Modeling I/O Interference in Data Intensive Map-Reduce Applications

Sven Groot
Institute of Industrial Science
The University of Tokyo
Tokyo, Japan
Email: sgroot@tkl.iis.u-tokyo.ac.jp

***Abstract*—Map-Reduce is a popular framework for very large-scale data mining and processing. Recently, some works have attempted to model the behavior of Map-Reduce, but these existing models ignore the non-linearity of disk I/O performance under contention, which is a critical aspect of estimating the performance of data intensive applications. Understanding I/O interference between tasks running on the same node is critical in optimizing task scheduling for improved resource utilization. In this paper, we present a model to estimate the I/O behavior of Map-Reduce applications that can be used to achieve these goals.**

***Keywords*-Map-Reduce; cloud computing; data intensive**

I. Introduction

Map-Reduce [1] continues to be a very popular platform for large-scale data processing. Hadoop [2] provides the most popular implementation, and is frequently used in cloud environments.

In order to analyze and improve workload management decisions in Map-Reduce, it is necessary to have a model of the behavior of jobs and tasks. When dealing with data intensive applications I/O costs and I/O contention will be important factors. Modern servers typically have many CPU cores, and Hadoop must run several simultaneous tasks on a single node to exploit these CPUs. These tasks will often experience a bottleneck when competing for the same limited I/O resources such as disks.

Servers typically have more CPU cores than disks, and the behavior of disks is very different from other resources such as CPUs and memory. They are also often the slowest component in the system, with a very large potential impact on the performance of an application.

Map-Reduce considers only data locality when scheduling tasks, but scheduling many I/O intensive tasks on a single node can have adverse affects on performance due to interference. By understanding the effects of I/O interference we can make better decisions about how many tasks to schedule on a node and how to mix tasks from different workloads, leading to better resource utilization. This can then improve execution time or reduce the number of resources necessary for a particular workload.

In this paper, we propose a cost model for Map-Reduce that is able to predict the effect of task scheduling decisions on I/O performance. In the following sections, we will investigate existing work in modeling Map-Reduce, describe our model approach, and evaluate it using a data intensive workload.

II. Related Work

Much of the work in modeling Map-Reduce is still very recent, and typically it is limited in scope depending on what the authors intended to use the model for.

Huai et al [3] propose a model that generalizes Map-Reduce and similar frameworks such as Dryad [4] into a matrix-based representation of the data flow, but which currently does not consider performance. Verma et al [5] propose a model to estimate job completion time based on the observed task completion times measured previously. Jindal et al [6] use a simple model that considers only reading of input data.

The model proposed by Herodotou et al [7], described fully in [8], is to our knowledge the most usable and detailed model currently available. It accurately models the data-flow between tasks and task phases, but its cost model does not distinguish between CPU and I/O costs.

III. Modeling Map-Reduce

Modeling the behavior of Map-Reduce is made difficult by the presence of I/O contention. Figure 1 shows what happens to tasks when the number of *task slots* (the number of tasks a single node can run in parallel) is increased. With only a single task slot, each task is the same length (in this example the tasks all process the same amount of data).

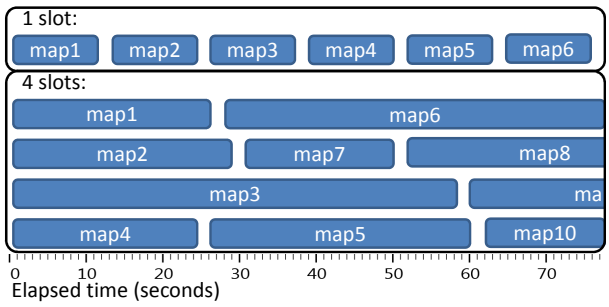
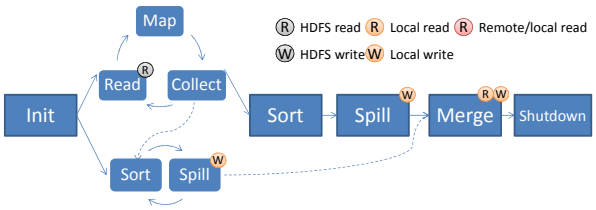
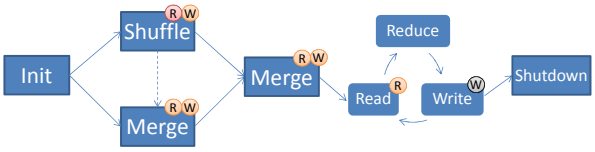


Figure 1. Actual time line of TeraSort map task execution using 1 and 4 map slots on a system with 8 CPUs.



(a) Map task



(b) Reduce task

Figure 2. Phases of map and reduce tasks.

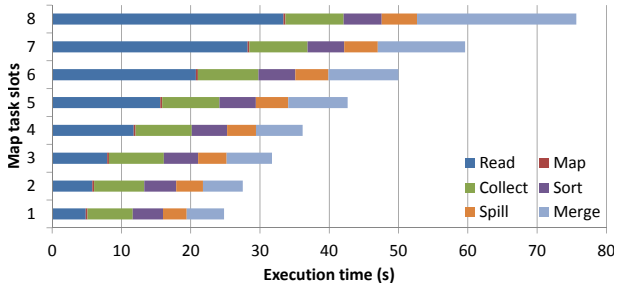


Figure 3. Breakdown of TeraSort map task execution time by phase. The initialization and shutdown phases are omitted for clarity.

When the number of slots is increased to four, the execution time for each task increases due to I/O contention and large variation between the tasks appears because not every task is affected equally by the interference.

Existing models [7][8] that do not consider I/O interference are unable to predict the increase in task time, and would therefore give the impression that any workload scales linearly on a single node. This means it is not possible to make meaningful decisions about the number of slots or task placement based on these models.

Accurately predicting interference is very difficult due to the many variables involved, including the behavior of the file system, page cache, I/O scheduler, and hardware. The large amount of variation between the tasks also makes it as good as impossible to predict individual task times. However, in order to analyze the effects of task scheduling decisions it is sufficient to know the average effect of the interference, which is possible to predict.

A. Map and Reduce Task Structure

In order to reason about I/O interference, we must consider how and when map and reduce tasks perform I/O. Processing of the tasks consists of several distinct phases which all have their own CPU and I/O usage characteristics. These phases are shown in Figure 2, a breakdown which is similar to the one used by Herodotou et al [8].

For each phase, it is indicated whether they perform read or write I/O to the distributed file system, or to local or remote disks.

B. CPU and I/O Cost Model for Phases

Not every phase has the same amount of I/O or does the same amount of CPU processing, which means that they will show different interference effects. Figure 3 shows the breakdown into phases of the average execution time of a map task—from the experiment described in Section IV—as the number of task slots is increased. The execution time of those phases that perform heavy I/O, particularly read and merge, increases significantly due to interference, while those phases that perform no I/O such as collect or sort remain the same. Also note that the spill phase remains relatively static despite the fact that it performs write I/O; this is because write I/O is cached, so most of the costs are incurred not by the spill phase itself but by other phases and later tasks.

Because the phases show different levels of I/O and CPU usage, we must consider the costs of each phase separately. Phases that are interleaved, such as the read/map/collect cycle in map tasks and the read/reduce/write cycle in reduce tasks, are considered as one due to how the CPU and I/O costs of these phases interact.

We split the costs of each phase into three components: CPU cost, read I/O cost, and write I/O cost. These costs are expressed per byte. The CPU cost for each phase is measured for a specific workload, and is currently assumed to be fixed for that workload. We also assume that CPU costs are not subject to interference, which holds as long as the number of task slots does not exceed the number of CPU cores. In practice there may be cache or memory contention, but this is minor compared to I/O contention and is not considered in this model.

The basic I/O costs are taken from the raw read and write throughput of the disks involved, which is sufficient because Map-Reduce performs only sequential I/O. To estimate interference, we first assume that bandwidth is divided evenly among tasks, and additional interference based on disk seek overhead and other factors is expressed by a hardware-dependent function.

Because the OS kernel—as well as the storage system hardware—perform asynchronous I/O and read-ahead caching, the CPU costs of a phase can overlap with the I/O costs. Therefore, we estimate the overall execution time of a phase as the maximum of the CPU or the I/O time: $T_{phase} = ((S_{read} + S_{write}) \cdot C_{cpu}) \vee (N_{slots} \cdot (S_{read} \cdot R_{io} + S_{write} \cdot W_{io})) + f_i$. Here, T_{phase} is the execution time of the phase, S_{read} and S_{write} are the size of the data being read and written in bytes, C_{cpu} is the CPU cost per byte, N_{slots} is the number of map or reduce task slots,

R_{io} and W_{io} are the read and write I/O costs per byte, and f_i is a function estimating the hardware-specific additional interference. This formula is used for all phases; since not all phases perform read or write I/O, some terms may evaluate to zero.

This calculation uses the assumption that when there are N slots, there are always N total instances of the same phase running at the same time. In reality this is not necessarily true due to the variation shown in Figure 1, which means that e.g. the map phase of one task can overlap with the merge phase of another. In addition, write caching means the cost of writes is not necessarily incurred by the phase that does the writing. However, these factors only move interference costs over the lifetime of the job; they don't change the overall interference. This means the formula we use gives a good indication of the average costs of each phase.

C. Intermediate Data Fragmentation

Reduce tasks read data that has been written by map tasks or merge phases of the same reduce task. This data was written by multiple simultaneous writers under heavy system load, which causes high levels of file fragmentation with a severe adverse effect on the I/O performance when reading these files. Because of this, we calculate the read I/O cost for the shuffle and read phases as ${N_{slots}}^F \cdot R_{io}$, were F is the factor by which fragmentation reduces performance. F is calculated by solving ${N_{slots}}^F = N_{slots} \frac{R_{fio}}{R_{io}}$ where N_{slots} is set to the maximum value that will be used and R_{fio} is the I/O cost for reading a fragmented file written using that number of slots. Improving our understanding of the effect of fragmentation is still a work in progress.

D. Task and Overall Execution Time

In order to calculate the average task time, we simply sum the time of all the individual phases except initialization and shutdown. This is the *work time* of the task.

At this point, we must separately consider the shutdown phase. The shutdown phase waits for a background thread which checks whether it needs to exit every three seconds, starting from the end of the initialization phase. To account for this, the time of a task is estimated by rounding up the work time to the nearest multiple of three, and then adding the initialization phase time.

The time that a task occupies a slot may be higher than its execution time, because a new task cannot be scheduled until the next heartbeat. For this reason, the task time is rounded up to the nearest multiple of the heartbeat interval (3 by default). Hadoop allows for sending out-of-band heartbeats when tasks are finished; when this is enabled the task times do not need to be rounded.

Finally, we estimate the overall execution time for the map or reduce stage by multiplying the slot time by the expected number of waves, which is $\frac{N_{tasks}}{N_{slots}}$; we assume the interference for the final wave is proportional to the number of tasks in that wave. This overall prediction can be used as an indication for the scalability of a job.

Reduce tasks may be started while the map tasks of the same job are still running, in which case they will be able to shuffle and merge some of the data in the background. While this can cause additional interference for the map tasks and increased waiting in the reduce tasks, it reduces the time spent in the reducers after the map tasks finish. The overall job time estimated by the averages is therefore still a good indication of performance.

System	
CPU	2x quad-core Xeon E5530 2.4GHz
Memory	24GB
Storage	
Controller	JCS VCRVAX-4F RAID
HBA	QLogic QLE2462 (4Gb/s fiber channel)
Disks	10x Hitachi HDS721010CLA332
Volume	RAID6 (7.2TB)
Read throughput	380MB/s
Write throughput	150MB/s

Table I
EXPERIMENTAL SETUP

IV. EVALUATION

A. Experimental Environment

For our experiments, we used the setup described in Table I. This system uses a RAID array with both DFS blocks and intermediate data stored on the same volume. We have chosen to use the RAID array because it provides more predictable performance for multiple streams and is less dependent on the behavior of the Linux I/O scheduler.

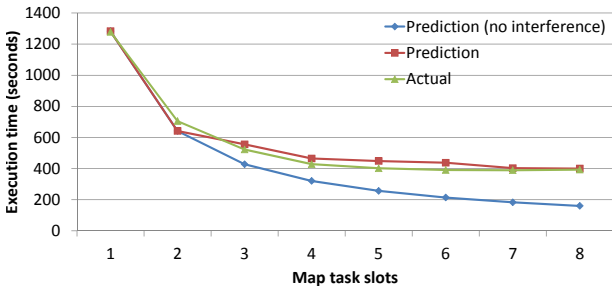
On this system, for reads, we estimate $f_i = 0.1N_{slots}$ based on an observed approximately 100ms read-ahead start-up cost for each stream, and for writes $f_i = 0$. The fragmentation factor was determined to be $F = 1.75$.

We configured a Hadoop cluster with a single slave node, which allowed us to focus on the costs of local I/O. Expanding the model for multiple nodes and non-local I/O is still a work in progress.

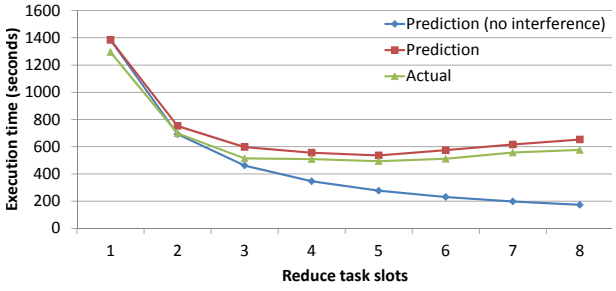
We evaluated the accuracy of our model using the TeraSort benchmark [9], using a 16GB data set with a 256MB split size. There are 32 map tasks and 8 reduce tasks. We measured baseline times using 1 slot, and then varied the number of slots between 1 and 8 to see if the model could correctly predict the effect of the interference. In order to evaluate reduce tasks separately, we set *mapred.reduce.slowstart.completed.maps* to 1.0 so that reduce tasks would not start until the map tasks had all finished. Out-of-band heartbeats were enabled. All other settings were left at their defaults.

B. Results

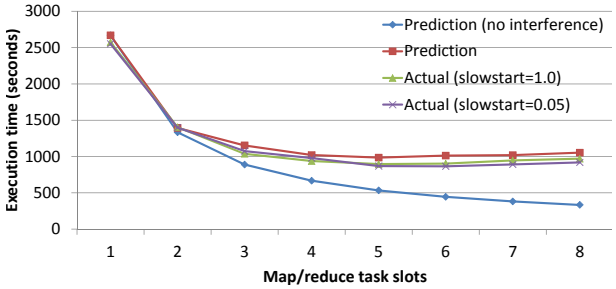
The map stage results are shown in Figure 4(a). We compared the prediction against the actual execution time



(a) Map tasks



(b) Reduce tasks



(c) Job overall

Figure 4. Predicted vs. actual execution times.

and a prediction that does not take interference into account. The prediction closely matches the actual execution time, but shows minor anomalies for the 2, 7 and 8 slots cases; these are caused by a small change getting exaggerated by the rounding. Most importantly, the prediction correctly captures the actual trend allowing us to predict how this workload will scale when per-node parallelism is changed for the map stage. When interference is not considered the prediction is far too optimistic, indicating the importance of considering the effects of I/O interference in the model.

The reduce stage results are shown in Figure 4(b). The interference prediction over-estimates the execution time to a larger degree than for map tasks, particularly for three slots, which is primarily due to caching; some of the intermediate data is still cached when it is read by the shuffle or reduce phases. When the number of slots increases, the amount of data per wave becomes big enough for caching to be less effective.

Figure 4(c) gives the overall time for the entire job with mapred.reduce.slowstart.completed.maps set to either 1.0 or its default value so reduce tasks start before all map tasks have finished. As the figure shows, this does not affect the accuracy of the prediction.

V. Conclusion and Future Work

In this paper, we have shown the complexity of modeling Map-Reduce when I/O interference is included. Existing models make simplifying assumptions regarding the presence (or lack of) I/O contention, which we believe are not realistic for data intensive workloads. CPU and I/O costs show very different scaling behavior when multiple tasks are competing for the same resources, and these costs must therefore be treated separately in the models.

We have introduced an analytical model for estimating I/O interference in Map-Reduce, which is able to predict the performance scalability of a job, which can help with making and analyzing scheduling decisions for a workload.

The model shown here is still a work in progress. We will extend it to deal with non-local I/O in a cluster with multiple nodes, which includes non-local map tasks, shuffling data from remote nodes, and output data replication.

It is our intention that this model can be used to better understand workload management decisions and help optimize resource usage with mixed workloads which are common in multi-tenant cloud systems.

VI. Acknowledgement

This research is partially supported by the project ”R&D on reliable and power-saving network control technologies for cloud computing” of the Ministry of Internal Affairs and Communications.

References

[1] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Apache, “Hadoop Core.” [Online]. Available: http://hadoop.apache.org/core

[3] Y. Huai, R. Lee, S. Zhang, C. H. Xia, and X. Zhang, “DOT: a matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems,” in *Proc. ACM SOCC*, 2011.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.

[5] A. Verma, L. Cherkasova, and R. H. Campbell, “ARIA: automatic resource inference and allocation for mapreduce environments,” in *Proc. ACM ICAC*, 2011.

[6] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, “Trojan data layouts: right shoes for a running elephant,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[7] H. Herodotou, F. Dong, and S. Babu, “No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics,” in *Proc. ACM SOCC*, 2011.

[8] H. Herodotou, “Hadoop performance models,” Duke University, Tech. Rep., 2010. [Online]. Available: http://www.cs.duke.edu/starfish/files/hadoop-models.pdf

[9] “TeraSort.” [Online]. Available: http://sortbenchmark.org