MapReduce Framework Optimization via Performance Modeling

Lijie Xu*
Institute of Software, Chinese Academy of Sciences
Graduate University of Chinese Academy of Sciences
Email: xulijie09@otcaix.iscas.ac.cn

***Abstract*—MapReduce framework has become the state-of-the-art paradigm for large-scale data processing. In our ongoing work, we attempt to solve the three interrelated problems: how to build an accurate MapReduce performance model, how to use it to automatically detect and optimize slow-running MapReduce jobs, and how to use it to help scheduler arrange job execution sequence. Currently, we mainly study the job execution time model and its training method. We also present several policies to optimize the job configuration and scheduler.**

I. INTRODUCTION

MapReduce [1] is a simple but efficient solution towards large-scale data processing and analysis. Apache Hadoop is an open-source implementation of GFS [2] and MapReduce. Hadoop's MapReduce framework consists of a job scheduler (JobTracker) running on the master node and a task manager (TaskTracker) running on each slave node. Each slave node is statically configured with fixed number of map slots and reduce slots according to CPU core number. Hadoop Distributed File System (HDFS) provides user-transparent and fault-tolerant data storage for MapReduce jobs. It breaks the large dataset into small blocks (typically 64MB) and stores them on slave nodes scattered with multiple replicas.

Although MapReduce framework frees the users from the labor of cluster management and job scheduling, its weak job profiling makes jobs' behavior unpredictable. Currently, Hadoop's job scheduler cannot tell the remaining time while jobs are running or the total time cost before running the jobs. But users need prior time estimation to predict how soon they may get the job's result or further decide when to run the job in a busy cluster. Similarly, the job scheduler needs time model to detect the slow-running tasks for speculative execution [3]. However, prior time estimation not only needs to predict time cost (e.g., each task's duration, overlapping time between map stage and reduce stage) but also consider a lot of influence factors (e.g., resource contention, data size, data locality, failures and heterogeneous clusters). **So it is a great challenge to tell the accurate time cost for different jobs with diverse data under different environments.**

Complex resource configurations also burden users on the optimization of job performance and resource utilization. Hadoop has more than 190 configuration parameters and one tenth of them can affect the job performance dramatically [4]. Due to lack of execution time model and automatic configuration tools, users have to detect the slow-running jobs by experience or run jobs under different configurations to find the best parameters. However, the parameter space of configurations is very large and resource utilization has bottleneck to detect. Searching for the right configurations is actually looking for the balance between job throughput and performance. In addition, performance degradation problem (in section 3) aggravates the difficulty to choose optimum parameters. **So it is a complex problem to automatically detect and optimize slow-running jobs' configurations with awareness of resource utilization.**

Based on execution time model, we can further optimize the job scheduler. Current schedulers do not adjust the job execution sequence except the job's priority is set by users. Keeping the original job sequence always incurs waste of resources in reduce tasks' shuffle phase. This problem can be solved by adjusting the job execution sequence, but jobs' makespan may be prolonged. **So it is necessary to design a scheduler assistant to arrange an appropriate job execution sequence considering the trade-off problem of job performance and resource utilization.**

In this PhD forum paper, we firstly introduce the MapReduce timeline and performance-related problems. Secondly, we present a primary execution time model and its fast training method. Thirdly, we provide the detecting and optimization methods for slow-running jobs. Finally, we address the optimization method towards job scheduler.

II. RELATED WORK

Since execution time model is useful for job scheduler to allocate tasks, Zaharia et al. [5] present a simple online formula to predict each task's complete time but not for the whole job. Morton et al. [6] propose a heuristic online method to predict the progress of parallel queries which can be converted into a series of MapReduce jobs. This method relies on running the query previously to get the intermediate data size and tasks' execution speed. Also, this method does not consider the overlap between map stage and reduce stage. Statistical method has been introduced in [7] to learn the correlation between SQL-like queries and their performance metrics from vast historical logs. But this method does not aim to model the single MapReduce job. Verma et al. [8, 9] study the single MapReduce job's progress in detail and present a theoretical bound based time model to predict job's minimum and maximum duration. This method works well in homogeneous environment without high resource contention. They also advocate a deadline scheduler to meet time goals through adjusting map and reduce slots.

Herodotou et al. [4] propose Hadoop's resource utilization model with numerous parameters and apply it in job's optimum configuration to meet specific performance goals. Based on subspace enumeration and optimized search

* Lijie Xu is a 1st-year PhD student, working with advisor Prof. Jun Wei, ISCAS, Beijing, China, wj@otcaix.iscas.ac.cn and co-advisor Dr. Jie Liu.

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.313

strategy, their configuration method can find the optimum parameters in few times. Instead of enumeration, our optimization approach is based on time model considering job performance and resource utilization simultaneously.

Different schedulers are proposed recently with different goals but the job sequence problem has not been well studied. FIFO is Hadoop's default scheduler aiming to run the coming jobs in sequence. Capacity scheduler [10] and Fair scheduler [11] make the cluster shareable via building multiple separate queues and pools. FLEX [12] provides a flexible slot allocation policy towards special performance requirement such as job makespan. Adaptive Scheduler [13] focuses on dynamically adjusting slots in order to improve resource utilization under jobs' completion time goals.

III. MapReduce Timeline and Related Problems

MapReduce framework executes the submitted jobs in two separate stages (map and reduce stages). In map stage, JobTracker will launch N^M (TotalDataSize/SplitSize) map tasks and each of them processes one logical split including one or few blocks. If total map slots number S^M is less than N^M, map tasks have to run in several waves in pipeline (in Fig. 1). When the first map task finishes, JobTracker will launch N^R reduce tasks on free reduce slots successively. N^R is a critical parameter for job performance but set statically by users. Reduce tasks may run in several waves with small reduce slots number S^R. Each reduce task will go through three (shuffle, sort and reduce) phases. In shuffle phase, reduce tasks are waiting for or fetching the corresponding <K, V> pairs from finished map tasks via HTTP. Shuffle phase finishes after map stage. In sort phase, reduce tasks sort and merge the grouped <K, V> pairs into <K, list(V)>. In reduce phase, reduce function processes the <K, list(V)> records and output final <K',V'> records onto HDFS. The shuffle and sort phases have no apparent boundary in Hadoop. So we integrate the two stages in time model.

The number of map/reduce slots on one slave node determines the max number of map/reduce tasks that can run simultaneously on this node. To improve the job throughput, we can let more tasks run in one wave by increasing slot number in multicore environment. However, more slots may reduce concurrent tasks' performance since more contention of CPU/Mem/IO/Net will occur. This "performance degradation problem" will affect the expansibility and stability of time model. Fig. 2 shows contention becomes more serious with increasing split size (64 to 256MB). It is unnecessary to improve the map slot number when split size is 256MB since the corresponding line's slope is above τ (this threshold will be studied in future). Fig. 3 shows that the wave pattern in map stage becomes messy because of high coefficient of variance (0.3) of map tasks' duration. Reduce tasks also suffer this problem. In this case, existing time models cannot predict each task's accurate duration, not to say the whole job's duration when resources are changed.

Multiple influence factors such as job type, data size, tasks' I/O ratio and slots number result in this degradation problem. In our practice, IO cost jobs with large map/reduce outputs suffer more degradation than others. We are now attempting to model this problem by studying the probability distribution of tasks' duration and resource metrics.

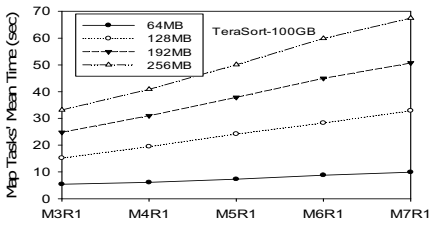

Figure 2. Running TeraSort [14] with different map slots per node.

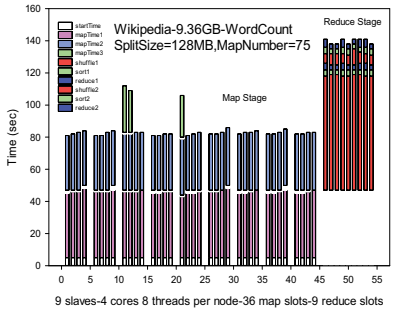

Figure 1. WordCount [1] on Wikipedia text

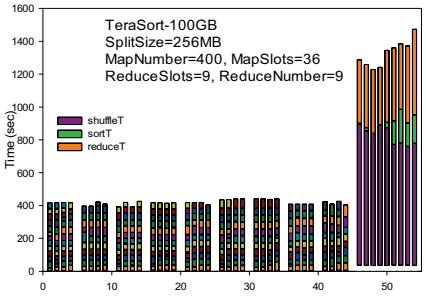

Figure 3. N^M=400, S^M = 36, S^R = 9, N^R = 9

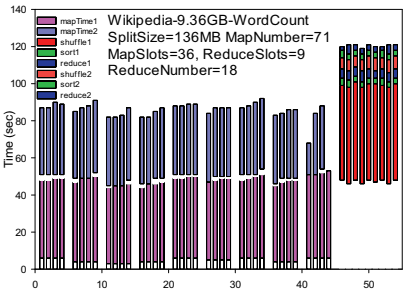

Figure 4. N^M=71, S^M = 36, S^R = 9, N^R = 18

IV. Performance Model and Training Method

Our final target is building an adaptive time model which can predict the accurate job duration both offline and online in heterogeneous environment. Currently, we mainly study the offline time model and training method in homogeneous environment. Due to scheduler's greedy policy of task assignment, first ready slot (when a task finishes on it) will receive the next task firstly. Fig. 1 and 3 are drawn based on this greedy policy from the view of logical slot. In general, the timeline difference between the first and last complete slot is at most the task's maximum duration. This observation is valid only if tasks in the first wave start almost at the same time. When slots are changing online, the time model should be modified dynamically (in future work).

Suppose a new job J will run with N^M map splits and N^R reduce tasks. In present, there are S^M map slots and S^R

reduce slots available. To estimate *J*'s duration, we run our fast sample-based training method as follows only once and use time model (step 8) to predict *J*'s actual execution time.

1. Randomly select N_S^M sample splits from the N^M splits.
   if $N^M < 2 * S^M$ **then** $N_S^M \leftarrow \alpha * N^M$($\alpha$=0.1 or more)
   else $N_S^M \leftarrow S^M$
2. Set the sample job's reduce number N_S^R.
   if $N^R < S^R$ **then** $N_S^R \leftarrow N^R$ **else** $N_S^R \leftarrow S^R$
3. Modify partition function to let reduce task j ($1 \le j \le N_S^R$) receive $j/\sum_{i=1}^{N_S^R} i$ percent of the total <K, V> pairs of map outputs.
4. Run job *J* with the sample splits and collect metrics. CPU/Mem/IO/Net metrics are collected constantly.
   for each map task i
   get its duration $T^M(i)$ and output data size $D^M(i)$
   for each reduce task j
   get its shuffle&sort duration $T^S(j)$,
   reduce duration $T^R(j)$ and its input data size $D^R(j)$.
5. Use regression to establish the correlation between reduce input data size and its duration.
   $T^S(j) = f(D^R(j))$ (received data size from $D^M(i)$ and corresponding transmission rate will be studied later)
   $T^R(j) = g(D^R(j))$
   The real reduce task's mean input data size D_E^R is
   $D_E^R \leftarrow \frac{N^M * \sum_{j=1}^{N_S^R} D^R(j)}{N_S^M * N^R}$.
6. Predict the parameters of time model.
   $T_E^M \leftarrow E(T^M(i))$ (mean value of $T^M(i)$)
   $T_E^S \leftarrow f(D_E^R)$, $T_E^R \leftarrow g(D_E^R)$
   Since reduce stage may overlap with map stage, we denote reduce tasks' shuffle&sort duration in the first wave as T_1^S. H is the interval time (2 or 3 seconds) between sequence tasks in the same slot.
   if $N^M \le S^M$ **then** $T_1^S \leftarrow T_E^S$
   else if $T_E^M + H \ge f\left(\frac{S^M * D_E^R}{N^M}\right)$ **then** $T_1^S \leftarrow f(\frac{S^M * D_E^R}{N^M})$
   else $T_1^S \leftarrow \left\lceil \frac{N^M}{S^M} \right\rceil f\left(\frac{S^M * D_E^R}{N^M}\right) + T_E^M + H - T_{avg}^{map\_stage}$
7. If job *J* has high I/O contention (e.g., variance of $T^M(i)$ is high) or the sample job's execution environment is highly different from the real one. We use Performance Degradation Model to adjust T_E^M, T_E^S and T_E^R later.
8. Predict the mean duration of map and reduce stages (their sum is *J*'s duration, formulas are similar to [8])

$$T_{avg}^{map\_stage} \approx \left\lceil \frac{N^M}{S^M} \right\rceil (T_E^M + H)$$

$$T_{avg}^{reduce\_stage} \approx T_1^S + \left\lfloor \frac{N^R - 1}{S^R} \right\rfloor T_E^S + \left\lceil \frac{N^R}{S^R} \right\rceil (T_E^R + H)$$

Without step 7, this training model works well while predicting low contention jobs' duration in our experiments (shown in Table 1, 9 salves, 36 map slots, 9 reduce slots). But the error is very high towards high contention jobs with large map outputs (e.g., TeraSort-100GB). Next, we will extend the time model to tolerate this degradation problem.

TABLE I. PREDICTED AND ACTUAL TIME ON TYPICAL JOBS

JobName / N^M/N^R / Predicted | Real	**Wikipedia WordCount (Fig. 1) 75 / 18**	**BuildInvertedIndex-9.36GB 150 / 9**	**TwitterGraphReverser-24GB 390 / 9**	**TeraSort-100GB (Fig. 3) 400 / 9**
T_E^M (s)	35 | 37.8	20 | 19.8	14 | 13.9	23 | 35
T_E^R (s)	4 | 4.2	99 | 107	90 | 90.8	190 | 411
$T_{avg}^{map\_stage}$ (s)	111 | 107	110 | 96	176 | 175	300 | 433
$T_{avg}^{reduce\_stage}$ (s)	28 | 35	120 | 139	107 | 113	228 |1094
TotalTime (s)	139 |142	230 | 235	283 | 288	528 |1527

V. JOB CONFIGURATION OPTIMIZATION

Our final goal is developing a tool to automatically detect the slow-running jobs and provide right configurations. Here, we give two primary solutions to optimize slow-running jobs. Since parameters of time model are different among different systems, this optimization aims at the particular system.

A. Map Stage Optimization

Since map splits are generated regardless of the available map slots, map tasks may run in several waves and remain one or few straggler tasks in the last wave. Fig. 1 shows only 3 map tasks run in the last wave so that reduce tasks' shuffle duration is prolonged. Straggler tasks also delay the next job's map stage because less map slots are available.

Our detecting and optimization method is as follows:

1. **if** $N^M \le S^M$ or $N^M mod\ S^M > \alpha (\alpha = 20\% * S^M\ or\ more)$ or $T_E^M + H < f\left(\frac{S^M * D_E^R}{N^M}\right)$
   then do not optimize and return.
2. Randomly generate N_S^M sample splits with increasing size and let i^{th} map split size be
   $Size_{split}(i) = i * Size_{block}/4\ (1 \le i \le N_S^M)$
3. Run the job and get i^{th} map task's duration $T^M(i)$.
4. Use regression to model $T_E^M(Size_{split}) = f(Size_{split})$
5. Use time model to find the optimum $Size_{split}$ (i.e. N^M) to achieve **minimum** $T_{avg}^{map\_stage}$
   subject to $N^M mod\ S^M > \alpha$ and $T_E^M + H \ge f\left(\frac{S^M * D_E^R}{N^M}\right)$

In step 1, if a job runs in one wave or shuffle speed is slower than map tasks' speed, we do not optimize its map stage. Or else, we run the increasing sample splits to model the relationship between map split size and map task's duration. So we can find the appropriate split size to avoid straggler tasks while decreasing the duration of map stage. In addition, we do not let one split become too large to avoid too much loss of data locality and network overhead. Fig. 4 shows job duration has declined and straggler tasks disappear when split size is up to 136MB. In future work, more advantages and disadvantages of this method will be studied.

B. Reduce Stage Optimization

It's a difficult problem for users to configure the right reduce task number N^R when facing different jobs. Cluster administrators also feel hard to set appropriate S^R because it is a trade-off problem between jobs' throughput and

performance. The time model tells us that more reducers lead to shorter duration of shuffle and reduce phases for each reduce task. However, for some jobs with fast shuffle speed (step 1) and small D_E^R, it is unworthy occupying too many reduce slots in order to gain less decline of reduce time. On the contrary, some IO cost jobs with large D_E^R may spend too much time in shuffle phase (shown in Fig. 3). The best solution is enlarging the concurrent reduce tasks so that time cost of shuffle and reduce phases dramatically decline (Fig. 5). But we should be aware of the network utilization U_{Net}. If network overhead occurs, we may set the reduce number 2 or 3 times of reduce slots in order to decrease the network contention. Step 1 gives a primary optimization framework.

1. **if** $T_1^S > \beta * T_E^M$ ($\beta \geq 3$) and $U_{Net} < \gamma$ ($\gamma \geq 0.9$) **then**
   if $N^R < S^R$ **then** $N^R \leftarrow S^R$
   else $N^R \leftarrow S^R \leftarrow f(Core_{cpu}, U_{Net}, Size_{data}, T_E^M)$

VI. Job Sequence Optimization

Our final goal is developing a scheduler assistant towards job sequence optimization. Now, we first illustrate this problem. When job A and job B with equal priority are submitted into the job queue successively, the FIFO scheduler runs job A first and then B. Similar situation exists in Capacity scheduler's certain queue and Fair Scheduler's certain pool. Hence, job B has to wait until job A releases the available slots, which may lengthen job B's makespan. If we run job B ahead of job A, job B's makespan will be reduced but the two jobs' completion time may be prolonged.

This problem is more common and serious when a "small" job and a "big" job come together, running the small one first may cause low resource utilization but gain less makespan. Fig. 6 and 7 show this situation occurs when running the BuildInvertedIndex [15] and WordCount on Wikipedia (9.36 GB). In Fig. 6, although job B's reduce stage is delayed by job A, the duration of the two jobs is shorter than that in Fig 7. The time-slower reason exists in job B. In Fig. 7, job B's first shuffle phase contains much waiting time so that the reduce slots has lower utilization compared with Fig. 6.

Whether to apply this adjustment depends on the cluster administrator and end-users' goals. If promoting resource utilization is more important in current cluster, we may let costly jobs' reduce tasks run as soon as possible. On the contrary, if decreasing small jobs' makespan is more critical, we may let small jobs run ahead. Based on time model, we will develop a scheduler assistant to optimize the running sequence of jobs according to different SLAs.

References

[1] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system," in *SOSP*, 2003.

[3] White, T. Hadoop: The Definitive Guide, Page 169.

[4] Herodotou, H., Babu, S. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs," in *VLDB*, 2010.

[5] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. "Improving MapReduce performance in heterogeneous environments," in *OSDI*, 2008.

[6] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. "Estimating the progress of MapReduce pipelines," in *ICDE*, 2010.

[7] Ganapathi, A., Chen, Y., Fox, A., Katz, R., Patterson, D. "Statistics-driven workload modeling for the cloud," in Proc. of *SMDB*, 2010.

[8] A. Verma, L. Cherkasova, and R. Campbell. "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *ICAC*, 2011.

[9] A. Verma, L. Cherkasova, and R. Campbell. "Resource Provisioning Framework for MapReduce Jobs with Performance Goals," in *Middleware*, 2011.

[10] Yahoo! Inc. Capacity scheduler, http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/

[11] M. Zaharia, et al. "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010.

[12] J. Wolf, et al. "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," in *Middleware*, 2010.

[13] J. Polo, et al. "Resource-Aware Adaptive Scheduling for MapReduce Clusters," in *Middleware*, 2011.

[14] TeraSort Benchmark, http://sortbenchmark.org/

[15] Cloud^9, http://lintool.github.com/Cloud9/

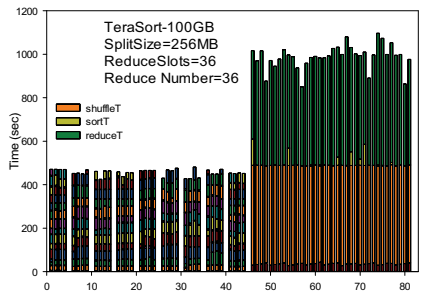

Figure 5. N^M=400, S^M=36, S^R=36, N^R=36

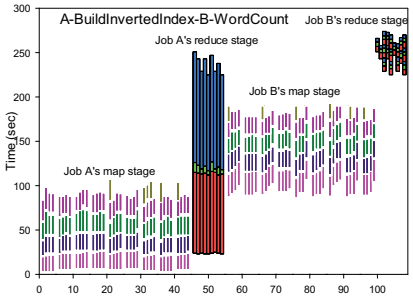

Figure 6. SplitSize=64MB, $N_A^M = N_B^M$=150

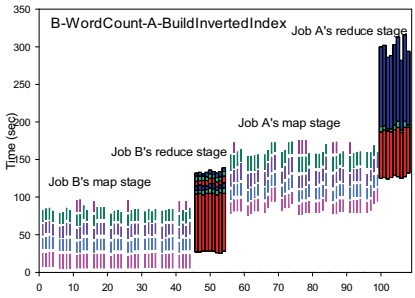

Figure 7. S^M= 36, S^R = 9, N_A^R = 9, N_B^R =18