Distribuited PageRank on MapReduce

Luca Saverio Esposito
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università di Roma Tor Vergata
Roma, Italia
lucasaverio.esposito@students.uniroma2.eu

***Abstract*—The World Wide Web continues to grow rapidly, searching right document from such an enormous number of web pages is a challenging task. PageRank is one of the methods used by Google to determine importance and relevance of web pages. The goal of this project is to design MapReduce version of PageRank that is distributed across a set of containers managed by Docker Compose and deployed on EC2 instance.**

Keywords—PageRank, WWW, Sink, Container, Docker, Map e Reduce, AWS, EC2, Virtualization

I. INTRODUCTION

Larry Page and Sergey Brin developed PageRank to address the problem they encountered with their search engine for the World Wide Web. Given a search query, it's not simple to quickly find the user's desired page. They wanted to incorporate a measure of page importance into the results to distinguish relevant pages from less useful ones. To do this, Google designed a system of scores called PageRank by using the link structure of the web. The more links point to a page, the greater the relevance of that page; links from relevant pages are more important than others [1]. For the project, the chosen programming model is MapReduce, which permits the parallelization of computation across a cluster of containers. These containers are deployed on an EC2 virtual machine from Amazon Web Services.

II. PRELIMINARIES

A. MapReduce

MapReduce is a specific programming model implemented in different systems. The computation takes a set of input key/value pairs and produces as output another set of key/value pairs [2]. The core of the model are two functions:

- Map: It takes as input pair and produces a set of intermediate pairs. Then, all the intermediate values associated with a key K are grouped together and passed to the Reduce function.
- Reduce: It accepts a key *K* and a set of values for that key, and it merges values together.

These functions are executed on multiple worker nodes in parallel. This implementation allows for achieving excellent results in terms of performance and provides redundancy, thus ensuring fault tolerance, by having multiple workers of the same type.

Here's an example picture:

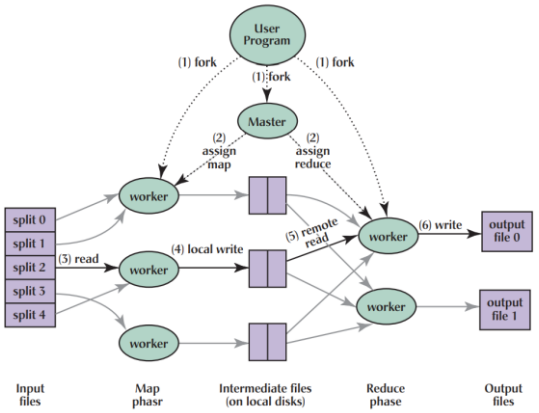

Figure 1 MapReduce execution overview.

Map function takes the pair $< node\ N\ ;\ N.pagerank >$ and emits $< node\ M\ ;\ pagerank\ share >$. To simplify the state inside of the worker as much possible, instead of passing all node N, map function works directly with adjacency list: $< N.adjacencylist\ ; N.pagerank >$ this permits to distribute the load arbitrarily among containers. After that, *Shuffle phase* [3], that happens directly on master program, groups together all the mappers outputs that sharing the same key. Then, reduce function takes in input the pair $< node\ N; [p1, p2, \dots] >$ to evaluate the new page rank. The master updates the values, and finally loop restarts.

Still remaining two problems to solve:

- Sink: Node or group of nodes having no out-links to other nodes [4].

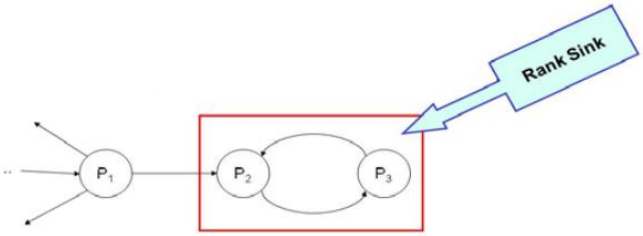

Figure 2 Sink example.

- Random jump factor: The possibility that surfer does not always follow a link.

To address these problems, a *cleanup pass* is needed, which modifies the node's actual page rank by adding the mass lost

due to sinks and a term that represents the random jump. So, the mapper check if the node is a sink; in that case returns the node page rank value. Once again, during the *shuffle phase*, combines data together and obtains sink's mass. Then, the reducer evaluates new page rank value with these new terms, and new iteration begins.
In summary, here's a recap of the model behavior:

- Map function: Evaluates the page rank contribution of each node.
- Shuffle phase: Merges together contributions associated with the same node.
- Reduce function: Evaluate the new actual page rank value for each node.
- Mapper's clean up phase: Retrieves mass from sink nodes.
- Shuffle phase: Merges together masses.
- Reducer's clean up phase: Once again, evaluates the new actual page rank value for each node.

B. PageRank algorithm

PageRank provides an advanced way to compute the importance of a web page, taking into consideration both the number of links pointing to that page and the relevance of those links. Starting from the basic model, we have a graph with N nodes, each of which initially has a page rank of 1/N. Each node divides its current page rank among its out-links and passes it to them. Each page updates its new page rank as the sum of the shares received. In particular, the PageRank formula is as follows:

$$R_{i+1}(u) = \sum_{v \in V_u} \frac{R_i(v)}{N_v}$$

Where:

- u: Single web page.
- V_u: Set of pages pointing to u.
- N_v: Number of out-links from v.

Then *scaled PageRank*, introduces the possibility of jumping to random pages, which is represented with a probability: $1 - c$. The updated formula is as follows:

$$R_{i+1}(u) = \sum_{v \in V_u} \frac{R_i(v)}{N_v} + (1 - c)E(u)$$

Where:

- *E(u):* It is vector of probabilities over web pages.
- c: It is called *dumping factor*.

Especially, for *E(u)* chosen as uniform distribution, $\frac{1}{N}$, and for c 0.85.
The *last version* of the algorithm includes the clean-up pass, so the formula takes account of sink mass. In particular:

$$p' = c\,(p + \frac{m}{N}) + (1 - c)\frac{1}{N}$$

Where:

- p: Current page rank value
- m: Mass lost due to sinks
- N: Number of nodes in the graph

Finally, a recap of algorithm:

- Set for each nodes an initial page rank (i.e., $\frac{1}{N}$)
- Each node divides its actual page rank and shares to other nodes.
- Each node evaluates with *shared PageRank* formula new page rank.
- Each node fixes its page rank by adding the contribution of sink mass with last formula.

These steps are executed by worker nodes with MapReduce pattern introduced earlier. The execution will stop after a fixed number of iterations or when the equilibrium is reached.

C. Docker Compose

The program is divided into different parts, each of which is deployed in a separate container managed by docker-compose. There are three main components, each with its specific docker file:

- Mapper
- Reducer
- Master

The Master component manages the execution and communicates with the Mapper and Reducer components using gRPC, functioning as the client of the system. Both Mapper and Reducer execute map and reduce functions, as well as the clean-up phase, and they provide a health monitoring service. The number of worker nodes (Mapper and Reducer) and the open ports they listen on can be configured by editing the configuration file. The docker-compose.yml file is automatically generated by pagerank.go based on the configuration file. All three components are part of the same network.

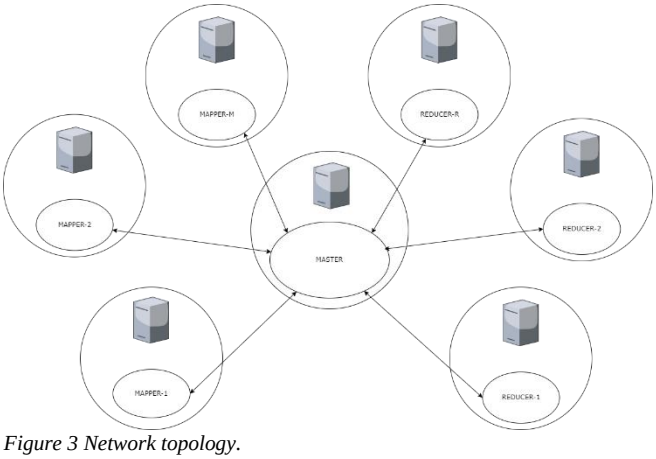

Figure 3 Network topology.

By using container combined to map reduce pattern the system becomes:

- Scalable: It's possible to adjust the worker's number based on graph size.
- Fault tolerant: Multiple containers perform the same task, so if one crashes, another one is available.
- Faster: Parallel execution is more powerful.

III. Implentation & Functionalities

The application evaluates the page rank value for an input graph. The graph is generated randomly by a simple algorithm, is possible to configure parameters like number of nodes, edges to attach and seed. Different seed generate different graph. The user also chooses the number of mapper and reducer, which is useful when graph size changes. There are three main components that's communicate with gRPC:

Master: The client of the application, running within a container, is responsible for sending requests to the Mapper and Reducer components and handling the shuffle phase with intermediate data. Once the graph is created, it initiates a loop where it invokes worker containers. For each node in the graph, the master launches a goroutine that requests the worker to execute a specific algorithmic task, such as mapping, reducing, or cleaning up. The scheduling is done in a round-robin manner; the client maintains a ring of connections to the containers. If a container is not ready, busy, or unreachable, the master attempts to send the request to the next available worker in the ring. If no containers are reachable, the program stops execution and must be restarted. After all requests are sent, the master waits until all goroutines have completed before proceeding to the next step of the algorithm. To synchronize all the goroutines, a waitGroup object is used. Goroutines are employed to enable parallel execution, thereby speeding up the program and taking advantage of the Map-Reduce computational model. The loops continue until the algorithm converges or reaches a maximum number of cycles. Convergence is determined when the page rank values remain the same for two consecutive rounds or when the difference between values is less than epsilon, an arbitrary parameter. Outside the loop, the master closes open connections and, if enabled by the flags, prints the graph, and saves the output results on S3.

Mapper: One of the two worker types, it listens on a specific port where it exposes map service.

Reducer: The other worker, it exposes reduce service.

Both workers also have ping service for health checking. As mentioned earlier, communication is based on gRPC, which is typically is synchronous and blocking, so here go-routine are used to work asynchronously.
For gRPC is necessary to generate pb files starting from proto file. So, both mapper and reducer have proto file that defines offered service, message structure and go files that implement the services.
When it comes to functionality, the application is resilient to multiple worker container faults. It requires only one container for each type. Notably, it utilizes a health check service to determine the status of each worker and directs requests accordingly. Once a faulty container is restored, the client can resume sending requests to it, thanks to round-robin scheduling. Additionally, the application features a logging system that records the output of algorithm executions, including the graph structure, intermediate PageRank values at each iteration, and the final results.

IV. Possible improvements

Container Scheduling and Dynamic Scaling: The system can benefit from an intermediate load balancer element instead of relying on a simple round-robin policy. This load balancer can efficiently distribute incoming tasks among worker containers based on their current load, ensuring optimal resource utilization. Furthermore, this load balancer should have the capability to start new containers dynamically when the workload increases and terminate containers when they are no longer necessary, providing automatic scaling based on demand.
Graph Printing Optimization: The current functionality for printing graphs works but is not optimized. In the case of large graphs, enabling this feature can significantly slow down the execution. To improve efficiency, consider implementing optimizations such as limiting the scope of graph printing or utilizing more efficient algorithms for generating graph output. This will allow the system to handle large graphs without sacrificing performance.
Algorithm Optimization: An important avenue for enhancing the performance of our PageRank implementation within the Map-Reduce paradigm is through the strategic application of code vectorization techniques. By optimizing the code structure for vectorization, we can leverage the parallel processing capabilities of modern processors and significantly boost the overall efficiency of our algorithm. As demonstrated in the seminal work by [5], which elucidates the matrix mathematics underpinning the PageRank algorithm, our efforts in code vectorization align with the fundamental principles of matrix-vector multiplication central to PageRank's computation.

V. Librarires and software platforms

A. Software platforms

- *Programming languages: Go.*
- *RPC:* gRPC.
- *Container:* Docker.
- *Container Orchestration:* Docker-Compose.

B. Libraries

- *git.sr.ht/sbinet*: for graph printing.
- *aws-sdk-go:* to save on s3 bucket.
- *golang.org/grpc*: for gRPC.
- *golang.org/protobuf*: to generate pb files.

VI. Bibliography

[1] B. a. M. H. K. Agarwal, «Analysis of Rank Sink Problem in PageRank Algorithm,» *International Journal of Scientific & Engineering Research*, vol. 4, pp. 251-256, 2013.

[2] J. a. S. G. Dean, «MapReduce: simplified data processing on large clusters,» *Communications of the ACM*, pp. 107-113, 51.

[3] K. C. a. D. X. Bahman Bahmani, «Fast personalized PageRank on MapReduce,» in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.

[4] S. S. a. M. Ripeanu, «No More Leaky PageRank,» *2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 27-34, 2021.

[5] D. F. Gleich, «PageRank beyond the web,» *siam REVIEW*, vol. 57, pp. 321-363, 2015.