Pagerank using Mapreduce

Musunuru Saurav   Rishikesh Reddy

1 Mapreduce and Pagerank

Mapreduce consists of 2 steps mainly map and reduce. Map takes a key and a value and produce another pair of key and value. The intermediate step takes common keys emitted by map and gives a key and list of values which is then passed on to reduce. The reduce step reduces the list given by the collate step and results in a key-value pair which is the final result. In this assignment we implement pagerank using mapreduce. In Pagerank we need to compute Mp where M is the p is the probability vector and M is the step-matrix. The step-matrix consists of 2 parts the weight of each outgoing link and the weight of all dangling pages. Since the dangling pages' pageranks have a common effect on all probabilities the sum of pageranks of all dangling pages is collected and it is added at the end. There is also a damping factor added to add into account the random page clicks.

2 Algorithm

2.1 Map and Reduce Functions

In the map function the key input is a node value and the value input is the adjacency list of that node. For each node in the adjacency list it emits the node as the key and pagerank / size as value. if there are no nodes in the adjacency list it's a dangling page and its pagerank is added to the dangling pagerank sum. The reduce gets the node as a key and different contributions of pagerank as value. The reduce function simply adds all the values in that list and emits it as the value and the node as key. To these final results the damping correction and the dangling page sum is added.

2.2 Parallelised algorithm

In the MPI part we try to parallelise the algorithm. We have a map and reduce function available on each core. So we divide the graph into N parts and send the respective part using MPI functions and then each core runs the map and reduce functions and results in 2 things - a partial sum of the dangling pages' pagerank and partial pageranks of different nodes. These 2 results are sent back the rank 0 processor where they are added to get the final pagerank result. The error of convergence is calculated and broadcasted to all processors to make sure all exit the loop properly and together.

3 Time complexity

Since the map goes through all adjacency lists. The map complexity gives O(E). The reduce goes through all contributions of each vertex which is of O(V). The loop in which we add the correction is also O(V). So the total complexity is O(V + E). In real life websites there are not more than 20 or 30 out-links from each node which makes the complexity of order O(V). But the constants can make a huge difference in cases of nodes of order 10^9

4 Results

We are only comparing the benchmarks of the barabasi files obtained from the pagerank repo. The other benchmark i.e. erdos is a very irregular benchmark as it has a lot of isolated nodes unlike barabasi which is not ideal for time calculations, since much of the graph is empty and does not exactly reflect the no. of nodes mentioned.

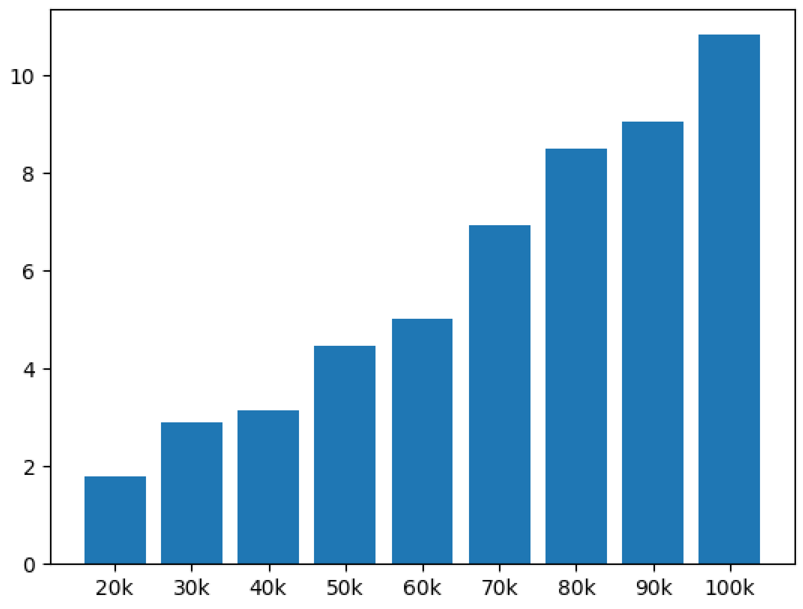

Figure 1: Timings for CPP library

5 Discussion on results

- The increase in time is almost linear in Figure 1 which is expected given the time complexity and the sizes of the nodes.
- In Figure 2 we can see that accross different cores the time taken is less but the speedup is very less. This is because the communication time is very much comparable to the calculation time because we are communicating a large amount of information i.e. initial pageranks for calculation, partial pageranks **for every iteration** and parts of the graph.
- Between 1 core in Figure 2 and Figure 1 the former has much better times. This is because the CPP library has many wrappers with a lot of object creations and calculations which causes

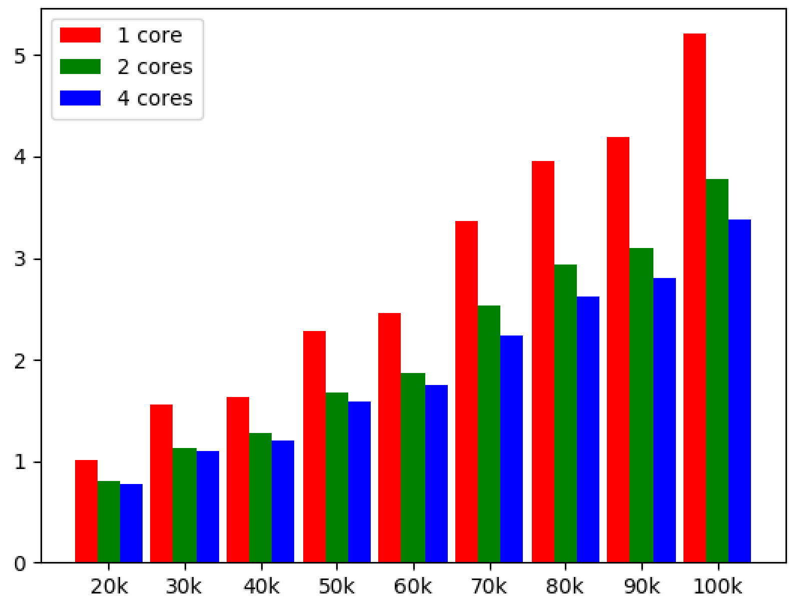

Figure 2: Timings for Own MPI implementation

unnecessary overhead and since each time a new object is being created for a new mapreduce the time taken is quite large compared to Figure 2

- In Figure 3 we can see the similar trend of the speedup not being very good due to rather large communication time same as Figure 2
- If you notice Figure 2 and Figure 3, the latter has much better time. This is mainly due to the communication of the partial results after the reduce. In the former each rank sends the results to rank 0 and thus all communication load falls on rank 0, whereas in the former we use the inbuilt gather function which is much more efficient. Also since this communication is done in every iteration this change makes quite a large difference.

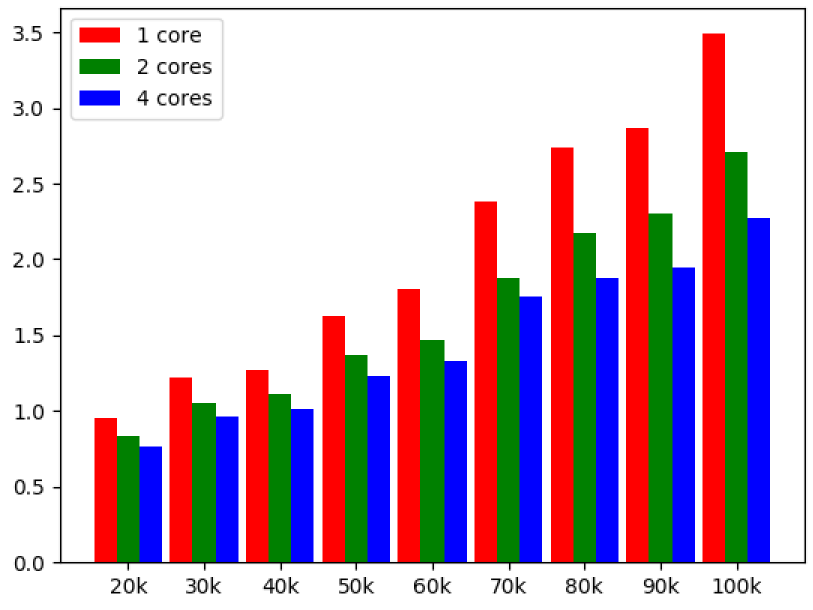

Figure 3: Timings for Given MPI library

6 Appendix

6.1 Computer Specifications

CPU(s):	8
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	6144K