2PTS: A Two-Phase Task Scheduling Algorithm for MapReduce

Byungnam LIM(a), Yeeun SHIM(b), Nonmembers, and Yon Dohn CHUNG(c), Member

SUMMARY For an efficient processing of large data in a distributed system, Hadoop MapReduce performs task scheduling such that tasks are distributed with consideration of the data locality. The data locality, however, is limitedly exploited, since it is pursued one node at a time basis without considering the global optimality. In this paper, we propose a novel task scheduling algorithm that globally considers the data locality. Through experiments, we show our algorithm improves the performance of MapReduce in various situations.

key words: MapReduce, task scheduling algorithm, data locality

1. Introduction

With the explosion of data, distributed/parallel systems have become essential for the large data processing. MapReduce[1] is one of the most famous distributed data processing frameworks. It is easy to use, independent to input data types, scalable and fault tolerant. For these reasons, MapReduce is widely used in industries as well as in scientific analysis[2], [3].

Apache Hadoop is a distributed storage and processing framework which is comprised of the storage part, HDFS (Hadoop Distributed File System)[4] and the data processing part, MapReduce. HDFS splits files into smaller blocks and replicates them in multiple nodes to ensure the data can be accessed even if there are failures such as unexpected power downs or disk breaks.

Hadoop MapReduce is an implementation of the MapReduce programming model for processing the data stored in HDFS. It is composed of two parts, Map and Reduce, which are comprised of map tasks and reduce tasks respectively. A map task is responsible for reading one data block in a node and writing its output into key-value pairs. Since the map task is only responsible for one data block, the number of map tasks is same to the number of the targetted data blocks. Figure 1 describes an overview of the Hadoop MapReduce program execution. Each map task reads its target data block and writes the output as key-value pairs, and then the reduce tasks are processed.

Assigning map tasks to nodes where the target data blocks are stored in its local disk is important for the fast execution. If a data block is not present at the node, the data should be fetched from other nodes through the network (we call this a remote access), which increases the execution time. Hence, minimizing remote accesses should be achieved through the task scheduling. We denote a task whose data block is stored in the local disk a local task. The task which requires a remote access is called a remote task.

In this paper, we propose a novel task scheduling algorithm that considers the global data locality. This algorithm first computes and gather information required for the scheduling, and schedules tasks with consideration of global optimality. Through the experiments, we show that our algorithm reduces the number of remote tasks and the execution time.

2. Backgrounds

2.1 Task Scheduling in Hadoop MapReduce

Hadoop MapReduce uses HDFS (Hadoop Distributed File System) for its storage system. In HDFS, the data are stored in several nodes and they are accessed through multiple map tasks. For the task distribution, Hadoop employes a task scheduler which coordinates and manages the tasks to be efficiently processed.

An effective task scheduling is important for Hadoop MapReduce since the data are splitted and distributed. The current task scheduling algorithm in Hadoop MapReduce considers the data locality (in a limited way) when assigning tasks. When a node requires a task, the scheduler firstly assigns a task which is local to the node if any. Algorithm 1 describes the scheduling algorithm used in the current Hadoop MapReduce. This algorithm takes the task request events from the worker nodes as input. For each worker node that needs a task to process, the scheduler finds the local task for...
Algorithm 1 The conventional MapReduce task scheduling algorithm

Input: Task request events from worker nodes
Output: Assignment of tasks for the worker nodes

1: for each worker node $w$ that needs a task to process do
2: if there exist a local task $t$ for the node $w$ then
3: assign $t$ to $w$
4: else
5: assign any remaining task to $w$
6: end if
7: end for

2.2 Problem Statement

The conventional task scheduling algorithm of Hadoop MapReduce, however, has a problem. It only considers the targeted node alone without concerning the other nodes. Note that assigning one task on a node also removes the opportunity of a local task for the other nodes which have the same data block. The current algorithm selects local tasks randomly, which could cause some nodes to lose its local tasks. When a node has no locally accessible data left, it has to take remote tasks after all.

Figure 2 shows an example of task scheduling with the current task scheduling algorithm. There are three nodes and three data blocks ($b_x$, $b_y$, and $b_z$) which are replicated in two nodes. Also there are three tasks ($t_x$, $t_y$, and $t_z$), where $t_x$, $t_y$, and $t_z$ access data blocks $b_x$, $b_y$, and $b_z$, respectively. Figure 2(a) shows an optimal case of the scheduling where all the tasks are assigned as local tasks. Figure 2(b) is another possible task assignment, which is not an optimal. When the task scheduler first sees Node 1, there are two possibility of local tasks, $t_x$ and $t_y$, and selects one randomly, in this example, $t_x$. The scheduler continues to schedule for Node 2 and it also sees two local tasks $t_y$ and $t_z$, and selects $t_z$. Then, Node 3 loses all the local tasks and it is forced to take the remaining remote task, $t_y$.

This example shows the problem of current scheduling algorithm that is the randomness of selecting local tasks for a node without concerning the others could result in a globally suboptimal assignment. In this paper, we propose a novel task scheduling algorithm for Hadoop MapReduce that considers the data placement over the entire cluster and schedules tasks with the consideration of the global data locality.

3. Related Work

The delay scheduling algorithm [6] is a famous way to improve the data locality, first developed by Facebook. The delay algorithm reduces remote tasks by skipping the task assignment when the idle node does not have any available local tasks. This is based on the inspiration that, instead of assigning a remote task for the node, it would wait for the other node to take the task, whereby the task can be processed as a local task.

For the delay scheduling method, it is important to determine the amount of delay time. If the delay time is too long, the delay itself is longer than the time required for processing the task. On the contrary, if the delay time is too short, then the delay would be useless because there would be few other nodes that can take the task. Note that an optimal delay time depends on the complexity of the job and the time required for a task to be processed, so it is almost impossible to find the optimal value that satisfies all of the job styles.

4. 2PTS: The Proposed Task Scheduling Algorithm

In this section, we propose a novel task scheduling algorithm that concerns the global data locality. Since the task scheduling problem is known to be NP-hard [5], we develop a heuristic algorithm. This algorithm is composed of two phases; it obtains global mapping information of the nodes and tasks in the first phase, and schedules tasks based on the information in the second phase.

1. In the first phase, we investigate the information of data blocks, tasks and the performance capability of each node so that we can calculate how many tasks can/should be given to the nodes respectively.
2. In the second phase, we iteratively pick tasks and determine nodes to process them based on the information we construct in the first phase.

The notations used in this paper are summarized in Table 1. We now explain and describe each of these steps in detail.

4.1 Phase 1: Information Investigation

The information investigation phase calculates the number of tasks to be processed in each node, TaskCap, sets of local tasks for each node, LocalTasks, and sets of local nodes for each task, LocalNodes. We can get the information of LocalTasks and LocalNodes from the MapReduce itself. So we only need to determine how many tasks can be processed in a node since the performance of each node may differ. Hence, we calibrate a node with its CPU and main memory, and numerate it to \( p(w_i) \) for node \( w_i \). After calculating \( p(w_i) \), we decide how many tasks would be processed for each node. We set TaskCap\((w_i)\) to be proportional to \( p(w_i) \) multiplied by the number of entire tasks, \( k \) (Eq. (1)).

\[
\text{TaskCap}(w_i) = \left[ \frac{p(w_i)}{\sum p(w)} \cdot k \right]
\]

(1)

4.2 Phase 2: Task Scheduling

The task scheduling phase computes the sets of tasks for each node and stores them. Algorithm 2 shows the procedure and detailed descriptions are in the below:

First, we pick a task \( t_j \) that has the smallest number of LocalNodes, except when the number is zero. Note that this task \( t_j \) has the lowest possibility of being a local task since other tasks have more nodes that can locally access the data, hence we schedule this task earlier than others. But if the task has no LocalNodes left, then we assume this task would have no chance of being a local task so abandon this task to be handled later. After selecting the task, we search for a node \( w_i \) that can locally access the data of \( t_j \) and has the smallest number of LocalTasks. You should notice that this node \( w_i \) has the highest probability of losing all of its local tasks and being forced to take remote tasks. After we find both the task and the node, schedule the task \( t_j \) to SchedTasks\((w_i)\) and remove it from the LocalTasks of other nodes because it should not be processed redundantly. We repeat this process until the size of SchedTasks\((w_i)\) reaches at TaskCap\((w_i)\), and then the node \( w_i \) is removed from the available node list \( W \) and the available local nodes LocalNodes for the remaining tasks. This is repeated until there are no more tasks to be scheduled.

After computing SchedTasks for each node, we start the actual processing of the MapReduce job. When a node \( w_i \) requires for a task, the scheduler takes a task from SchedTasks\((w_i)\) and assigns to the node. (obviously, it will always be the local task of the node). Eventually, there can happen one or more nodes that finish their work earlier than others. In this case, we use the original MapReduce scheduling algorithm and assign remaining tasks.

5. Performance Evaluation

We here show how the proposed algorithm can reduce remote tasks compared to the previous scheduling algorithms. We set the delays to be 3, 5 and 10 seconds for the delay scheduling algorithm.

5.1 Experiment Setup

We used Apache Hadoop version 2.6.0 [7] to implement our
algorithm. This version of Hadoop contains both HDFS and MapReduce of version 2. We used our in-house 32-node cluster where each node comprises Intel(R) Core(TM) i5-2500 3.30 GHz CPU, 16 GB of RAM, and two 2TB disks, and connected with 1 Gbps LAN. We configured the replication factor to 3 and the sizes of the dataset are approximately 10, 30 and 100GB. In experiments, we used three types of application; wordcount, sort, and scan where the wordcount and sort have CPU-heavy workloads and scan has IO-heavy workloads.

5.2 Experiment Result

Figure 3 plots the execution time and the reduction ratio of remote tasks. We represent the execution time as bar graphs and the remote task reduction ratio as vertical lines. For all the cases, execution times are improved about 10% compared to the current MapReduce task scheduling. Delay algorithms shows the better improvements in the reduction of remote tasks, but they are consistently inferior at the actual execution time than the proposed 2PTS algorithm. This is because the delay algorithm suffer from more nodes that have run out of local tasks than the proposed algorithm and pays more time for delays. Moreover, when the data are large and the job has CPU-heavy workload, we can see the proposed algorithm works better in both aspects - execution times and remote task reduction ratios. This shows that the proposed algorithm is independent to the workload style while the delay algorithm can be underutilized when nodes require longer execution time to finish each of tasks and fails to find nodes to take local tasks between the delay time.

6. Conclusion

In this paper, we proposed a two-phase task scheduling algorithm that reduces the number of remote tasks. In the first phase, we investigate information of tasks and nodes and calibrate each node. In the second phase, the proposed algorithm schedules tasks with the global consideration of both the tasks and nodes so that nodes do not suffer from being ran out of local tasks earlier than others. By experiments, we found that our proposed two-phase task scheduling can improve the execution time compared to the previous task scheduling algorithms by globally considering the locality of tasks. Also the result shows that the proposed algorithm is robust to the size of data and types of workloads.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A1A11053657).

References