Parallel PageRank Estimator Using Adjacent List

Jin Tao 11474660

Introduction

The rapid growth of the World Wide Web is posing greater challenges to searching within the document or web page collections based on specific queries and extracting useful information. The World Wide Web can be viewed as a special document collection which is huge, dynamic, self-organized and, most importantly, hyperlinked. Currently the Indexed Web contains billions of pages and it keeps growing rapidly, roughly, doubling every 16 months. Based on the expectation, only a few pages would appear in response to the user's query, among which some are much more relevant than others. The linking feature is very essential to web search engines and hyperlinks enable focused and effective search. How to leverage its link structure and properties for better information retrieval has been paid great attention to and a lot of web page ordering research is being done. The most popular algorithm is Larry Page's PageRank algorithm which effectively utilizes the link structure of the Web to generate a global "importance" ranking of each web page and guides search and navigation of better quality. It has become a natural part of modern search engines and is widely applied to web spam detection, crawler configuration and so on. The main idea of the PageRank algorithm is the propagation of importance from one Web page towards others through its out-going hyperlinks. Efficient computation of PageRank is required because of the huge size of web graphs and parallelization can be applied to accelerate the computation of PageRank. PageRank inherently is massively parallelizable due to the strict host-based link locality of the Web. Each web surfer is independent on each other and each performs a series of targeted search based on different queries. Due to this property, the PageRank can be parallelized based on the independence of its surfers. In this report, I introduce a new approach to computing the PageRank is a parallel fashion. To be detailed, I design, implement and test a multithreaded parallel estimator for PageRank calculation using OpenMP and also design the algorithm for the distributed memory version using MPI.

Problem Statement

Input: Graph G(V, E) of web pages containing |V|=n nodes (webpage) and |E|=m edges

Output: A sorted array pr_sorted based on pr which contains n elements: pr[0], pr[1], …pr[i]…pr[n-1]; where pr[i] = the pagerank estimated for vertex i

Key Challenges in Parallelization

As I seek to develop and implement the parallelized PageRank application, there are several major challenges required to be considered for this project: Firstly, aiming for good load balance across all the threads/processors is challenging. Potential highly imbalanced load on different threads/processors because of highly non-uniform data distribution or improper task allocation should be considered. Balancing the workload among threads/processors is critical to computing performance. Sharing the workload equally across all threads/processors aims to minimize the idle time on different threads/processors and thus the computation can advance efficiently, leading to improved performance. However, it is non-trivial to achieve perfect load balance which depends on the parallelism of the parallel

implementation, the application and workload. To address this problem, in my implementation I use static scheduling of iterations. Secondly, how to avoid possible race conditions which occurs in a parallel program execution when two or more threads/processes try to access and modify a common resource is very challenging. In the context of this project, when each thread wants to update the visit count of each node in my final count array, race condition is likely to happen. Because race condition is very hard to debug, I need to avoid race conditions by careful implementation design instead of trying to fix it later. To prevent race conditions, critical section and atomic operation can be applied. Critical section can ensure the serialization of blocks of code and can be extended to serial groups of blocks using the name tag but it is slower than atomic operation. Atomic operation can only guarantee the serialization of a particular set of operations but it is much faster. In my project, I use atomic operation to prevent race conditions during count array update. Thirdly, how to minimize the communication for distributed memory implementation is challenging. It aims to reduce the number of messages passed and reduce the amount of data contained in messages. Thus, in my distributed memory version using MPI, each processor can have their own copy of graph in their memory and in the end, only the count array will be gathered in the root processor. Finally, how to achieve good resource management is also challenging. For this project, the graph size is not trivial because it is based on the World Wide Web. Therefore, proper representations of huge graphs are highly desired. Though adjacent matrices can represent graphs in an easier manner, it should not be used in this project because it will consume more space O(V^2) where V is the number of vertices. The sparse graph will still take that much space. The representation of adjacent list is suitable for this project. There is an array of V adjacent lists, one adjacency list for each vertex. It takes up O(V+E) space where V is the number of vertices and E is the number of edges. This is memory efficient in both multithreaded version using OpenMP and distributed memory version using MPI.

Proposed Approaches

Graph Representation G(V, E): Considering the large size of datasets, I use adjacent list representation which was mentioned in the section above and it is suitable to cases where many nodes don't have any out-going links and take up O(V+E) space where V is the number of vertices and E is the number of edges. Because the node number is a unique number and is not necessarily in the range of [0, the total number of vertices -1], I map each node number to the unique number within that range using unordered map in C++. Without the mapping, I will have to specify which node is in the head list, which will bring unnecessary work. Mapping enables me to use the index in the head list to represent the node. Then I construct my adjacent list representation using the mapped node number.
Multithreaded version using OpenMP:

Pseudocode:

Input: Graph G(V, E) of web pages containing |V|=n nodes (webpage) and |E|=m edges

Length of random walk = K

Damping ratio = D (the probability to jump to any node instead of to only neighbors)

Output: An n-element array count[n] of PageRank value for each node (webpage) and a sorted n-element array of PageRank.

0. Preprocessing data: map all the node numbers to [0, n-1]

1. #pragma omp parallel for num_threads(p) schedule(static)

2. **For** i ← 0 **to** n-1 **do**

```
3.      start ← i
4.      totalWalk ←K
5.      while (totalWalk > 0)
6.          r ←random number generated between [0,1]
7.          if r < D then
8.              jump to random number between [0, V-1]
9.          else
10.             if the node start has no neighbors then
11.                 jump to node num which is a random number between [0,V-1]
12.             else
13.                 jump to any neighbor num of the node start based on equal probability
14.             end if
15.         end if
16.         start = num
17.         totalWalk = totalWalk - 1
18.         #pragma omp atomic
19.         count[num]++;
20.     end while
21. end for
22. count_sorted ← sort the count array
23. Map the node number back to its original number
24. return count_sorted
```

Note: based on the requirement, my final project only return the top 5 nodes with their pagerank values. I am using static scheduling and atomic operation which has been covered in the last section Key Challenges in Parallelization. Moreover, the fast walk matrix in my code is based on the two dimensional vector. Each row has the neighbor vector for the node having the row number as its mapped node number. It will avoid traversing the linkedlist and the access time will be O(1), which makes the implementation more efficient.

Space complexity analysis: (1) The adjacent list will take up O(n + m) where n is the total number of vertices and m is the total number of edges. (2) The node number to [0, n-1] map will take up O(n). (3) The fast walk matrix will take up O(m). (4) The final count array will take up O(n). Therefore, in total the space complexity is O(n + m).

Run-time complexity analysis: each thread is responsible for n/p of nodes. Each while loop contains O(K) work. Run time would be O(K*n/p).

Distributed memory version using MPI:

Input: Graph G(V, E) of web pages containing |V|=n nodes (webpage) and |E|=m edges

Length of random walk = K

Damping ratio = D (the probability to jump to any node instead of to only neighbors)

Output: An n-element array count[n] of PageRank value for each node (webpage) and a sorted n-element array of PageRank.

0. Preprocessing data: map all the node numbers to [0, n-1]

1. Send G(V, E) from the root processor to other processors using one-to-all Broadcast and each processor maintains **its own count array** and does the following up to line 20:

2. **For** i ← 0 **to** n-1 **do**

3. start ← i
4. totalWalk ←K
5. **while** (totalWalk > 0)
6. r ←random number generated between [0,1]
7. **if** r < D **then**
8. jump to random number between [0, V-1]
9. **else**
10. **if** the node start has no neighbors then
11. jump to node num which is a random number between [0,V-1]
12. **else**
13. jump to any neighbor num of the node start based on equal probability
14. **end if**
15. **end if**
16. start = num
17. totalWalk = totalWalk - 1
18. count[num]++;
19. **end while**

20. **end for**

21. The root processor collects the count array from all other processors using Gather

22. The root processor combines the count of the same index together and generate the final

21. count_sorted ← sort the count array

22. Map the node number back to its original number

23. **return** count_sorted

Space complexity analysis (for each processor): (1) The adjacent list will take up O(n + m) where n is the total number of vertices and m is the total number of edges. (2) The node number to [0, n-1] map will take up O(n). (3) The fast walk matrix will take up O(m). (4) The final count array will take up O(n). Therefore, in total the space complexity for each processor is O(n + m).

Run-time complexity analysis:

(1) Communication time: one-to-all broadcast + gather=O((t+mu*x)lgp)+O(t*lgp+mu*y*p), x is the size of graph O(V+E), y is the size of count array which is |V|.

(2) Computation time: O(K*n/p)

Run time will be the sum of (1) and (2).

Experimental Results and Discussion

Results: Directed graph using web-Google.txt:

Google

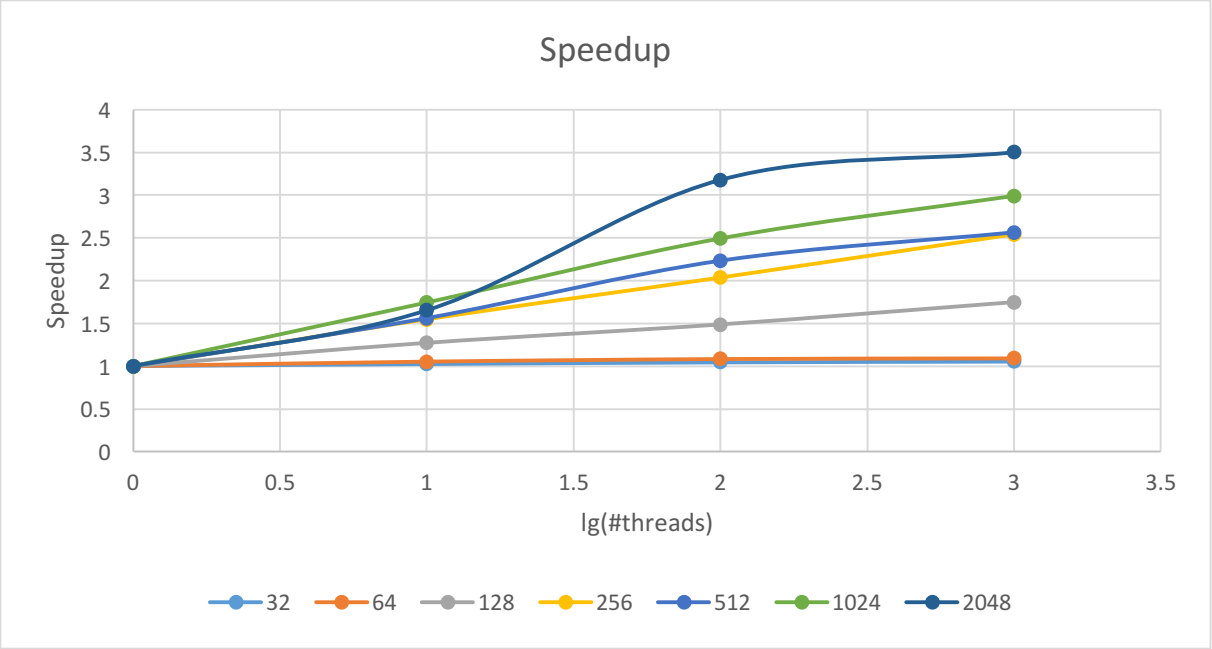
Time in sec	P=1	P=2	P=4	P=8
32	0.819167	0.798114	0.782107	0.774326
64	1.699107	1.611826	1.565443	1.554102
128	3.551441	2.792398	2.387563	2.030911
256	7.904892	5.107236	3.882017	3.108927
512	16.203281	10.385621	7.263489	6.325129
1024	32.386458	18.574239	12.990125	10.826732
2048	64.012832	38.712345	20.139521	18.283461

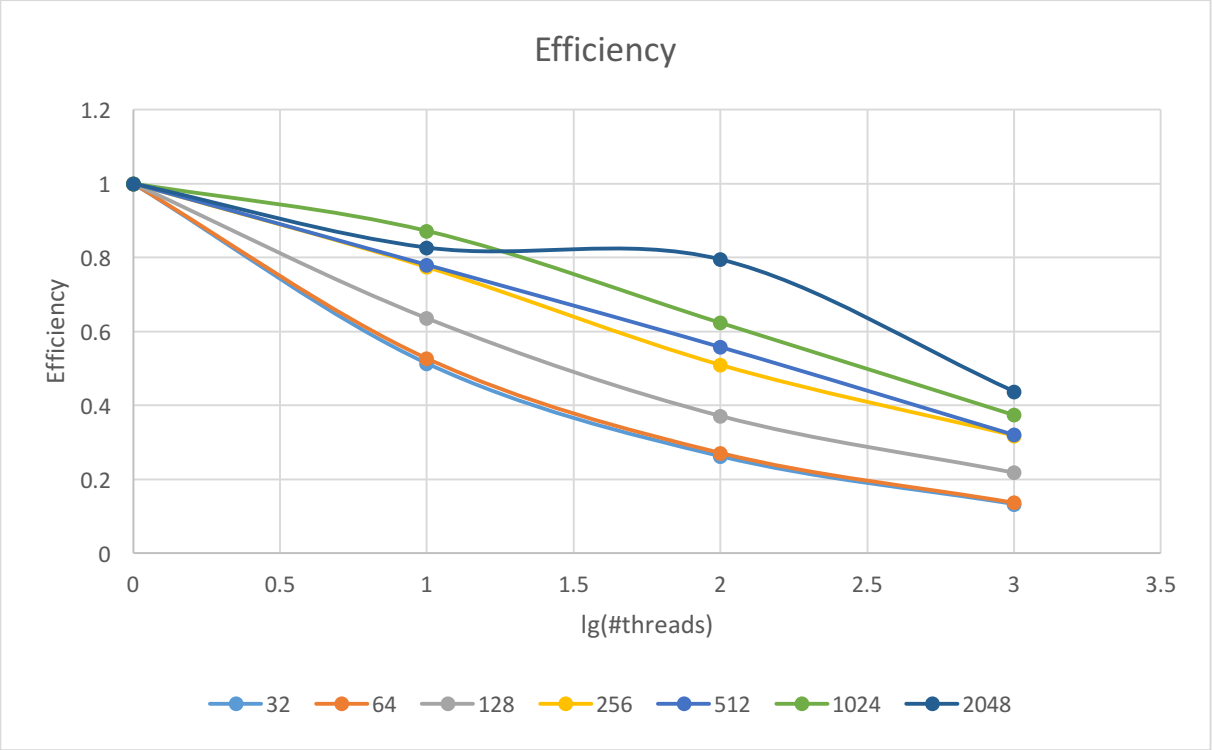
Speedup

K	P=1	P=2	P=4	P=8
32	1	1.026378437	1.047384821	1.057909718
64	1	1.054150386	1.085384137	1.093304687
128	1	1.27182479	1.487475304	1.748693567
256	1	1.547782793	2.036284746	2.542643169
512	1	1.560164866	2.23078482	2.56173131
1024	1	1.743622336	2.493159843	2.991341986
2048	1	1.653550876	3.178468445	3.501133183

Efficiency

K	P=1	P=2	P=4	P=8
32	1	0.513189219	0.261846205	0.132238715
64	1	0.527075193	0.271346034	0.136663086
128	1	0.635912395	0.371868826	0.218586696
256	1	0.773891396	0.509071186	0.317830396
512	1	0.780082433	0.557696205	0.320216414
1024	1	0.871811168	0.623289961	0.373917748
2048	1	0.826775438	0.794617111	0.437641648





Top 5 nodes in the last run with the corresponding count and page rank value:

Walk length K=2048,

Node Number	Visit Count	Page Rank Value
163075	709647	0.000395686
597621	649794	0.000362313
504140	647112	0.000360818

819223	494273	0.000275597
54147	491323	0.000273953

Evaluation of results:

Speedup analysis: The speed up increases when I increase the number of threads which meets my expectation.

Efficiency analysis: The efficiency decreases and the rate of decrease reduces as I increase the number of threads which meets my expectation.

Analysis on the influence of the walk length K: Firstly, when K is smaller, the top five nodes vary a lot. However, when K is large, there is little change in my top five nodes and it becomes almost steady in the end. I think the increase in K can reduce the effect of randomness and can give me a better evaluation of the importance of webpages. Speedup for the longer length is greater and efficiency for the longer length is also higher.

About how to deal with the dead-end nodes: The high-level approach listed in the class website suggests being stuck at the dead-end nodes. This is not fair and it will give higher page rank to these dead-end nodes. Therefore, the final result will be biased. My way to do this is when I jump into a dead-end node, next time I will jump to any node (webpage) among the entire set of nodes with equal probability. As a result, the page rank value will not favor these dead-end nodes and the result will be more fair.

About values of D: The proper value for damping ratio should be in (0, 0.5) because practically it is more likely for the online surfer to follow one of the out-links of a web page and correspondingly it is less likely to jump to any node (webpage) among the entire set of nodes with equal probability. I tested 0.3, 0.2, 0.15 and 0.1 for the damping ratio. If D is 0.15, then there is 0.15 chance that a typical user won't follow any links on the page and instead navigate to a new random page. Both the score of each node and its rank will change as the damping ratio changes. Smaller damping ratio decreases the possibility of following long paths of links without jumping out, which tends to increase the page rank value of nodes along the paths. It turns out 0.15 works the best among all these values.

References

[1] Page, Lawrence, et al. *The PageRank citation ranking: Bringing order to the web*. Stanford InfoLab, 1999.