal circumstance when PageRank score differences are smaller than 0.001.

The simplified implementation of the PageRank algorithm does not cover the Table 1 shows the result after three iterations:

Iteration	0	1	2	PageRank
A	1/4	1/12	1.5/12	1
B	1/4	2.5/12	2/12	2
C	1/4	4.5/12	4.5/12	4
D	1/4	4/12	4/12	3

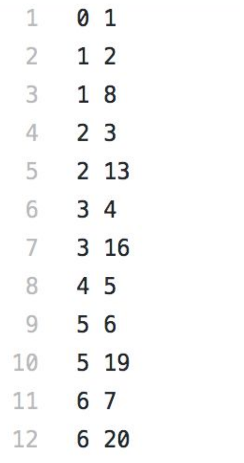
Table. 1 results after three iterations

After introducing the fundamental idea of this algorithm, we are going to utilize it in a more complicated context. Here is the pseudocode of this algorithm:

```
for iteration in range of num_iteration:
    if the iteration is the first one:
        set all the (current) scores to 1/n;
    else:
        set all the previous scores with corresponding scores;
        set all the scores to 0;
        for node A in all the nodes:
            for node B in node A's predecessors:
                A.score += B.prev_score / B.num_successors;
```

In the provided example, the iteration number is constant and large (e.g., 500 or above). In real-world problems, the iteration will be done until the results (PageRank score) roughly converge to constant numbers. The reason in our case we did not do this way is because we want to compare the difference of execution times for doing the same amount of work.

The input can be fairly simple, we choose the edges list contained in the text file to be our input format with a list of two-column number sets separated by space where the first column represents the source nodes, and the second column represents the target nodes that are pointed. The output is the list of page nodes along with their final PageRank scores. Figure 2 is an example of a small size test file. Frankly, the order of the nodes is automatically sorted ascendingly.



```
0 1
1 2
1 8
2 3
2 13
3 4
3 16
4 5
5 6
5 19
6 7
6 20
```

Figure. 2 simple test file example

Matrix Based Implementation [3]

To turn the formula mentioned above into a program, one way is to construct the graph data into a 2-dimensional array based transitional matrix for computing PageRank scores. As we store the scores of every node into one dimensional array, we can also treat it as a 1xN matrix. The score then can be updated by simply multiplying with the transitional matrix.

The transitional matrix contains the entries a_{ij} as described below:

$$\bar{a}_{ij} = \begin{cases} \frac{1}{L(j)} & \text{if there is a link from node } j \text{ to node } i, \\ \frac{1}{N} & \text{if node } j \text{ is a dangling node,} \\ 0 & \text{otherwise,} \end{cases}$$

Suppose we have a example network:

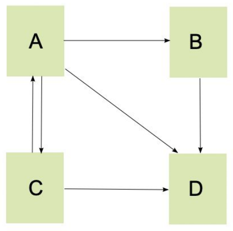


Figure. 3 network example

By applying the rules we defined above, the initial scores and transitional matrix will look like:

$$\begin{bmatrix}1/N\\1/N\\\vdots\\1/N\end{bmatrix} \quad \begin{bmatrix}0 & 0 & 1/2 & 1/4\\1/3 & 0 & 0 & 1/4\\1/3 & 0 & 0 & 1/4\\1/3 & 1 & 1/2 & 1/4\end{bmatrix}$$

And if we compute our scores with the matrix multiple times we will have:

iteration	x_A	x_B	x_C	x_D
0	0.25	0.25	0.25	0.25
1	0.75	0.5833	0.5833	2.0833
2	0.8125	0.7708	0.7708	1.6458
3	0.7969	0.6823	0.6823	1.8385
4	0.8008	0.7253	0.7253	1.7487
5	0.7998	0.7041	0.7041	1.7920

Table. 2 results at each iteration

For the example shown above, we will take results at the fifth iteration as the final scores.

Design Decisions

Naive Implementation

We applied a Struct of Array (SoA, using vector instead of array) data structure as our naive implementation, which turns out to be slow:

```
struct Nodes
{
        vector<vector<int>> nodesFrom;
        vector<int> countTo;
        vector<float> score;
        vector<float> scorePrev;
};
```

Code snippet. 2 SoA

Optimized Single-core and CPU Parallelism Implementations

We then applied a matrix-based (2-D array) data structure as our optimized single-core implementation, as the algorithm described in the previous section. By transforming the Page Rank algorithm into a set of matrix operations, we have managed to make our implementation multi-core parallel as well. There are essentially two main parts of our implementation of the algorithm: *entry_update* section and *cal_pagerank* section.

The *entry_update* section, as shown in the pseudocode snippet below, updates the probability of the entry from each page to every other page from the visited matrix we extracted from the raw inputs.

```
entry_update:
#pragma omp parallel for num_threads( NUM_T )
for node A in all the N nodes:
    for node B in all the N nodes:
        if node B has no outgoing node:
            set entries from A to B to 1 / N
        else if node A has outgoing to node B:
            set entries from A to B to 1 / B.outgoing_degree
        else:
            set entries to 0
    end for
end for
```

Since this algorithm does not have any shared data, we can easily apply the OpenMP[2] *pragma omp parallel for* as shown in the above pseudocode in first for loop to make the whole section parallel and theoretically reduce the execution time without worrying about any race condition.

The *cal_pagerank* section, as shown in the pseudocode snippet below, calculates the score of each page node in the score matrix by doing matrix multiplication with the updated entries matrix we have mentioned previously. This process gives up one iteration of the PageRank calculation. To have it calculated multiple iterations, we will have to wrap it with an outer for-loop.

cal_pagerank:
```
for num_interation of times:
    initialize 2d vector old_scores
     /* store our last scores in old_scores matrix */
    #pragma omp parallel for
    for node A in all the N nodes:
        set old_scores of A to last_scores of A
    end for
    initialize sum to 0
    #pragma omp parallel for reduction( +:sum ) num_threads( NUM_T )
    for node A in all the N nodes:
        set sum to 0
        for node B in all the N nodes:
            sum +=B's  old_scores * entries from A to B
        end for
            Update A's score to damping_factor * A's old_scores + (1-damping factor) * sum
    end for
end for
```

In the original implementation, we noticed that we would have to do the matrix multiplication, and basically we will have to multiply two values and add it to a sum. So in order to protect the sum, we used OpenMP reduction technique that prevents each thread from blindly accessing this variable while it is already occupied, in our case, we mitigated the race condition for each thread to access the sum variable and want to perform multiplication as shown in the pseudocode above.

GPGPU Parallelism Implementation
Recall that our PageRank algorithm has two critical stages: 1. initializing the transition matrix (update_entries()), which identifies and handles the dangling nodes; 2. updating the transition matrix and updating the scores through iteration (page_rank()). We consider applying slightly different techniques to achieve GPU parallelism, based on the data accessing flow in each of the stages. We provide pseudocodes to demonstrate the kernel functions and explain how the functions should be called by the host in detail. The complexities of both stages are both O(m^2).
In kernel:

```
update_entries( trans_m, vis_m, out_t, N ):
    i = ceil( idx / N )
    j = dix / N
    if out_t[ j ]=0:
        trans_m[ i ][ j ] = 1 / N
    else if vis_m[ j ][ i ] = 1:
        trans_m[ i ][ j ] = 1 / out_t[ j ]
```

where trans_m is an N x N transition matrix (Hidden Markov matrix which holds the probability for each node to visit every other node), vis_m is an N x N visited matrix that represents whether a node has a connection to another node, out_t is the size N outgoing table (number of outgoing nodes for each node), N is the number of nodes, and idx is the index of the thread.

The task is to update the entries in the N x N transition matrix, thus we launch N^2 threads in a single kernel function call from the host. Although the data in all the parameters are shared by all the threads, since each thread only modifies their own designated entry (with matrix indexes i and j calculated from the thread index) in the transition matrix, the process of updating entries is considered thread-safe (i.e. no race conditions).

In Device Kernel:

```
pagerank( socre_t, old_score_t, trans_matrix, d, N ):
    i = idx
    sum = 0
    for j in 0 to N-1:
        sum += old_score_t[ j ] * trans_m[ i ][ j ]
    score_t[ i ] = d * old_score_t[ i ] + (1-d) * sum
```

In Host:

```
for each iteration:
    old_score_t = score_t
    /* call pagerank() with N threads. */
```

where score_t is the score table and old_score_t is the previous-iteration score table (both are 1-D arrays with a length of N), d is the damping factor.

In this stage, we update the score of each node by summing up each of its neighbors' previous score multiplied by the probability of going to that neighbor. In this time, we only assign N threads, for the N nodes. To avoid dealing with the summation with parallelism, we adopt this kind of simplified implementation, instead of using N^2 threads like in the previous stage. The main reason is we have not explored how to use atomic addition in kernel functions. An alternative way is creating another N x N score matrix, and fill in the sub-score for each node to each of its neighbors in the kernel call with N x N threads, then sum up the sub-scores and transform to our score table in host function when each iteration is done. This way seems trivial and not likely to improve the runtime by too much since we have already reached very good performance with our current implementation.

Experiments

Datasets and Experiment Setup

The datasets are obtained from Stanford Large Network Dataset Collection, and they are all Gnutella peer to peer network datasets, from different months of 2002.

ID	Dataset	Nodes	Edges
0	p2p-Gnutella08	6,301	20,777
1	p2p-Gnutella05	8,846	31,839
2	p2p-Gnutella04	10,876	39,994
3	p2p-Gnutella25	22,687	54,705
4	p2p-Gnutella24	26,518	65,369
5	p2p-Gnutella30	36,682	88,328

Table 3. Datasets

We perform tests on Nvidia Tesla V100 GPU with 16 GB of memory. We fix the damping factor to 0.85, and the number of iterations to 10. For the GPU parallelism, the number of threads per block is fixed to 512.

For each of the naive, baseline, CPU and GPGPU implementations, we recorded the **total runtime**, and time spent on executing the computational iterations (**PR calculation time**). We consider the speedups from the baseline to CPU and GPGPU as the ratio of the PR calculation times, for running each dataset, because we would not consider the effects of I/O and pre-processing in this case.

Experiment Results

Figure 4 shows the runtimes for the naive implementation. From the results, the PR calculation times are almost 100% of the total runtimes, and the growth is almost linear.

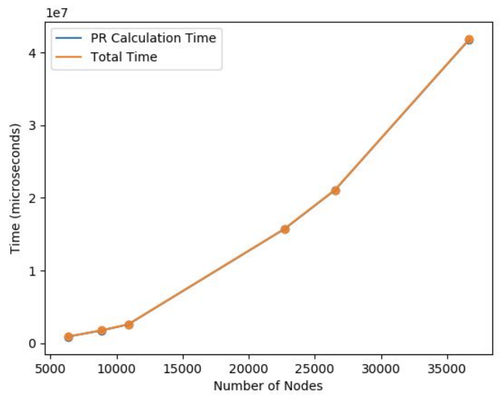


Figure. 4 Naive: single-core implementation

Figure 5 shows the runtimes for the optimized single-core implementation, which is the baseline for examining the optimization of the parallelisms. Compared to the results of the naive implementations above, we gained a 2x speedup. The time spent on I/O and preprocessing has become noticeable.

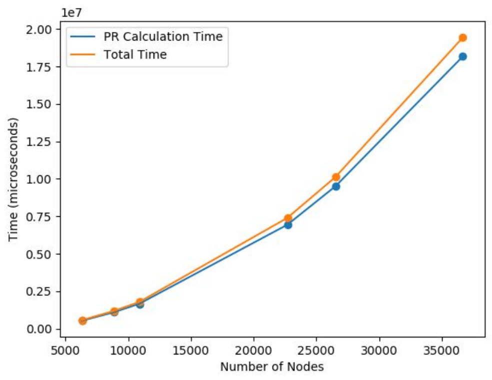


Figure. 5 Baseline: single-core optimized implementation

Figure 6 and and Figure 7 shows the runtimes for the CPU and GPGPU multi-core parallelism implementations, respectively. Figure 8 compares the PR calculation times among the 3 models: baseline, CPU and GPGPU, where the bars represent the runtimes, and the lines represent the speedups. The x-axis is the ID of the dataset (sorted by number of nodes), which does not have a linear relationship to the number of nodes. As we can see, the speedup for the CPU remains at about 4.5x no matter how the size of the dataset grows, while the speedup for GPGPU increased from about 40x to 80x.

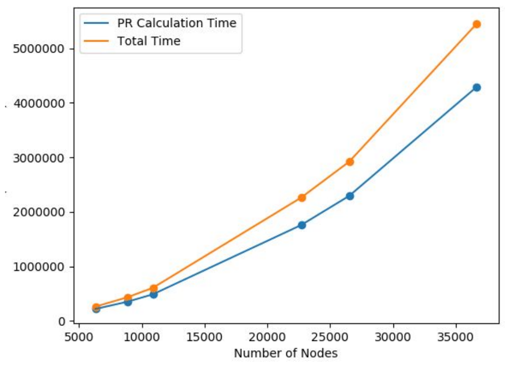


Figure. 6 CPU implementation

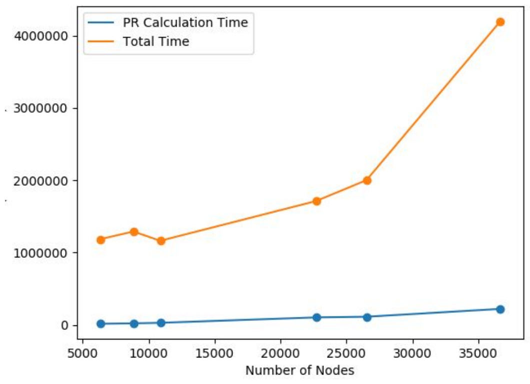


Figure. 7 GPGPU implementation

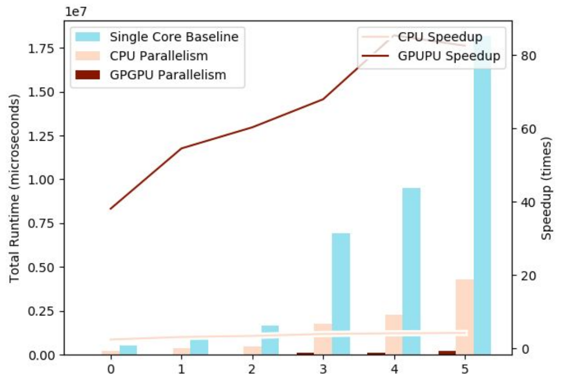


Figure. 8 Compare the speedup for CPU and GPGPU models

Discussion

During the early stage, we have designed and implemented a std::map based baseline model which deals with pointers and thus made the programming speed quite fast compared to the matrix implementation. However, the goal of this course is to expose ourselves to the very low level of the system and we argue that with matrix implementation using all primitive data types, we can have more room to deal with the parallelism. Another reason we have made the changes from map based to matrix based is because it is impossible to perform GPU parallelization with vector and map. For two reasons stated above, we have been developing the optimization on the matrix based implementation throughout the term project.

References

[1] GeeksforGeeks: https://www.geeksforgeeks.org/page-rank-algorithm-implementation/. (last visited on 02/22/2020).

[2] OpenMP website: https://www.openmp.org/. (last visited on 03/13/2020)

[3] K. Shum. Notes On Pagerank Algorithm. http://home.ie.cuhk.edu.hk/~wkshum/papers/pagerank.pdf. (last visited on 02/22/2020)