SMapReduce: Optimising Resource Allocation by Managing Working Slots at Runtime

Feng Liang
Department of Computer Science
The University of Hong Kong
Hong Kong
fliang@cs.hku.hk

Francis C.M. Lau
Department of Computer Science
The University of Hong Kong
Hong Kong
fcmlau@cs.hku.hk

***Abstract*—Hadoop version 1 (HadoopV1) and version 2 (YARN) manage the resources in a distributed system in different ways. HadoopV1 executes MapReduce tasks in working slots that are statically configured; YARN uses a set of task containers to encapsulate its memory and CPU resources. However, neither of them considers the runtime performance of the cluster when deciding the proper number of concurrent tasks to run on each node to achieve the optimal throughput. In order to gain higher performance, the users of Hadoop usually need to use their experience to carefully configure the resources of the cluster and the resources needed by their jobs. But as the workload is typically always changing in the cluster, rarely could such a manual configuration lead to optimized performance. In this paper, we study the MapReduce job performance in HadoopV1 and YARN with different resource configurations, and model the cluster throughput in terms of the resource capacity of the cluster. We propose SMapReduce, which can dynamically manage a proper number of concurrent tasks running on each node. SMapReduce can gain the maximum job throughput by considering the thrashing phenomenon and the balancing between map and reduce tasks. Evaluation results show that SMapReduce can yield significant performance speedup comparing to both HadoopV1 and YARN for various MapReduce workloads.**

***Keywords*-Resource Management; MapReduce; Hadoop; YARN; Performance Modeling**

I. Introduction

Nowadays, big data processing and analysis is critical for many scientific and industrial applications. MapReduce [1], a popular distributed computing framework proposed by Google, because of easy programming, high performance and fault tolerance, is a popular tool for big data analysis [2], [3], [4]. Programmers only need to adapt the computation tasks to the map and reduce interfaces, and MapReduce will take care of managing and running the tasks efficiently in the distributed system.

Hadoop [5] implements MapReduce as an open-source project. Hadoop version 1 (HadoopV1) uses a slot-based design. Each working node (called task tracker) in HadoopV1 executes map tasks and reduce tasks in map slots and reduce slots, respectively. The numbers of the map slots and reduce slots are configured statically before system startup and cannot be changed at runtime. Map tasks and reduce tasks are assigned to the free slots accordingly. As the workload of many MapReduce jobs can vary greatly during runtime, the simple design and inflexibility of static working slots cannot adapt to the dynamic working behaviour, resulting easily in underutilisation of available resources or depletion of some resources.

Hadoop evolved to version 2 which is known as YARN [6]. YARN is container-based and treats the resources in the cluster as a combination of memories and CPU cores. The users of YARN can configure the memories and CPU cores of the containers which are used to run the tasks of the job. Thus, YARN offers a more precise control of the resources of the cluster than HadoopV1. However, to determine the suitable amount of resources to assign to the containers is often largely a guesswork by the users. If too little resources is assigned, some tasks might fail due to the lack of memories at runtime; if too much is assigned, a few containers would fill a working node, and the allocated resources could end up poorly utilized. In practice, users tend to configure the container resources lavishly, and thus underutilization of rosources is a common phenomenon, which means that optimal throughput is hardly achieved.

HadoopV1 and YARN use different methods to allocate the resources to the jobs. But the resource configuration problem in either version can be seen as the same fundamental problem: to find out the *proper* number of concurrent tasks (or working slots) to run on a working node in order to achieve higher job throughput. The challenge is that this number cannot be decided statically in a dynamic environment. Figuring out this proper number of concurrent tasks is not easy as the jobs can be very different in terms of their compute logic and the data I/O size.

Another problem with MapReduce is that there is a synchronised barrier between the map phase and the reduce phase. The outputs of all map tasks need to be stored locally in some intermediate files and the reduce tasks must shuffle all the map outputs of a particular partition of the problem to the working nodes before these nodes can start reducing the data. This barrier prevents the reduce function from executing in parallel with the map function of the same job which can increase the total execution time of the job if the map outputs are plenty. YARN sets a higher scheduling priority for map tasks than reduce tasks to make sure the maps tasks can obtain more resources than the reduce tasks. But the optimal scheduling policy for MapReduce jobs to achieve a higher job throughput has not been addressed.

In this paper, we would not try to break this barrier between the map phase and reduce phase. Breaking this barrier either introduces another barrier or requires extra resources to increase logical parallelism, but optimal throughput cannot be guaranteed. Instead, we study the behaviour and performance of MapReduce under different resource configurations; we model the job execution time in terms of the resource capacity and resource allocation, and propose a working slot allocation method which can help the system to achieve the maximum parallelism on both sides of the barrier. We develop SMapReduce, which can dynamically manage the working slots for map and reduce tasks at runtime to achieve the maximum utilisation of the CPU and network bandwidth for MapReduce jobs.

This paper is organised as follows. Section II introduces Hadoop and the thrashing phenomenon in multithreading systems, and gives the motivation for managing the working slots at runtime. Sections III and IV describe the design and implementation of the SMapReduce system. Section V evaluates SMapReduce using various benchmarks. Related works on improving the performance of MapReduce are discussed in Section VI. We highlight some possible future work and conclude the paper in Section VII.

II. Background and Motivation

In this section, we introduce the architecture and working mechanism of Hadoop. Understanding how MapReduce works, we then discuss our motivation for developing SMapReduce, a working MapReduce system with dynamic slot management. We also discuss the thrashing phenomenon in multithreading systems, which could affect the runtime behaviour of MapReduce and is used as a guide in the design of the slot management policy.

A. Hadoop

1) Overview of MapReduce: MapReduce is one of the major components in Hadoop. MapReduce divides any given computation into two primitives: map and reduce. The map primitive organises a list of values by their keys and the reduce primitive aggregates the list of values to a single value for each key.

In HadoopV1, the structure of MapReduce consists of two main types of components: a job tracker (also called the master) and several task trackers (also called the worker nodes). The job tracker is the master computer node that manages the running of the whole system. It maintains the runtime information of the whole system, assigns the map and reduce tasks to the task trackers (based on the heartbeat mechanism between the task tracker and the job tracker), and coordinates the running of the MapReduce jobs. The task trackers are the computer nodes that actually work on the tasks. The task trackers run the map and reduce tasks in map and reduce working slots accordingly.

In YARN, the structure is like that of HadoopV1, except that there is not a single central job tracker. The role of managing the processing of a job is taken up by the application master, which is started on a random node when a job is submitted for running. A single resource manager manages the resources of memories and CPU cores of the cluster and is in charge of allocating these resources to the jobs. The role of the task trackers in HadoopV1 is replaced by that of the node managers. Node managers run the tasks in the resource containers. For uniformity, in the following discussions, we use the *slot* to denote the slot in HadoopV1 and the container in YARN, which runs a task in a working node.

By default, when a map task is assigned to the task tracker by the job tracker, the task tracker will find a vacant map slot to launch a new thread that runs a Java Virtual Machine (JVM); the map task will run on this JVM. The same for the reduce task. The map task can be divided into three (sub-)phases: the map phase, the sort and spill phase, and plus optionally the combine phase. All the phases do mainly computing, for which the allocated CPU time is the decisive factor of their performance. The reduce task consists of three (sub-)phases: the shuffle phase, the sort phase and the reduce phase. The shuffle phase reads a partition of map outputs of all the map tasks from all the task trackers. The network bandwidth of the cluster is the decisive factor of the performance of this phase. The running of the shuffle phase can overlap with the running map tasks, but it will not end (and hence the remaining two phases cannot start) until all the map tasks finish. This is the synchronisation barrier between the map tasks and the reduce phase. With a limited number of map slots and reduce slots in the cluster, and given many map tasks and reduce tasks for the job, there will be multiple waves of map tasks and reduce tasks. For the first wave of the reduce tasks, their shuffle phase can overlap with the running of all waves of the map tasks, but it can only proceed to the remaining phases of the reduce tasks after all the map tasks finish.

2) Discussion on the Working Slots: We focus on the duration during which the map tasks and the shuffle phases of the first wave of reduce tasks are running in parallel. We refer to this as the *"overlapped section"*. We want to maximise the utilisation of the required resources for the execution of the tasks in spite of the existence of the synchronisation barrier. The required resources are the CPU time slices (for the mapping) and the network bandwidth (for the shuffling). If we can coordinate the progresses of these two types of tasks, thus making a full utilisation of the CPU and network resources simultaneously, we can achieve high performance when running MapReduce jobs.

For map-heavy jobs, the volume of output data needed to be shuffled is small comparing to the volume of input data the map tasks need to handle. In this case, when there are still map tasks working (i.e., before the barrier is reached), the reduce slots will spend most of their time waiting for mapped output data to shuffle. We could allocate more CPU time slices to the map tasks, that is, more map slots, so that the system can finish the map tasks, cross the synchronisation barrier faster, and go on to the reduce tasks. For reduce-heavy jobs, the data needed to be shuffle are large. The shuffle phases require both CPU time and network bandwidth to sort and transfer the data for reducing. If there are too many map slots, besides the overhead of multithreading scheduling, the shuffle phases may not be able to finish transferring most of the data needed for the corresponding reduce phases, and will have to spend additional time to shuffle data after the barrier. After the synchronisation barrier, when the shuffle phases are still running, there are no more map tasks, the reduce phase is waiting, and the system will spend most of the time transferring data through the network, while the CPU is being left idle.

If the Hadoop system can realise the existence of the map and reduce synchronisation barrier and allocate the proper numbers of working slots for the map tasks and the reduce tasks dynamically at runtime, the system could make full utilisation of the CPU and network resources available and achieve higher performance. This is the motivation behind SMapReduce, an enhanced MapReduce with dynamic slot management.

B. *Thrashing*

Thrashing [7] happens when there are too many threads in the system, and the virtual memory system is in a constant state of paging between the disk and the memory; this causes the CPU utilisation to drop drastically. A similar phenomenon regarding CPU throughput can happen in the MapReduce system. When the number of working slots for map tasks and reduce tasks increases, CPU throughput increases; but when the number of working slots reaches a thrashing point, the throughput of the working tasks begins to decrease.

Generally, if we allocate more working slots to the map tasks or reduce tasks, the throughput of the tasks will increase. The increase of the number of working slots, however, is accompanied by a corresponding increase in the thread scheduling overhead, which creates a threshold in the number of working slots. When the number reaches this threshold, the scheduling overhead starts to outweigh the benefit gained, and the throughput will begin to decrease with the rising number. Fig. 1 shows the thrashing phenomenon in HadoopV1 and YARN with three of the benchmarks we used in our workbench (to be introduced in Section V). We can see from the figure that the map task throughput is sensitive to the number of configured map slots. Finding a proper number of slots for the task is meaningful in order to increase the throughput of the system. In all of the benchmarks, the throughput increases proportionally to the number of map slots. When the number of map slots reaches the thrashing point, the throughput stops increasing or begins to fall. The thrashing points of different job types can be different. In general, map-heavy jobs have a higher thrashing point than reduce-heavy jobs. This is because reduce-heavy jobs spend more resources on shuffling and reducing than map-heavy jobs and suffer an early map thrashing point.

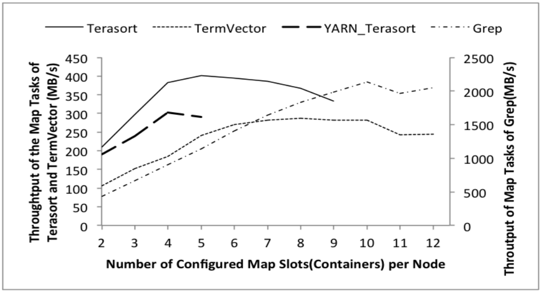

Figure 1. Thrashing: In the Terasort, TermVector, and Grep benchmarks, the curves of the throughput of the map slots versus the number of map slots in each node begins to fall when the number of map slots reaches the thrashing point.

The thrashing point will guide us to decide on the number of working slots to allocate to the map tasks or reduce tasks when we manage the working slots in SMapReduce at runtime.

III. Design of SMapReduce

In this section, we present the architecture of SMapReduce and the design of the working slot management algorithm, and discuss several important issues that affect the design.

A. *Overview of SMapReduce*

We design and implement SMapReduce based on the existing framework of the slot-based design of HadoopV1 by adding a working slot managing mechanism. Fig. 2 shows the architecture of SMapReduce. The blocks in white colour are components that are the same as in MapReduce, the blocks in gradient grey are components that are modified from HadoopV1, and the blocks in dark grey are components that are new in SMapReduce.

The job tracker of SMapReduce consists of three main components: the task scheduler, the heartbeat handler, and the slot manager. The task scheduler decides which task should be run on which task tracker. It is the same as that in MapReduce. The heartbeat handler collects heartbeat

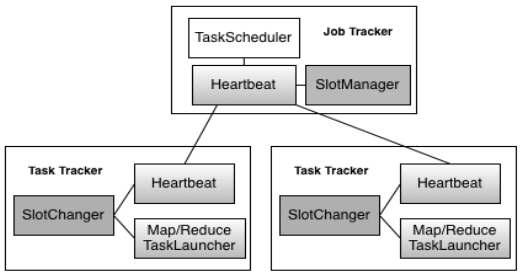

Figure 2. The Architecture of SMapReduce

information from the task trackers and responds with the commands for the task trackers to execute. The commands from the job tracker to the task trackers include the tasks to run or clean up. In SMapReduce, job tracker also sends commands of changing the working slot number to the task trackers by heartbeats. The slot manager is a thread that makes decisions on how many slots should be in the task tracker in order to optimise the resource utilisation.

The task tracker of SMapReduce is made up of these components: the heartbeat sender, the slot changer and the map task launcher and reduce task launcher. In addition to the task tracker's status and the running status of tasks, the task trackers also supply statistics of the running tasks to the job tracker by heartbeats. These task statistics help the slot manager in the job tracker to determine the types of the running jobs and the numbers of slots needed for both map tasks and reduce tasks. The slot changer changes the number of working slots in the task tracker. the task launchers launch map tasks and reduce tasks on the available working slots.

B. Slot Manager

The slot manager is the critical component in managing the working slots in SMapReduce. The slot manager makes use the runtime information of the MapReduce jobs to determine the proper map slots and reduce slots for the jobs at that moment.

1) Balancing between Map and Shuffle Throughput: One important aim of SMapReduce is to properly allocate the working slots for the map tasks and reduce tasks in the overlapped section, so that the jobs can progress to the barrier faster and thus have a shorter execution time.

Based on the workload, the total progress of a MapReduce job can be divided into two parts: the front stretch is from the start of the job to the end of execution of the shuffle phases of the first wave, and the tail stretch is from this point to the end of execution of the job. A job is so divided because in the front stretch, the map phases and the shuffling phases of the same job are running in parallel, while in the tail stretch, only reduce tasks are running.

For the front stretch, SMapReduce tries to balance the allocation of working slots for the map tasks and reduce tasks so that the execution time of this part is minimised. For map-heavy jobs, the map output data are small and it should be easy for the shuffle rate to match the map output rate. If they do match, when the map tasks finish, the shuffle phase is also more or less finished. In this case, the time needed to finish the work of the front stretch of the progress is:

$$t = \frac{M}{T_m},$$

where M is the workload of the map tasks and T_m is the throughput of the map tasks. When the shuffle rate can match the map output rate, since the map workload M is a constant, to achieve minimum execution time of the front stretch, we only need to maximise the throughput of the map tasks in this part. As the map throughput increases, the map output rate also increases. It may happen that the map output rate increases to a point where the shuffle rate cannot catch up. This in fact would be the situation of reduce-heavy jobs.

For reduce-heavy jobs, the data needed to be shuffled in the front stretch is heavy. It is possible that the shuffle rate cannot catch up with the map output rate—i.e., the shuffle phases still have a lot of data to transfer after all map tasks have finished. In the case when shuffle rate cannot match the map output rate, the time needed to finish the work of the former progress part is:

$$t = \frac{M}{T_m} + \frac{R - \frac{M}{T_m} \times T_{r1}}{T_{r2}},$$

where M is the workload of the map tasks, T_m is the throughput of map tasks, R is the workload of the first wave of the shuffle phases, T_{r1} is the shuffle throughput of the shuffle phases when the map tasks are still running, and T_{r2} is the shuffle throughput after the map tasks finish. Note that T_{r2} is a constant in the system since the system only needs to run reduce tasks and there will not be any resource sharing between the map tasks and the reduce tasks. It can be assumed that total throughput of the system is a constant, no matter how the resources are divided between the map tasks and the reduce tasks, and thus we have $T = T_m + T_{r1}$, where T is the total throughput of the system. The above equation can be simplified to:

$$t = \frac{R + M}{T_{r2}} - \frac{(T - T_{r2}) \times M}{T_m \times T_{r2}}.$$

Since all the variables except T_m are constant, if we want to reduce the time t, the map task throughput T_m should decrease. This makes sense. When map tasks and shuffle phases work together to the full extent, the system is making full use of the available resources working for the job. If the map tasks progress too quickly and finish before the shuffle phases finish, the system will only use part of the resources for the shuffle phases afterwards.

When the map throughput is reduced to shorten the execution in the reduce-heavy case, the map output rate also falls. It can come to the situation where the shuffle rate can match the map output rate again. The state when the shuffle rate can just catch up with the map output rate is called the *Balanced State*. When the system is in the balanced state, the running map tasks and shuffle phases are making full utilisation of the resources of the system, and can achieve the minimum execution time of the front stretch of the progress.

As discussed in Section II, the throughput of the map tasks can be controlled by the number of map slots in the system, as long as the map tasks have not reached the thrashing point.

2) Detecting the Thrashing Point: Increasing the slot number does not always lead to increase in the throughput of the tasks. The slot manager needs to detect the occurrence of thrashing when it tries to control the throughput of the map tasks or reduce tasks by adjusting the number of working slots.

Detecting the thrashing point is easy. Take the map tasks for instance. For every number of map slots, the slot manager records the average processing rate of all the map tasks in the system for that slot number. If the number of map slots increases, the slot manager calculates the average processing rate of the map tasks running on the new number of map slots and compares it with the previous average processing rate. If the current rate is smaller than the previous one, the slot manager knows that the system has reached a thrashing point and will stop increasing the number of map slots.

3) Switching Map Slots to Reduce Slots: When the front stretch comes to the end, the number of unfinished map tasks decreases and fewer map slots are needed to run the map tasks. In the tail stretch, there are only reduce tasks running. Under this circumstance, the slot manager can reduce the number of map slots and increase the number of reduce slots appropriately to accelerate the execution the remaining unfinished reduce tasks.

However, we will only increase the reduce slots in the tail stretch when the job shuffle size is small. When the job shuffle size is large, increasing the reduce slots will add a large number of threads copying the map outputs, which can jam the network and thus reduce the shuffle rate on the contrary.

C. Messages in Heartbeat

The slot manager requires statistics of the running job, such as the shuffle rate, in order to balance the map and reduce throughput, to detect the occurrence of thrashing, and to judge the progress of the job. These statistics need to be collected from the task trackers from time to time. We inherits the heartbeat mechanism between the job tracker and the task trackers of MapReduce to support the running of the slot manager.

In addition to the original heartbeats of MapReduce, the task trackers of SMapReduce adds the following information to each heartbeat message: the map input processing rate, the shuffle rate and the map output rate. The slot manager can aggregate these data from all the task trackers.

In the job tracker's end, the heartbeat handler detects whether the number of map slots or reduce slots of a task tracker is different from the number decided by the slot manager. If it is, the heartbeat handler will send a command to ask the task tracker to update its number of working slots as the heartbeat response. After receiving the command to update the working slot number, the task tracker passes it to the slot changer to handle the changing of the number of working slots.

D. Lazy Slot Changing in Slot Changer

The slot changer of the task tracker has two roles: one of changing the number of map slots and the other of changing the number of reduce slots. The slot changer does not change the number of the slots directly. Instead, it sends a signal to the task launcher indicating that the number of working slots has changed. The task launcher will then change the working slot number, but lazily.

There is a map task launcher and a reduce task launcher in the task tracker. They launch map tasks and reduce tasks using the available working slots, respectively. When the task launcher receives a signal from the slot changer, the signal can ask the task launcher either to increase or to decrease the number of the working slots. When it is the increase signal, the task launcher should be able to add some working slots which are ready for launching tasks. When it is the decrease signal, the task launcher may be in a state that all the slots are working on the tasks, which means that there is no free slot at that moment; if the task launcher shuts down one slot immediately, the running task, which is in the middle of its progress, must be terminated and rescheduled in another free slot later. This rescheduling overhead should be avoided because it wastes the resources that have already been allocated to that task. If the slot changing action is frequent, the rescheduling cost can be substantial.

The task launcher therefore applies the lazy policy in changing the number of the working slots. When it needs to reduce the number of the slots and there are not enough idle slots to shut down, it will shut down the idle slots first and remember the number of slots that still need to be shut down when there are idle slots later. When the busy slots finish running the tasks and become idle, the task launcher will know that it is safe to shut down some idle slots if the total number of slots is greater than expected. In the case that the task launcher wants to increment the number of slots, it will be safe for the task launcher to add the slots immediately.

IV. Implementation of SMapReduce

This section will present some of the implementation details of SMapReduce. We implement SMapReduce based on the source code of Hadoop, version 1.0.4, a recent stable version. We have modified mainly the JobTracker class, the TaskTracker class and the MapTask class.

A. *Implementation of JobTracker*

In the JobTracker class, much of the work is spent on implementing the slot manager. The slot manager is a thread in the job tracker. The slot manager needs to detect whether the system is in a thrashing state and decides on the proper number of the slots periodically. After every time period, the slot manager is almost certain that all the task trackers have updated their statuses in the job trackers since the last period, and thus can make a more accurate judgement on the state of the system.

There are some issues we need to deal with when detecting the thrashing phenomenon and deciding on the proper number of working slots for the MapReduce jobs.

1) Slow Start: At the beginning of the execution of the job, the data reported from the task trackers may not be substantive enough for the slot manager to base on to make a decision. Such data can lead to wrong decisions, and thus impact the performance. For example, soon after the job has started and yet no map tasks have finished, the shuffle rate of the system is zero while the map output rate is non-zero. This can lead to a wrong conclusion that the shuffle rate cannot match the map output rate, and the job is suspected to be reduce-heavy.

The slot manager adopts the “slow start” approach to avoid this potential problem. The slot manager will only start working after a certain portion of the map tasks have finished executing and reported their running statistics to the job tracker. In SMapReduce, the value of the start threshold is 10% by default. The slot manager will start working after 10% of the total map tasks have finished.

2) Suspected Thrashing: As discussed in Section III, to detect the occurrence of map thrashing, the slot manager compares the average map processing rates when the number of map slots increases. The map processing rate is measured by the input rate of the map tasks. However, immediately after responding to the slot change command, the map processing rate of the task trackers will drop slightly at first. So it is not advisable that the map processing rate just after a slot change be used for comparison, which will almost always give the result of the occurrence of thrashing. According to our observation, the map processing rate after a slot change will grow gradually to a stable range after some time. Only then should the slot manager begin to consider whether the system has indeed entered map thrashing.

Because of the nondeterministic nature of distributed systems, the slot manager still cannot conclude that the system is approaching the thrashing state once the map processing rate has grown to the stable range and is smaller than what was before incrementing the map slot number. Instead, under this circumstance, the slot manager will mark the current state as suspected of thrashing, and give the system another chance. If the system is detected to be suspected of thrashing continuously, the slot manager can then announce with confidence that the system is in a thrashing state.

The slot manager does not need to consider any reduce thrashing problem, because the number of reduce slots in the system is usually set to a small number, which is not likely to cause thrashing. The number of reduce slots is set low because one reduce slot can generate several threads, to copy the map output data from all the other nodes in the cluster during the shuffle phase. A large number of reduce slots can cause network jam in the cluster.

3) Deciding the Slot Numbers: Initially, the slot manager has a specific number of map slots and reduce slots as configured by the user, just like in HadoopV1. It gradually adjusts the number of slots based on the running status of the jobs. To decide the proper number of map slots or reduce slots for the job, the slot manager considers the two progress stretches as discussed in Section III.

In the front stretch of the progress, when many map tasks are running in parallel with the first wave of shuffle phases, the slot manager needs to find out whether the shuffle rate can catch up with the map output rate of a partition of the reduce data. The map output rate of a partition of reduce data is the data needed to be shuffled in the first wave of shuffle phases. It is estimated under the assumption that every shuffle phase needs to transfer the same amount of map output data. Therefore, we have

$$R_m = \frac{n}{N} \times R_t,$$

where R_m is the map output rate of a partition of reduce data, n is the number of the running reduce tasks, N is the number of total reduce tasks, and R_t is total map output rate. The balance level of the shuffle phases and the map tasks can be indicated by:

$$f = \frac{R_s}{R_m},$$

where R_s is the shuffle rate. When the system is not in a thrashing state, if the factor f is greater than an upper bound, we say that the shuffle rate can catch up with the map output rate, and this is the map-heavy case. The slot manager will then increment the map slots by 1. If the factor f is smaller than a lower bound, we say that the shuffle rate cannot match the map output rate, and it is the reduce-heavy case. In this case, the slot manager will decrement the map slots. When the factor f is somewhere between the upper bound and the lower bound, the system is considered in the balanced state between shuffle and map, and will do nothing.

At the end of the front stretch and in the tail stretch, when there are only a few or no map tasks, the map slots are reduced and reduce slots can be increased.

B. *Implementation of TaskTracker and MapTask*

The main modification to the TaskTracker class is to add the lazy policy of changing the slot number in the task launcher without affecting the normal execution of the tasks. The lazy policy is implemented by the consideration of the expected slot number the task tracker is going to have. Whenever a busy slot is released, the task launcher checks whether the released slot should be shut down to meet the expected slot number. Whenever a task wants to engage a slot, it makes sure the free slot is not the one that should be shut down because there are already as many slots running as expected, and updates the number of working slots.

In the MapTask class, we added some extra statistics needed by the slot manager. For example, the class records the map output data size of the map task upon completion so that the slot manager can use it to calculate the map output rate of the system.

V. Evaluation

The evaluation workbench is configured as follows: The computer cluster consists of 18 nodes, each of which is configured with 4 quad-core 2.53GHz Intel CPUs and 32GB DDR3 memory, running the CentOS 2.6 operating system. One node serves as the name node of HDFS [8], and one as the job tracker of SMapReduce; the remaining 16 nodes are the data nodes of HDFS as well as task trackers of SMapReduce. The nodes are connected by switches with 16 GbE ports. We evaluate SMapReduce in a physical cluster instead of in the virtual environment like that of Amazon in order to minimise the affect of the virtual client machine scheduling and virtual network. To reduce the impact of the TCP incast problem in the network, the TCP minimum retransmission timeout (RTO_{min}) is modified from 200ms to 1ms [9].

We evaluate the performance of SMapReduce and compare it with HadoopV1 and YARN using the benchmarks from "Purdue MapReduce Benchmarks Suite" (PUMA) [10]. This benchmark suite includes various practical jobs and input data from real life. As the recommended reduce task number is 99% of the number of reduce slots in the cluster, and the default reduce slot number in the 16 task trackers is 2, the reduce task number is set to 30 in all the benchmarks. The block size of HDFS is set to 128MB.

We evaluate the performance of SMapReduce in terms of the execution times of different benchmarks, progress speed, the performance with different data input sizes, the performance under different resource configurations, the effects of the detection of thrashing and the slow start policy, and multiple concurrent job workloads. All the experiment results are the average values of the data collected from two trials.

A. *Performance in Various Benchmarks*

To have an overall understanding of the performance of SMapReduce, We run SMapReduce on these benchmarks and compare their execution times in each phase with HadoopV1 and YARN. The configuration of all the jobs in

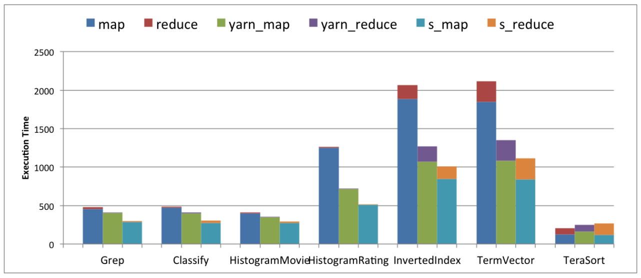

Figure 3. The Execution Time of Each Benchmark in HadoopV1, YARN and SMapReduce

HadoopV1 and SMapReduce is: 3 initial map slots and 2 initial reduce slots in each task tracker. Equivalently, YARN is configured to able to run 3 map containers and 2 reduce containers concurrently. Fig. 3 shows the experiment results. The map time stands for the execution time when map tasks run with the shuffle phases in parallel. The reduce time stands for the execution time after the barrier when only reduce tasks are running.

From the figure, SMapReduce has shorter map time, shorter reduce time and higher job throughput than both HadoopV1 and YARN in almost all benchmarks. Generally, map-heavy jobs and medium reduce-heavy jobs have even higher total performance increase than reduce-heavy jobs. This is because map-heavy jobs usually have a higher thrashing point, and any misconfiguration of map slots and reduce slots leaves plenty of space for optimisation. For instance, in the HistogramRating benchmark, In terms of throughput, SMapReduce has 140% and 72% performance increase than HadoopV1 and YARN, respectively. Terasort is the only exception here, where SMapReduce execution time is slightly longer than that of HadoopV1 and YARN. This is because the current slot configuration happens to be optimal for this job, and the management of slots in SMapReduce adds a small overhead that has affected the job throughput. But the overhead is so small that it should be negligible.

The total performance increase is mainly contributed to by the map throughput increase. Because of the policy of increasing the reduce slots appropriately when map tasks are few or all finished, we can have reduce throughput increase in many benchmarks, such as in InvertedIndex. This policy failed to increase the reduce throughput in some benchmarks, but the effect is very small.

B. *Progress Speed*

To undertand further the effect of slot management on the performance of MapReduce jobs, we record the progress percentage of the finished part of the jobs throughout their execution. The total progress percentage of the job is 200%, 100% for the map tasks and 100% for the reduce tasks. Fig. 4 shows the progress percentages over time of the HistogramMovie benchmark running on MapReduce and

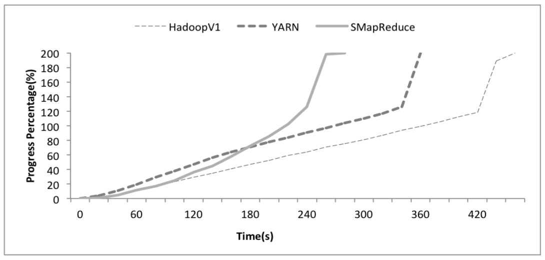

Figure 4. The Progress Percentage along the Time of the HistogramMovie Benchmark on HadoopV1, YARN and SMapReduce

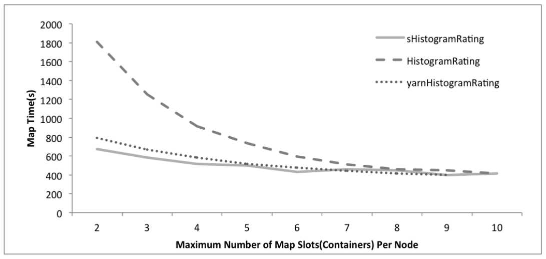

Figure 5. Map Time of the HistogramRating Benchmark under Different Map Slot Configurations

SMapReduce, respectively. At the beginning, SMapReduce progresses at approximately the same speed as HadoopV1 and YARN. However, as time goes by, the slot manager of SMapReduce dynamically adjusts the number of slots for the map tasks so that progress of the job is sped up. As the configuration of slots is getting close to the optimal, the speedup rate increases over time. Without slot management, the job progresses at a constant speed in HadoopV1 and YARN. Note the sharp turns of all the curves at the point slightly above the 100% mark, which is when all map tasks finish.

C. Different Resource Configurations

In the above evaluations, we configure the number of map slots to 3 and the number of reduce slots to 2. It is possible that HadoopV1 and YARN performs poorer than SMapReduce simply because of the mis-configuration of the number of slots (or container memories in YARN). We evaluate SMapReduce under different configurations of the number of map slots (or containers) and find that SMapReduce still has a shorter map time in most cases. Fig. 5 shows the map times of HadoopV1, YARN and SMapReduce under different map slot configurations for the HistogramRating benchmark.

In the HistogramRating benchmark, when the map slot number is between 2 to 6, the map throughput of SMapReduce is 10%-18% higher than that of YARN and 30%-160% higher than that of HadoopV1. For the configuration where HadoopV1 and YARN can achieve the minimum map time—i.e., it happens to be the optimal map slot

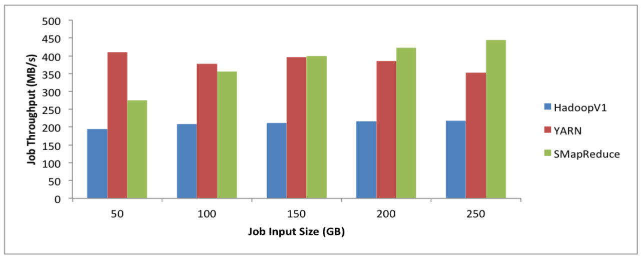

Figure 6. HistogramRating Job Throughput of HadoopV1, YARN and SMapReduce with Different Input Data Sizes

configuration—SMapReduce can still achieve the same map performance as MapReduce.

We did not measure the performance of SMapReduce vs. HadoopV1&YARN under different reduce slot configurations because the number of reduce slots on each task tracker is usually set to a small number (e.g., 2) to avoid too many concurrent reduce tasks, which can jam the network.

D. Different Input Sizes

We also evaluate the scalability of SMapReduce. We measure the job throughputs of SMapReduce with input data of different sizes with the HistogramRating benchmark. Fig. 6 shows the result. As the input size increases, the job throughput of SMapReduce increases, while those of HadoopV1 and YARN remain almost unchanged. It is because when the input size is large, SMapReduce has more time to adjust the slots to the optimal configuration. When the input size grows to 250GB, the job throughput of SMapReduce is about 1.3 times that of YARN and 2.0 times that of HadoopV1. One conclusion we can make is that the larger the input data size, the more benefits we can get from managing the slots at runtime.

E. Detecting Thrashing and Slow Start

Detecting the occurrence of thrashing and the slow start policy are important considerations in our design of the slot managing algorithm. We run experiments to see the necessity and effects of detecting thrashing and the slow start policy. Fig. 7 shows the results of the experiment of comparing the map times of two benchmarks with and without detecting thrashing, and with and without slow start. In both benchmarks, without detecting thrashing, the map time of SMapReduce is much longer than that of HadoopV1 and YARN. The mechanism of detecting thrashing can greatly improve the performance of SMapReduce. Without the slow start policy, the map time of the SMapReduce may be either larger or smaller than that of HadoopV1 and YARN, depending on whether the slot manager has made a right decision at the beginning when little runtime information is available. Generally, SMapReduce applying the slow start policy runs the map tasks faster than when without the policy.

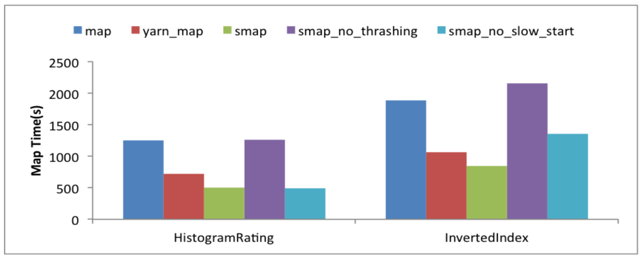

Figure 7. Map Time with and without Detecting Thrashing and Slow Start Policy

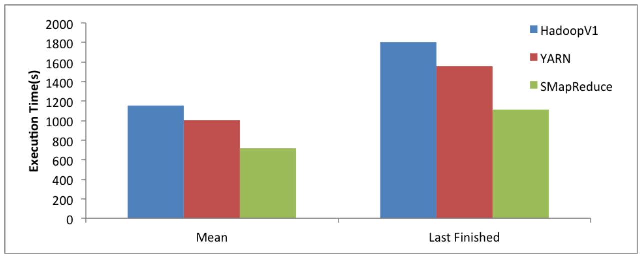

Figure 8. Mean and Last Finished Execution Time of Multiple Concurrent Job Workload of Grep Jobs

F. Multiple Concurrent Jobs

Hadoop is a shared environment, and in most use cases, multiple concurrent jobs run in the system. We evaluate SMapReduce with synthetic multiple job workloads. In the multiple job workloads, we submit 4 jobs of the same benchmark in total to the system, and each job is submitted 5 seconds after the previous job. In SMapReduce and HadoopV1, we use the FIFO scheduler for multiple jobs. In YARN, we use the capacity scheduler. Both of them are the default schedulers respectively. The capacity scheduler is similar to the FIFO scheduler, which tries to schedule containers for early submitted jobs first. But the capacity scheduler further considers the map tasks having a higher scheduling priority than the reduce tasks. We compare the mean execution time and the execution time when the last job finishes. Fig. 8 and Fig. 9 show the performance of the three systems running multiple Grep jobs and multiple InvertedIndex jobs, respectively. SMapReduce has a shorter mean execution and total execution time than HadoopV1 and YARN in both cases. In the Grep workload, for instance, the mean time and the time needed for the last job to finish in SMapReduce are both only about 60% of that in HadoopV1, and only about 70% of that in YARN. SMapReduce clearly outperforms them for multiple concurrent job workloads.

VI. Related Work

There are many existing works exploring different ways to improve the performance of MapReduce.

YARN [6], the developed version of Hadoop, acts as a resource manager of the cluster. Comparing to the static slot-based implementation of HadoopV1, tasks run within the

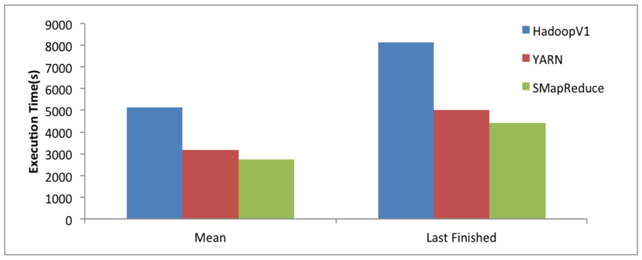

Figure 9. Mean and Last Finished Execution Time of Multiple Concurrent Job Workload of InvertedIndex Jobs

resource containers and make better utilisation of the cluster resource. However, different configurations of resources for each container can yield performance that varies greatly and the best resource configuration is generally unpredictable. YARN also assigns map tasks at a higher priority than the reduce tasks to prevent too many reduce tasks blocking the process of the map tasks. But unlike the dynamic allocation decision in SMapReduce, the priority design is simple and does not consider the runtime performance to determine an optimal allocation policy in order to gain a higher job throughput.

Many researchers also tried to optimise the task scheduling policy for different application scenarios. They designed schedulers to serve different goals, including finishing the jobs before the deadlines [11] [12], maintaining the fairness in the shared environment [13], and allowing priority in jobs [14]. Some improved the scheduler for a specific use case. HaLoop [15] makes the task scheduler aware of the input data locality to support fast multiple-loop iterative MapReduce jobs. And some developed scheduling algorithms in specific hardware environments, such as the hybrid computing environment [16] and the heterogeneous cluster environment [17]. Instead, SMapReduce aims at achieving the optimal job throughput in the homogeneous cluster environment. It tries to reach this design goal by considering how many slots should be allocated for the tasks dynamically, so that the resources in the system are fully utilised.

Breaking the synchronisation barriers between the map, shuffle, sort, and reduce phases can improve the potential parallelism level of the system [18] [19]. Wang et al. [20] proposed a total ordering method when copying outputs of the map for reducing. Total ordering enables parallel execution of shuffle, merge and reduce phases, but requires an extra synchronisation between map and shuffle. HPMR [21] introduces prefetching and pre-shuffling into MapReduce in the shared environment. However, these solutions only maximise the logical parallelism of different phases, but do not consider the full utilisation of available physical resources in the system, and thus are unlikely to have an optimal execution throughput of the jobs. SMapReduce, on the other hand, can balance the resources even with the existence of the synchronisation barrier and is optimal in terms of the job throughput.

VII. Future Work and Conclusion

Currently, SMapReduce only considers the case where the cluster is homogeneous and the data are random in distribution. We are working to extend SMapReduce to the heterogeneous environment, which may be a common setting in some small clusters.

In this paper, we study how the number of concurrent tasks can affect the performance of MapReduce and build a mathematics model to working out a way to properly allocate the map slots and reduce slots so as to make full utilisation of the resources in the system to achieve the minimum execution time of a MapReduce job. We implement SMapReduce, which can dynamically manage the working slots for different types of job at runtime to gain a higher job throughput. Evaluation results show that SMapReduce has remarkable performance improvement comparing to both HadoopV1 and YARN.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science &amp; Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[3] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE, 2008, pp. 277–284.

[4] J. Lin and C. Dyer, "Data-intensive text processing with mapreduce," *Synthesis Lectures on Human Language Technologies*, vol. 3, no. 1, pp. 1–177, 2010.

[5] T. White, *Hadoop: the definitive guide*. O'Reilly, 2012.

[6] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[7] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 915–922.

[8] D. Borthakur, "Hdfs architecture guide," *Hadoop Apache Project. http://hadoop. apache. org/common/docs/current/hdfs_design. pdf*, 2008.

[9] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM, 2009, pp. 73–82.

[10] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," 2012.

[11] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010, pp. 373–380.

[12] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 235–244.

[13] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[14] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Job scheduling strategies for parallel processing*. Springer, 2010, pp. 110–131.

[15] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[16] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "Moon: Mapreduce on opportunistic environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 95–106.

[17] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[18] Y. Guo, J. Rao, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," *10th International Conference on Autonomic Computing*, pp. 107–117, 2013.

[19] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online." in *NSDI*, vol. 10, no. 4, 2010, p. 20.

[20] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop acceleration through network levitated merge," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 57.

[21] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–8.