# Optimizing PageRank Algorithm

CSC 485C/586C

Kevin(Zhe) Chen
Yiming Sun
Xiangwen Zheng
Mar 12, 2020

## Interim Report 1: Single-threaded Optimization

### 1.1 Introduction

Given a huge set of website data, it can be very difficult for one to retrieve the webpage with the desired media contents. If we treat such web search problems as a simple graph problem, the task would be to find the centrality of the graph network according to one's interests. 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, etc.

The existing algorithm we found has several problems: the running time for a large dataset can be improved and the cache hit rate is low. The area that will be investigated is the data structure using and the algorithms. The existing algorithm utilizes an array of struct (AoS) to restore the data, and some of the attributes (properties) of the objects are not even frequently used. Also, the tiling technique is possible to be implemented in the main algorithm to maximize the cache hit rates when the code is iterating through the nodes.

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.

### 1.2 Algorithm Description

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.

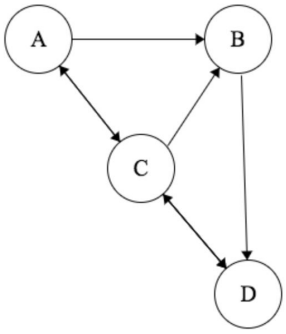


*Figure. 1 simple graph example*

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 a number. The reason in our case we did not do this way is because we want to compare the difference of performance and the perf results when the iteration number is different.

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*

In terms of storing and structuring the input data, AoS is the data structure being used. Firstly, an object called Node is used to contain the page information, which includes id (page ID), a node vector called nodesFrom that contains a list of nodes point to this node, an integer type called countTo that counts the number of this node is pointing to, a variable called scorePrev that stores the PageRank score of this node from the previous iteration, and the variable score that stores score in current iteration. The following is how the array of the structure looks like:

```
using Id = int;
using Count = int;
using Score = float;

struct Node
{
    Id id;
    vector< Node* > nodesFrom;
    vector< Node* > nodesFrom;
    Count countTo;
    Count countFrom;
    Score scorePrev; // Score in
                     // iteration i-1.
     Score score; // Score in iteration
i.
};
```

*Code snippet. 1 AoS*

### 1.3 Experiment Setup

In order to test the algorithm for its accuracy and correctness, only the time of the algorithm part will be measured, instead of including the time measurement of reading files and storing dataset. Additionally, several different datasets with different cases are necessary. For example, ordered data, unordered data, dangling type data, and

sink type data. Firstly, a fair large test file is necessary for stress tests. Therefore, we planned to generate around twenty different text test files with different sizes of nodes inside from 5 to 40,000 nodes. At the current stage, the focus was put on a 10,000-nodes and a 50,000-nodes dataset, both with 500 iterations. All the connections and directions between nodes are totally random. The test files with small sizes will make sure that the algorithm is correct and doing the right things, and the content of the text files are easy to be changed for testing in different cases. As we observed, the running time and cache hit results can be varied if the test files are small. Therefore, performing in a larger dataset is crucial for visualizing the difference before and after optimization. In the Linux-Perf[3] command, the following two figures (Figure 3 and Figure 4) are generated respectively for AoS and SoA. For our experiment, instruction per cycle, branch miss rate and L1-cache miss rate are the focus.

```
     124.40 msec task-clock                #    0.935 CPUs utilized
          2      context-switches          #    0.016 K/sec
          0      cpu-migrations            #    0.000 K/sec
        365      page-faults               #    0.003 M/sec
  468,918,147      cycles                    #    3.770 GHz                      (48.82%)
  296,173,339      instructions              #    0.63  insn per cycle           (61.68%)
   75,956,895      branches                  #  610.604 M/sec                    (61.68%)
    4,469,435      branch-misses             #    5.88% of all branches          (61.68%)
   99,741,314      L1-dcache-loads           #  801.802 M/sec                    (54.39%)
   20,598,975      L1-dcache-load-misses     #   20.65% of all L1-dcache hits    (25.46%)
   14,034,584      LLC-loads                 #  112.821 M/sec                    (25.72%)
       22,435      LLC-load-misses           #    0.16% of all LL-cache hits     (37.27%)

  0.133052256 seconds time elapsed

  0.116719000 seconds user
  0.008049000 seconds sys
```

*Figure. 3 AoS (Baseline) Perf result*

```
      70.11 msec task-clock                #    0.864 CPUs utilized
          5      context-switches          #    0.071 K/sec
          0      cpu-migrations            #    0.000 K/sec
        306      page-faults               #    0.004 M/sec
  252,824,111      cycles                    #    3.606 GHz                      (51.60%)
  344,384,679      instructions              #    1.36  insn per cycle           (64.54%)
   52,049,147      branches                  #  742.349 M/sec                    (64.54%)
    2,317,033      branch-misses             #    4.45% of all branches          (65.80%)
   96,485,671      L1-dcache-loads           # 1376.124 M/sec                    (48.40%)
    8,405,182      L1-dcache-load-misses     #    8.71% of all L1-dcache hits    (22.80%)
    1,099,125      LLC-loads                 #   15.676 M/sec                    (22.81%)
          664      LLC-load-misses           #    0.06% of all LL-cache hits     (34.76%)

  0.081160789 seconds time elapsed

  0.058018000 seconds user
  0.012432000 seconds sys
```

*Figure. 4 SoA Perf result*

For the future experiment, special cases of the dataset will be tested for more details and purposes. The first special case is called dangling node case, which means there are some nodes in the dataset that are not pointing to any other nodes. The second special case is called the sink node case, which means there are some nodes in the dataset that are not pointed by other nodes.

### 1.4 Single-threaded Optimization

In our results from the baseline, we noticed that the L1 cache miss ratio is significantly high (20%), thus increasing the spatial locality is the priority. In the conventional AoS data structure, the data that we access frequently (i.e. the scores) is not contiguous. As a result, the data that we access next is not likely loaded in the cache line.

So, as our first optimization step, we decided to change the data structure that stores nodes from AoS to the struct of array (SoA). The key is that, even though the access of nodes is random, all the scores are stored in the same array (vector), so it is ensured that we can load

as many scores to the cache as possible. Compared to Code snippet 1 in the Algorithm Description section, Code snippet 2 is the SoA implementation. Some attributes are deleted here because they are not going to be used at all, and it would be a big waste if those irrelevant data is kept operating. Figure 3 and Figure 4 are the Perf results when AoS and SoA are used.

```
struct Nodes
{
        vector< vector< Id > >
nodesFrom;
        vector< Count > countTo;
        vector< Score > score;
        vector< Score > scorePrev;
};
```

*Code snippet. 2 SoA*

In addition, we have explored an alternative way to store the current scores and previous scores with only one vector. For a graph of N nodes, use a vector of size 2*N. For each node with ID = i, the index of the previous score of such node is 2*i, and the index of the current score is 2*i + 1. The purpose is that since we assign the previous scores with the current scores, and the current scores back to 0's in every iteration, putting each current score next to its corresponding previous score should theoretically increase locality. However, the results turn out to be making no difference than the two-vector-score structure, so the idea will not be discussed further.

### 1.5 Overall Comparison

The following table (Table 2) is generated from the Perf results we got. In this case, we compare when the dataset sizes are 10,000 and 50,000. As we observe (from Figure 3 and Figure 4) in the size interval between 10,000 and 50,000. The number of branch misses, the number of cache misses, and runtimes increase almost linearly, which is expected because there is much more work to do as the dataset has greater size. But switching data structure from AoS to SoA significantly improves the code performance. Since every time we ran the perf, the results vary a little bit, so the result we take the average out of ten attempts. The branch miss is not improved much because our branch situation is rare, there are not many chances to improve for this part. Clearly, SoA reduces the cache miss massively, it is over a half less compared to AoS. That is expected for switching AoS to SoA since SoA's feature is maximizing the cache hits and improving the performance. The running time reduced to 1/3, which is also desired.

| | AoS | | SoA | |
|---|---|---|---|---|
| | N = 10,000 | N = 50,000 | N = 10,000 | N = 50,000 |
| Instruction / Cycle | 0.63 | 0.57 | 1.36 | 1.36 |
| Branch Miss Rate | 5.88% | 6.39% | 4.45% | 5.60% |
| L1 Cache Miss Rate | 20.65% | 22.29% | 8.71% | 10.21% |
| Runtime (microseconds) | 98,938 | 516,765 | 36,902 | 183,370 |

*Table. 2 Benchmarking*

### 1.6 Diagnostics

From the experimental results, back-end (memory) boundness is the primary cause of latencies. In the beginning stage of our project, we implemented a full Node struct which allows a page node to have complete access to both its incoming and outgoing node lists. However, we identified that some of the data were not accessed by the algorithms at all, hence, we have removed those cold data in order to reduce the memory load. and After implementing our baseline of naive PageRank algorithm, by using both Linux-Perf and our manual inspection, we have noticed that the L1 cache miss ratio to be more than 20%. This is because our code accesses data in different places in the Node vector numerous times during the computation iteration process which further affects the program spatial locality performance.

### 1.7 Limitations and Future works

As we have discussed from the sections above, our works have been proved to generate several observable improvements over the baseline algorithm. However, due to the limited timeframe set on this project, there are several paths we have not yet done but would like to try to improve our current works. Our work tested on very little input data and only mainly focused on the 10,000-node and 50,000-node ones. To enhance our findings and statements, in our next step, we would like to generate and test input sets with differences in the number of nodes/edges and max degree (number of outgoing edges). The current implementation on the PageRank algorithm is experiencing a technical issue which results in inaccurate PageRank scores due to an edge case regarding dangling nodes that holds no outgoing edges, which makes the score of such node to be not accurate. In order to solve this problem, we plan to apply matrix operation on the PageRank Algorithm introduced by Kenneth Shum [2]. For our existing implementation, there is very little chance we could take the advantage of tiling technique we learned from the class, but this will be discussed more for the future step where we have adopted the matrix version of the implementation of the PageRank algorithm.

### 1.8 References

[1] GeeksforGeeks: https://www.geeksforgeeks.org/page-rank-algorithm-implementation/. (last visited on 02/22/2020).
[2] Kenneth Shum. Notes on PageRank Algorithm 1 Simplified. http://home.ie.cuhk.edu.hk/~wkshum/papers/pagerank.pdf . (last visited on 02/23/2020).
[3] Linux-Perf manual. http://man7.org/linux/man-pages/man1/perf.1.html . (last visited on 02/19/2020).

# Interim Report 2: Multi-threaded optimization

## 2.1 Introduction

Often when a single-threaded program hits a bottleneck that comes from CPU computing power, one would optimize its program to fit into multi-core/multi-thread development. By adding more cores/threads into the computation, one would expect every core/thread to work in a parallel fashion and make the program to have a significant improvement over the sequential instruction performance.

In this report, we will include a discussion about the optimization of utilizing multiple-threaded and parallelism methodologies. This report consists of six sections. Section 2.2 describes our continued works on refining the baseline model including introducing a matrix version of the implementation of the algorithm and improving score precisions by fixing the dangling node problem. Section 2.3 describes how we could take advantage of multi-core CPU parallelism into our implementation of PageRank. Section 2.4 is our experiment setup. Section 2.5 compares the parallel scalabilities across different levels of parallelism with the control of the numbers of threads used. Section 2.6 shows evidence of work efficiency. Section 2.7 describes the problems we have encountered and the limitation of our implementation using a 2d vector to represent the matrix. And section 2.8 is future work.

## 2.2 Baseline Refinement

2.2.1 Deal with the Dangling Nodes

In our previous implementation, we did not deal with the dangling node problem, and the sum of the scores was not 1 (usually converged to about 0.2). Dangling node is the node that has zero degrees, that is this node has no outgoing links. In our case, the dangling node is the webpage that does not point to any other pages. In the PageRank algorithm, we define it to have the possibility of 1/n to be visited. Even though there are no links to those nodes, they are still possible to be visited. Therefore, before the code starts initializing or updating the content in certain data structures (in our case, using SoA and 2D array matrix), the scores of those nodes that have zero entry will be set as 1/n where n is the number of nodes.

In pseudocode, here is how we modify the nodes' information, before running the PageRank iterations, for the SoA implementation:

```
for node A in all the N nodes:
    if the count of the outgoing links for A is 0:
        set the count of the outgoing links for A to N
        for node B in all the nodes:
            add A to B's predecessor list
        end for
end for
```

Applying this modification to the implementation, the runtime increased hugely: the refined version is about 1400 times slower than the previous one, when

testing on the 10,000-node dataset, as there are actually over 6,000 dangling nodes. The reason is that we have forced the time complexity to be strictly O(n^2). Therefore, we decided to reimplement the algorithm, in a faster and thread-friendly method, with a 2D matrix.

### 2.2.2 Implement with 2D Matrix

In order to optimize the performance for calculating scores for every iteration. Matrix version of the data structure is introduced to use in our example. In the C++ programming language, we can conveniently use the 2d vector to store and represent our entry matrix. By having our data in a matrix format, it would give us the advantage in multi-core / multi-thread parallelism. The reason for it is when we have both scores table and entries table in the matrix, we can perform matrix operation and based on the nature of it, we can implement a true parallelized code that dispatches parallel instructions to each of the threads to go through the Page Rank algorithm.

The dangling node problem is also solved in the matrix version, as follows:

```
if ( table->num_entries[ j ] == 0 )
{
        table->ij_entries_matrix[ i ][ j ]
= 1.0f / N;
}
```

*Code snippet. 1 Dangling node solution*

### 2.2.3 Split into Hot-cold Data

Additionally, the hot-cold data structure is added for further optimization and reduces cache miss rates. Separation of Hot and cold data is the optimization technique that classifies the frequency of accessing data in cache or storage. Hot data is the data that will be accessed more frequently, and cold data is the data that rarely be used. Following is the code fragment of hot-cold data structure that has been used in our case:

```
struct Tables_Hot
{
    Score score; // 8 bytes
    std::vector< Entry > entries_col;
};
struct Tables_Cold
{
    Count num_entry; // 4 bytes
    std::vector< Count > visited_col;
};
struct Matrix_soa
{
    std::vector< Tables_Hot > hot;
    std::vector< Tables_Cold > cold;
};
```

*Code snippet. 2 Hot-code data structure*

Clearly, our purpose is to calculate the score for every page and rank them up. Therefore, the frequent use of data will be score and nodes that this node is pointing to (the entries of this node). These two data are set to be in hot data. In comparison, the number of node entries and the vector that is used to record down visited nodes are not frequently used, so they are stored in cold data. After these two steps, we put these two groups of data into a Matrix SoA.

### 2.2.4 Asymptotic Complexity

For the computation complexity of one's algorithm to be asymptotic, it will have a limiting behavior of its execution time of the algorithm when the size of the input data grows larger. In our matrix implementation of the Page Rank algorithm, we have tested several

different sizes of the input data on our algorithm and as the size or number of iterations increase, the execution time also grows accordingly as shown in Table 1.

| Number of Nodes | Runtime (us) |
|---|---|
| 10 | 5 |
| 100 | 120 |
| 1,000 | 13,864 |
| 10,000 | 1,798,538 |
| 20,000 | 7,742,427 |
| 50,000 | 64,286,037 |

*Table. 1 Runtimes for different dataset*

### 2.3 Parallel Algorithm Description

By transforming the Page Rank algorithm into a set of matrix operations, we have managed to make our implementation parallel. 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[1] *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 Page Rank 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 have noticed that we would have to do the matrix multiplication, and basically we will have to add 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.

**2.4 Experiment Setup**

The experiments were performed on the 6-core Azure cloud instance. We compiled our codes with optimization level Og, and C++17. We chose to keep using the 10,000-node dataset, and only vary the number of threads.

Since in our C++ code, the number of used threads can be simply controlled by changing the variable number in pragma code. It is useful to test through different numbers of threads and compare their results to see the best optimization and its limits.

At the beginning of the experiment of multithreading, we have noticed that there is a significant performance overhead due to the initialization of threading. As Figure 1 shows, the difference in the number of instructions, from the original sequential program (0 thread) to thread-enabled program (1 thread), is 0.75 billion. The difference in the running time is less significant: only 64 ms.

In section 2.5, we will further determine the effects of adding more threads in the program. At this point, we can assume that the program will have the best performance as the number of threads reaches 6, the same as the number of cores on the instance. Due to the overhead, the best acceleration on the running time should be less than 6 times, but close.

**2.5 Parallel Scalability**

The number of threads from 1 to 10
For better visualization and comparison, we generate the line graph to show the change. The two following line graphs (Figure 1) show the results that are generated when different numbers of threads are used. The thing needs to be noticed is that we test the codes both in our local machine (4-core machine) and in the 6-core Azure server. The graphs show the result from the 6-core server.

After monitoring both two cases, we can observe that the runtime drops significantly when the number of threads approaches to 6, which is expected because the 6-core server is using for this case. And after 6 threads, the results are somehow converging into a consistent runtime interval, which is near 370 ms. It is not a linear change, it is more like a log graph, which means the change is getting smaller and smaller when the thread

number approaches to the core number. And for a 4-core machine, runtime goes to nearly flat after using 4 threads. For 6-core instance, the runtime performance has been optimized nearly 5 times (from 1,906 ms to 349 ms), which is really impressive and matches the expectation and estimation from 2.4.

For the number of instructions, the change is not really obvious. The number of instructions is distributed nearly in the range of 16 billion to 16.8 billion. This is also expected because using multiple threads does not reduce the amount the work is done, but distribute the work to different threads and they execute them together.

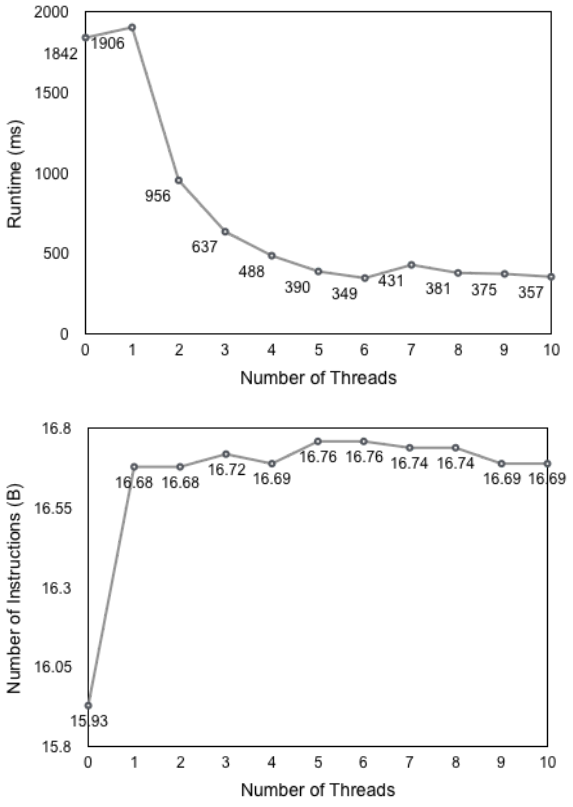


*Figure. 1 The effects of parallelism*

### 2.6 Evidence of Work Efficiency

From the above, we can simply prove the boost of work efficiency by applying parallelism with the two metrics: runtime of the algorithm part, and the total number of instructions for running the program. By comparing the state-of-the-art baseline single-threaded implementation (0 thread) to the best-performance 6 thread parallelization, we have achieved an acceleration of 5.27 times (1842 ms / 349 ms), with only adding 4.7% work overhead (15.93 B to 16.68 B instructions).

### 2.7 Limitations

As mentioned section 2.2, both the time complexity and the space complexity of the matrix implementation of PageRank are O(n^2), which is obvious: for processing an N-node graph, we need a N x N matrix to store the information of the graph, and we need to access and update all the entries in the matrix in every iteration.

The graph with the maximum number of nodes that our program can process (on our local machine) so far is a 50,000-node graph. In terms of real memory usage, processing such a graph requires 10 GB of memory (4 x 50,000 x 50,000 as each float entry that represents the probability would take 4 bytes of memory). In this condition, we failed to process a 100,000-node dataset, as when the number of nodes doubles, the required memory increases to 4 times (40 Gb), which exceeds the physical memory capacity on the machine.

### 2.8 Future Works

As we turned the Page Rank algorithm into a set of matrix operations, we have observed a huge improvement in the performance just by making our implementation parallel with multi-core/multip-thread. One future path is to take advantage of hundreds or thousands of cores in a graphics processing unit that is able to accelerate floating-point computation even further.

### 2.9 References

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