Coalescing HDFS Blocks to Avoid Recurring YARN Container Overhead

Wonbae Kim, Young-ri Choi, Beomseok Nam
Ulsan National Institute of Science and Technology (UNIST)

***Abstract*—Hadoop clusters have been transitioning from a dedicated cluster environment to a shared cluster environment. This trend has resulted in the YARN container abstraction that isolates computing tasks from physical resources. With YARN containers, Hadoop has expanded to support various distributed frameworks. However, it has been reported that Hadoop tasks suffer from a significant overhead of container relaunch. In order to reduce the container overhead without making significant changes to the existing YARN framework, we propose leveraging the *input split*, which is the logical representation of physical HDFS blocks. Our *assorted block coalescing* scheme combines multiple HDFS blocks and creates large input splits of various sizes, reducing the number of containers and their initialization overhead. Our experimental study shows the assorted block coalescing scheme reduces the container overhead by a large margin while it achieves good load balance and job scheduling fairness without impairing the degree of overlap between map phase and reduce phase.**

I. Introduction

Apache Hadoop is designed to run fault tolerant distributed tasks for large scale datasets stored in HDFS. Since its initial release that resembled Google's MapReduce framework [1], Hadoop has been continuously developed and improved. As Hadoop has become one of the most popular distributed job processing frameworks, it has transitioned from a dedicated cluster environment into a shared cluster environment.

In order to run with other applications in a shared cluster, Hadoop had to be used along with generic job schedulers such as Torque [2]. With such external job schedulers, each Hadoop job specifies the configuration of a small-sized compute cluster and the schedulers create a new Hadoop cluster for each job. However, these job schedulers do not account for the locality of HDFS blocks. Hence, it has been reported that Hadoop version 1 in large scale data centers fails to leverage data locality [3].

Appusmay et al. [4] analyzed the job sizes from real-world MapReduce deployments and demonstrated that most jobs are under 100 GB in size and the median job size is just 14 GB. Since such medium-sized datasets are replicated in a relatively small number of HDFS nodes compared to the total number of nodes in the data centers (i.e., 900,000 servers in Google data centers), traditional job schedulers rarely allocate servers that contain replicas of input datasets. As a result, the lack of locality requires remote data access, degrades scalability, hurts the cluster utilization, and causes many other critical performance problems.

In order to resolve this problem, Hadoop was extended to a more generic resource management framework - YARN [3]. YARN provides a container abstraction that isolates computing tasks from physical computing resources. YARN container is a process that executes an application-specific task using constrained computing resources. With the container abstraction, YARN could expand its scope to support more diverse programming models such as Hive, Giraph, REEF, and many others.

For such various programming models, YARN splits up the functionality of JobTracker and TaskTracker of old Hadoop version 1 into a generic resource manager, per-node slave node managers, per-application application masters, and per-task containers running on each node manager. That is, YARN delegates scheduling functionalities to application-specific components, but arbitrates resource contention between tenants.

However, inevitably there is a price to pay in becoming a generic resource manager. YARN is ignorant of application-specific semantics and it trades the fine-grained control of a specific high-level framework for versatility. For instance, Kim et al. [5] reported that MapReduce programming model in YARN suffers from the high overhead of container initialization which repeats for each map or reduce task. In our testbed cluster, we also observed the initialization overhead of a container that runs a Grep task for a 128 MB HDFS block is about 5x higher than the task scheduling overhead (6.51 sec vs. 1.28 sec) and about 1.7x higher than the pure task execution time (6.51 sec vs. 3.85 sec). Since we repeat the container initialization for each HDFS block, the high container overhead often becomes a dominant fraction of job execution time. Even if we consider that failures can occur at task level, 6.51 sec initialization overhead for just a 128 MB block seems just too high.

The lifespan of a task can vary dramatically. If a job is computation intensive and the lifespan of a task is longer than an hour, the container initialization overhead can be ignored. However, as we discussed above, most Hadoop jobs are short-lived and the container initialization overhead can account for a dominant portion of its lifespan.

In order to reduce YARN container overhead, we can take various approaches. First, we may re-design a high-level framework so that it can reuse containers, as in Spark [6], or we can improve YARN to reduce the container initialization overhead via ad-hoc optimizations. Both approaches are feasible but they require significant amounts of effort to restructure existing Hadoop ecosystem frameworks. Instead, in this work, we propose a simple but effective optimization scheme - *HDFS block coalescing via input split* for Hadoop MapReduce jobs.

The basic idea of HDFS block coalescing is that we can reduce the container initialization overhead by reducing the number of containers. Our HDFS block coalescing scheme logically combines multiple HDFS blocks and makes a single container process them. That is, the HDFS block coalescing can avoid re-creating containers as if it is reusing containers for multiple HDFS blocks. However, although increasing input sizes reduces the container overhead, it is vulnerable to the problems of coarse-grained scheduling - load imbalance, low degree of overlap between map and reduce phases, and unfair job scheduling.

In order to resolve these problems, we propose a novel *assorted block coalescing* scheme, which reduces the container overhead while achieving good load balance. The contributions of this work are summarized as follows:

- In this work, we show YARN container overhead accounts for a dominant portion of Hadoop job lifespan, and large input splits can mitigate the problem.
- We propose a novel assorted block coalescing scheme that creates large input splits of various sizes, which helps effectively reuse YARN containers for multiple HDFS blocks. The proposed assorted block coalescing scheme does not require major modifications to the current YARN design.
- Through extensive performance study, we show HDFS block coalescing improves the average job response time of concurrent jobs by up to 36%.

The rest of this paper is organized as follows. In Section II, we describe the motivation of this work. In Section III, we propose a simple but effective maximum block coalescing scheme that reduces container overhead. Section IV discusses the problems of coarse-grained scheduling. Then, in Section V and Section VI, we present an alternative scheme - the assorted block coalescing scheme and its performance results. In Section VII, we discuss related works, and finally we conclude in section VIII.

II. YARN Container Overhead

In order to start a Hadoop map task in YARN, YARN exchanges many messages with various components of YARN - NodeManager, MRAppMaster, ApplicationMaster, ContainerLauncher, and ContainerManager. Once a container starts, the container performs the following tasks: i) it loads job configuration, ii) it initializes various container parameters including UserGroupInformation, iii) it loads credentials from UserGroupInformation, iv) it creates a JVM task, v) it configures the JVM task, vi) it creates a UserGroupInformation instance for the JVM task, vii) it sets a job class loader, viii) it creates a log synchronizer, ix) it launches the JVM task, and then finally x) it starts a map task.

1) Experimental Setup: In order to measure how long these 10 initialization steps take, we submitted a Grep job that processes 250 GB datasets in our testbed Hadoop cluster. Our testbed cluster consists of one NameNode and 32 DataNodes. Each node runs CentOS 5.5 and has two quad-core 2.13GHz Intel Xeon CPUs, 20GB RAM, and a two 7200 RPM HDDs - one for OS partition and the other for HDFS. In each node, we set the number of slots to the number of cores - eight. The NameNode and 19 DataNodes are connected by a Gigabit Ethernet switch and the other 13 DataNodes are connected by another Gigabit Ethernet switch. The two 1G switches are connected by another Gigabit Ethernet switch, which forms two level network hierarchy.

2) YARN Container Overhead: These 10 initialization steps take 6.51 seconds in our testbed cluster. This overhead is not negligible because sequential read throughput of hard disks these days is often higher than 100 MB/sec and we can read a 128 MB HDFS block in a second. Via a back-of-the-envelope calculation, YARN seems to spend 5x as much time initializing a container as reading a 128 MB HDFS block.

For long-lived tasks that last for several minutes, a 6.51 seconds container initialization overhead can be considered negligible. However, for most I/O intensive representative Hadoop applications, we find that re-occurring container initialization causes significant overhead. One fundamental way of resolving this high container overhead is to make YARN reuse containers. However, this requires a significant amount of effort to restructure the current design of YARN MapReduce. Moreover, it will prevent other programming models from re-authenticating each task with different credentials.

III. HDFS Block Coalescing via Input Split

HDFS is a distributed file system that partitions a large file into a sequence of blocks and distributes them across DataNodes. The block size and the number of replicas can be configured per file. But once the partitioned blocks are stored, the size of blocks cannot be modified unless we format the file system and upload the file again.

A. Input Split

HDFS blocks are fixed-sized blocks. Hence, a single line in a text file block can spill over into another block. Since a map task often needs to access data across multiple blocks, Hadoop provides a logical representation of partitioned data blocks, which is referred to as *input split*.

When a Hadoop job is submitted, the job submitter calls `getSplits()` method of `InputFormat` class, which generates logical input splits based on data-specific logical boundaries. The default behavior of the `getSplits()`

method is to find the next logical boundary of physical HDFS blocks. With the input split, Hadoop can access the truncated data in the next block by figuring out the location of the next block and it completes to read a whole record.

Each input split contains the location information of HDFS blocks and their replicas along with the offset information of the blocks. The split information provides map tasks with a record-oriented view of input data so that map tasks can read a record in its complete form, not a partial record.

B. *Maximum Coalescing of HDFS Blocks*

The size of input splits can be dynamically and arbitrarily configured unlike the physical HDFS block size because they are logical partitions. Creating a logical large block serves the purpose of arbitrarily increasing the workload of each container. With the logical input split, we make a single container process multiple HDFS blocks without re-partitioning and uploading a file onto HDFS. By increasing the size of logical input splits, we can effectively reduce the number of map waves and the number of containers.

In the *maximum coalescing* scheme that we propose, we maximize the input split size so that only a single wave of map tasks can complete the entire map phase. That is, the size of input splits is set to the input file size divided by the total number of slots ($totalNumBlocks/(numSlotsPerNode \times numNodes)$). We refer to this size as *maxSplitSize*. In the maximum coalescing scheme, no more than a single input split is assigned to each slot. Therefore, each job creates only one container per slot and it minimizes the container overhead while maximizing the slot utilization of a cluster.

C. *Performance Evaluation of Maximum Coalescing*

In this section, we analyze the performance effect of the maximum coalescing scheme in the Linux cluster that we described in section II-1. We implemented the maximum coalescing scheme in Hadoop 2.7.2, and set the replication factor to 3.

In the experiments shown in Figure 1, we ran WordCount for a 250 GB text file while varying the HDFS block size. With the Hadoop's *default* input split creation method, the number of input splits (map tasks) varies from 7989 to 125 as we increase the HDFS block size from 32 MB to 2 GB. With the help of `Combiner` class in Hadoop, the volume of data to shuffle in WordCount application is significantly smaller than the input file size. Besides this, the reduce phase of WordCount accounts for a very small portion of the job execution time. Therefore, the map phase execution determines the overall job execution time of WordCount.

Figure 1 shows that WordCount application runs faster as we increase the HDFS block size from 32 MB to 1 GB. When the HDFS block size is 1 GB, each slot runs a single map wave, which minimizes the container initialization overhead. Therefore, WordCount runs 1.7x and 1.16x faster compared to when the HDFS block size is 32 MB and 128 MB respectively. This result confirms that the container initialization overhead accounts for about 16∼70% of job execution time.

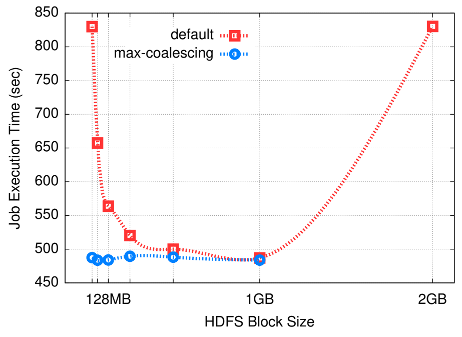


Figure 1: Job Exec. Time vs Block Size (WordCount)

However, when the HDFS block size is set to 2 GB, WordCount runs about two times slower than the HDFS block size of 1 GB. This is because the number of input splits becomes smaller than the number of available slots. I.e., almost half of the slots are idle.

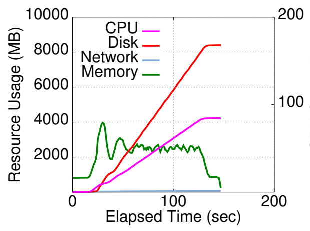
For various default HDFS block sizes, we run the same experiments again but this time we used the maximum coalescing scheme, generating large input splits and reducing the number of map waves down to one for all HDFS block sizes.

As we expected, no matter how small we set a HDFS block size, the maximum coalescing scheme shows similar job execution times to when the default HDFS block size is 1 GB. I.e., the maximum coalescing scheme makes job execution time independent of a HDFS block size.

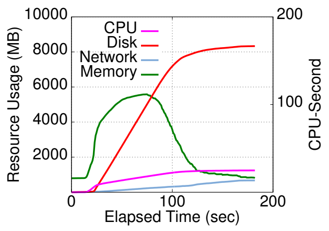
IV. Problems of Coarse-Grained Task Scheduling

Although the maximum coalescing scheme can help minimize the container overhead, it makes concurrent applications suffer from the problems of coarse-grained task scheduling. More specifically, if we increase the size of input splits, YARN suffers from mainly three problems - i) load imbalance, ii) low degree of overlap between map and reduce phases, and iii) unfair job scheduling.

First, a coarse-grained job scheduler fails to balance the loads. The execution time of each task can vary depending on the characteristics of computation and the contents of each block. With a large number of waves that process small blocks, YARN can resiliently react to the dynamic load change by scheduling the rest of tasks to available slots. But, in contrast, if we have a single large task per slot, there is no opportunity to balance the loads even if one slot finishes much earlier than the other slots. Figure 2 shows the resource usage patterns of Grep application with two



(a) Grep with 128 MB HDFS blocks



(b) Grep with 1 GB HDFS blocks

Figure 2: Resource usage pattern for map tasks in Grep application

different HDFS block sizes. Using `dstat`, we measured the average resource usage across 32 nodes. For memory usage, we show the fluctuations in usage over time, but for CPU time, and disk access, and network traffic, we show the cumulative resource usage. From the experiments, we can see that a large 1 GB block size helps access more data in a shorter time, i.e., about 90% of input data are processed within 100 sec, but Grep with a 128 MB HDFS block size processes only 75% of input data within 100 sec. However, interestingly, a 1 GB block size results in a longer execution time due to the variance of map task completion time.

Second, the degree of overlap between map phase and reduce phase will become low. If there are multiple waves of map tasks, shuffle phase starts when the progress of the map phase exceeds 5% in Hadoop's default configuration. That is, when a map task completes, the intermediate result of the map task is transferred to reduce tasks while the next map tasks are running. However, as we increase the size of input splits, the number of map waves will be decreased and the starting point of shuffle phase will be delayed. For an instance, if there is only a single map wave, the map phase and reduce phase cannot overlap at all.

Third, large HDFS blocks may negatively impact job scheduling fairness. YARN does not schedule a larger number of tasks than a fixed number of slots and it does not preempt running tasks. Hence, new tasks need to wait indefinitely until previous tasks release slots. That is, increasing an input split size results in longer running tasks, and the long running tasks will make waiting tasks wait longer which will hurt the job scheduling fairness.

V. Assorted Coalescing to Enable Fine-Grained Task Scheduling

In order to mitigate the drawbacks of the maximum coalescing scheme, we designed and implemented an alternative HDFS block coalescing scheme - *assorted coalescing*. In the assorted coalescing scheme, we generate input splits of various sizes and assign the assorted large and small input splits to each node. With various input split sizes, small tasks can start shuffle phase early and the small tasks can yield their slots to other tasks waiting in the queue. As a result, assorted input split sizes can improve job scheduling fairness, load balancing, and shuffle phase pipelining.

The detailed algorithm is shown in Algorithm 1. The algorithm starts from calling `getSplitTypes()` in line 2, which generates an array of various input split sizes using a geometric progression and the number of slots assigned for each input split size. The split type structure that `getSplitTypes()` generates contains an array of various input split sizes and the number of slots assigned for each split size. In our implementation that employs a simple geometric progression, the input split sizes are set to $\{maxSplitSize, maxSplitSize/2, maxSplitSize/4, ...\}$.

Algorithm 1 Assorted Coalescing: Creating Input Splits of Various Sizes

```
1:  procedure CREATESPLITS
2:      $splitTypes \leftarrow$ getSplitTypes();
3:      $numCoalescedBlocks \leftarrow 0$;
4:      $splitList.initialize()$;
5:      for $i = 0$; $i < splitTypes$.size(); $i$++ do
6:          $<splitSize, numSlots> \leftarrow splitTypes[i]$;
7:          sortNodesForLoadBalancing($nodes$);
             // Nodes with fewer blocks come first
8:          for all $node \in nodes$ do
9:              $atLeast = 0.9$;
10:             $n = MIN(numSlots, node.numAvailableSlots)$
                    $\times (maxSplitSize/splitSize)$;
11:             for all $i = 0$; $i < n$; $i$++ do
12:                 $blocks \leftarrow getLocalBlocks(node,$
                        $splitSize)$;
13:                 if $blocks.size() < (splitSize$
                        $\times atLeast)$ then
14:                     break;
15:                 end if
16:                 while $blocks.size() < splitSize$ do
17:                     $blocks.add(getRemoteBlock$
                            $(nodes))$;
18:                 end while
19:                 $split \leftarrow createSplit(blocks)$;
20:                 $splitList.add(split)$;
21:                 $numCoalescedBlocks$ += $splitSize$;
22:             end for
23:         end for
24:     end for
        // Create single block splits for remaining blocks.
25:     if $numCoalescedBlocks < totalNumBlocks$ then
26:         $createDefaultSplitsForRemainings(splitList)$;
27:     end if
28: end procedure
```

In the first iteration of the outermost loop shown in line 5, we process the maximum input split first. In the assorted coalescing, a configurable number of slots in each node are used for the maximum input splits. In our implementation, 50% of available slots are used for the maximum input splits by default. But the number of slots used for each input split size can be configured according to workload and an application type. In the next iteration, the second largest input splits will use another group of slots, and we keep this iteration for all split sizes until all slots become busy.

On each iteration, given the number of assigned slots, we create input splits of the current size across data nodes in a greedy fashion. Note that it is not desirable to create a large input split if a data node does not have an enough number of local blocks for the current input split size. Since HDFS blocks are replicated and it is not guaranteed that each data node has an equal number of blocks, we sort data nodes according to the number of local HDFS blocks for the current input file (line 7) and we give a higher priority to those nodes that have a lower number of data blocks so that they can select their local blocks for their input split first. This greedy strategy increases the probability of creating larger input splits. If we allow data nodes that have a large number of blocks to choose first, the data nodes that have a lower number of blocks may find out that their local blocks have been already taken by other data nodes. Hence, the data node will fail to create a large input split using local blocks. This problem can cause the load imbalance problem.

In the for loop shown in line 8 $\sim$ 23, each data node selects the next candidate block to coalesce. Note that HDFS blocks in a logical input split does not have to be contiguous, but they can be chosen randomly. During the iterations of the for loop, the selected candidate blocks are temporarily stored in the `blocks` array. If a data node fails to create an input split of a specific size, the data blocks stored in `blocks[]` will be released so that they can be used for the next smallest input split size.

In our implementation, we allow 10% of remote blocks to be included in each input split and this ratio can be configured. If more than 10% of remote blocks are needed, we abort creating input splits of the current size and move on to the next smaller input splits. If there exist HDFS blocks that are not included in any input split at the end of the assorted coalescing scheme, we fall back to the default input split creation algorithm for those blocks as shown in line 26.

Figure 3 shows an example of input splits generated by the maximum coalescing scheme and the assorted coalescing scheme. Suppose a file consists of 36 physical HDFS blocks and they are distributed over three data nodes with replication factor 2. When each node has 2 slots, the total number of available slots is 6. In the maximum coalescing scheme, we create two input splits for each node, i.e., one input split per slot. Thus, a single map wave will process 6 HDFS blocks without re-creating containers.

In the assorted coalescing scheme, we create the maximum input split for half of the slots. But the size of the rest of the input splits progressively decreases by half (or another configurable common ratio). With varying input split sizes,

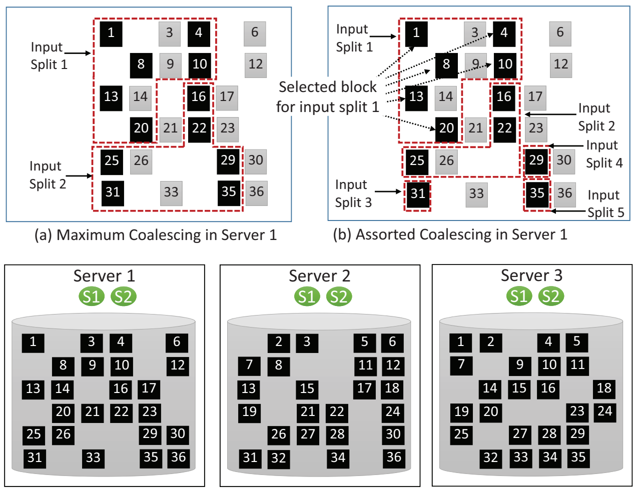


Figure 3: An Example of Input Splits with Maximum Coalescing and Assorted Coalescing

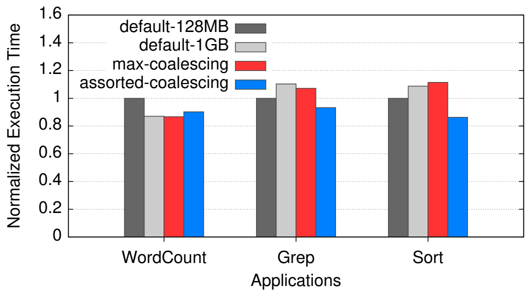


Figure 4: Normalized Job Exec. Time with Block Coalescing

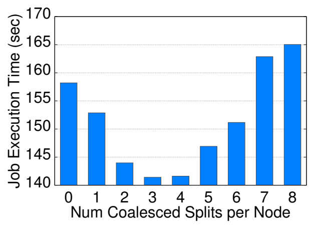
we allow YARN to schedule multiple concurrent tasks in a more fair way while mitigating the container initialization overhead.

VI. Performance Evaluation of Assorted Coalescing

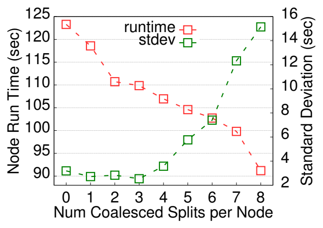
In this section, we evaluate the performance of *assorted coalescing* scheme by evaluating the performance of WordCount, Grep, and Sort applications on the Linux cluster that we described in section II-1. These three applications have distinct resource usage patterns with regard to I/O, computation, and network usages.

Figure 4 shows the normalized job execution time of each application with different coalescing schemes. Overall, the assorted coalescing scheme outperforms the maximum coalescing scheme and the default Hadoop scheduling for Grep and Sort applications because it reduces container overhead while avoiding the problems of coarse-grained scheduling. Now, let us look in more detail at the behaviors of the assorted coalescing scheme in terms of load balancing, MapReduce pipelining, and job scheduling fairness.

In the experiments shown in Figure 5, we ran Grep using the assorted coalescing scheme with only two types of input



(a) Job Execution Time



(b) Execution Time of Fastest Node and Standard Deviation of Execution Times

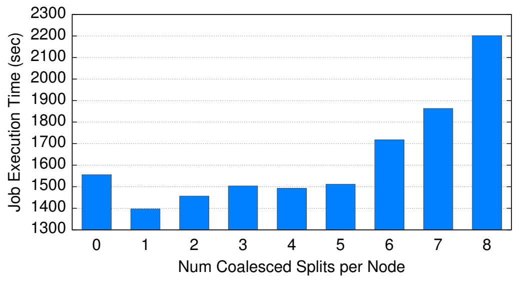
Figure 5: Load Balance vs. Container Overhead with Grep application

split sizes including *maxSplitSize* and default HDFS block size. We vary the number of slots for each type to evaluate the effect of each input split size.

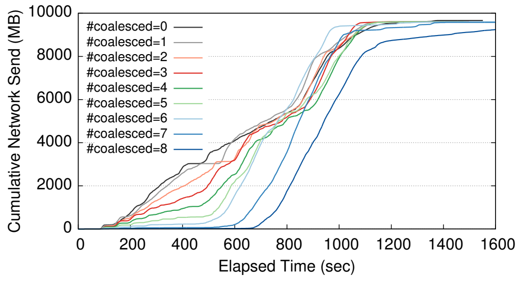
As we discussed in Section IV, a large input split can hurt load balance and it can delay data transfer from map tasks to reduce tasks. When the number of the coalesced input splits is 8 (the number of the slots per node), all slots process a single map wave as in the maximum coalescing scheme. When the number of the coalesced input splits is 0, its performance is no different from the performance of the default Hadoop. Unlike the WordCount application, the Grep application shows worse performance when we use the maximum size input splits for all slots. This is because each task of Grep spends various amounts of time on the computation even if we assign the same amount of input data to each task.

Figure 5a shows the total job execution time over various numbers of the coalesced input splits, and Figure 5b shows the execution time of the fastest node that completed all of its tasks for Grep. As the number of coalesced input splits increases, the task completion time of the fastest node decreases since the container overhead is reduced. The shortest completion time is 92 seconds when the maximum size input splits are used for all slots. But, if we do not coalesce any HDFS blocks, the shortest completion time is 123 seconds. Although some tasks run faster when we use more maximum size input splits, the standard deviation of task completion times increases as shown in Figure 5b. This result shows that the maximum coalescing scheme can suffer severely from the load imbalance because it fails to adaptively balance the load during the task executions.

This result implies that we need to take both the container overhead and the load balance into account at the same time. When we create the maximum size input splits for only half of the slots, i.e., when the number of coalesced input splits is 3 or 4 in the result, we can compromise the container overhead reduction and the load balance. The slots that process the maximum size input splits benefit from the reduced container overhead, while the slots that process small input splits help balance the system load. This results validates the proposition of the assorted coalescing scheme.



(a) Job Execution Time.



(b) Cumulative Network Send Usage

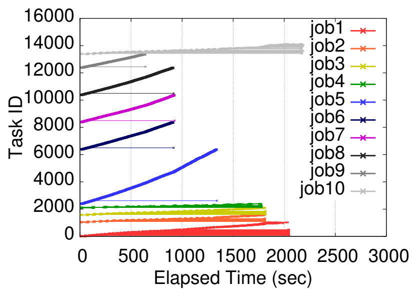
Figure 6: Degree of Overlap between Map Phase and Shuffle Phase vs. Container Overhead (Sort)

Another downside of a large input split size is that the starting time of data transfer between map tasks and reduce tasks is delayed because the execution time of the first map wave becomes longer. For the applications that generate small map outputs, the low degree of overlap between map phase and reduce phase does not degrade overall job execution time. But if applications generate large outputs in map phase as in Sort, the shuffle phase constitutes a critical bottleneck and the delayed shuffle phase can hurt the performance.

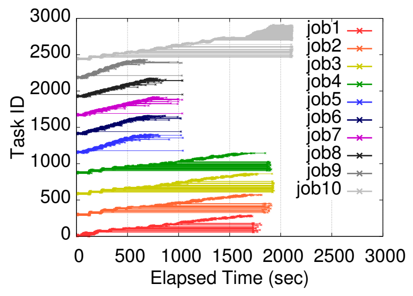
In the experiments shown in Figure 6, we use two types of input splits and vary the number of slots for each type as in the previous experiments. We measure the job execution time and the cumulative network usage.

With the default HDFS block size (the number of maximum size input splits = 0), the Sort application finishes in about 1550 sec, but when we create a single maximum size input split, the job execution time decreases down to 1400 sec. However, as we increase the number of maximum input splits, the job execution time slightly increases but it is still faster than the default block size unless the number of large input splits is larger than 5. If the number of maximum input splits is larger than 5, the cluster suffers from burst transfers of map outputs in the delayed shuffle phase.

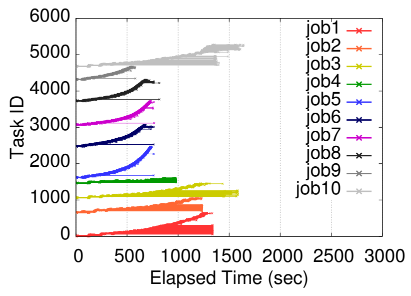
Figure 6b shows the cumulative amount of transferred data during the job execution. As the number of slots that process maximum input splits increases, we find map outputs are transferred in a shorter time. When all slots process the maximum size input splits, the shuffle phase is delayed for



(a) Default

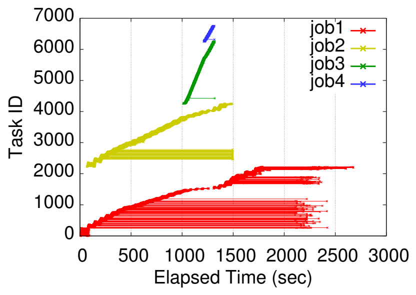


(b) Maximum Coalescing

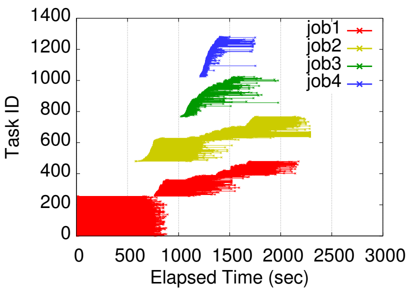


(c) Assorted Coalescing

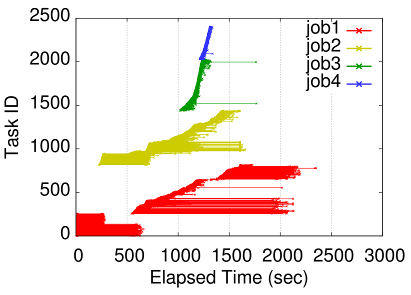
Figure 7: Lifespan of Tasks with Multiple Concurrent Jobs (10 Jobs)



(a) Default



(b) Maximum Coalescing



(c) Assorted Coalescing

Figure 8: Lifespan of Tasks with Multiple Concurrent Jobs (4 Jobs)

about 600 seconds, which explains why the job execution time increases for about 600 seconds.

In order to evaluate the fairness of job schedules that the assorted block coalescing schemes generates, we ran experiments varying the number of jobs and input files using the Hadoop fair scheduler. The fair scheduling algorithm assigns resources to all jobs so that all jobs get an equal share of resources over time on average.

In the experiments shown in Figure 7 and Figure 8, we submit 10 jobs at the same time and 4 jobs with varying the submission times (0 sec, 10 sec, 1000 sec, 1200 sec), respectively. The 10 jobs consist of 4 WordCount applications, each of which reads a different 64 GB input file, 5 Grep applications, each of which reads its own 250 GB input file, and one Sort application that accesses a 64 GB input file. And the 4 jobs consists of one WordCount, one Sort, and two Grep applications. Again, each job accesses a different input file.

Figure 7a and Figure 8a show the lifespan of map and reduce tasks when the default HDFS block size (128 MB) is used for input splits without block coalescing. The default Hadoop creates about 14,000 tasks and 7,000 tasks in each of the experiments, respectively. The jobs share the available slots and make progress concurrently. With the fine-grained block sizes, the default Hadoop achieves fairness although they suffer from the container overhead.

Figure 7b and Figure 8b show the lifespan of tasks when we employ the maximum coalescing scheme. Since it minimizes the number of tasks, the total number of tasks in the experiments is less than 3,000 and 1,400, respectively. The maximum coalescing scheme does not consider the job scheduling fairness. But, since we submit all jobs at the same time in Figure 7b, each slot in a node is used for different jobs. In Figure 8b, we observe that some jobs suffer from increased waiting time due to its unfair scheduling. Because of the low container overhead, most of the HDFS input blocks are processed earlier than when we use the default Hadoop. However, we observe that some tasks suffer from longer waiting times, delayed shuffle phases, and load imbalance problems. Hence their job execution times are higher than those of the default Hadoop.

Figure 7c and 8c show that the assorted coalescing scheme has better fairness than the maximum coalescing scheme. With the good load balancing and the reduced container overhead, the assorted coalescing scheme improves the job execution time by 36.3% over the default Hadoop shown in Figure 7.

VII. Related Work

Recent analysis of production Yahoo! Hadoop clusters [4], [7] reported that the median input size of Hadoop jobs is smaller than 14 GB and over 80% of the jobs complete execution in under 10 minutes. Besides, the failure rates in Yahoo! Hadoop clusters are only about 1% per month. Considering the small job size and failure rates, YARN in its current form needs to be redesigned for short jobs.

The overhead of container initialization has been a major challenge especially for interactive database queries. Tenzing [8] is a SQL query engine built on top of MapReduce for ad hoc analysis in Google data center. Tenzing avoids the overhead of spawning new tasks by employing a pool of ever-running worker tasks. It has been pointed out that using ever-running worker tasks have two shortcomings. One is that they often waste computing resources because of the fixed number of reserved workers. And the other is that the reserved workers have limitations in leveraging data locality.

Piranha [7] is a Hadoop extension designed for DAG execution of short jobs. Instead of running tasks in two-level mappers and reducers, Piranha spawns map-only tasks and connects *initial tasks*, *intermediate tasks*, and *terminal tasks* using ZooKeeper [9]. Similar to our approach, Piranha employs Hadoop's *input split generator*. For the intermediate and terminal tasks, Piranha generates zero-length input splits, that are not associated with any HDFS blocks, so that they can be scheduled on any available slots in the cluster.

AJIRA [10] is a middleware designed to support generic programming models and to improve the performance when dealing with small to medium data volumes. It shares the goal of our work, but AJIRA achieves this goal by avoiding the use of a coordination master, and minimizing the extent of fault tolerance. Our work is different from AJIRA in that we do not compromise any valuable features of Hadoop such as fault tolerance and security.

Starfish [11] is an optimization framework that searches for good parameter configurations by employing dynamic job profiling to capture the run-time behavior of MapReduce jobs. However, Starfish requires users to configure Hadoop parameters by static settings.

In addition to these works, there exist many other related projects such as Quincy [12], iShuffle [13], Manimal [14], and many more. Our work is different from these works in that our approach solely focuses on reducing the container overhead. We believe our work complements these works.

VIII. Conclusion

In this work, we reduce YARN container overhead which accounts for a significant portion of MapReduce job execution time. In order to reduce the container overhead, we propose a novel block coalescing scheme that combines multiple HDFS blocks to generate a large input split. Thereby, it effectively reduces the number of containers and the container overhead. Our experimental study shows the assorted block coalescing scheme reduces the container overhead by a large margin while it achieves good load balancing and job scheduling fairness without impairing the degree of overlap between map phase and reduce phase.

Acknowledgment

This research was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-SRFC-IT1501-04.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[2] G. Staples, "TORQUE resource manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, 2006.

[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2013.

[4] R. Appuswamy, C. Gkantsidis, D. Narayana, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for Hadoop: Time to rethink?" in *4th annual Symposium on Cloud Computing (SOCC)*, 2013.

[5] J.-S. Kim, N. Cao, and S. Hwang, "Moha: Many-task computing meets the big data platform," in *Proceedings of the 12th International Conference on e-Science*, 2016.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2010 USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[7] K. Elmeleegy, "Piranha: Optimizing short jobs in Hadoop," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 11, pp. 985–996, 2013.

[8] 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," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 12, pp. 1318–1327, 2011.

[9] "Apache ZooKeeper," *http://zookeeper.apache.org/*.

[10] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal, "AJIRA: A lightweight distributed middleware for mapreduce and stream processing," in *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*, 2014.

[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 *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[13] Y. Guo, J. Rao, and X. Zhou, "iShuffle: Improving hadoop performance with shuffle-on-write," in *10th USENIX International Conference on Autonomic Computing (ICAC)*, 2013, pp. 107–117.

[14] M. J. Cafarella and C. Ré, "Manimal: Relational optimization for data-intensive programs," in *13th International Workshop on the Web and Databases (WebDB)*, 2010.