Pagerank with PySpark and MPI

1st Tingcong Jiang
ECE
Rutgers University
New Brunswick, United States
tj215@scarletmail.rutgers.edu

2nd Shuren Xia
ECE
Rutgers University
New Brunswick, US
sx67@scarletmail.rutgers.edu

***Abstract*—PageRank is an algorithm used by Google Search to rank web pages in their search engine results. PageRank algorithm assigns weight to each web page, and it determines the importance of a page by the number of its incoming links and the weights carried by the links. Nowadays, there are countless web pages hosted on the internet. Thus, sequential method to calculate the rank of each page is slow. To tackle this problem, programmers used distributed framework to process links to calculate pagerank, and famous framework includes PySpark and Hadoop. Although MPI(Message Passing Interface) is not a a distributed framework, it is an interface that explores parallelism across CPU cores and machines.**

***Index Terms*—Pagerank, PySpark, Hadoop, MPI, Cloud Computing**

I. Introduction

The objectives of this project are to find which factors in MPI or PySpark affect the performance of pagerank, and where can we further speedup the pagerank algorithm. Pagerank is an algorithm proposed by Sergey Brin and Lawrence Page [1], and it aims to cooperate with Google Search Engine to give users a more satisfying searching result. The core idea behind the page rank algorithm is that the importance of each page is determined by the the importance of its neighbors, and the neighbors of a web page are defined as the web pages that have an outbound link to it. In section III, we provide details of the Pagerank algorithm. Since pagerank involves a lot of calculation, parallel computation is a plausible way to speedup the calculation. In the scope of this project, we used PySpark [2] and MPI [3] to implement the pagerank algorithm and test what are the factors that affect the performance.

II. Background

MPI stands for Message Passing Interface, and its goal is to offer the ability of communication between CPU cores or machines when there is no shared memory across them. The data travels through different cores or machines via message rather than memory address in shared memory machine. Furthermore, rather than being an implementation, MPI is a standardized and portable message-passing standard, and it defines the behavior of each function and the scope of functionality. Thus, there are various implementation of MPI available for different platforms. In general the workflow of MPI can be conclude in the following steps:

1) At beginning, multiple workers are spawned by the root process and each worker has a unique ID called rank
2) Root and workers processes have the exactly same code (behaves like system-call fork())
3) Root and workers can execute different task by examine their ranks.

However, since the execution order of workers is not undefined, synchronization is required at certain point to avoid race condition.

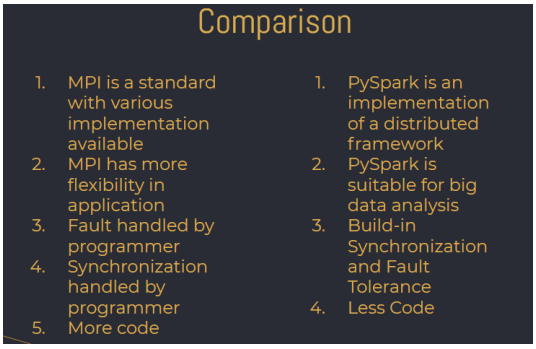

Fig. 1. Difference Between MPI and PySpark

PySpark is an open-source distributed general-purpose cluster-computing framework [4]. Unlike MPI, PySpark is an implementation, and it offers built-in fault tolerance and synchronization without programmer's interruption. As a programmer, the users of PySpark only need to define the workflow of data such as where to read data, how to transform data, how to calculate based on data, and where to write the result. Furthermore, the PySpark will determine the number of partitions of input data and workers by examine the underlying hardware of executing machine. However, users can also specify the scale of resources that can be used by PySpark.

The Figure 1 summarize the difference between MPI and PySpark

III. Algorithm

The pagerank Algorithm consists of two stages: initialization and iteration. In initialization, pagerank set the rank of each page to 1. Then, in iteration, pagerank calculates the contribution of each page by calculating $contribution =$

$\frac{rank}{number\_of\_outboundlinks}$. Then, each page aggregates the contribution received from its inbound links, and update its rank by calculating $rank = damping\_factor + (1 - damping\_factor) \times total\_contribution$. In general, damping factor = 0.15.

To implement pagerank algorithm with MPI, we do following:

1) Root Process read graph file and store it in a list that each tuple contains a partition of links, and the number of partitions equals the number of total processes (includes root)
2) Links are partitioned by their source page ID, and links are reduced so that each tuple in a partitions has <source ID, [dest1, dest2, ... , destN]>
3) Root process send one partition to one worker, and root process also take one partition
4) Initialize each page's pagerank to 1
5) Iteration Start:
6) Calculate contribution of each web page <[[dest1, dest2, ... , destM], contribution1], [[dest1, dest2, .. , destN], contribution2]>
7) Each process broadcast its contribution to all other processes
8) Each process update local pagerank by iterating through the whole contribution list
9) Each process calculate convergence of its local partition, and send result to root process
10) If the root concludes that the result is not converged from results of each worker, start next iteration

Similarly, for PySpark, we do following:

1) Read input, and transform it to a RDD called links which each tuple is <source, [dest1, dest2, ... , destN]>
2) Create rank RDD which each tuple is <source, rank> from the links RDD, and initialize rank to 1 by mapvalues()
3) Iteration Start:
4) Join rank RDD and links RDD so that the result RDD has following structure: <source, ([dest1, dest2...], pagerank of source)>
5) Calculate contribution list based the joined RDD, and contribution list has structure [(dest1, contribution), (dest2, contribution) ....]
6) Use reduce by key to aggregate contribution
7) Update rank RDD based on aggregated contribution
8) If the result is not converged from results of each worker, start next iteration

IV. Setup and Experiment

In our experiment, we used one local machine to run all tasks scheduled by PySpark or MPI. The local machine has a 4 cores CPU with HyperThreading and 3.4GHz base frequency. For software, we installed PySpark 3.0.1, Hadoop 2.7.0, Microsoft MPI, mpi4py library, and Python 3.7.6. Furthermore, the experiment of our implementation consists of two parts. The first part is using MPI with various numbers of workers and PySpark to process a graph with 10000 vertices and recording the average time for each iteration. Moreover, the second part is recording the average time of iterations that takes for MPI with 8 workers and PySpark to process a graph with 60000 vertices. Intuitively, the first experiment can show the relationship between number of workers and MPI performance, and the first experiment offers a baseline performance which can be used with the result of second experiment to show how the performance change when we increase the input size.

V. Evaluation Results

The results of our experiments are shown in Table I and Table II. In our experiment 1, the result shows that the MPI performance has a linear relationship with the number of workers. However, the percentage of improvement getting smaller when we keep increase the number of workers indefinitely. The explanation for this is that the situation has to do with how CPU schedule processes. In a modern computer, it uses context switching to show the multi-tasking ability to its user. Instead of keep running a process until it is terminated, each of the CPU threads schedules a process from the process pool and runs the scheduled process for a short period of time. After the process reached its time limit or it is terminated beforehand, the status of the process and its data inside registers will be saved into the main memory, and a scheduled process will be executed. The scheduled process's status will be resumed from the main memory, and the process of switching the execution of different processes is called context switching. Because context switching invokes memory I/O, it has an significant overhead. Thus, when we have much more number of workers than the number of physical threads, there are more context switching, and at certain point, the benefit from parallelism covered by the overhead from context switching between workers. So, this is why we see an decrease in percentage of improvement when we increase the number of workers indefinitely. However, the reason of why MPI has better performance when we have 16 workers rather than 8 workers is that MPI with 16 workers has a greater chance being scheduled by the CPU than it only has 8 workers. The prove of this statement is that we see more jump in CPU usage when we only schedule 8 workers than when we schedule 16 workers. Theoretically, both 8 workers and 16 workers should constantly use the full power of the underlying CPU because there are only 8 physical threads available. However, the CPU will schedule some Windows background services at some point, so CPU usage decrease as background services are not computation-intensive. Furthermore, we also can see that the PySpark is significantly outperformed by MPI given a graph with 10000 vertices. The reason behind this is that the PySpark only uses 1 core under Windows environment while MPI uses 8 cores when we schedule 8 or 16 workers.

The result getting more interesting when we compare experiment 1 and 2. In experiment 2, we can see that PySpark

TABLE I
EXPERIMENT PART 1

Implementation	Num_Workers	Num_Vertices	Average Time(Sec)
MPI	1	10000	20.9233
MPI	2	10000	10.728
MPI	4	10000	6.28255
MPI	8	10000	5.91
MPI	16	10000	3.4755
PySpark	N/A	10000	8.263

has much better performance than MPI when we increased the input size, although PySpark still only uses one core. The reason of this is that our implementation of MPI has a lot of communication overhead. In the step 7 of MPI implementation, each worker process sends data N-1 times to other processes while it also receives data N-1 times from other processes where N is the number of processes. Thus, the communication overhead has a time complexity of $O(N^2)$. Furthermore, at the end of each iteration, there is an overhead caused by barrier which is used to avoid race condition. Moreover, our implementation of MPI is not efficient. In step 7, MPI broadcast N-1$\times$ data than the data actually needed. In the case of PySpark, PySpark has more efficient operations within memory with minimum number of operation. Thus, PySpark outperforms MPI by a significant portion. However, there is one part of the performance of PySpark we cannot figured out that the performance of PySpark is affects a little with the dramatically increased input size. Our assumption is that PySpark has a well-optimized join operation. However, PySpark join operation also causes one problem. In our original implementation, we did not re-partition the joined RDD into a fixed number of partitions. Then, the execution time increase dramatically when the number of partitions increase. Because PySpark adds up the number of partitions of the two RDDs involved in the join operation, the performance of PySpark getting worse and worse when the number of iteration increase. Thus, our optimization to this is that we re-partition the joined RDD to a fixed size at each iteration, and the result shows that the time taken by each iteration becomes being stable.

In conclusion, we concludes that the MPI has 5 performance factor: Data Transmitted, Number of Workers, Synchronization, and the underlying hardware. The data transmitted and number of workers affects the performance dramatically compared with Synchronization under a fixed hardware platform. Thus, for a MPI program, decreasing the size of transmitted data and frequency of communication is the key of performance. Also, as a programmer, we should choose the optimized number of workers that can be determined by examine the underlying hardware. For PySpark, since already offers efficient operation to the programmers, the programmers should focus on how to optimize the number of partitions for each data frame or RDD they have created.

TABLE II
EXPERIMENT 2

Implementation	Num_Workers	Num_Vertices	Average Time(Sec)
MPI	8	60000	350.29175
PySpark	N/A	60000	9.80571

VI. FUTURE WORK

In this section, we propose three methods that we can think of to improve our MPI implementation. The first method is input split, and it is inspired by the PySpark implementation. For PySpark, in order to achieve efficiency, it will do transformation or calculation on data that has being placed into the main memory rather than data is placed on disk. So, when we are given a very large input that is not viable to place it into the main memory, we can separate the input into small chunks so that each chunk can be placed into the memory. However, there is a problem that if the MPI program only execute within one local machine, the input split with multiple processes can be disk I/O bounded, especially if we are working with a hard drive rather than a SSD. Because of the design of a hard drive, the hard drive is good at sequential writing and reading rather than random read or write. Thus, multiple processes doing multiple random reads or writes concurrently can cause a lot of overhead.

The second method is communication table. The reason of why we need every process keeps one copy of the contribution list in step 7 is that each process does not know which process it need to communicate with in order to receive the contribution. In order to understand this, we need to dive deeper in our implementation. In our implementation, each process holds a list of web pages that it will handle, and it also holds the outbound links of that pages. However, in order to know which processes that one process need to receive the contribution from, it also need to know the inbound links of the pages that it holds. But, since holding inbound links can causes a lot of memory usage, we choose not to do so. Thus, we need a communication table to determine the source of an inbound link given a destination of it.

The third method is matrix method. Matrix methods is the way of representing the calculation of pagerank, as shown in Figure 2. The matrix in Figure 2 is called transition matrix, and the vector on the right is the rank of each page. Then, the calculation becomes that the rank of each page equals one row of the transition matrix dot product with the rank vector. Because the transition matrix is most likely a sparse matrix when the number of vertices is huge, we can optimise the dot product by utilizing the sparsity of the transition matrix in our MPI implementation. With matrix method, each worker only need to store a range of rows from the matrix and the latest rank of page. Furthermore, workers do not need to store anything to know that which process it needs to communicate in order to fetch the needed row to calculate the rank, because the assignment of row can be assigned by range or hashing. Thus, we believe that the matrix method is the best bet of

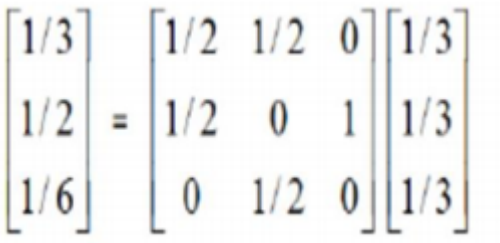

Fig. 2. Matrix Method

improve our implementation of pagerank with MPI

REFERENCES

[1] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, pp. 107–117, 1998. [Online]. Available: http://www-db.stanford.edu/ backrub/google.html

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2934664

[3] Wikipedia, "Message Passing Interface — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Message%20Passing%20Interface&oldid=994210455, 2020, [Online; accessed 16-December-2020].

[4] ——, "Apache Spark — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Apache%20Spark&oldid=993765552, 2020, [Online; accessed 16-December-2020].