Pagerank Over MapReduce

Rajat Jaiswal
cs5170415@cse.iitd.ac.in

Rajbir Malik
cs1170416@cse.iitd.ac.in

May 5, 2020

Introduction

In this Assignment, we used the MapReduce paradigm to solve the Pagerank algorithm. There were three parts to this assignment, and in each of the part we had to use a different implementation of MapReduce, namely:

- MapReduce C++ library: Open-source implementation of MapReduce paradigm in C++.
- Own Implementation: Implemented our own MapReduce library with MPI.
- MapReduce-MPI library: Open-source implementation of MapReduce written on top of standard MPI message passing.

The input was read from the file and stored in a graph, which is a pointer to a vector, in the form of $< X, < A, B >>$, where A and B are nodes in the graph, and X corresponds to the fraction of A's weight that is transferred to B.

Map Task: Each Map task ran on an exclusive part of the graph vector emitting $< key, value >$ pair where key is a node in the graph and value is a part of it's final weight. The part of the graph vector that each map task will operate on is decided by the input parameter to that map task. The key corresponding to the input of map task is the process number/id and the value corresponding to the input is a pair that contains start and end position of the graph vector over which the map task has to operate.

Reduce Task: The MapReduce paradigm itself clubs all the values emitted by the Map Task and which corresponding to a particular key in the form of a list and is presented to the reduce task as $< key, valuelist >$, where key is the node in the graph and valuelist contains list of weights corresponding to the key. When this value list is summed, it gives the final weight/pagerank for that particular key/node.

Convergence Criteria: If the sum of square difference of pageranks in consecutive iteration was less than the LIMIT($= 10^{-13}$) in our case), the algorithm was considered to have converged.

Latency Comparison

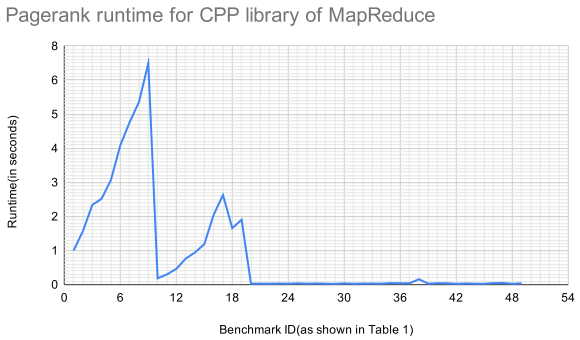

Fig.1. Pagerank runtime using MapReduce C++ library for different Benchmark ID

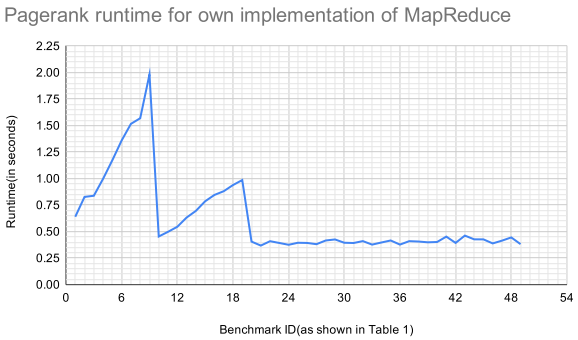

Fig.2. Pagerank runtime using our own implementation of MapReduce for different Benchmark ID

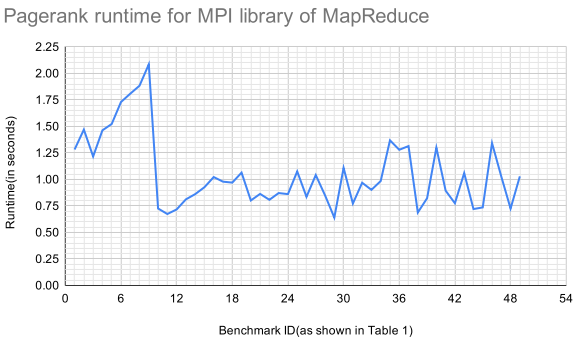

Fig.3. Pagerank runtime using MapReduce-MPI library for different Benchmark ID

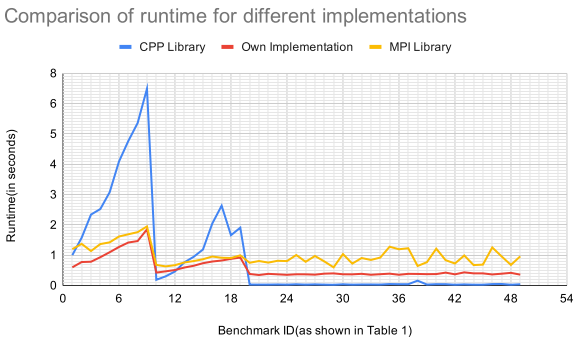

Fig.4. Pagerank runtime using MapReduce C++ library for different Benchmark ID

Observations

We classify our dataset into three categories:

- **Barabasi dataset:** These are the dataset comprising of benchmark ID 1 to 9, as shown in Table 1. They are high entropy dataset compared to Erdos dataset. Number of nodes in each of the file is greater than 20000.
- **Erdos dataset:** These are the dataset comprising of benchmark ID 10 to 19, as shown in Table 1. They have low entropy compared to Barabasi dataset. Number of nodes in each of the file is greater than 10000.
- **Famous dataset:** These are the dataset comprising of benchmark ID 20 to 49, as shown in Table 1. They have very less number of nodes(less than 200).

The benchmarks were tested on a machine with 4-cores and all the 4 cores were utilized while testing with any above-mentioned implementation. Based on the latencies observed in this run, **the following points are concluded:**

- For the Barabasi and Erdos dataset, the implementation using CPP library had significantly high latency compared to the other two implementations. Whereas the other two implementations gave almost similar performance, but still the implementation using our own MapReduce library was slightly better than the MapReduce-MPI library.
- For the Famous dataset, CPP library implementation was faster than the other two implementations. The latency observed for this dataset in CPP library implementation was close to zero, and it was almost 30 times faster than the MapReduce-MPI library implementation, which had the worst latency for this dataset.
- As the size of the dataset increases, the latency also increases, though the factor by which it increases is highest in CPP library implementation. The factor of increase is also high for Barabasi dataset due to its high entropy compared to Erdos dataset.
- For the Famous dataset, the latency across all the benchmarks is almost constant in CPP library implementation and our own implementation. There is no visible trend in MapReduce-MPI library implementation.
- The latency for Barabasi benchmark with equal number of nodes as that of Erdos benchmark is much higher across all the implementations. This may owe to the fact that the Barabasi dataset has high entropy.
- Overall it can be concluded that our implementation is better than that of pre-existing libraries, because it gives much better performance for Barabasi and Erdos dataset, which have large number of nodes. And is almost comparable to CPP library implementation for smaller famous dataset.

The latency for all the benchmark IDs across all the implementations are reported in the table on the next page.

ID	File Name	Runtime [CPP Library](in s)	Runtime [Own Implementation](in s)	Runtime [MPI Library](in s)
1	barabasi-20000.txt	1.07177186	0.6403746605	1.282238007
2	barabasi-30000.txt	1.68679738	0.8275928497	1.470044613
3	barabasi-40000.txt	2.505387545	0.8382053375	1.216393232
4	barabasi-50000.txt	2.701224089	0.9984548092	1.463128805
5	barabasi-60000.txt	3.293782234	1.173936129	1.522726297
6	barabasi-70000.txt	4.380320072	1.359719753	1.729859352
7	barabasi-80000.txt	5.10600543	1.516526461	1.808632135
8	barabasi-90000.txt	5.741603374	1.569322348	1.883943319
9	barabasi-100000.txt	6.953551054	1.982263803	2.085427046
10	erdos-10000.txt	0.1995644569	0.454015255	0.7257857323
11	erdos-20000.txt	0.3216640949	0.4982538223	0.6737930775
12	erdos-30000.txt	0.4936349392	0.5455799103	0.7168464661
13	erdos-40000.txt	0.8160393238	0.6326472759	0.8119683266
14	erdos-50000.txt	1.011268854	0.6953594685	0.8612709045
15	erdos-60000.txt	1.270769119	0.7861526012	0.926633358
16	erdos-70000.txt	2.191371679	0.845893383	1.022348404
17	erdos-80000.txt	2.819102049	0.8809854984	0.9788639545
18	erdos-90000.txt	1.774767399	0.9390044212	0.9698884487
19	erdos-100000.txt	2.041962624	0.9863038063	1.064139843
20	bull.txt	0.03166365623	0.4057607651	0.8013875484
21	chvatal.txt	0.02979421616	0.3683841228	0.8634910583
22	coxeter.txt	0.02969479561	0.4096646309	0.8078083992
23	cubical.txt	0.03461933136	0.3928074837	0.8714177608
24	diamond.txt	0.02883863449	0.3758487701	0.8613529205
25	dodecahedral.txt	0.04095864296	0.3948168755	1.075278282
26	folkman.txt	0.0290555954	0.393091917	0.8361210823
27	franklin.txt	0.03793740273	0.3814642429	1.041344643
28	frucht.txt	0.02909326553	0.4163262844	0.8498589993
29	grotzsch.txt	0.02359747887	0.4261045456	0.6400783062
30	heawood.txt	0.04064798355	0.3949923515	1.109084845
31	herschel.txt	0.02765440941	0.3925020695	0.7724013329
32	house.txt	0.03345394135	0.4104640484	0.9687361717
33	housex.txt	0.03399085999	0.3775188923	0.9017999172
34	icosahedral.txt	0.0312564373	0.3973183632	0.9848489761
35	krackhardt_kite.txt	0.04872131348	0.4163901806	1.368774414
36	levi.txt	0.04311966896	0.3776869774	1.279588461
37	mcgee.txt	0.04334306717	0.4099271297	1.314483166
38	meredith.txt	0.1692459583	0.4064610004	0.6877608299
39	nonline.txt	0.03094577789	0.3991825581	0.8226184845
40	noperfectmatching.txt	0.04462242126	0.4020390511	1.299406767
41	octahedral.txt	0.04384565353	0.4527323246	0.8938343525
42	petersen.txt	0.02526831627	0.3934779167	0.7751140594
43	robertson.txt	0.03645658493	0.4624018669	1.060019016
44	smallestcyclicgroup.txt	0.02746629715	0.4272079468	0.7206172943
45	tetrahedral.txt	0.02664279938	0.4267907143	0.7360656261
46	thomassen.txt	0.04776120186	0.3888013363	1.342869997
47	tutte.txt	0.05136370659	0.4148085117	1.027151108
48	uniquely3colorable.txt	0.02521371841	0.4451544285	0.7252619267
49	walther.txt	0.04201197624	0.3819580078	1.029062748

Table 1: **Pagerank runtime for different implementations on a 4-core machine**