An Improved Algorithm for Mutual Friends Recommendation Application of SNS in Hadoop

LEI, YUAN

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
10
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2015-03-24
(URL)
https://doi.org/10.15002/00011667

An Improved Algorithm for Mutual Friends Recommendation Application of SNS in Hadoop

YUAN LEI
Faculty of Computer and Information Sciences, Hosei Unversity
Tokyo, Japan
yllbxx9072@gmail.com

***Abstract*— In Social network system, existing "people you might know" or "Mutual friends" recommending applications are commonly utilized to list two-hop mediate relationships that one could have with another in order to tighten the bonds among groups. However, as the number of SNS users increase dramatically, the relationship data get so huge that the performance of Mutual Friends recommendation system becomes an urgent problem considering the developers' requirements. Here we propose a sorting algorithm in Hadoop----a parallel computing framework, to enhance the efficiency of "Mutual friends" recommendation process by taking advantage of the novel map reduce model. In the revised application, original sorting algorithm of intermediate data, merge sort, is replaced by a more time saving sorting approach which introduces a B-Tree like data structure, 2-3 Tree to store the user friendship data and conducts the sorting process. As number of user increases, the revised user defined map reduce functions perform better than the conventional design in time consuming aspect.**

Keywords*— *Mutual Friends, Recommendation System, Map Reduce, Hadoop, 2-3 Tree

I. INTRODUCTION

As the number of SNS users increases, the relations among users will grow up to heterogeneous graphs, which outstrips the storing and computing ability of even large-scale parallel processing machines. In "People You Might Know" friendship recommendation, an efficient Reducer side graph processing algorithm on Hadoop distributed system is proposed, which takes advantage of the novel database structure, 2-3 Tree, to store and sort common friends entries used in user defined Reduce function instead of array data structure. As a result, average reduce processing time can be reduced.

The application of Mutual Friends is a paralleling computing process which reads users' friends list as input data and calculates the potential common relationship each user could have with another. And then application recommends the "people you might know" to users ordered by the number of common friends they both could have. In this application, people and the relationships between two persons can be regarded correspondingly as vertices and edges between vertices in graph theory. As a result, the human network is abstracted as a non-directional graph without edge weights.

Existing algorithms designed for map reduce scenario to find friends in common such as Steve Krenzel [1] and Pulkit Goyal [2] emit inadequate output information. The former one generates just the specific mutual friends' ids without a sum. For example, "A, B → C, D" is a typical output line which means C and D are common friends between A and B. But if the developer would like to show the number of common relationships, another map reduce cycle is necessary. The later one, Goyal's algorithm is just short of something in an opposite way. Only number of mutual friends is written to the output file without the specific mutual friends list one by one.

The paper [3] written by Zhao GuoGuang et al. proposed another way of finding "common ancestor" in a given tree graph, which resembles the situation as "finding mutual friends", because the "common friend" to two users is like the ancestor to two of its children nodes. But this paper use different format of input data and data structure to store the graph information, so both algorithms have their own characters and they are just not able to be compared in a simple way.

Both Rishan Chen et al. [4] and Kisung Lee et al. [5] focused on the partition strategy as an approach to achieve the improvement. We also came up with some similar idea about changing its previous partitioning way in Mapper side which will be demonstrated later. But we found this application has its own special features, so achieving the improvement in the Reducer side would be more applicable, described as following.

Another algorithm about counting "people you might know" application is published by Jure Leskovec [6], whose design will be compared in the following chapter 4. Our revised version is based on the fundamental program of this one. During the reduce part of this algorithm, users and their mutual friends with key are stored and sorted by the array collection, which utilizes merge sort as sorting algorithm. And our improvement is focused on the modification of his sorting algorithm.

The rest of paper is organized as follows. We review the preliminary and related works on map reduce program model, a popular framework called Hadoop which uses the model in section II, also we introduce the Mutual Friends algorithm in both map and reduce in the same section. We describe the original sorting algorithm and present the revised version of sorting algorithm using 2-3 Tree in section III. We show the experiment environment in section IV. We present the experiment process with results and analysis in section V, and conclude this thesis with future work in section VI.

II. PRELIMINARY AND RELATED WORK

A. *Map Reduce Model*

The basic core design ideas of map and reduce is that some rules and limitations must be obeyed to achieve the parallelization of huge assignments. Users specify the computation in terms of a map and a reduce function. Map processes a set of key-value and generates a new, corresponding set of key-value as the intermediate results, and then Reduce function merges the results with same key [7]. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The Map Reduce library groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function. The reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges these values together to form a possibly smaller set of values [8].

B. *Hadoop Distributed File System and Hadoop way of Map Reduce*

Hadoop is the software for reliable, scalable, distributed computing [9]. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. An HDFS cluster has two types of nodes operating in a master-worker pattern: a name node (the master) and a number of data nodes (workers). The namenode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. Datanodes are the workhorses of the file system. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing. A client accesses the file system on behalf of the user by communicating with the namenode and datanodes. The Map Reduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

C. *Mutual Friends Counting in Parallel*

Here's how the application counts the mutual friends for each user with others in a Map Reduce way. The input data will contain the adjacency list and has multiple lines in the format of <USER><SPACE><FRIENDS>, where <USER> is an unique ID for an unique user, and <FRIENDS> is the list of users separated by comma who are the friends of <USER>. The following Fig 1 is an input example.

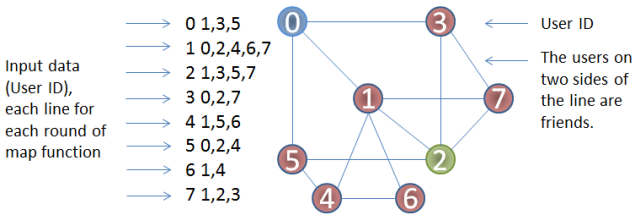

Fig. 1. Input example and a graph representing it

In the graph, you can see User 2 is not friends of User 0, User 4 and User 6, but User 2 and User 0 have mutual friends 1, 3 and 5; User 2 and User 4 have mutual friends 1, 5; User 2 and User 6 have mutual friend 1. As a result, we would like to recommend User 0, User 4 and User 6 as friends of User 2.

The output common friends list will be given in the following format shown in Fig 2: <USER><SPACE><Recommended USER (Number of mutual friends) [the ids list of mutual friends separated by comma]; >. The output result is sorted according to the number of mutual friends. Those who have the most mutual friends with <USER> will be on top of the list. And if two have the same number of common relationships they will be ordered by the natural order of id such as ascending. For instance, User 0 has two common friends with both User 4 (1, 5) and User 7 (1, 3), since User 4 has the ID smaller than that of User 7's, we hence put User 4 in front of User 7 in the list.

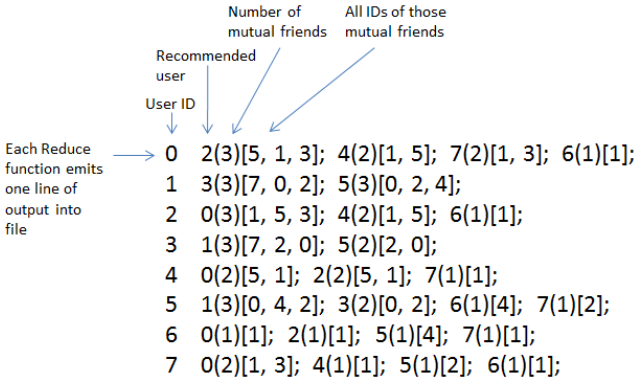

Fig. 2. Output example

User 0 has friends User 1, User 3 and User 5; as a result, the pair of <1, 3>, <1, 5> and <3, 5> have mutual friend of User 0. As a result, we can emit <key, value> = <k=1, {v1=3, v2=0}>, <k=1, {v1=5, v2=0}>, <k=3, {v1=5, v2=0}>, where v2 means the common relationship as mutual friend between k and v1. In the following implementation, v2 = -1 is used when they are already friends instead of using extra field.

A. *Map phase:*

For each f in {f} = {f1,f2,f3…fn}

Emit <k, {v1= f, v2=-1}>;

Let's say the number of f is n; then we'll emit n records for describing that k and f are already friends. So we set v2 as -1.

For each f1 and f2 (f1 ≠ f2) in {f}

Emit <f1, {v1=f2, v2=k}>;

Emit <f2, {v1=f1, v2=k}>;

They have mutual friend, k. It will emit n*(n – 1) records.

Totally, there are n^2 emitted records in the map phase, where n is the number of friends <USER> has.

B. *Reduce phase:*

Just summing how many mutual friends they have between the k and v1, and make a list to contain all the v2. If any of

them has mutual friend -1, that is, v2 = 1, we don't make the recommendation since they are already friends.

Pick up all the keys and values in the hash and sort them based on the size of value. And if two values in the hash have the same size, he who has the smaller key can stand in the front of the larger one.

However, by calculating the time each phase spends, the current Reduce part occupies the majority of whole job time, which we believe is not so efficient adequately; Sorting all the keys and values in the hash takes advantage of the merge sort as sorting algorithm and normal array as data structure, which has its limitation, that is, not efficient enough. There still remains potential for improvement. And we think 2-3 Tree, by which the improved design will be realized, is more promising, because it sacrifices space and trades that for speed. A contrast between these two sorting algorithms is about to be explained as following chapter.

III. 2-3 TREE SORT AND MERGE SORT

The difference between two designs is that the original utilizes the merge sort to sort intermediate data in reduce, while the revised one adopts 2-3 tree to do so. And merge sort offers O (nlogn) performance while the time complexity of the revised one is O (logn) (The time complexity of insertion method of 2-3 tree). Therefore, we make the prediction that the revised algorithm will end up with a performance improvement.

A. 2-3 Tree Sort

2-3 Tree is a type of data structure like B-Tree widely used to improve the performance in insertion, deletion, searching and sorting of data. 2-3 Tree has 3 main features. Its structure is shown in Fig 3.

- For any non-leaf node, it has either 2 or 3 children strictly;
- All leaves lie on the same level (the bottom level);
- Records can be comparable with each other. Each node has six fields, as shown in Fig 3. P1 points to the left child, P2 points to the middle child and P3 points to the right child. V1 represents the max/min record in the sub-tree of P1. V2 corresponds to P2 and V3 to P3. Note that V3 and P3 can be empty. Only leaves contain the records. Intermediate nodes and root only contain the references and max/min value.

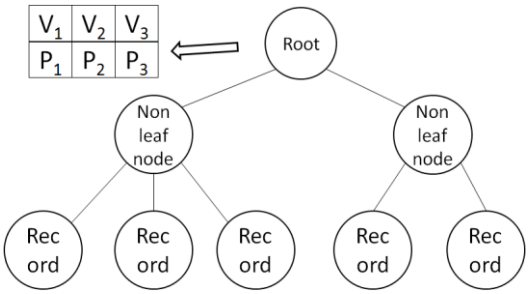

Fig. 3. 2-3 Tree Strcuture

Here is the pseudo code for the 2-3 Tree insert algorithm. The function has two parameters, one is the tree and the other is the element to be inserted to the tree. The function return the previous tree with x inserted correctly.

2-3Tree Insert (2-3 Tree tree, Element x)

If tree is null, put x to a leaf p, tree = p;

If tree has only a root which is a leaf, let tree be a 2-leaf tree with a root. One leaf contains the original leaf's record, the other leaf contains the x;

Add (2-3 Tree tree, Element x, 2-3 Tree p');

If p' != null, make a new root R, let tree be the left child of R, p' be the middle child of R, tree = R;

The insertion algorithm of 2-3 tree is described as above, in which there is a recursive function Add. The detail of the Add function is presented as Fig 4.

```
Add (2|3 Tree p, Element x, 2|3 Tree p'){
  If (p is the father of leaf){
    If (p has 2 children){
      p' = null;
      insert x;
    }
    If (p has 3 children) {
      Cut one child off from p, and make p' has two children
      from which one is x and the other is the cut off child;
    }
  }
  Case (x <= v1)
    p'' = p1;
  Case (v1 < x <= v2 || v3 == null)
    P'' = p2;
  Otherwise:
    P'' = p3;
  Add (p'', x, p0);
  If (p0 != null) {
    If (p has 2 children)
      Add p0 as a 3rd child properly;
    If (p has 3 children)
      Divide the 3 children of p (p1, p2 and p3) and
      p0 (in total 4 sub trees) into 2 sub trees g1 and g2 properly,
      let g1 = p, g2 = p', each has two sub trees.
  }
}
```

Fig. 4. Recursive function Add in the insertion algorithm

Finally as the tree constructed, we can see all the records in the leaves are sorted by the defined element comparing rules from left leaves to right leaves. Then the tree is required to be traversed to emit all the records in the reduce phase.

Because the 2-3 Tree requires more space to store extra nodes except for those who contain records, optimization has been applied when realizing the algorithm. For instance, we use the custom class instead of numeric to present the record. Also, StringBuilder instead of String is used in the traversal method.

B. Comparison between 2-3 Tree Sort and Merge Sort

On the other hand, the original design puts all entries from the hash described in the end of the recommendation algorithm into an array collection and sorts the elements by taking advantage of the merge sort.

Merge sort [10] is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort [11] [12].

The original merge sort is realized in Java which sorts the specified list according to the order induced by the specified comparator. This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort. This algorithm offers guaranteed O (nlogn) performance.

Because we can see a difference in two versions that original process uses merge sort which offers O (nlogn) performance while the revised one utilizes the 2-3 Tree with time complexity of insertion method as O (logn) to sort values in reduce function, we make the prediction that the revised algorithm will end up with a performance improvement. Therefore related experiments are designed and planned to test the assumption.

IV. Experiment Environment

A. *Cluster Configurations*

We use a cluster of 8 nodes. One is name node and job tracker, one is secondary name node and the other 6 nodes are data nodes and task trackers as slave nodes for HDFS storage and MapReduce task execution. We set the HDFS block size to its default value 64 MB. Each slave node was configured with 2 map slots and 2 reduce slots, resulting in a total capacity of running 12 map and 12 reduce tasks simultaneously in the cluster. Each machine has 4GB RAM, two 2.4GHz Intel® Core™2 CPU 6600 64-bit processor and one 7200 rpm SATA disk with 160GB. The network typology is shown in Fig 5 from which the rack structure is displayed. All nodes were interconnected by a Gigabit Ethernet. Each node is installed the Linux Ubuntu 12.04 operating system, JDK 7, Hadoop stable release version 1.2.1and SSH. The hostnames, Linux UserName, machine NO.s and IP addresses are shown as table nodes' information.

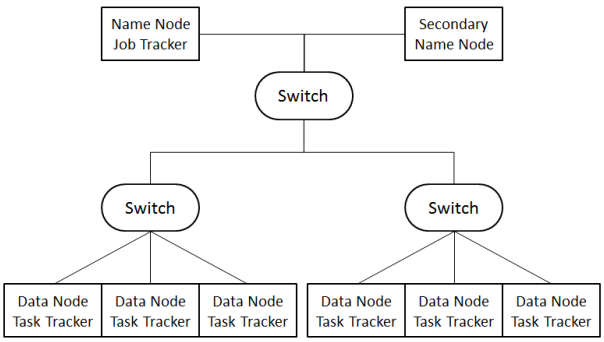

Fig. 5. Network Typology

B. *Dataset: Generation of Input Friends List*

The input data should be similar to the real world SNS user relationships. So the number of friends one user could have is by no means a random number. And how many friends one could have should follow some rules and disciplines. Since the scenario is one of the applications from those bi-directional relationship pattern SNS such as Facebook, we take a look at the Facebook friends statistics.

According to The Anatomy of the Facebook Social Graph by Johan Ugander et al. [13], they measured how many friends people have with a conclusion that the average friend count is 190. And according to the Data Science of the Facebook World [14] as shown in Fig 6 and the deduction by Self-learner project: Friendship Paradox [15], the distribution can be summarized as gamma distribution with 3 features: 1. The distribution should be self-similar; 2. The shape of the distribution of the friends of friends should resemble the shape of distributions of friends of friends of friends. The distribution must be positive; you can't have less than 0 friends; 3. The distribution should have a long right tail.

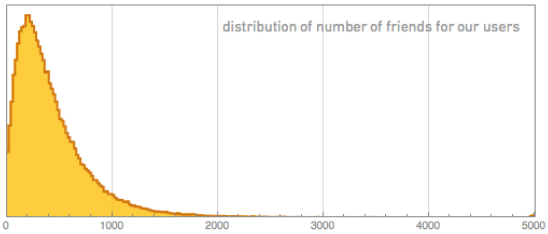

Fig. 6. Facebook - distribution of number of friends

Therefore we generate the user friends list to make the number of friends follow a Gamma (3, 67) distribution. Scale is 3 and shape is 67 so that the average number of friends one user could have is about 200. And the mode is Shape*(Scale-1) = 134. The distribution of number of friends we adapted to the input data is shown in Fig 7.

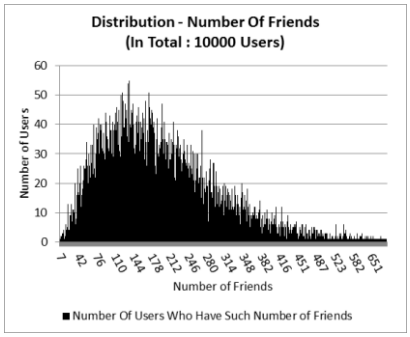

Fig. 7. Our data – distribution of number of friends

C. *Measurements*

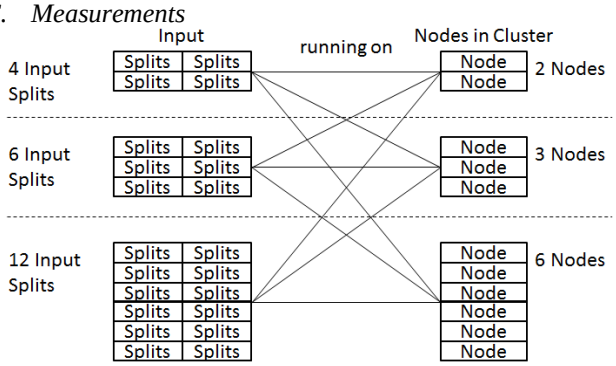

Fig. 8. Experiment Group Design

We separately test the original version of application and the improved one by using 12, 6, 4 input splits on 6, 3, 2 data nodes (also the number of task trackers) cluster as shown in Fig 8. The reason we choose these variables can be illustrated by

the fact that different input splits and the number of nodes determines whether all the map task slots available can be fully taken advantage of in the cluster. In our experimental configuration, the maximum task each machine has is 2 so that 6-Node cluster is capable of loading 6 * 2 = 12 map tasks at one time. Hence, if we use 12 splits input data, all files can be read into the map tasks, so the cluster can be described as completely utilized. But on the other occasion, we remove 3 nodes from the cluster, which makes it only 3 left, and there are still 12 splits, as a result, only 6 of them will be loaded first. The system will still create 12 map tasks, but 6 of them will not relieve pending status until some map task has been finished and there is some map slot available for another pending map task to run. That means some tasks have to wait and the system is overloaded. On the third situation, we still have 3 active nodes but with just 2 splits. It is not hard to image, only two map tasks will be running and because the less number of splits, the larger each split will be, it will take much longer time to process each map task, and we call that underutilized. All input data are the same with 10000 users and have the number of friends' distribution: Gamma (3, 67). We calculated each reduce task execution time in each scenario. Then the average execution time of user defined reduce functions are calculable. All required calculations are shown as below.

However, the reduce task contains three phase: shuffle, sort and the user defined function. So the time of user defined reduce function can be counted as the formula:

Time (User Defined Reduce Function) = Time (Whole Reduce Phase) – Time (Shuffle) – Time (Sort) (1)

And the average time each reduce task spent is:

AVG Time (Reduce) = sum of time of each reduce task/ number of reduce tasks (2)

In addition, to compare the different memory usage of both applications, the sum of physical memory each reduce task occupied is recorded.

Physical Memory Usage (Reduce) = sum of physical memory usage of each reduce task (3)

We test the average reduce process time and memory usage in the application which has the original sorting algorithm and in that which has the improved sorting algorithm under the Cartesian product of two-dimension configurations, that is, comparing both time as number of map tasks and cluster nodes changes.

V. Results Analysis

The Fig 9 shows the detailed experimental results from which we can see the different time consumption of each part.

Configuration	Algorithm Used	Whole Job Time (sec)	Reduce		
			Avg Reduce Task Time (sec)	Avg Reduce Function Time (sec)	Physical Memory Usage (bytes)
12 splits 6 Nodes	Original	883	346.14	225.76	4,965,109,760
	Revised	841	324.57	199.86	5,182,189,568
12 splits 3 Nodes	Original	1202	518.60	261.80	1,509,027,840
	Revised	1123	507.80	241.10	1,575,174,144
12 splits 2 Nodes	Original	1493	662.71	213.71	999,854,080
	Revised	1426	607.29	168.43	1,051,287,552
6 splits 6 Nodes	Original	1037	351.86	240.00	3,155,374,080
	Revised	1030	322.86	212.90	3,348,054,016
6 splits 3 Nodes	Original	1355	446.80	274.90	1,464,164,352
	Revised	1207	354.10	234.60	1,546,625,024
6 splits 2 Nodes	Original	1685	599.14	209.71	1,026,908,160
	Revised	1576	553.86	167.86	1,114,402,816
4 splits 6 Nodes	Original	1260	347.90	230.33	3,154,132,992
	Revised	1209	315.10	205.67	3,371,548,672
4 splits 3 Nodes	Original	1712	480.80	254.10	1,469,228,448
	Revised	1563	387.50	211.40	1,609,179,136
4 splits 2 Nodes	Original	1668	463.00	200.00	1,005,510,656
	Revised	1646	400.29	150.00	1,117,442,048

Fig. 9. Experimental Results

Fig 10 shows the physical memory usage of all combinations of the experimental variables such as the number of input splits, i.e., number of map tasks, and the number of active nodes.

From the fig we can see for the same-node scenarios, no matter how many splits there are, the physical memory usage are almost the same among all original design as well as among all revised design except for the 12 splits in 6 nodes scenarios. It results from the facts that more reduce tasks are killed in 12 splits-6 nodes scenario than that in 6 splits-6 nodes and 4 splits-6 nodes scenarios. Those tasks may confront with some runtime error and end up being failed. In a word, extra tasks cause more memory usage.

Also as a crucial discovery, under the same input splits and active nodes environment, the revised version required cost more physical memory than the original one did. While compared within each version and each input split scenario, the physical memory usage increased as the number of active nodes rises. The reason is because 2-3 Tree data structure requires more memory for indexing. It treats the space for efficiency. There are plenty of intermediate nodes which take extra room besides the store-in-leaf records. As a result, the initialization and the garbage collection of these objects need more time. While looking at the merge sort version, the space array occupies is the same as the amount of records' demand. Therefore it is reasonable that the improved application makes use of more physical memory that the original one does.

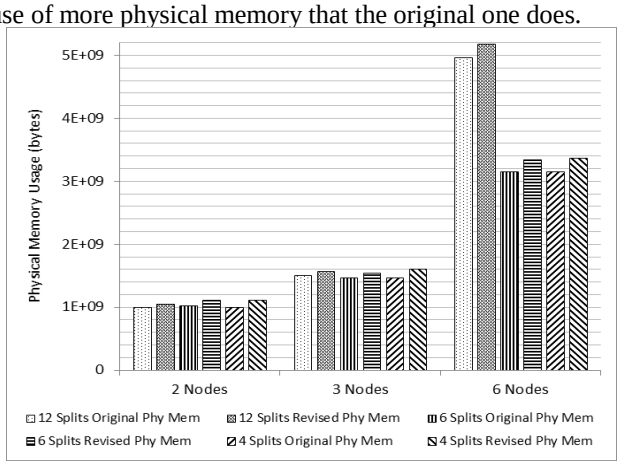

Fig. 10. Physical memory usage

Fig 11 shows the comparison among average user defined reduce function running time of all combinations of the experimental variables.

The difference can be told from the fig that the revised application performs better than the original one under the same number of input splits and the same number of active nodes situation. Noted that when the cluster contains only 2 nodes, the performance is much better than that under which scenario the cluster contains more than 2 nodes. In addition, the performance of 6 nodes scenarios outweigh 3 nodes ones grouped by each kind of version while the performance of 3 nodes scenarios do worse than that of 2 nodes ones. This is because when there are 3 or 6 active nodes in the cluster, the number of data replicas is 3, all output data will be copied twice more for the data redundancy, but it can only be 2 when there are only 2 active nodes and all output data will be copied just once more, which saves a lot of time. As a result, the performance of 2 nodes scenarios overshadows that of other configurations.

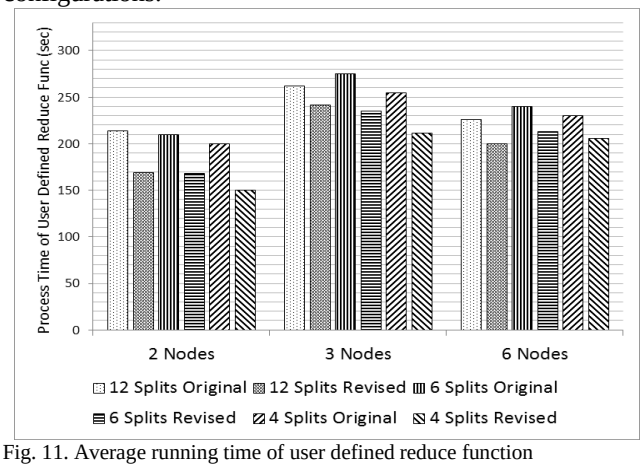

Fig. 11. Average running time of user defined reduce function

VI. Conclusions and Future Work

We present an improved algorithm for mutual friends recommendation application of SNS in Hadoop by taking advantage of the 2-3 Tree data structure to conduct the sorting algorithm in reduce phase rather than the original merge sort algorithm by array data structure.

From the result analysis of the experiments, we draw the conclusion as below. Under the same number of active nodes and the same number of input splits situation, the improved algorithm is better than the original one.

From a dynamic perspective, when the number of data splits is fixed, the more available nodes in the cluster to process the application there are, the better we should apply the improved algorithm. When the number of nodes is fixed, the less input splits there are, the more appropriate it is to utilize the revised design.

In future work, we plan to increase the applicability of our work to a wider range of MapReduce applications and the next generation of MapReduce -- MapReduce 2. As the upgrade of hardware, we are considering promoting the usage of the improved algorithm to the Map Reduce 2 and even in in-memory database configuration, which allows reduce phase to copy and store intermediate data in ram instead of hard disk, under which circumstance the I/O will be considerably cut down. Also, it is possible that multiple tasks can run on one processor at the same time. As a result, we would like to move further steps to set up experiments to see if the number of splits can be increased and the task trackers can be further utilized.

Acknowledgment

The paper is directed and supervised by Prof. Nobuhiko KOIKE. All research and experimental materials are supported by the KOIKE's parallel computing laboratory and the Faculty of Computer and Information Sciences, Hosei Unversity.

References

[1] Map Reduce - Finding Friends. http://stevekrenzel.com/finding-friends-with-mapreduce

[2] Writing scalable recommender system with Hadoop. http://pulkitgoyal.in/writing-scalable-recommender-system-with-hadoop/

[3] Z. GuoGuang and D. Bu, "CloudLCA: finding the lowest common ancestor in metagenome analysis using cloud computing," Protein & Cell, (2012), vol 3(2), pp. 148-152.

[4] C. Rishan and X. Weng, "Improving large graph processing on partitioned graphs in the cloud," Proceedings of the Third ACM Symposium on Cloud Computing. ACM, (2012).

[5] K. Lee, Ling Liu, "Efficient data partitioning model for heterogeneous graph in the cloud," SC 13 November 17-21 (2013).

[6] "People You May Know" Friendship Recommendation with Hadoop. http://importantfish.com/people-you-may-know-friendship-recommendation-with-hadoop/

[7] D. Jeffrey, and S. Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008), pp. 107-113.

[8] W. Tom. "Hadoop: the definitive guide: the definitive guide." O'Reilly Media, Inc., 2009.

[9] Welcome to Apache Hadoop. http://hadoop.apache.org/

[10] R. Raghu, J. Gehrke, and J. Gehrke, "Database management systems." New York: McGraw-Hill, Vol. 3, (2003).

[11] Merge Sort. http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm

[12] Merge Sort. http://en.wikipedia.org/wiki/Merge_sort

[13] U. Johan, "The anatomy of the facebook social graph." arXiv preprint arXiv:1111.4503 (2011).

[14] Data Science of the Facebook World . http://blog.stephenwolfram.com/2013/04/data-science-of-the-facebook-world/

[15] Self-learner project: Friendship Paradox. http://badmomgoodmom.blogspot.jp/2013/01/self-learner-project-friendship.html