Dynamic Slot Allocation Technique for MapReduce Clusters

Shanjiang Tang, Bu-Sung Lee, Bingsheng He
School of Computer Science&Technology
Nanyang Technological University
{*stang5, ebslee, bshe*}*@ntu.edu.sg*

Abstract—**MapReduce is a popular parallel computing paradigm for large-scale data processing in clusters and data centers. However, the slot utilization can be low, especially when Hadoop Fair Scheduler is used, due to the pre-allocation of slots among map and reduce tasks, and the order that map tasks followed by reduce tasks in a typical MapReduce environment. To address this problem, we propose to allow slots to be dynamically (re)allocated to either map or reduce tasks depending on their actual requirement. Specifically, we have proposed two types of Dynamic Hadoop Fair Scheduler (DHFS), for two different levels of fairness (i.e., *cluster* and *pool* level). The experimental results show that the proposed DHFS can improve the system performance significantly (by $32\% \sim 55\%$ for a single job and $44\% \sim 68\%$ for multiple jobs) while guaranteeing the fairness.**

Keywords-**MapReduce, Hadoop, Fair Scheduler, Dynamic Scheduling, Slots Allocation.**

I. Introduction

In recent years, MapReduce has become the parallel computing paradigm of choice for large-scale data processing in clusters and data centers. A MapReduce *job* consists of a set of map and reduce *tasks*, where reduce tasks are performed after the map tasks. Hadoop [1], an open source implementation of MapReduce, has been deployed in large clusters containing thousands of machines by companies such as Yahoo! and Facebook to support batch processing for large jobs submitted from multiple users (i.e., MapReduce workloads).

In a Hadoop cluster, the compute resources are abstracted into map (or reduce) slots, which are basic compute units and statically configured by administrator in advance. Due to 1) the slot allocation constraint assumption that map slots can only be allocated to map tasks and reduce slots can only be allocated to reduce tasks, and 2) the general execution constraints that map tasks are executed before reduce tasks, we have two observations: (I). there are significantly different performance and system utilization for a MapReduce workload under different job execution orders and map/reduce slots configurations, and (II). even under the optimal job submission order as well as the optimal map/reduce slots configuration, there can be many idle reduce (or map) slots while map (or reduce) slots are not enough during the computation, which adversely affects the system utilization and performance.

In our work, we address the problem of how to improve the utilization and performance of MapReduce cluster without any prior knowledge or information (e.g., the arriving time of MapReduce jobs, the execution time for map or reduce tasks) about MapReduce jobs. Our solution is novel and straightforward: we break the former first assumption of slot allocation constraint to allow *(1). Slots are generic and can be used by map and reduce tasks. (2). Map tasks will prefer to use map slots and likewise reduce tasks prefer to use reduce slots.* In other words, when there are insufficient map slots, the map tasks will use up all the map slots and then borrow unused reduce slots. Similarly, reduce tasks can use unallocated map slots if the number of reduce tasks is greater than the number of reduce slots. In this paper, we will focus specifically on Hadoop Fair Scheduler (HFS). This is because the cluster utilization and performance for the whole MapReduce jobs under HFS are much poorer (or more serious) than that under FIFO scheduler. But it is worth mentioning that our solution can be used for FIFO scheduler as well.

HFS is a two-level hierarchy, with task slots allocation across "pools" at the top level, and slots allocation among multiple jobs within the pool at the second level [2]. We propose two types of Dynamic Hadoop Fair Scheduler (DHFS), with the consideration of different levels of fairness (i.e., pool-level and cluster-level). They are as follows:

- *Pool-independent DHFS (PI-DHFS).* It considers the dynamic slots allocation from the cluster-level, instead of pool-level. More precisely, it is a typed phase-based dynamic scheduler, i.e., the map tasks have priority in the use of map slots and reduce tasks have priority to reduce slots (i.e., intra-phase dynamic slots allocation). Only when the respective phase slots requirements are met can excess slots be used by the other phase(i.e., inter-phase dynamic slots allocation).
- *Pool-dependent DHFS (PD-DHFS).* It is based on the assumption that each pool is selfish, i.e., each pool will always satisfy its own map and reduce tasks with its shared map and reduce slots between its map-phased pool and reduce-phased pool (i.e., intra-pool dynamic slots allocation) first, before sharing the unused slots with other overloaded pools (i.e., inter-pool dynamic slots allocation).

We have designed and implemented the two DHFSs on top of default HFS. We evaluate the performance and fairness of our proposed algorithms with synthetic workloads. Both schedulers, PI-DHFS and PD-DHFS, have shown promising results. The experimental results show that the proposed DHFS can improve the system performance significantly (by $32\% \sim 55\%$ for a single job and $44\% \sim 68\%$ for multiple jobs) while guaranteeing the fairness.

Organization. The rest of the paper is organized as follows.

Section II reviews the MapReduce background and related work. Section III introduces our two types of Dynamic Hadoop Fair Scheduler, namely, PI-DHFS and PD-DHFS. Section IV reports on the performance improvement of proposed DHFS obtained from our experiments. Section V discusses fairness and slots movement for PI-DHFS and PD-DHFS. Finally, Section VI concludes the paper and gives our future work.

II. Preliminary and Related Work

A. MapReduce

MapReduce is a popular programming model for processing large data sets, initially proposed by Google [16]. Now it has been a de facto standard for large scale data processing on the cloud. Hadoop [1] is an open-source java implementation of MapReduce. When a user submits jobs to the Hadoop cluster, Hadoop system breaks each job into multiple map tasks and reduce tasks. Each map task processes (i.e. scans and records) a data block and produces intermediate results in the form of key-value pairs. Generally, the number of map tasks for a job is determined by input data. There is one map task per data block. The execution time for a map task is determined by the data size of an input block. The reduce tasks consists of shuffle/sort/reduce phases. In the shuffle phase, the reduce tasks fetch the intermediate outputs from each map task. In the sort/reduce phase, the reduce tasks sort intermediate data and then aggregate the intermediate values for each key to produce the final output. The number of reduce tasks for a job is not determined, which depends on the intermediate map outputs. We can empirically set the number of reduce tasks for a job to be $0.95\times$ or $1.75\times$ reduce tasks capacity [17].

There are several job schedulers for Hadoop, i.e., *FIFO*, *Hadoop Fair Scheduler* [2], *Capacity Scheduler* [18]. The job scheduling in Hadoop is performed by the jobTracker (master), which manages a set of taskTrackers (slaves). Each taskTracker has a fixed number of map slots and reduce slots, configured by the administrator in advance. Typically, there is one slot per CPU core in order to make CPU and memory management on slave nodes easy [2]. The taskTrackers report periodically to the jobTracker the number of free slots and the progress of the running tasks. The jobTracker allocates the free slots to the tasks of running jobs. In particular, the map slots can only be allocated to map tasks and reduce slots can only be allocated to reduce tasks.

Hadoop Fair Scheduler [2] is a multi-user MapReduce job scheduler that enables organizations to share a large cluster among multiple users and ensure that all jobs get roughly an equal share of slot resources at each phase. It organizes jobs into pools and shares resources fairly across all pools based on max-min fairness [3]. By default, each user is allocated a separate pool and, therefore, gets an equal share of the cluster no matter how many jobs they submit. Each pool consists of two parts: map-phase pool and reduce-phase pool. Within each map/reduce-phase pool, fair sharing is used to share map/reduce slots between the running jobs at each phase. Pools can also be given weights to share the cluster non-proportionally in the configuration file.

B. Related Work

There is a large body of research work that focuses on the performance optimization for MapReduce jobs. Broadly, it can be classified into the following two categories.

• ***Data Access and Sharing Optimization***.

Jiang et al. [4] propose a set of general low-level optimizations including improving I/O speed, utilizing indexes, using fingerprinting for faster key comparisons, and block size tuning. Thus, they were focused on fine-grain tuning on different parameters to achieve performance improvements. Agrawal et al. [5] proposed a method to maximize scan sharing by grouping MapReduce jobs into batches so that sequential scans of large files are shared among as many simultaneous jobs as possible. MRShare [6] is a sharing framework that provides three possible work-sharing opportunities, including scan sharing, mapped outputs sharing, and Map function sharing across multiple MapReduce jobs, to avoid performing redundant work and thereby reduce total processing time. MapReduce Online [7] is such a modified MapReduce system to support online aggregation for MapReduce jobs that run continuously by pipelining data within a job and between jobs. LEEN [21] addresses the fairness and data localities.

All these studies are complementary to our study and our approach can be incorporated into these modified MapReduce frameworks (e.g., MRShare [6], MapReduce Online [7]) for further performance improvement. In contrast, our work belongs to the computation and scheduling optimization. Specifically, we focus on improving the performance for MapReduce workloads by maximizing the cluster computation utilization.

• ***Computation and Scheduling Optimization***.

There are some computation optimization and job scheduling work that are related to our work. [8], [9], [10], [19], [20] consider job ordering optimization for MapReduce workloads. They model the MapReduce as a two-stage hybrid flow shop with multiprocessor tasks [13], where different job submission orders will result in varied cluster utilization and system performance. However, there is an assumption that the execution time for map and reduce tasks for each job should be known in advance, which may not be available in many real-world applications. Moreover, it is only suitable for independent jobs, but fails to consider those jobs with dependency, e.g., MapReduce workflow. In comparison, our DHFS is not constraint by such assumption and can be used for any types of MapReduce workloads (i.e., independent and dependent jobs).

Hadoop configuration optimization is another approach, including [11], [12]. For example, Starfish [11] is a self-tuning framework that can adjust the Hadoop's configuration automatically for a MapReduce job such that the utilization of Hadoop cluster can be maximized, based on the cost-based model and sampling technique. However, even under an optimal Hadoop configuration, e.g., Hadoop map/reduce slots configuration, there is still room for performance improvement of a MapReduce job or workload, by maximizing the utilization of map and reduce slots.

Guo et al. [15] propose a resource stealing method to enable running tasks to steal resources reserved for idle slots and give them back proportionally whenever new tasks are assigned, by adopting multithreading technique for running tasks on multiple CPU cores. However, it cannot work for the utilization improvement of those purely idle slave nodes without any running tasks. Polo et al. [14] present a resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization by extending the abstraction of traditional 'task slot' of Hadoop to 'job slot', which is an execution slot that is bound to a particular job, and a particular task type (map or reduce) within that job. In contrast, in our proposed schedulers, we keep the traditional task slot model and maximize the system utilization by dynamically allocating unused map (or reduce) slots to overloaded reduce (or map) tasks.

III. Dynamic Hadoop Fair Scheduler (DHFS)

In MapReduce, each job consists of a set of map and reduce tasks, with reduce tasks performed after map tasks. Map tasks are run on map slots and reduce tasks are run on reduce slots. However, this leads to severe under-utilizations of the respective slots. As the number of map and reduce tasks varies over time, the number of slots allocated for map/reduce can be greater than the number of map/reduce tasks. Our dynamic slots allocation policy is based on the observation that at different periods of time there may be idle map (or reduce) slots, as the job proceeds from map phase to reduce phase. We can use the unused map slots for those overloaded reduce tasks to improve the performance of the MapReduce workload, and vice versa. For example, at the beginning of MapReduce workload computation, there will be only computing map tasks and no computing reduce tasks. In that case, an ideal MapReduce framework should utilize the idle reduce slots for running map tasks, whereas current MapReduce does not. That is, we break the implicit assumption of current MapReduce framework that the map tasks can only run on map slots and reduce tasks can only run on reduce slots. Instead, we modify it as follows: *both map and reduce tasks can be run on either map or reduce slots.*

In addition to utilization, fairness is also another important consideration for Hadoop clusters. Based on different levels of fairness for our DHFS, in this section, we first propose two kinds of Dynamic Hadoop Fair Scheduler (DHFS), namely, pool-independent DHFS(PI-DHFS) and pool-dependent DHFS (PD-DHFS).

A. Pool-Independent DHFS (PI-DHFS)

Traditional Hadoop Fair Scheduler (HFS) is a two-level hierarchy. At the first level, HFS allocates task slots across pools, and at the second level, each pool allocates its slots among multiple jobs within its pool [2]. It adopts max-min fairness [3] to allocate slots across pools with minimum guarantees at the map-phase and reduce-phase, respectively. *Pool-Independent DHFS (PI-DHFS)* extends the Hadoop Fair Scheduler by allocating slots from the cluster global level, i.e., independent of pools. As shown in Figure 1, it presents the slots allocation flow for PI-DHFS. It is a typed phase-based dynamic slots allocation policy. The allocation process consists of two parts, as shown in Figure 1:

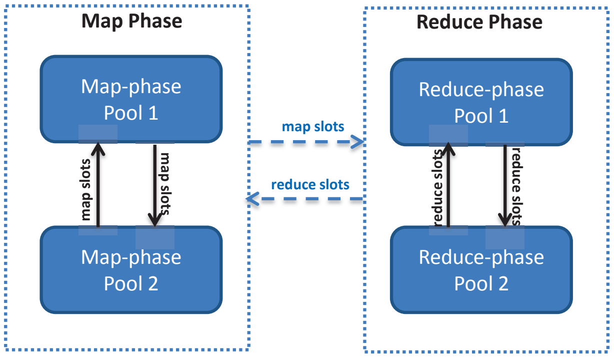


Fig. 1: Example of the fairness-based slots allocation flow for PI-DHFS. The black arrow line and dash line show movement of slots between the map-phase pools and the reduce-phase pools.

(1). *Intra-Phase dynamic slots allocation*. Each pool is split into two sub-pools, i.e., map-phase pool and reduce-phase pool. At each phase, each pool will receive its share of slots. An overloaded pool, whose slot demand exceeds its share, can dynamically borrow some unused slots from other pools of the same phase. For example, an overloaded map-phase Pool1 can borrow map slots from map-phase Pool 2 when Pool 2 is under-utilized, and vice versa.

(2). *Inter-Phase dynamic slots allocation*. After the intra-phase dynamic slots allocation for both the map-phase and reduce-phase, we can now perform dynamic slots allocation across typed phases. That is, when there are some unused reduce slots at the reduce phase and the number of map slots at the map phase is insufficient for map tasks, it will borrow some idle reduce slots for map tasks, to maximize the cluster utilization, and vice versa.

Thus, there are four possible scenarios. Let N_M and N_R be the number of map and reduce tasks respectively, while S_M and S_R be the number of map and reduce slots configured by users respectively. The four scenarios are as follows:

Case 1: When $N_M \leqslant S_M$ and $N_R \leqslant S_R$, the map tasks are run on map slots and reduce tasks are run on reduce slots, i.e., no borrow is needed across map and reduce slots.

Case 2: When $N_M > S_M$ and $N_R < S_R$, we satisfy reduce tasks for reduce slots first and then use those idle reduce slots for running map tasks.

Case 3: When $N_M < S_M$ and $N_R > S_R$, we can schedule those unused map slots for running reduce tasks.

Case 4: When $N_M > S_M$ and $N_R > S_R$, the system should be in completely busy state, and similar to (1), there will be no movement of map and reduce slots.

Next, it will perform intra-phase dynamic slots allocation for those borrowed map or reduce slots using max-min fairness within the phase.

The pseudocode for this algorithm is shown in Algorithm 1.

Whenever a heartbeat is received from a compute node, we first compute the total demand for map slots and reduce slots for the current MapReduce workload. Particularly, the demand for map slots is computed based on the number of pending map tasks plus the total number of currently used map slots, rather than the number of running map tasks. The reason is that in our dynamic slot allocation policy, the map slots can be used by reduce tasks, and map tasks can be running using reduce slots. For each tasktracker, the number of used map slots can be calculated based on the formula: $\min\{runningMapTasks, trackerMapCapacity\} + \max\{runningReduceTasks - trackerReduceCapacity, 0\}$. The formula is similarly used in the computation for reduce slots. We can then compute load factors for map tasks and reduce tasks. We next determine dynamically the need to borrow map (or reduce) slots for reduce (or map) tasks based on the demand for map and reduce slots, in terms of the above four scenarios. The specific number of map (or reduce) slots to be borrowed is determined based on the number of unused reduce (or map) slots and its map (or reduce) slots required. To minimize the possible starvation of slots for each phase, instead of borrowing all unused map (or reduce) slots, we add configuration variables *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots* for the percentage of unused map and reduce slots that can be borrowed. The updated map (or reduce) load factor can be computed with the inclusion of borrowed map (or reduce) slots. Finally, we can compute the number of available map and reduce slots that should be allocated for map and reduce tasks at this heartbeat for that tasktracker, based on the current map and reduce slots capacity as well as used map and reduce slots.

Algorithm 1 The dynamic task assignment policy for tasktracker under PI-DHFS.

```
When a heartbeat is received from a compute node $n$:
 1: compute its clusterUsedMapSlots, clusterUsedReduceSlots, mapSlotsDemand, reduceSlotsDemand, mapSlotsLoadFactor and reduceSlotsLoadFactor.
 2: /*Case 1: both map slots and reduce slots are sufficient.*/
 3: if ($mapSlotsLoadFactor \leqslant 1$ and $reduceSlotsLoadFactor \leqslant 1$) then
 4:     //No borrow operation is needed.
 5: /*Case 2: both map slots and reduce slots are not enough.*/
 6: if ($mapSlotsLoadFactor \geqslant 1$ and $reduceSlotsLoadFactor \geqslant 1$) then
 7:     //No borrow operation is needed.
 8: /*Case 3: map slots are enough, while reduce slots are insufficient. It calculates borrowed map slots for reduce tasks.*/
 9: if ($mapSlotsLoadFactor < 1$ and $reduceSlotsLoadFactor > 1$) then
10:     currentBorrowedMapSlots = clusterUsedMapSlots - clusterRunningMapTasks;
11:     extraReduceSlotsDemand = min{ max{ floor{ clusterMapCapacity * percentageOfBorrowedMapSlots} - currentBorrowedMapSlots, 0}, reduceSlotsDemand - clusterReduceCapacity}
12:     updatedMapSlotsLoadFactor = (mapSlotsDemand + extraReduceSlotsDemand) / clusterMapCapacity;
13: /*Case 4: map slots are insufficient, while reduce slots are enough. It calculates borrowed reduce slots for map tasks.*/
14: if ($mapSlotsLoadFactor > 1$ and $reduceSlotsLoadFactor < 1$) then
15:     currentBorrowedReduceSlots = clusterUsedReduceSlots - clusterRunningReduceTasks;
16:     extraMapSlotsDemand = min{ max{ floor{ clusterReduceCapacity * percentageOfBorrowedReduceSlots} - currentBorrowedReduceSlots, 0}, mapSlotsDemand - clusterMapCapacity}
17:     updatedReduceSlotsLoadFactor = (reduceSlotsDemand + extraMapSlotsDemand) / clusterReduceCapacity;
18: compute availableMapSlots and availableReduceSlots based on the updated map/reduce load factor and used slots.
```

B. Pool-dependent DHFS (PD-DHFS)

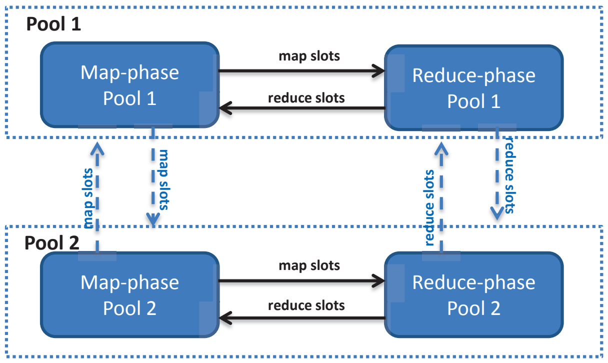


Fig. 2: Example of the fairness-based slots allocation flow for PD-DHFS. The black arrow line and dash line show the borrow flow for slots across pools.

In contrast to PI-DHFS that considers the fairness in its dynamic slots allocation independent of pools, but rather across typed-phases, there is another alternative fairness consideration for the dynamic slots allocation across pools, as we call Pool-dependent DHFS (PD-DHFS), as shown in Figure 2. It assumes that each pool, consisting of two parts: map-phase pool and reduce-phase pool, is selfish. That is, it always tries to satisfy its own shared map and reduce slots for its own needs at the map-phase and reduce-phase as much as possible before lending them to other pools. PD-DHFS will be done with the following two processes:

(1). *Intra-Pool dynamic slots allocation.* First, each typed-phase pool will receive its share of typed-slots based on max-min fairness at each phase. There are four possible relationships for each pool regarding its demand (denoted as *mapSlotsDemand*, *reduceSlotsDemand*) and its share (marked as *mapShare*, *reduceShare*) between two phases:

Case (a). $mapSlotsDemand < mapShare$, and $reduceSlotsDemand > reduceShare$. We can borrow some unused map slots for its overloaded reduce tasks from its reduce-phase pool first before yielding to other pools.

Case (b). $mapSlotsDemand > mapShare$, and $reduceSlotsDemand < reduceShare$. In contrast, we can satisfy some unused reduce slots for its map tasks from its map-phase pool first before giving to other pools.

Case (c). $mapSlotsDemand \leqslant mapShare$, and $reduceSlotsDemand \leqslant reduceShare$. Both map slots and reduce slots are enough for its own use. It can lend some unused map slots and reduce slots to other pools.

Case (d). $mapSlotsDemand > mapShare$, and $reduceSlotsDemand > reduceShare$. Both map slots and reduce slots for a pool are insufficient. It might need to borrow some unused map or reduce slots from other pools through *inter-Pool dynamic* *slots allocation* below.

(2). *Inter-Pool dynamic slots allocation.* It is obvious that, (i). for a pool, when its *mapSlotsDemand* + *reduceSlotsDemand* $\leqslant$ *mapShare* + *reduceShare*. The slots are enough for the pool and there is no need to borrow some map or reduce slots from other pools. It is possible for the cases: (a), (b), (c) mentioned above. (ii). On the contrary, when *mapSlotsDemand* + *reduceSlotsDemand* $>$ *mapShare* + *reduceShare*, the slots are not enough even after *Intra-Pool dynamic slots allocation*. It will need to borrow some unused map and reduce slots from other pools, i.e., *Inter-Pool dynamic slots allocation*, to maximize its own need if possible. It can occurs for pools in the following cases: (a), (b), (d) above.

The pseudocode for PD-DHFS implementation is shown in Algorithm 2. When tasktracker receives a heartbeat, instead of allocating map and reduce slots separately, it treats them as a whole to allocate for pools. That is, it first computes the total slots demand *totalSlotsDemand* for all map and reduce tasks from all pools. The running number of tasks *trackerRunningTasksNum* and current slots capacity *trackerCurrentSlotsCapacity* for a tasktracker can be determined by considering the load balance for a cluster as well as its own maximum slots capacity *trackerSlotsCapacity*. Then we can compute the maximum number of free slots that can be allocated at each round of heartbeat for a tasktracker by subtracting *trackerRunningTasksNum* from *trackerCurrentSlotsCapacity*. For each slot allocation, we first scan sorted pools. For each pool, we try the following possible slot allocations:

Case (1): We first try the map tasks allocation if there are idle map slots for the tasktracker (i.e., *trackerRunningMapTasksNum* $<$ *trackerMapCapacity*), and there are pending map tasks for the pool (i.e., *p.getMapDemand* $>$ *p.getRunningMapTasks*).

Case (2): If the attempt of Case (1) fails since the condition does not hold or it cannot find a map task satisfying the valid data-locality level, we continue to try reduce tasks allocation when the following conditions hold: *trackerRunningReduceTasksNum* $<$ *trackerReduceCapacity* and *p.getReduceDemand* $>$ *p.getRunningReduceTasks*.

Case (3): If Case (2) still fails due to the required condition does not hold, we try for map task allocation again. Case (1) fails might be that there are no idle map slots available (i.e., *trackerRunningMapTasksNum* $\geqslant$ *trackerMapCapacity*). In contrast, Case (2) fails might be due to no pending reduce tasks (i.e., *p.getReduceDemand* $\leqslant$ *p.getRunningReduceTasks*). In this case, we can try reduce slots for map tasks of the pool.

Case (4): If Case (3) still fails, we try for reduce task allocation again. Both Case (1) and Case (3) fail might be that there are no valid locality-level pending map tasks available, whereas there are idle map slots. In contrast, Case (2) might be that there are no idle reduce slots available (i.e., *trackerRunningReduceTasksNum* $\geqslant$ *trackerReduceCapacity*). In that case, we can allocate map slots for reduce tasks of the pool.

The slot allocation flow for the above cases is shown in Figure 3. The number labeled in the graph denotes the corresponding case. Moreover, there is a special case that needs to be particularly considered:

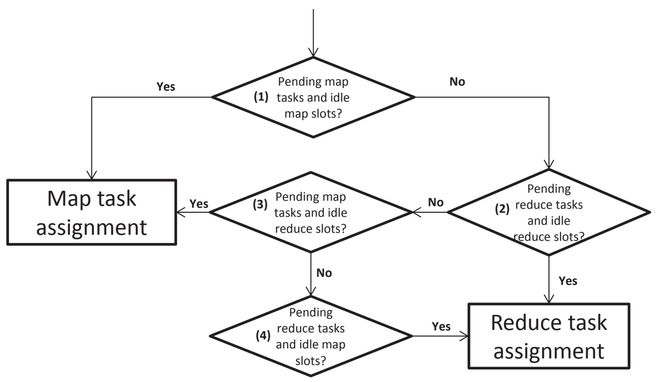
Case (5): Note it is possible that all the above four possible slot allocation attempts fail for all pools, due to the data locality consideration for map tasks. For example, it is possible that there is a new compute node added to the Hadoop cluster. It may be empty and does not contain any data. Thus, the data locality for all map tasks might not be satisfied and all pending map tasks cannot be issued. The failures of both Case (2) and Case (4) indicate that there are no pending reduce tasks available for all pools. However, there might be some pending map tasks available. Therefore, there is a need to run some map tasks by ignoring the data locality consideration on that new compute node to maximize the system utilization. To implement this, we make a mark *visitedForMap* for each job visited for map tasks. The data locality will be considered when *visitedForMap* does not contain scanned job. Otherwise, it will relax the data locality constrain for map tasks.

Algorithm 2 The dynamic task assignment policy for tasktracker under PD-DHFS.

```
When a heartbeat is received from tasktracker $tts$:
1: Compute its totalSlotsDemand, totalSlotsCapacity, trackerSlotsCapacity,
   trackerRunningTasksNum and trackerCurrentSlotsCapacity.
2: /* Return when there are no idle slots. */
3: if trackerRunningTasksNum $\geqslant$ trackerCurrentSlotsCapacity then
4:     return NULL;
5: for (i = 0; i $<$ trackerCurrentSlotsCapacity - trackerRunningTasksNum;
   i++) do
6:     Sort pools by distance below min and fair share
7:     for (Pool p : pools) do
8:         /* Case (1): allocate map slots for map tasks from Pool p*/
9:         if (there are pending map tasks and idle map slots) then
10:            Task task = assignTask(tts, currentTime, visitedForMap);
11:            if (task != null) then
12:                foundTask = true; tasks.add(task); break;
13:        /* Case (2): allocate reduce slots for reduce tasks from Pool p*/
14:        if (there are pending reduce tasks and idle reduce slots) then
15:            Task task = assignTask(tts, currentTime, visitedForReduce);
16:            if (task != null) then
17:                foundTask = true; tasks.add(task); break;
18:        /* Case (3): allocate reduce slots for map tasks from Pool p*/
19:        if (there are pending map tasks) then
20:            Task task = assignTask(tts, currentTime, visitedForMap);
21:            if (task != null) then
22:                foundTask = true; tasks.add(task); break;
23:        /* Case (4): allocate map slots for reduce tasks from Pool p*/
24:        if (there are pending reduce tasks) then
25:            Task task = assignTask(tts, currentTime, visitedForReduce);
26:            if (task != null) then
27:                foundTask = true; tasks.add(task); break;
28:    /* Case (5): schedule the non-local map tasks when its node-local
       tasks cannot be satisfied. */
29:    if (!foundTask) then
30:        for (Pool p : pools) do
31:            if (there are pending map tasks) then
32:                Task task = assignTask(tts, currentTime, visitedForMap);
33:                if (task != null) then
34:                    foundTask = true; tasks.add(task); break;
```

IV. Experimental Evaluation

In this section, we evaluate the performance benefit of our proposed dynamic slot allocation techniques.



Yes
(1) Pending map tasks and idle map slots?
No
Map task assignment
Yes
(3) Pending map tasks and idle reduce slots?
No
(2) Pending reduce tasks and idle reduce slots?
No
Yes
(4) Pending reduce tasks and idle map slots?
Yes
Reduce task assignment

Fig. 3: The slot allocation flow for each pool under PD-DHFS.

J_i	**Benchmark**	**DataSize** (GB)	**BlockSize** (MB)	$\lvert J_i^{\mathcal{M}}\rvert$	$\lvert J_i^{\mathcal{R}}\rvert$
J_1	wordcount	10	64	160	30
J_2	sort	20	64	320	200
J_3	grep	30	64	480	120
J_4	wordcount	40	64	640	100
J_5	sort	30	64	480	200
J_6	wordcount	10	128	80	15
J_7	sort	20	128	160	100
J_8	grep	30	128	240	80
J_9	wordcount	40	128	320	50
J_{10}	grep	10	128	80	40
J_{11}	wordcount	30	64	480	60
J_{12}	sort	30	64	480	200
J_{13}	grep	20	64	320	80
J_{14}	wordcount	20	128	160	30
J_{15}	sort	10	128	80	60
J_{16}	wordcount	10	256	40	20
J_{17}	sort	20	256	80	40
J_{18}	grep	30	256	120	60
J_{19}	wordcount	40	256	160	40
J_{20}	sort	10	256	40	10

TABLE I: The job information for testbed workloads.

A. Experiments Setup

We ran our experiments in a cluster consisting of 10 compute nodes, each with two Intel X5675 CPUs (6 CPU cores per CPU with 3.07 GHz), 24GB memory and 56GB hard disks. We configure one node as master and namenode, and the other 9 nodes as slaves and datanodes. Moreover, we configure 10 map and 2 reduce slots per slave node. We generate our testbed workloads by using three representative applications, i.e., **wordcount** application (computes the occurrence frequency of each word in a document), **sort** application (sorts the data in the input files in a dictionary order) and **grep** application (finds the matches of a regex in the input files). We take wikipedia article history dataset[1] with four different sizes, e.g., 10GB, 20GB, 30GB, 40GB as application input data. As there is one map task per data block in Hadoop, we upload each data into HDFS with different block sizes of 64MB, 128MB, 256MB to have different number of data blocks and varied block sizes. Table I lists the job information for our testbed workloads. It is a mix of three benchmarks together with different sizes of input data and varied block sizes.

B. Performance Improvement Evaluation

Figure 4 presents the evaluation results for our proposed DHFS for a single MapReduce job (e.g., J_1, J_2, J_3) as well as MapReduce workloads with multiple jobs, e.g., 5 jobs ($J_1 \sim J_5$), 10 jobs ($J_1 \sim J_{10}$) and 20 jobs ($J_1 \sim J_{20}$). All speedups are calculated with respect to the original Hadoop. We can see that both PI-DHFS and PD-DHFS can improve the performance of MapReduce jobs significantly, i.e., there are about $32\% \sim 55\%$ for a single job and $44\% \sim 68\%$ for MapReduce workloads with multiple jobs. For the traditional Hadoop, the map/reduce slot configuration has a big influence in the cluster utilization and performance for MapReduce jobs, whereas our DHFS is not influenced by map/reduce slot configuration.

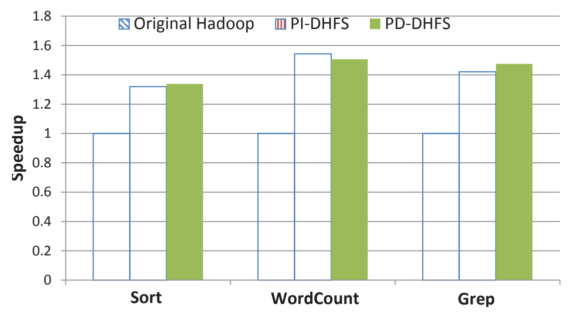
Take a single job for example. Let N_M and N_R denote the number of map tasks and reduce tasks. Let t_M and t_R denote the execution time for a single map task and reduce task. Let S_M and S_R denote the number of map slots and reduce slots. Moreover, we assume that there is one slot per CPU core and thus the sum of map slots and reduce slots is fixed for a given cluster. Then for the traditional Hadoop cluster, the execution time will be $\lceil \frac{N_M}{S_M} \rceil \cdot t_M + \lceil \frac{N_R}{S_R} \rceil \cdot t_R$. In contrast, it will be $\lceil \frac{N_M}{S_M+S_R} \rceil \cdot t_M + \lceil \frac{N_R}{S_M+S_R} \rceil \cdot t_R$ for our DHFS. Based on the formula, we can see varied performance from the traditional Hadoop under different slot configurations. However, there is little impact on the performance for different slot configurations under DHFS.

[1]http://dumps.wikimedia.org/enwiki/

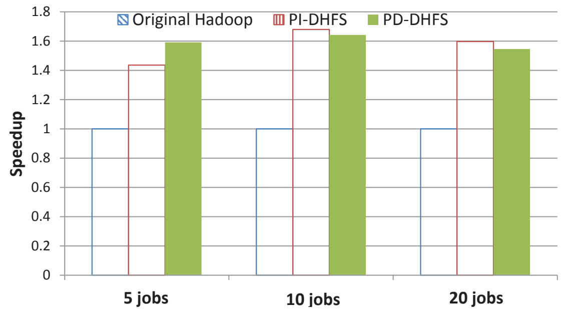
C. Dynamic Tasks Execution Processes for PI-DHFS and PD-DHFS

To show different levels of fairness for the dynamic tasks allocation algorithms, PI-DHFS and PD-DHFS, we perform an experiment by considering two pools, each with one job submitted. Figure 5 shows the execution flow for the two DHFSs, with 10 sec per time step. The number of running map and reduce tasks for each pool at each time step is recorded. For PI-DHFS, as illustrated in Figure 5(a), we can see that, at the beginning, there are only map tasks, with all slots used by map tasks under PI-DHFS. Each pool shares half of the total slots (i.e., 54 slots out of 108 slots), until the 3^{th} time step. The map slots demand for pool 1 begins to shrink and the unused map slots of its share are yielded to pool 2 from the 4^{th} time step to the 7^{th} time step. Next from 9^{th} to 15^{th} time step, the map tasks from pool 2 takes all map slots and the reduce tasks from pool 1 possess all reduce slots, based on the typed-phase level fairness policy of PI-DHFS(i.e., intra-phase dynamic slots allocation). Later there are some unused map slots from pool 2 and they are used by reduce tasks from pool 1 from 16^{th} to 18^{th} time step(i.e., inter-phase dynamic slots allocation).

For PD-DHFS, similar to PI-DHFS at the beginning, each pool obtains half of the total slots from the 1^{th} to 3^{rd} time

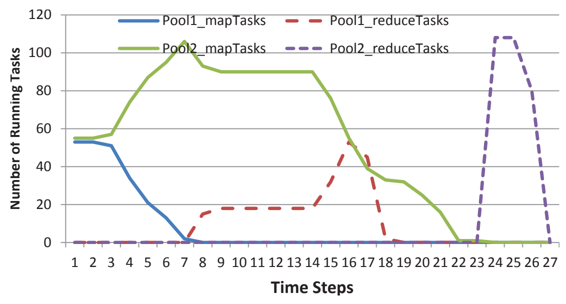


(a) A single MapReduce job

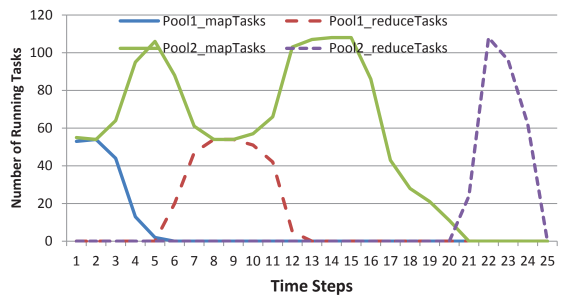


(b) MapReduce workloads with multiple jobs

Fig. 4: The performance improvement with our dynamic scheduler for MapReduce workloads.



(a) PI-DHFS



(b) PD-DHFS

Fig. 5: The execution flow for the two DHFSs. There are two pools, with one running job each.

step, as shown in Figure 5(b). Some unused map slots from pool 1 are yielded to pool 2 from 4^{th} to the 7^{th} time step. However, from the 8^{th} to 11^{th}, each of the map tasks from pool 2 and the reduce tasks from pool 1 takes half of the total slots, subject to the pool-level fairness policy of PD-DHFS (i.e., intra-pool dynamic slots allocation). Finally, the unused slots from pool 1 begins to yield to pool 2 since 12^{th} time step (i.e., inter-pool dynamic slots allocation).

D. Discussion on the Performance of Different Percentages of Borrowed Map and Reduce Slots

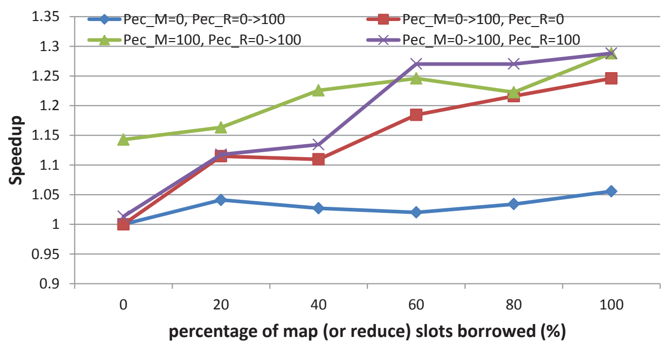


Fig. 6: The performance results with different percentages of map (or reduce) slots borrowed.

In Section III-A, instead of borrowing all unused map (or reduce) slots for overloaded reduce (or map) tasks, we provide users with two configuration arguments *percentageOf-BorrowedMapSlots* and *percentageOfBorrowedReduceSlots* to limit the amount of borrowed map/reduce slots, and ensure that tasks at the map/reduce phase are not starved. It is meaningful and important when users want to reserve some unused slots for incoming tasks, instead of lending all of them to other phases or pools. To show its impact on the performance, we perform an experiment with *sort* benchmark (320 map tasks and 200 reduce tasks) by varying values of arguments.

Let Pec_M and Pec_R denote *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots* respectively. Figure 6 presents the performance results under varied argument configurations. All speedup results are calculated with respect to the case when $Pec\_M = 0$ and $Pec\_R = 0$. We consider four cases: (1). Vary the value of Pec_R from 0 to 100 while fix $Pec\_M = 0$; (2). Vary the value of Pec_R while set $Pec\_M = 100$; (3). Vary the value of Pec_M while fix $Pec\_R = 0$; (4). Vary the value of Pec_M while fix $Pec\_R = 100$. We can see that, the performance improves by increasing either Pec_M or Pec_R. Particularly, there is a significant performance improvement(i.e., approximate 29%) when we fix the value of Pec_R and increase the value of Pec_M. It is because there are plenty of reduce tasks (e.g., 200 reduce tasks) but only 18 reduce slots. Thus increasing the percentage value of map slots (Pec_M) that can be borrowed would let more reduce tasks be scheduled using borrowed map slots, reducing the number of computation waves of reduce tasks and improving the utilization as well as performance of the Hadoop cluster.

V. Discussion

The goal of our work is to improve the utilization and performance for MapReduce clusters while guaranteeing the fairness across pools. PI-DHFS and PD-DHFS are our first two attempts of achieving this goal with two different fairness definitions. PI-DHFS follows strictly the definition of fairness given by traditional HFS, i.e., the slots are fairly shared across pools within each phase (i.e., map phase or reduce phase). However, the slot allocations are independent across phases. In contrast, PD-DHFS gives a new definition of fairness from the perspective of pools, i.e., each pool shares the total number of map and reduce slots from the map phase and reduce phase fairly with other pools.

Because of different definitions of fairness, the possibility of slots movement between map-phase and reduce-phase in dynamic slots allocation is different between PI-DHFS and PD-DHFS. For PI-DHFS, the slots movement can only occur when one typed slots (e.g., map slots) are enough while the other typed slots (e.g., reduce slots) are insufficient, i.e., two cases: Case 2 and Case 3 in Section III-A. However, for PD-DHFS, the slots movement can occur in all four possible scenarios (e.g., Case 1, Case 2, Case 3, Case 4) mentioned in Section III-A, i.e., there are slots movements even in the case that both total map slots and reduce slots are sufficient (or insufficient) for PD-DHFS. Thus, PD-DHFS tends to have more slot movement than PI-DHFS.

VI. Conclusion and Future Work

This paper proposes Dynamic Hadoop Fair Schedulers (DHFS) to improve the utilization and performance of MapReduce clusters while guaranteeing the fairness. The core technique is dynamically allocating map (or reduce) slots to map and reduce tasks. Two types of DHFS are presented, namely, PI-DHFS and PD-DHFS, based on fairness for cluster and pools, respectively. The experimental results show that our proposed DHFS can improve the performance and utilization of the Hadoop cluster significantly. As for future work, we are interested in extending our dynamic slot allocation algorithms to heterogeneous environments. Cluster/cloud has become heterogeneous with different architectures. We plan to extend our previous study [22] to handle the slot configuration on CPUs and GPUs.

The DHFS source code is publicly available for downloading at *http://sourceforge.net/projects/dhfs/*.

VII. Acknowledgment

This work was supported by the "User and Domain driven data analytics as a Service framework" project under the A*STAR Thematic Strategic Research Programme (SERC Grant No. 1021580034).

Bingsheng He was partly supported by a startup Grant of Nanyang Technological University, Singapore.

References

[1] Hadoop. http://hadoop.apache.org.
[2] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy,S. Schenker,I. Stoica, *Job Scheduling for Multi-user Mapreduce Clusters*. Technical Report EECS-2009-55, UC Berkeley Technical Report (2009).
[3] Max-Min Fairness (Wikipedia). http://en.wikipedia.org/wiki/Max-min_fairness.
[4] D.W. Jiang, B.C. Ooi, L. Shi, and S. Wu.*The Performance of MapReduce: An Indepth Study*, PVLDB, 3:472-483, 2010.
[5] P. Agrawal, D. Kifer, and C. Olston. *Scheduling Shared Scans of Large Data Files*. In VLDB, 2008.
[6] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. *MRShare: Sharing Across Multiple Queries in MapReduce* . Proc. of the 36th VLDB (PVLDB), Singapore, September 2010.
[7] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein. *MapReduce online*. In Proceedings of the 7th USENIX conference on Networked systems design and implementation, pp. 21C21, 2010.
[8] B. Moseley, A. Dasgupta, R. Kumar, T. Sarl, *On scheduling in map-reduce and flow-shops*. SPAA, pp. 289-298, 2011.
[9] A. Verma, L. Cherkasova, R.H. Campbell, *Orchestrating an Ensemble of MapReduce Jobs for Minimizing Their Makespan*, IEEE Transaction on dependency and secure computing, 2013.
[10] A. Verma, L. Cherkasova, R. Campbell. *Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance*. MASCOTS 2012.
[11] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. *Starfish: A Self-tuning System for Big Data Analytics*. In CIDR, pages 261C272, 2011.
[12] H. Herodotou and S. Babu, *Profiling, What-if Analysis, and Costbased Optimization of MapReduce Programs*. in Proc. of the VLDB Endowment, Vol. 4, No. 11, 2011.
[13] C. Oğuz, M.F. Ercan, *Scheduling multiprocessor tasks in a two-stage flow-shop environment*. Proceedings of the 21st international conference on Computers and industrial engineering, pp. 269-272, 1997.
[14] J. Polo, C. Castillo, D. Carrera, et al. *Resource-aware Adaptive Scheduling for MapReduce Clusters*. Proceeding Middleware'11 Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware, pp. 187-207, 2011.
[15] Z.H. Guo, G. Fox, M. Zhou, Y. Ruan.*Improving Resource Utilization in MapReduce*. 2012 IEEE International Conference on Cluster Computing (CLUSTER). pp. 402-410, 2012.
[16] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, In Proceedings of the 6th Symposiumon Operating SystemsDesign and Implementation (OSDI), 2004.
[17] HowManyMapsAndReduces. http://wiki.apache.org/hadoop/HowMany MapsAndReduces.
[18] *Capacity Scheduler Guide*. http://hadoop.apache.org/common/docs/r0.20.1/capacity scheduler.html, 2010.
[19] S.J. Tang, B.S. Lee, and B.S. He. *MROrder: Flexible Job Ordering Optimization for Online MapReduce Workloads*. in Euro-Par, pp. 291-304, 2013.
[20] S.J. Tang, B.S. Lee, R. Fan and B.S. He. *Performance Optimization for MapReduce Workloads*, CORR (Technical Report), 2013.
[21] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He and Li Qi: *LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud*, In Proc. of CloudCom 2010, pp. 17-24.
[22] Yu S. Tan, Bu-Sung Lee, Bingsheng He and Roy H. Campbell. *A Map-Reduce Based Framework for Heterogeneous Processing Element Cluster Environments*. In Proc. of CCGRID 2012, pp. 57-64.