2017 IEEE International Parallel and Distributed Processing Symposium

MRapid: An Efficient Short Job Optimizer on Hadoop

Hong Zhang
Department of Computer Science
University of Central Florida
Orlando, FL, USA
hzhang1982@knights.ucf.edu

Hai Huang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
haih@us.ibm.com

Liqiang Wang
Department of Computer Science
University of Central Florida
Orlando, FL, USA
lwang@cs.ucf.edu

***Abstract*—Data have been generated and collected at an accelerating pace. Hadoop has made analyzing large scale data much simpler to developers/analysts using commodity hardware. Interestingly, it has been shown that most Hadoop jobs have small input size and do not run for long time. For example, higher level query languages, such as Hive and Pig, would handle a complex query by breaking it into smaller ad-hoc ones. Although Hadoop is designed for handling complex queries with large data sets, we found that it is highly inefficient to operate at small scale data, despite a new Uber mode was introduced specifically to handle jobs with small input size.**

In this paper, we propose an optimized Hadoop extension called MRapid, which significantly speeds up the execution of short jobs. It is completely backward compatible to Hadoop, and imposes negligible overhead. Our experiments on Microsoft Azure public cloud show that MRapid can improve performance by up to 88% compared to the original Hadoop.

***Keywords*-Hadoop; MapReduce; Short Job; Uber Mode; Distributed Mode**

I. Introduction

MapReduce [1] is a parallel programming model that uses a reliable distributed architecture to process data. Hadoop [2][3] is an open source implementation of that. It consists of three components: a distributed file system (HDFS), a resource management system (Yarn), and a parallel processing framework (MapReduce).

Although Hadoop is designed to process very large data sets, a majority of jobs are short in the real world. For example, the MapReduce jobs at Google in 2004 took 634 seconds on the average, and over 80% of Yahoo's jobs finished within 10 minutes [1][4][5]. This is mainly due to the input data size being small, especially when it is spread across the entire HDFS cluster and processed in parallel. Moreover, SQL-like query systems, such as Pig and Hive, that operate on top of MapReduce could break a longer running job into a collection of shorter jobs [6][7]. More recently, Uber mode was introduced in Hadoop 2 to specifically deal with jobs with small input size (less than 1 data chunk, to be precise). This special mode runs all tasks of a job within one container. However, even Uber mode is not efficient enough to handle small jobs. We summarize the inefficiencies of running short jobs on Hadoop as follows:

- Hadoop scheduler does not take data locality into account for short job, thus unnecessary data transfer could significantly slow down the execution of short jobs.
- One-time task setup and tear down overheads, which are often negligible in a long running job, can no longer be overlooked for short jobs.
- Piggybacking requests and responses to periodic heartbeat messages is designed for cluster scalability, but waiting a few seconds here and there adds up quickly. Short-circuiting these paths can be beneficial for short jobs.
- In Uber mode, running all tasks sequentially within a single container does not take full advantage of all local resources.
- Moreover, in Uber mode, intermediate data incur disk I/Os, such as spill operation, could significantly degrade performance.

Short jobs have been studied in the past [1][4][5]. Although there is no exact definition, a short job roughly means that its completion time ranges from seconds to minutes, rather than hours. Hadoop's Uber mode gives a more quantitative definition: a small job has less than 10 mappers, only 1 reducer, and the input size is less than the size of one HDFS block. However, this definition still cannot help users decide whether to run MapReduce jobs in Uber mode or not. We consider that the definition of short job is relevant to resource available for users, and the threshold between short job and large job varies depending upon the available resource in the cluster. For instance, if the cluster contains 10 DataNodes, each of which can launch 2 containers, we can run 20 Map tasks in parallel. But if the cluster consists of 100 DataNodes with the same configuration, 200 Map tasks can be executed in one wave.

In this study, we propose an efficient short job optimization on Hadoop, called *MRapid*, to speed up the execution of MapReduce short jobs. In our system, we design two improved modes based on Hadoop: Improved Distributed (D+) mode and Improved Uber (U+) mode. Our contributions are summarized as follows:

- In D+ mode, we design a new scheduler to schedule

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.100

Map tasks according to the resource distribution situation and data locality. When an ApplicationMaster requests resources from Yarn, instead of waiting for report from NodeManager, our new scheduler allocates resources according to the current resource availability and data distribution in ResourceManager, and responds the request in the same heartbeat, rather than waiting for at least two heartbeats in Hadoop. Our algorithm not only avoids load imbalance problem for short jobs, but also reduces the communication cost. The benefits of spreading out Map tasks and data-locality awareness are significant, especially when a short job can be executed in one wave.

- In U+ mode, rather than executing Map tasks sequentially, we run multiple Map tasks in parallel on the same node. The degree of parallelism depends on the available resources on the node.
- In U+ mode, we cache intermediate data into memory instead of writing them to disk and reading them back later as intermediate data are usually small for short jobs.
- We design a job submission framework, which reserves an ApplicationMaster pool for reuse and avoids the long waiting time to initialize new ones for short jobs.
- For a short job, deciding which mode (D+ or U+) runs faster is a grand research challenge. Our job submission framework handles it by supporting speculative execution. Specifically, the framework can execute an application initially in both D+ and U+ modes. During the execution, a profiler records the execution and data I/O information for each mode. When the framework is confident that one mode is behind the other, the slower one will be terminated. The winner mode can be designated to the short jobs for the future run.

Given a job, users may submit it as a short job as MRapid can always bid the performance of the original Hadoop, except for the overhead of running both D+ and U+ modes at the short initial stage of the execution. However, if the execution history has recorded the same job before, our system can easily decide the faster mode to execute.

The rest of this paper is organized as follows. Section II gives the background information about the job submission process in Hadoop. We then describe the details of our design in Section III including D+ mode, U+ mode, and a speculative job submission framework. Section IV shows experimental results. Section V provides a review of related work. Conclusions and future work are summarized in Section VI.

II. Background

Yarn, a cluster resource manager, is a key component of Hadoop 2, and MapReduce is one of computing frameworks that runs on Yarn. In this section, we give a comprehensive overview of the job submission process, and discuss the root causes of its inefficiency. As shown in Figure 1, there are 6 steps to submit a job.

1) *Job Submission:* To submit a new job, a client first communicates with the ResourceManager (RM) to generate a new job ID. which in turn checks the specification of the job, uploads input splits, job Jar file, and configuration to HDFS. The client then submits the job to the RM.
2) *ApplicationMaster (AM) Allocation:* When the RM receives the job submission request, the scheduler allocates a container to set up and launches an AM instance.
3) *Launching AM:* The designated NodeManager (NM) of the allocated container launches AM for the job. Once AM is started, it downloads input splits, job Jar file, and configuration from HDFS, and initializes itself.
4) *Request Containers:* If the job is not configured to run in Uber mode, the AM requests containers for Map and Reduce tasks from the RM, which schedules resources based on data locality that allocates each task to be near its input data. Hadoop employs CapacityScheduler by default, which allows multiple tenants to share a large cluster and allocate resources under constraints of specified capacities for each user.
5) *Task Assignment:* After tasks have been assigned to run in certain containers by the RM's scheduler, the AM starts the containers by contacting the corresponding NMs.
6) *Task Execution:* After downloading configuration and Jar file from HDFS, the Map or Reduce task is executed as a Java application in JVM.

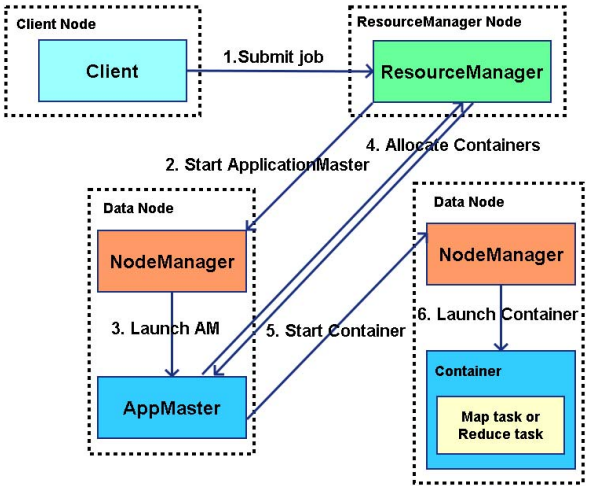

Figure 1. Hadoop job submission.

Submitting job in Hadoop system is inefficient and the time consumption of creating containers for small amount of input data is relatively expensive. To resolve this, Uber

mode runs all tasks of a job in the same JVM as the AM in order to avoid container allocation, start up overhead, and the transferring of intermediate data from Map phase to Reduce phase. However, the original Uber mode forces tasks to be sequentially executed within a single container, which is another weakness.

III. Design and Implementation

For a short job running in Hadoop, it is difficult to decide which way is more efficient: distributing all Map tasks to the cluster uniformly or executing them in a single container. Spreading Map tasks to the whole cluster maximizes resource utilization; however, requesting and launching containers will consume a large amount of unnecessary time, and shuffling intermediate data from the Map phase to the Reduce phase is also expensive. An alternative way is to execute all Map and Reduce tasks in a single container in Uber mode, but the current Uber mode executes all tasks sequentially, which is extraordinarily inefficient. Therefore, we design two computing modes: one is a new resource and data-locality aware strategy that distributes and executes Map tasks in parallel in the cluster, which is called the Improved Distributed mode (D+ mode); another is the Improved Uber Mode (U+ mode) that executes Map tasks in a single container in parallel using multiple threads, and stores intermediate data in memory rather than to disk to speed up job execution.

No matter what mode to choose, it is inevitable to launch an AM for each Hadoop job. From experiments, we notice that the time on initializing a short job and launching its AM is expensive compared to the overall execution time of the short job. Therefore, to reuse AM, we introduce a new framework to reserve AM objects in a pool rather than allocating a new one for each short job.

A. Distributed Mode

In the D+ mode, our resource and locality aware scheduler allocates Map tasks to different nodes as distributed as possible in order to avoid resource contention like CPU, memory, and disk I/O. Due to data-locality awareness, it also increases the number of data-local Map tasks and reduces data transferring between DataNodes.

Figure 2 shows steps how to request containers from RM in the original Hadoop. The AM requests containers and obtains resources from the RM. This request is wrapped into a heartbeat and invoked periodically. The heartbeat contains description of new request, list of released containers, and update information of blacklist nodes.

When the RM receives such a kind of heartbeat, it sends a CONTAINER_STATUS_UPDATE event to the ResourceScheduler (RS). The RS puts the container request to the corresponding queue.

As one of NMs reports its status to the RM by heartbeat, the RM sends a NODE_STATUS_UPDATE event to RS,

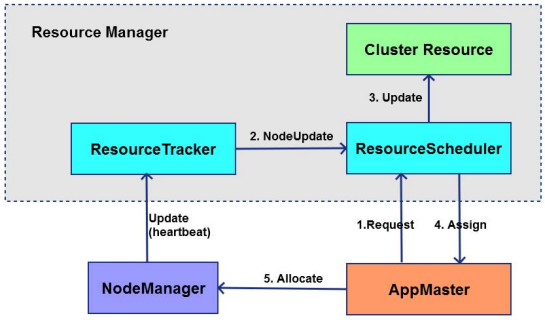

Figure 2. Resource request in Hadoop.

then the RS allocates available resources of this node to the container request in front of the request queue. Note that one request may ask for multiple containers. The corresponding AM will obtain these containers at the next heartbeat. Then the AM tells the selected NM to start Map or Reduce Tasks. The NM will register launched containers to the RM later.

From the description above, we know that the RM does not respond to the container request immediately, it has to wait until one NM with available resources reports its status, then allocates these resources to the container request. However, such a resource scheduling scheme has several major defects, especially for short jobs. First, waiting for NM status report is a waste of time, which causes the AM cannot obtain resources at the current heartbeat, even there are massive idle DataNodes. The time consumption of communication between the AM and the RM is expensive for short jobs and could not be ignored. Secondly, this scheme can lead to container allocation imbalance, so that some DataNodes may be squeezed with many containers, but others could be idle. Last but not least, this algorithm is not aware of data locality for short jobs. Although this method in the original Hadoop is not bad if the input data are large and spread uniformly in the cluster, lack of data locality is a fatal problem for short job since transferring input data is inevitable especially when the size of cluster is not small.

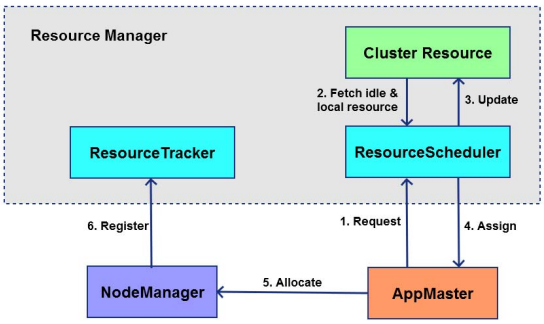

Figure 3. Resource request in D+ Mode of MRapid.

Figure 3 illustrates our improved distributed mode to allocate resources to small jobs more evenly and efficiently. The AM sends a request of resources to the RM, which in turn generates a CONTAINER_ STATUS_UPDATE event and sends it to the RS. In Step 2, when the RS realizes that this is a request for short job, instead of waiting for available DataNodes to report their status, the RS can allocate resources from Cluster Resource, which is a special structure designed to store the current resource information of each node and decide how to allocate resources using Algorithm 1. The resource status for each node is updated by each heartbeat, so it is sufficient to represent the latest resource status. Step 3 in Figure 3 shows that the RS updates the resource usage for every DataNode. After the RS finishes resource allocation, the RM sends resource allocation information immediately to the AM, as shown in Step 4. The rest steps are the same as the original Hadoop to launch containers on the selected NMs.

Algorithm 1 Scheduler algorithm for distributed mode

Input: $request$, $nodes$
Output: $response$

```
types = {NodeLocal, RackLocal, ANY}
for each type in types do
    Decide which resource is the current dominant resource;
    Sort nodes by available dominant resource in descending order;
    for each node in nodes do
        for each task in request do
            container = getResource(task, node, type);
            if (container is not null) then
                response.add(container);
                request.del(task);
            end if
            if (request is empty) then
                return response
            end if
        end for
    end for
end for
return response
```

Algorithm 1 shows our improved CapacityScheduler. The original Hadoop scheduler allocates containers from each available DataNode by a greedy algorithm, which deploys tasks to DataNodes as few as possible. Thus it does not consider data locality and container allocation balance in a global view. In our algorithm, we sort nodes by available dominant resource in descending order so that we assign Map tasks to relatively idle nodes. Dominant resource is a kind of resource such as CPU or memory that has the highest usage ratio in the cluster. Note that our dominant resource definition is based on the whole cluster, which is different from dominant resource [8] for each user.

HDFS's default replica is three. Its placement policy usually stores one replica on a node in the local rack, another replica on a node in a different rack, and the last on a different node in the same remote rack. Then there are three resource types corresponding to the preferred node of resource request. NodeLocal means that the preferred node is the same with the resource node. RackLocal is the type that the requested node and the resource node are in the same rack. ANY type is that we can designate any resource node to execute Map tasks. So in our approach, we schedule Map tasks to the NodeLocal resource first, then RackLocal, at last ANY in order to take data locality into account adequately until this resource request is satisfied. For each task, we assign resource by "getResource" if the task preferred type (NodeLocal, RackLocal, or ANY) matches the current node with available resources. After one type of resource request has been served, we calculate the dominant resource and sort nodes again to place the current relatively idle nodes in front before serving the next kind of request.

Our D+ mode spreads tasks of a short job across the cluster uniformly, which avoids work overload on specific nodes. In addition, our approach responds to AM requests in one heartbeat, whereas Hadoop usually needs two or more heartbeats. Another important advantage is that our design fully considers data locality before assignment rather than afterwards redistribution, which involves lots of data movement.

B. Improved Uber Mode

An Uber task is that the AM uses its own JVM to run the whole Map and Reduce tasks for a short job. Rather than executing each mapper and reducer task in a separated container, the AM container runs Map and Reduce tasks within its own process to avoid the overhead of requesting, launching, and communicating with remote containers.

Figure 4 describes the procedure to run a Hadoop job in the original Uber mode. Since there is only one container available, the AM has to execute Map and Reduce tasks sequentially. Another reason causing inefficiency of the original Uber mode is that intermediate data of Map tasks are spilled to local disks.

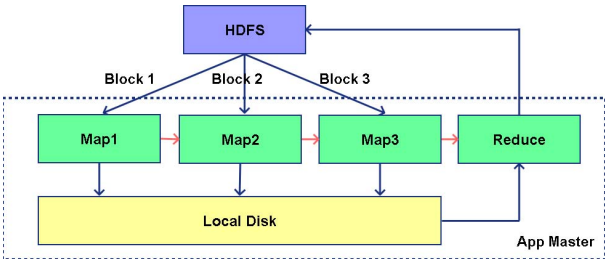

Figure 4. Hadoop's Uber mode.

To eliminate these inefficiency problems, we design an improved Uber Mode (U+ mode) that inherits the single container feature from the original Uber Mode, but is extended with the support of multithreading. Figure 5 shows the details of U+ mode. When an AM is launched, it parses the job configuration, fetches input data from HDFS, and then executes Map tasks concurrently using multithreading. The number of Maps per wave for the U+ mode (n_u^m) depends on cpu_vcores (n^c, the virtual CPU cores, which can be configured by users) of the AM. Let n_c^m denote the number of Map tasks running simultaneously on each cpu_vcore. Thus, $n_u^m = n^c * n_c^m$ indicates how many Maps per wave. For a small amount of intermediate data, we store them into memory instead of writing them to local disks. Thus, the Reduce task can fetch results of Map tasks from memory directly to decrease shuffle overhead.

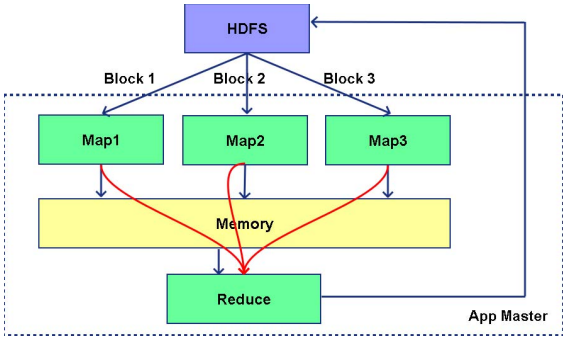

Figure 5. U+ Mode in MRapid.

C. Job Submitting Framework and Speculative Execution

As shown in Figure 1, after a client uploads job files (*e.g.*, jar file, configuration file) to HDFS, it submits the job to the RM. Then the RM launches an AM in a DataNode to manage job execution. The cost to request a container and launch AM is high for a short job, so we design a novel job submission framework using Spring Hadoop [9]. Our framework consists of three major modules. (1) The proxy is used to maintain an AM pool that contains a reasonable number of AMs reserved for short jobs and allocate an AM for each short job. (2) The client module is responsible for uploading the jar file and configuration files to HDFS and submitting short job to the proxy. (3) The AMSlave module is the module to accept and execute AM from the proxy instead of the RM of the original Hadoop. We implemented a RPC (remote procedure call) to allow the proxy to communicate with the AMs.

Due to the unpredictability of execution time for different kinds of short jobs, we employ a speculative execution mechanism to execute short jobs in both D+ mode and U+ mode. Figure 6 shows the workflow of speculative execution in our system.

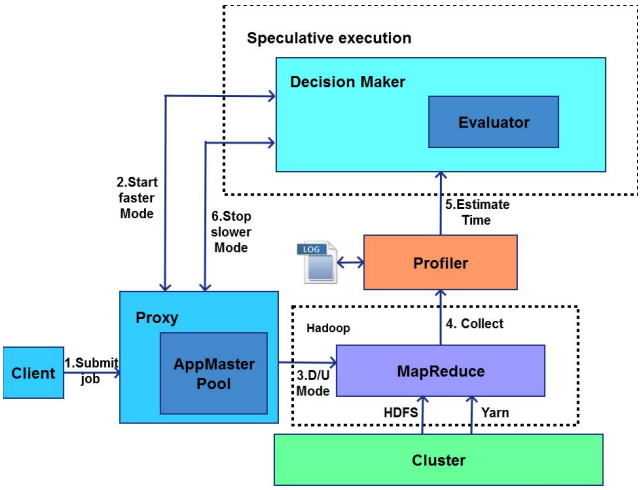

Figure 6. Speculative execution in MRapid.

1) **Job Submission:** When Hadoop starts, it launches a proxy service and creates an AM pool that reserves a certain number of AM containers for short jobs. The number of AMs is configured by Hadoop administrator, which is 3 by default. Users can use the client of our submitting framework to submit a short job to the proxy, request a job ID from HDFS, and upload the jar file and configuration files to HDFS.
2) **Pre-decision Making:** When the proxy receives job submission request, it consults the decision maker for which mode will be more efficient based on the execution records of the same job, even if they were executed with different input data.
3) **Launching AM:** If the decision-maker gives a clear answer on which mode is preferred, the proxy chooses one AM container from the pool to submit the job. Otherwise, it submits the job in both U+ and D+ modes.
4) **Profiling:** The designated AM receives the job information from the proxy, then downloads input splits, job Jar file, and configuration from HDFS, and executes the job. We designed a specific Hadoop profiler using ASM [10], which is a small and fast Java bytecode manipulation framework. Our profiler collects Hadoop application execution information including input/output data size and the average execution time for Map and Reduce tasks, and uploads them to HDFS.
5) **Evaluation:** We estimate the total execution time for a job in both U+ mode and D+ mode using the record collected by the profiler. The detailed estimation algorithm will be discussed later. The decision maker evaluates the performance of the two modes, and asks the proxy to terminate the inefficient one.
6) **Terminating Slower Mode:** After the proxy receives a notice from the decision maker, it kills the slower

Table I
NOTATIONS USED IN THE ESTIMATION ALGORITHM

t^{job}	the total execution time for job
t^{AM}	the AM setup time
t^{Map}	the Map phase execution time
$t^{Shuffle}$	execution time of shuffling phase
t^{Reduce}	the Reduce phase execution time
n^m	number of Map tasks
n^c	number of available containers
n^w	number of waves
n_u^m	number of Maps per wave for the U+ mode
t^l	execution time for launching container
t^m	execution time for map sub-phase
d^i	disk input rate
d^o	disk output rate
b^i	bandwidth
s^i	average input data size of Map tasks
s^o	average output data size of Map tasks
t_u	execution time for job in U+ mode
t_d	execution time for job in D+ mode

mode and releases allocated resources.

$$\begin{aligned} t^{job} &= t^{AM} + t^{Map} + t^{Shuffle} + t^{Reduce} \\ &= t^l + (t^l + s^i/d^o + t^m + s^o/d^i + s^o/d^o \\ &\quad + s^o/d^i) * n^w + (s^o * n^c)/b^i + t^{Reduce} \end{aligned} \tag{1}$$

Table I shows the notations used in our estimation algorithm. Equation 1 gives an evaluation of time consumption for a MapReduce job. The AM setup time can be expressed by the container launch time t^l. The execution time of Map tasks t^{Map} includes 5 sub-phases: setup, read, map, spill and merge. The setup sub-phase can be shown as the container launch time t^l. The read sub-phase can be calculated by the input data size s^i divided by the disk output rate d^o. The map sub-phase is symbolized by t^m, which can be evaluated by history records. The spill sub-phase writes the intermediate data into disk, *i.e.*, s^o/d^i. The merge sub-phase is to read the spilled data back for merging and write the merged data into disk again, *i.e.*, $s^o/d^o + s^o/d^i$, if the intermediate data is too large to spill once. The above analysis of Map phase is to calculate one wave, then we multiple it by the number of waves (n^w). The shuffle phase is intermediate data size divided by the bandwidth in one wave, other waves are not considered because there are overlaps between the Map phase and Shuffle phase.

$$t_u = t^m * (n^m/n_u^m) \tag{2}$$

$$t_d = (t^l + t^m + s^o/d^i) * (n^m/n^c) + (s^o * n^c)/b^i \tag{3}$$

Our algorithm to estimate the performance of the U+ and D+ modes are shown in Equations 2 and 3, respectively. Since we only consider one Reduce task, its execution time for both U+ and D+ modes are exactly the same, which can be omitted in Equations 2 and 3. The submission framework also removes the AM setup time (t^{AM}) from the Equation 1 for both modes. The setup sub-phase and Shuffle phase can be ignored due to a single container for the U+ mode. As the intermediate data are cached instead of dumping them into disk in the U+ mode, the time consumption of Spill and Merge, *i.e.*, $s^o/d^i + s^o/d^o + s^o/d^i$, are trivial. n^m/n_u^m shows the calculation of the number of waves for U+ mode. The overall performance t_u is calculated as Equation 2. The evaluation of time consuming for the D+ mode is shown in Equation 3. For a short job, a majority of Map tasks only spill to disk once, we can ignore the Merge sub-phase *i.e.*, $s^o/d^o + s^o/d^i$, and only consider the Spill sub-phase, *i.e.*, s^o/d^i. The Shuffle phase can be computed as $(s^o * n^c)/b^i$ due to the overlap between the Map phase and Shuffle of two adjacent waves. At last, the decision maker compares the execution time for the U+ mode and D+ mode to kill the slower one.

Table II
MICROSOFT AZURE INSTANCE TYPES

Instance Type	Cores	Memory	Disk	Price
A1	1	1.75GB	70GB	$0.09/hr
A2	2	3.5GB	135GB	$0.18/hr
A3	4	7GB	285GB	$0.36/hr

IV. EXPERIMENTS

A. Experimental Setup

The experiments were conducted on Microsoft Azure [11], which supports different types of servers such as A1, A2, A3. These instances differ in the number of cores, memory size, disk size, and price, as shown in Table II.

Our experiments were performed on two different clusters, which have the same usage charge per hour. One cluster consists of 1 NameNode and 4 DataNodes of A3 instances. The another cluster consists of 1 NameNode and 9 DataNodes of A2 instance nodes. Each node runs CentOS Linux Server 7, JDK version 1.7, and Apache Hadoop version 2.2. To evaluate our optimization techniques, we ran three different benchmark applications from Hadoop example package, *i.e.*, WordCount, TeraSort, and PI. WordCount is a MapReduce program that counts words in input files. TeraSort samples the input data generated by TeraGen, and uses MapReduce to sort them into a total order. PI is a MapReduce program that estimates pi using a quasi-Monte Carlo method.

B. Experimental Results on A3 Cluster

In this experiment, we create a cluster consisting of 1 NameNode and 4 DataNodes of A3 instances. In Figure 7, the number of files varies from 1 to 16, and the size of each file is 10MB. We execute the WordCount benchmark to compare the performance under the original Hadoop and

our MRapid. Our D+ mode gains an improvement of 36.36% compared to the original Hadoop when the file size is 8. The reason is that our improved scheduler chooses DataNodes that are relatively idle and have better data locality to allocate Map tasks. Our new submission framework also reduces the setup and allocation overhead of AMs. When the number of input files is 4, our U+ mode improves the performance by 59.26% compared to the original Uber mode. This is due to our parallel computing mechanism, which can execute Map tasks in parallel and avoid spilling intermediate data into disk when they are small. Figure 7 shows that when the number of input files is very large, the D+ mode can only gain performance improvement by the submission framework, since the original Hadoop behaves nearly to our D+ mode in terms of data locality and resource usage. When the total input file size is 160 MB, the U+ mode has to spill intermediate data into the disk, which is similar to the original Uber mode, the improvement is 11.43%. From our experiment, when the number of files is 8, the D+ mode and the U+ mode have similar performance; when the number is more than 8, the U+ mode performs worse, even though it is still better than the original Uber mode.

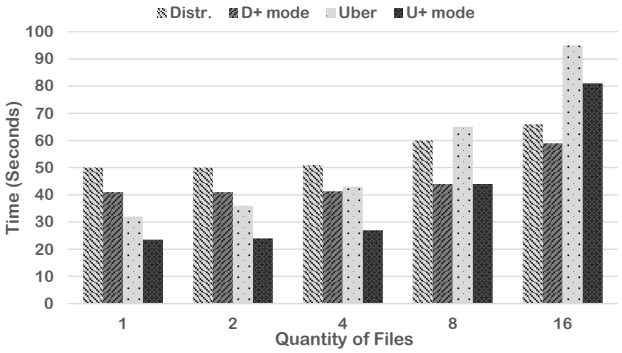

Figure 7. WordCount performance when varying the number of files but fixing the file size to 10MB.

Figure 8 demonstrates the performance of Hadoop and MRapid with 4 input files but the file size varies from 5 MB to 40 MB. The D+ mode can outperform the original Distributed mode by 43.40% when the file size is 40 MB, which is also 11.32% faster than the U+ mode. We observe that the D+ mode gains more performance improvement on larger file size. This is because Algorithm 1 schedules Map tasks as uniformly as possible; however, the original Hadoop only employs the resource of recently reported nodes, which can cause serious allocation imbalance for short jobs. The performance of the D+ mode is better than the U+ mode when the total input data size is large, as the D+ mode can use cluster resource more efficiently than the U+ mode, which uses only one container to execute all tasks.

Figure 9 shows the performance when the total file input

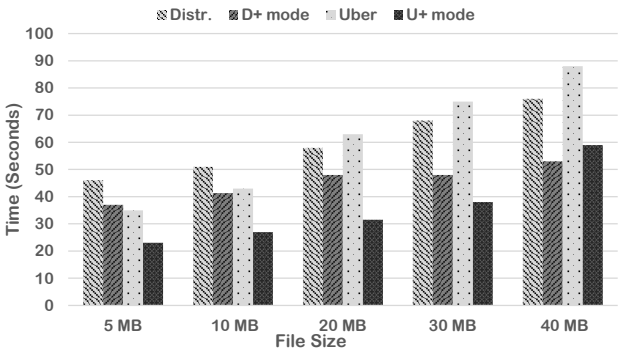

Figure 8. WordCount performance by fixing the number of files to 4 but varying file size.

size is fixed to 60 MB, and the number of files varies from 2 to 4. The performance of 4 files with the file size 15 MB is the best for the D+ mode, where we achieve 79.41% improvement due to better parallelism. The performance of the U+ mode is better when the number of files is 4 due to multithreading parallelism, which outperforms the original Uber mode by 88.89%.

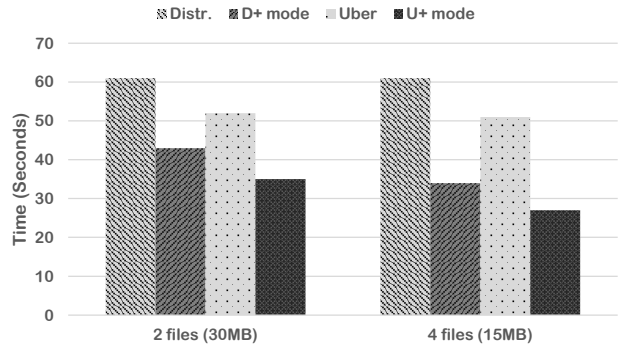

Figure 9. WordCount performance when fixing the input size to 60 MB.

Figure 10 shows the performance of another benchmark called TeraSort. We vary the number of 100-byte rows from 100k to 1,600k with 4 blocks, which designates 4 Map tasks. When the number of rows is 100k, the D+ mode gains 59.42% improvement compared to the original Hadoop. We also observe that the U+ mode is always better than the D+ mode; specifically, the U+ mode outperforms by 67% when the number of rows is 800k because such kind of applications do not require massive computation, and one container can handle it. We notice that the benchmark PI has the similar property, as shown in Figure 11. We vary the random number size from 100m to 1,600m for benchmark PI. When the random number size is more than 200m, for the original Hadoop, it is better to run PI in the original Distributed mode rather than the original Uber

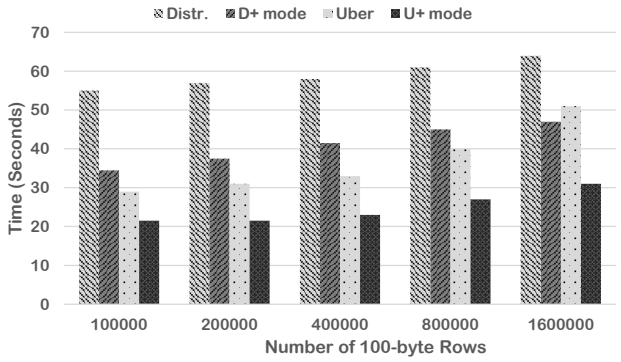

Figure 10. TeraSort performance with different numbers of rows.

mode. However, for MRapid, when the random number size is large, *e.g.*, 1,600m, the U+ mode is still the better choice, which indicates that MRapid alleviates the limitation of the original Uber mode.

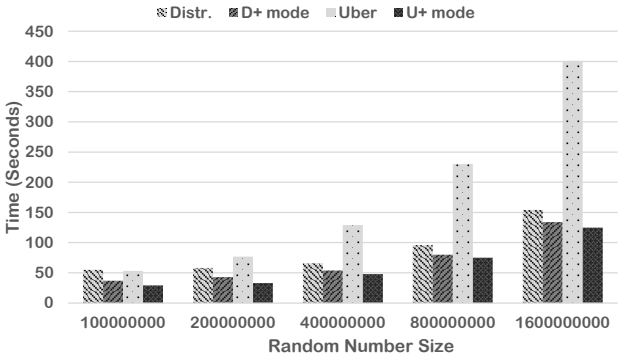

Figure 11. PI performance when varying the number of seeds.

C. Experimental Results for A2 Cluster

In this section, we discuss our experiments on another cluster consisting of 1 NameNode and 9 DataNodes of A2 instances.

Figure 12 compares the performance of our system with the original Hadoop when the number of containers allocated for each core is varied from 1 to 2 in A2 cluster. We find that the performance for MRapid does not fluctuate obviously, especially for the U+ mode when executing WordCount with four 10MB files. This is because the U+ mode only uses one container, and the D+ mode usually selects the relatively idle nodes to launch Map tasks. But for the original Hadoop, when the number of containers per core is 2, the performance of the original distributed mode becomes much worse due to its greedy scheduling.

For public cloud users, the cluster cost is often a concern. We compare the performance for a 10-node A2 cluster and a 5-node A3 cluster, which have around the same cost. As

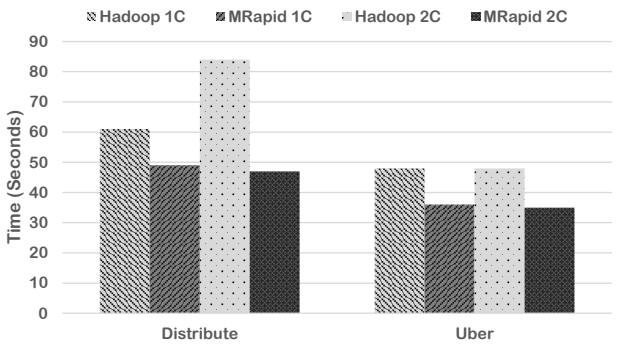

Figure 12. WordCount performance when varying the number of containers for each core.

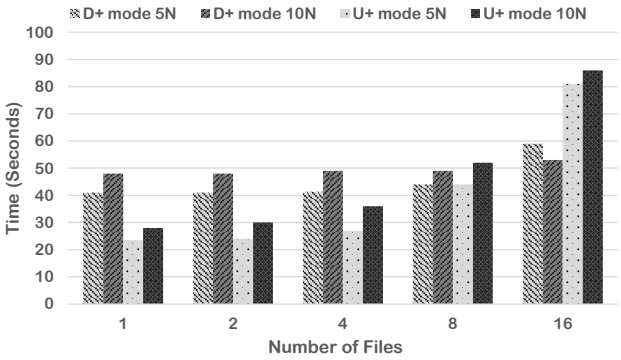

Figure 13. WordCount performance with different numbers of nodes.

shown in Figure 13, for the U+ mode, it is always better to select the A3 cluster. For D+ mode, if the number of files is few and the cluster is relatively idle, it is better to use A3 cluster rather than A2 cluster; otherwise, it is better to deploy A2 cluster. The reason is that although the U+ mode just uses one container to execute the short job, if there are more resources available on the same node, the container for the U+ mode may steal these resources if allowed. For the D+ mode, if the number of files is large, such as 16 in Figure 13, although a cluster with more nodes at the same cost degrades the capability of each node, disk I/O and network contentions could be reduced; thus a cluster with more nodes may read input data and shuffle map results more efficiently.

D. Contribution Analysis

Figure 14 shows the contribution of each optimization technique in the D+ mode when there are 5 nodes in the cluster, there are eight 10 MB files as input for the WordCount benchmark. Our new scheduler using a round-robin technique rather than the greedy approach, which contributes 50% to the performance improvement. The second most significant contribution factor is the job submission framework that creates an AM pool for reusing, which contributes 31%.

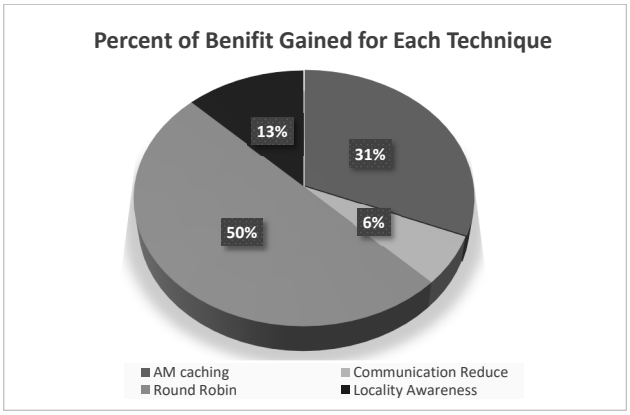

Figure 14. The contribution comparison of different optimization techniques on performance improvement under the D+ mode.

Locality awareness and reducing communication contribute 13% and 6%, respectively.

In Figure 15, there are 4 optimization techniques in the U+ mode. Running tasks in parallel contributes the most, which is 64%. And the submission framework is the second most influential factor for U+ mode, which is 23%. Storing intermediate data into memory and reducing communication between DataNodes and RM contribute 9% and 4%, respectively.

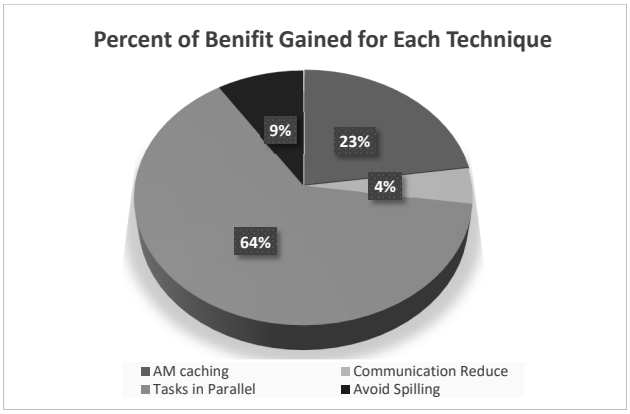

Figure 15. The contribution comparison of different optimization techniques on performance improvement under the U+ mode.

V. Related Work

MRapid is related to the four research areas in optimizing performance of MapReduce short jobs: short job scheduling [5, 12, 13], data-locality awareness [14–16], cache mechanism [17], and reusing resource [18].

Elmeleegy [5] designed a system called Piranha that avoids checkpointing intermediate results to disk. It also supports a simple fault-tolerance mechanism, and employs self-coordination to reduce the cost of high-latency polling protocol. However, this system reduces only cost of spilling data into disk, and the Uber mode is not considered. Yao *et al.* [12] propose a job-size based scheduling algorithm. It leverages the knowledge of workload patterns to reduce average job response time by dynamically tuning the resource sharing among users. But this approach cannot reduce useless overhead caused by Hadoop itself. Yan *et al.* [13] implement an optimized version of Hadoop to reduce the time cost during the initialization and termination stages of a job, and replace the pull-model task assignment mechanism with a push-model approach. This system just reduces the communication between Driver to NameNode and JobTrack to TaskTrack, but cannot reuse previous jobs' execution environment to speed up.

Hammoud *et al.* [14] designed a Locality-Aware Reduce Task Scheduler (LARTS), which collocate Reduce tasks with the maximum required data after recognizing input data locations and sizes. This method is useful only when the input data are skewed, and the performance improve is not significant. Zhang *et al.* [15] proposed a next-k-node scheduling (NKS) method to reserve nodes for Map tasks to satisfy node locality policy. It is not enough for short job by just considering data locality. Maestro [16] is another scheduling algorithm that schedules map tasks in two waves: first, it fills the empty slots of each data node based on the number of hosted map tasks; second, runtime scheduling takes into account the probability of scheduling a map task depending on the replicas of the tasks input data. But for many short jobs, there is only one wave to be executed.

Spark [19][17] is a fast and general engine that can be deployed on Hadoop Yarn. It organizes data into a distributed data structure called resilient distributed dataset (RDD), which can be cached in memory, and be reused across different computations. But we observed that the performance of Spark on Yarn is still slow for short jobs because of the high overhead to launch containers for AMs and executors.

HJ-Hadoop [18] is designed to exploit multicore parallelism at the intra-JVM level, while limiting the number of JVMs created on each node. In our U+ mode, we employ a similar technique, which executes Map tasks of a container in parallel rather than a sequential way.

Besides optimizing MapReduce performance, Hadoop can be improved in many other aspects, such as network [20], HDFS [21], middleware [22], and query optimization [23].

VI. Conclusions and Future Work

In this paper, we introduce an optimized Hadoop system to improve the performance of short jobs in two modes: D+ mode and U+ mode. In D+ mode, we design a new scheduler to schedule Map tasks based on the resource distribution situation and data locality. Instead of waiting for heartbeats reported from NMs to decide how to schedule tasks, our scheduler can allocate resources immediately. Our algorithm not only avoids allocation imbalance problem for short jobs,

but also reduces the communication cost. For the U+ mode, rather than executing Map tasks sequentially, we employ multi-threading to run Map tasks in parallel. In addition, we cache the intermediate data into memory instead of writing them into disk and reading them later when the intermediate data are small. Moreover, we implement a new job submitting framework and speculative execution system to reduce the setup cost for short jobs. Our experiments show that our system can obtain significant performance improvement by 11% to 88% compared with the original Hadoop for short jobs.

Nowadays, Spark has become another data processing engine in Hadoop ecosystem as an alternative to the traditional MapReduce batch processing model. Several optimization techniques of our system can also improve the performance of Spark on Yarn such as the submission framework and the improved CapacityScheduler. In the future, we plan to migrate MRapid to Spark to handle applications such as real-time stream data processing and interactive queries more efficiently.

VII. ACKNOWLEDGEMENT

This work was supported in part by NSF-CAREER-1622292.

REFERENCES

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 1–13, 2004.

[2] Apache Hadoop website. http://hadoop.apache.org/.

[3] Hadoop from Wikipedia website. https://en.wikipedia.org/wiki/Apache_Hadoop.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[5] K. Elmeleegy. Piranha: optimizing short jobs in hadoop. In *VLDB Endowment*, pages 985–996, 2013.

[6] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *ACM SIGMOD*, pages 13–24, 2013.

[7] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. In *VLDB Endowment*, pages 1318–1327, 2011.

[8] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, pages 323–336, 2011.

[9] Apache Spring website. http://projects.spring.io/spring-hadoop/.

[10] Asm website. http://asm.ow2.org/.

[11] Microsoft Azure website. https://azure.microsoft.com.

[12] Y. Yao, J. Tai, B. Sheng, and N. Mi. Lsps: A job size-based scheduler for efficient task assignments in hadoop. In *IEEE Transactions on Cloud Computing*, pages 411 – 424, 2014.

[13] J. Yan, X. Yang, R. Gu, C. Yuan, and Y. Huang. Performance optimization for short mapreduce job execution in hadoop. In *CGC*, pages 1–7, 2012.

[14] M.Hammoud and M. F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *CloudCom*, pages 570–576, 2011.

[15] X. Zhang, Z. Zhong, S. Feng, B. Tul, and J. Fan. Improving data locality of mapreduce by scheduling in homogeneous computing environments. In *Int. Symp. on Parallel and Distributed Processing with Applications*, pages 120–126, 2011.

[16] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu. Maestro: Replica-aware map scheduling for mapreduce. In *CCGRID*, pages 435–442, 2012.

[17] S. Ryza, U. Laserson, S. Owen, and J. Wills. *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, 2015.

[18] Y. Zhang. Hj-hadoop: An optimized mapreduce runtime for multi-core systems. In *SPLASH*, pages 111–112, 2013.

[19] Apache Spark website. http://Spark.apache.org/.

[20] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *ACM SIGCOMM*, pages 407–420, 2015.

[21] H. Zhang, L. Wang, and H. Huang. Smarth: Enabling multi-pipeline data transfer in hdfs. In *ICPP*, pages 1–10, 2014.

[22] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. Ajira: A lightweight distributed middleware for mapreduce and stream processing. In *ICDCS*, pages 545–554, 2014.

[23] H. Zhang, Z. Sun, Z. Liu, C. Xu, and L. Wang. Dart: A geographic information system on hadoop. In *IEEE CLOUD*, pages 1–8, 2015.

[24] H. Ma, S. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. Symbolic analysis of concurrency errors in openmp programs. In *ICPP*. IEEE, 2013.

[25] H. Huang, L. Wang, B. Tak, L. Wang, and C. Tang. CAP3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *IEEE Cloud*, pages 228–235. IEEE, 2013.

[26] V. Subramanian, L. Wang, E. Lee, and P. Chen. Rapid processing of synthetic seismograms using windows azure cloud. In *CloudCom*, 2010.

[27] V. Subramanian, H. Ma, L. Wang, E. Lee, and P. Chen. Rapid 3d seismic source inversion using windows azure and amazon EC2. In *SERVICES*, pages 602–606. IEEE, 2011.