Improving Resource Utilization in MapReduce

Zhenhua Guo, Geoffrey Fox, and Mo Zhou

School of Informatics and Computing,
Indiana University, US
{zhguo,gcf,mozhou}@cs.indiana.edu

Abstract. MapReduce has been adopted widely in both academia and industry to run large-scale data parallel applications. In MapReduce, each worker node hosts a number of task slots to which tasks can be assigned. So they limit the maximum number of tasks that can execute concurrently on each node. When all task slots of a node are not used, the resources "reserved" for idle slots are wasted. To improve resource utilization, we propose resource stealing to enable running tasks to steal resources reserved for idle slots and give them back proportionally whenever new tasks are assigned. Resource stealing makes the otherwise wasted resources get fully utilized without interfering with normal job scheduling. MapReduce uses speculative execution to improve fault tolerance. Current Hadoop implementation decides whether to run speculative tasks based on the progress rates of running tasks, which does not take into consideration the absolute progress of each task. We propose Benefit Aware Speculative Execution which evaluates the potential benefit of speculative tasks and eliminates the unnecessary runs. We implement our proposed algorithms in Hadoop and conduct experiments to show that our algorithms can significantly shorten job execution time and reduce the number of non-beneficial speculative tasks.

Keywords: MapReduce, Hadoop, scheduling, resource utilization, speculative execution

1 Introduction

Data deluge has been observed in many science areas such as particle physics, astronomy, and biology. A significant amount of computation power is demanded to process the collected data. Message Passing Interface (MPI) [1] has been used widely in High Performance Computing (HPC) as a programming model. For data parallel applications, MapReduce [2] has gained popularity in both academia and industry, and has been used in bioinformatics [3], machine learning [4], etc. Hadoop is a widely-used implementation of MapReduce and thus our research target. Besides Hadoop, other data parallel systems including Dryad and Sector/Sphere have been developed with different features.

In MapReduce, multiple tasks can run concurrently on each node to explore the processing capability of modern multi-core processors. To limit the task concurrency on each node and thus avoid intense resource contention, each node

hosts a configurable number of map and reduce slots where tasks can run. A slot gets *occupied* when a task is assigned to it, and gets *released* when the task completes. This approach results in resource underutilization when there are no enough tasks to fill all slots, which is mitigated by our proposed resource stealing.

In MapReduce, speculative execution is adopted to support fault tolerance. The master node keeps track of the progresses of all scheduled tasks. When it finds a task that runs unusually slow compared with other tasks of the same job, a speculative task is launched to process the same input data with the hope that it will complete earlier than the original task. The speculative tasks, which complete later than original tasks and do not benefit the overall job execution, are termed *non-beneficial speculative tasks*, the number of which should be minimized to maximize efficiency. In Hadoop, the default mechanism incurs the execution of many non-beneficial speculative tasks and is inefficient. We propose Benefit Aware Speculative Execution to solve it.

The rest of this paper is organized as follows. Related work is discussed in sector 2. The details of our proposed resource stealing algorithm and BASE are discussed in sector 3. The experiments we have conducted and their results are presented in sector 4. Finally we conclude in sector 5.

2 Related Work

The term speculative execution has been used in different contexts. For example, at instruction level, branch predictors [5] guess which branch a conditional jump will go to and speculatively execute the corresponding instructions; and automatic I/O hint generation [6] exploits idle processor cycles to dynamically analyze the I/O behavior of the application that is stalled on I/O and predict its future data access. For distributed systems where communication overhead significantly impacts performance, task duplication [7] redundantly executes some tasks on which other tasks critically depend. So task duplication mitigates the penalty of data communication by running the same task on multiple nodes. Speculative execution in MapReduce employs a similar strategy but is mainly used for fault tolerance.

To improve MapReduce performance in heterogeneous environments, Longest Approximate Time to End (LATE) [8] is proposed which aims to robustly perform speculative execution by prioritizing tasks to speculate, selecting fast nodes to run on and limiting the number of speculative tasks. Our BASE algorithm improves upon LATE to further maximize performance.

Work stealing [9] enables idle processors to steal computational tasks from other processors and is more communication efficient than its work-sharing counterparts. Our proposed resource stealing shares similar motivations. But the execution model of MapReduce is logically independent of underlying hardware while work stealing is closely coupled with processors. Cycle stealing [10] enables busy nodes to take control of idle nodes, supply them with work, and receive results. The motivation is to harness the otherwise wasted resources of idle nodes.

Task splitting yields better load balancing across nodes by dynamically adjusting task granularities [11]. Our proposed resource stealing is applied at a lower level to the resources located on a single node.

In grid systems, batch scheduling has been used extensively. When a job is scheduled, the requested number of nodes are reserved for a specific period of time even though the resource usage may vary across the phases of the job. Backfilling [12] moves small jobs ahead to leapfrog big jobs in front to alleviate fragmentation and improve resource utilization. Backfilling should not delay the first job or any job waiting in the queue depending on its aggressiveness. Resources are shared among jobs in MapReduce while grid systems adopt reservation-style resource allocation. In resource stealing, jobs are not re-ordered or moved in the queue; and stealing is done at task level without impacting job scheduling at all. So resource stealing is a finer-grained and lower-level optimization of resource usage.

3 Our approaches

3.1 Resource Stealing

How to tune Hadoop parameters automatically has been studied in [13, 14]. In this paper, we assume the number of task slots are set optimally so that optimal resource utilization is achieved when all slots are occupied. Resource utilization is proportional to the number of occupied slots approximately. According to the trace of production clusters [15], resource utilization is way below 100% and varies across time periods, so mostly there exist idle slots in large systems. It implies that the capability of resources can be further exploited to minimize execution time. The portion of the resources that sit idle on a slave node is termed *residual resources* which can be utilized without incurring severe usage contention or degrading overall performance. We can consider that residual resources are reserved for prospective tasks that will be assigned to currently idle slots. One advantage of resource reservation is that whenever a new task is assigned resource availability is guaranteed. However, an obvious drawback is that residual resources are left unused until new tasks are assigned.

We propose *resource stealing* to improve resource utilization. The resource usage of running tasks (if any) on each node is dynamically expanded or shrunk according to the availability of task slots. When there are idle slots, running tasks temporarily steal resources reserved for prospective tasks so that residual resources are fully utilized. If a node is perfectly loaded by using resource stealing, to assign a new task obviously will overload it and degrade the performance of running tasks. Our solution is to adjust the resource usage of running tasks by making them relinquish stolen resources proportionally. In this way, resource stealing does not violate the assumption made by Hadoop that resources are guaranteed for new tasks, which is critical to efficient Hadoop scheduling. To summarize, the overall philosophy is to steal residual resources if corresponding map/reduce slots are idle, and hand them back whenever new tasks are launched to fill the idle slots. From the perspective of the task scheduler, idle slots are still

idle and new tasks can be assigned to them, so resource stealing is transparent to the task scheduler and can be used in combination with any Hadoop scheduler directly such as fair scheduler and capability scheduler. Resource stealing is applied periodically with the up-to-date information of task execution and system status. So it is adaptive in the sense that it reacts to real-time changes of the system state.

3.2 Allocation Policies of Residual Resources

Given residual resources and the number of running tasks on a node, the next issue is how to distribute residual resources among running tasks, e.g. which tasks should get how much. The policies can range from simple to complex in their use of system state information. Complex policies have the potential to take full advantage of the processing capability of each node. The disadvantages include high overhead cost and the risk that a well tuned policy may behave unpredictably when inaccurate state information is collected. We come up with several policies summarized below.

Even: This policy equally divides residual resources among running tasks. It is inherently stable because of not relying on the collection or prediction of system state (and thus not impacted by the information inaccuracy).

First-Come-Most (FCM): This policy orders running tasks by start time. The task with the earliest start time is given residual resources. The heuristic is to make tasks complete in the order of job submission with best efforts.

Shortest-Time-Left-Most (STLM): Firstly, the remaining execution time of tasks is estimated, where different mechanisms can be plugged in. Here we adopt the same mechanism used in [8] which assumes each task progresses at a constant rate across time and predicts the time left based on progress rate and current progress. The task with the shortest time left is given residual resources. The heuristic is to make close-to-completion tasks complete as soon as possible to make way for long-running tasks.

Longest-Time-Left-Most (LTLM): This policy is the same as STLM except that the task with longest time left is given residual resources.

Speculative-Task-Most (STM): Speculative execution in MapReduce aims to mitigate the impact of slow tasks by duplicate their processing on multiple nodes. The basic idea of STM policy is that speculative tasks are given more resources than regular tasks with the hope that they will not hurt the job execution time. Because speculative tasks are given more resources, they can run faster and will not be stragglers any longer hopefully. If there are no speculative tasks on a node, it falls back to the regular case and other policies can be applied. If there are multiple speculative tasks running on a node, residual resources are allocated to them evenly.

Laggard-Task-Most(LTM): In this approach, we do not distinguish between regular tasks and speculative tasks. Instead, for each job we use the estimated remaining execution time of all its scheduled tasks (both regular and speculative tasks) to calculate the *fastness* of a running task T using (1). Fast-

ness reflects the expected order of task completion for each job; and a task with small fastness will complete later than a task with large fastness.

The fastness of a task cannot be computed locally by a slave node because it requires the information of all other tasks belonging to the same job. Job tracker maintains the statues of all tasks so that it is the ideal component to compute fastness. Each slave node reports the statuses (e.g. progress, failure) of its running tasks to the job tracker in heartbeat messages. After collecting the information of all tasks, the job tracker calculates the fastness of each task and returns it to the corresponding slave node. Upon receiving fastness information, slave nodes order tasks by fastness. The tasks whose fastness is smaller than threshold *SlowTaskThreshold* (a user configurable parameter) are called *laggards* and given residual resources. If there are multiple laggards on a node, residual resources are evenly allocated to them.

$$fastness = \frac{\#\ \ of\ \ \ tasks\ \ \ that\ \ \ will\ \ \ complete\ \ \ later\ \ \ than\ \ \ T}{\#\ \ \ of\ \ \ running\ \ \ tasks} \quad (1)$$

As we discussed, the motivation of speculative execution is to improve performance by running duplicate processing. There are several drawbacks. Firstly, if speculative execution is triggered, the completion of any task renders the work done by other duplicate tasks to be wasted. Secondly, if the slowness of tasks is caused by intermittent and temporary resource contention, it is highly likely that they do not lag much behind and still complete earlier than their speculative tasks, which subverts the motivation of speculative execution. Thirdly, sometimes speculative execution deteriorates performance rather than improve it [8]. LTM reduces the invocations of speculative execution by proactively allocating more resources to laggards whenever possible and thus accelerating their execution. Fourthly, the tasks of a job may be heterogeneous intrinsically in that their execution time varies greatly depending upon both data size and the content of the data. For example, easy and difficult Sudoku puzzles have similar input sizes (9 x 9 grids) but require dramatically different amounts of computation. Speculative execution is not helpful because the variation of execution time is not mainly caused by extrinsic factors (e.g. faulty nodes) and the execution time will not be reduced significantly no matter how many speculative tasks are run. In that case, the tasks demanding the most computation progress slower than other tasks and thus are the laggards. LTM speeds up their execution by assigning more resources. By balancing the workload within each job, LTM reduces job execution time and the number of speculative tasks. Assignments of new tasks decrease the amount of residual resources while the completion of running tasks increases the amount of residual resources. They both trigger the re-allocation of residual resources.

3.3 The BASE Scheduler

Speculative execution is not a simple matter of running redundant tasks for sufficiently slow tasks. To make it effective, two issues need to be addressed: i) detect slow tasks; ii) choose the tasks to speculate. Hadoop identifies the tasks whose progress rates are one standard deviation lower than the mean of all tasks

as slow tasks. Then it chooses the task with the longest remaining execution time to speculate. It does not take into consideration whether speculative tasks will complete before the original tasks. Assume a job has two tasks A and B; task A is 90% done but progresses slowly with rate 1; and task B progresses fast with rate 5. Because task A progresses slow, the master node decides to start a speculative task A' for A which progresses with rate 5. By doing a little math, we can easily figure out that task A will complete earlier than A' although A progresses slowly. The reason is that task A is close to completion when A' is launched. This inefficiency was observed in our tests, where a large portion of speculative tasks were killed before their completion because the original tasks completed earlier. Those speculative tasks were not beneficial at all and their execution resulted in the waste of resources. To overcome the issue, we propose Benefit Aware Speculative Execution (BASE) in which speculative tasks are launched only when they are expected to complete earlier than the original tasks. The estimation of the remaining execution time of a running task has been discussed above. We propose a mechanism to estimate the execution time of prospective speculative tasks. It depends upon two factors: 1) the progress rates of other tasks of the same job; 2) the node where the speculative task will run. The key is to estimate the progress rate which can be directly used to calculate run time. Slow tasks can be identified using the mechanism described in [8]. Given a slow task T of job J and a slave node N_i, following algorithm solves the problem whether a speculative task T' should be launched on N_i for T.

1. If some tasks belonging to J are running or have run on node N_i, the mean of their progress rates is calculated and used as the progress rate of T'.
2. Otherwise, progress rates of all scheduled tasks of job J are gathered and normalized against the reference baseline. The normalization of progress rates is needed when nodes are heterogeneous and is computed based on hardware processing power (e.g. weighted sum of the capabilities of processors, disks and network interface cards). Then the mean of normalized progress rates is calculated. Because the mean is against the reference baseline, we de-normalize it against the specification of node N_i to compute the expected progress rate of T'. We assume the scheduling order of tasks is stochastic approximately and thus the mean of scheduled tasks reflects the expectation of real progress rate.
3. No matter which of 1) and 2) is applied, the estimated progress rate of T' has been calculated so far. The execution time is $1/progressrate$. If it is shorter than the remaining execution time of T, T' is launched on N_i. Otherwise, do not run T' on N_i.

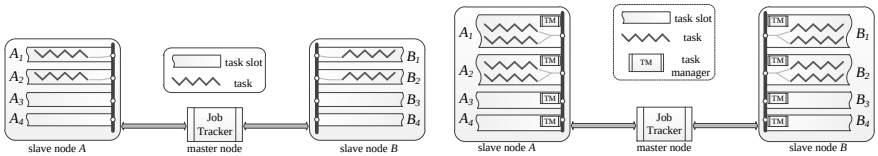
To predict the run time of T' via the mean of progress rates actually is equivalent to the harmonic mean of the run time of scheduled tasks.

3.4 Implementation

Our implementation is optimized for compute-intensive applications and thus processors and cores are the critical resources. Multithreading technique is adopted

to explore the parallel processing capability of modern servers. In Hadoop, each task is run in a separate process to isolate its execution environment. Within each task process, one thread is started to process data. Fig. 1 shows an example. There are two slave nodes each of which has 5 cores. Each node has 4 slots among which 2 slots are idle. For node A, Slots A_1 and A_2 are busy; and slots A_3 and A_4 are idle. In Hadoop, each task process only runs one thread even if there are lightly-utilized cores (shown in Fig. 1(a)). Resource stealing starts multiple threads within a task process that concurrently process input data (shown in Fig. 1(b)). One extra thread is created for both A_1 and A_2, and each of the four threads can be scheduled to an individual core. For each task, *thread manager* periodically adjusts the number of threads dynamically based on the latest system status.

Resource stealing and BASE are transparent to end users. Regular MapReduce applications can be run directly without any modification. Additional configuration parameters are added and exposed to make administrators able to tune various aspects of our improvements. For example, administrators can enable/disable resource stealing and/or BASE, and change the allocation policy of residual resources.



(a) Native Hadoop (b) Resource stealing

Fig. 1: Scheduling with native Hadoop and resource stealing

4 Experiment

We conducted extensive experiments to evaluate our proposed algorithms. Instead of directly measuring resource utilization (e.g. CPU usage), we measure user-perceivable job execution time which indirectly reflects the improvement or deterioration of resource utilization. Although simple applications (e.g. grep, web crawler) are used in our tests below, their results are applied to not only them per se but also other applications of the same types. For instance, for compute-intensive applications, what is computed concretely (e.g. pattern searching, sequence alignment) is not important, and what matters is computation dominates overall execution. So we believe our findings are applicable to several categories of applications under our consideration.

4.1 Scheduling of Map-only Jobs

On FutureGrid Hotel cluster, we deployed Hadoop which comprised one master node and twenty slave nodes that were homogeneous in terms of both hardware and software. Each node had 20GB memory and 8 cores one of which was reserved for HDFS and MapReduce daemons. According to the best practice that the number of slots should be between 1x and 2x the number of cores, each node

was configured to host 7 map slots and 7 reduce slots. So there were 140 map slots and 140 reduce slots total. Block size of HDFS was set to 128MB.

We ran *grep* without reduce phase to eliminate the impact of shuffling and merging and exactly measure the effectiveness of resource stealing for map-only jobs. A large portion of MapReduce jobs (over 70%) are map-only jobs[16]. In our tests, each map task processed 128MB text data and was tuned to run approximately for 5 minutes by repeating regular expression matching in map operations to simulate the interactive job types in MapReduce [2]. To avoid confusion, we name it *rep-grep*. Please note that grep is IO intensive while rep-grep is compute intensive. Multiple rep-grep jobs were run with the number of map tasks varied. We set the number of map tasks to 35, 70, 105 and 126 which yield system workload 25%, 50%, 75% and 90%.

Rep-grep without BASE We ran rep-grep without BASE and show job execution time in Fig. 2(a). Execution time is not significantly influenced by workload for native Hadoop, which means that processing 4.375GB, 8.75GB, 13.125GB, and 15.75GB data takes similar amounts of time. The reason is resource usage is proportional to the number of tasks and residual resources are not utilized at all. Resource stealing shortens run time by 64%, 32%, 13%, and 6% respectively for policy Even. The lower the workload is, the more resource stealing outperforms native Hadoop. So the performance benefit of resource stealing is negatively related to system workload, which matches our expectation well. We also calculated the average processing time per GB data by dividing job execution time by data size. Increasing workload can drastically improve the efficiency for native Hadoop, while it approximately keeps invariant for resource stealing. Different allocation policies exhibit different performance. Overall, STLM and LTLM perform the worst and LTM performs well for all tests. It implies that it is inefficient to blindly allocate residual resources evenly or simply enforce FIFO order. When the workload gets relative high (e.g. 75%, 90%), the performance difference becomes smaller.

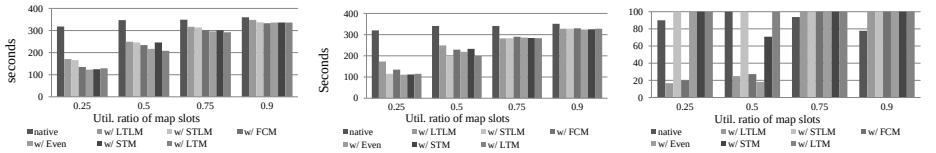
In our setup, all nodes were on the same rack and blocks were randomly placed on nodes by HDFS with its default block placement strategy. Data locality aware scheduling in Hadoop co-locates compute and data with best efforts. As a result, MapReduce tasks were evenly distributed across all slave nodes approximately so that each node ran a similar number of tasks. This is beneficial to resource stealing because its gain is not substantial if the resources of a node are fully loaded already.

Rep-grep with BASE We ran the same tests as above except BASE was enabled and present results in Fig. 2(b). The plot has similar characteristics to Fig. 2(a) in that native Hadoop performs the worst and the performance superiority of resource stealing decreases with the increased sytem workload. By comparing 2(b) and 2(a), we observe that BASE shortens execution time and the improvement is increased as system workload is also increased.

For the cases where BASE is disabled and enabled, we counted the number of non-beneficial speculative tasks and computed the difference shown in Fig. 2(c).

BASE drastically eliminates the launches of non-beneficial speculative tasks. For workload 75% and 90%, almost all unneeded speculative tasks are removed.

we conclude that BASE reduces the number of non-beneficial speculative tasks significantly while yielding shorter execution time. Because a fewer number of speculative tasks are launched, the saved resources can be allocated to normal tasks to speed up their execution. It also indicates that the estimation of execution time is approximately accurate so that BASE rarely removes the runs of beneficial speculative tasks.



(a) Runtime w/o BASE (b) Runtime w/ BASE (c) Reduction of non-beneficial spec. tasks w/ BASE(%)

Fig. 2: Run map-only rep-grep in a homogeneous environment

4.2 Scheduling of Map-only Jobs with Straggler Nodes

In this experiment, background load is generated to slow down some nodes and simulate stragglers. We wrote a load generator that can generate user-specified load of computation, network and disk IO. We ran two CPU-hogging threads per core which resulted in nearly 100% core utilization, and one IO-intensive thread reading/writing data continuously from/to disks. The background load significantly slowed down the nodes without rendering them thoroughly unresponsive. We ran rep-grep jobs that utilize 75% of all map slots and thus 8.75GB data was processed total in each run.

Firstly, two slave nodes were slowed down. Job execution time is shown in Fig. 3(a). Again resource stealing improves performance over native Hadoop significantly no matter which resource allocation policy is used. LTM performs well stably for the cases with and without BASE. Fig. 3(b) shows BASE can save runs of nearly all unnecessary speculative tasks, which implies the estimation of execution time is accurate when only a small number of nodes are stragglers.

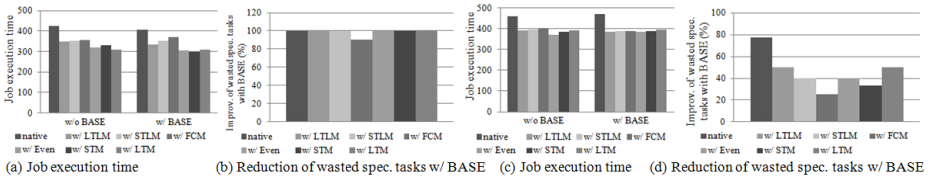
Secondly, four slave nodes were slowed down. Fig. 3(c) shows execution time. The jobs ran longer compared with the previous test because more map tasks were slowed down. Resource stealing is still effective to speed up job execution. The performance disparity of different resource allocation policies becomes marginal and they perform equally well approximately. Fig. 3(d) shows BASE can eliminate 20% - 50% of non-beneficial speculative tasks. Compared with the previous case, BASE becomes less effective. It indicates our estimation of execution time gets inaccurate as more straggler nodes incur larger variation of task execution. In addition, resource stealing aggravates the situation because of the dynamic nature of the (re-)allocation of residual resources.

4.3 Scheduling of Reduce-mostly Jobs

In this test, we ran reduce-mostly jobs and used modified *grep* (not *rep-grep*) as the test application. Operations in the reduce phase of grep were run repeatedly

to make reduce phase dominate overall execution. Each job comprised 10 reduce tasks and 70 map tasks each of which processed 128MB data, and ran for 5 minutes approximately. For resource stealing, only policy Even is compared below because it is simple and performs among the best based on the results above.

Job execution time and the number of non-beneficial speculative tasks are shown in Fig. 4(a) and Fig. 4(b) respectively. BASE reduces job execution time marginally, but reduces the number of non-beneficial speculative tasks by 90% (from 10 to 1) compard with native Hadoop. Because of the drastic reduction of resource waste, more useful tasks can be run concurrently and thus the efficiency of resource usage is improved. This demonstrates the effectiveness of BASE. Fig. 4(b) shows resource stealing thoroughly eliminated non-beneficial speculative tasks, and Fig. 4(a) shows resource stealing substantially shortens job execution time by 70% - 80%. There are many more nodes than reduce tasks which are well spread out so that each node runs one reduce task at most on average. For each reduce task, resource stealing creates 6 new reduce tasks (remember the number of reduce slots is 7 on each node) to run in parallel, which should yield 7x speedup optimally. In reality, we only got 4x–5x speedup because of additional overhead. Reduce threads compete for the same input stream and only one thread can read from the stream at any time. To alleviate the contention, in our implementation each thread locks the input stream, copies next (key, values) tuple to its local buffer, unlocks the input stream and processes the data in local buffer without interfering with other threads. But this approach incurs extra memory copies. In addition, reduce threads belonging to the same task contend for the same output stream as well. To investigate advanced mechanisms to mitigate contention further is among future work.



(a) Job execution time (b) Reduction of wasted spec. tasks w/ BASE (c) Job execution time (d) Reduction of wasted spec. tasks w/ BASE

Fig. 3: Run map-only rep-grep with straggler nodes. There are two straggler nodes for (a) and (b), and four straggler nodes for (c) and (d)

4.4 Experiments with Other Workload

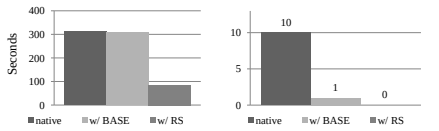
Besides compute-intensive applications. We also ran jobs of other types to comprehensively evaluate our approaches.

Network-Intensive Workload: We wrote a distributed web crawler *mr-wc*. Its input is a set of URLs of the webpages to download. Mr-wc does not have reduce phase; and its map tasks download web pages and save them into HDFS. Network is the most critical resource for mr-wc. Lemur project published a data set of unique URLs [17]. We use a small portion of it as the input of mr-wc. The same testbed as above was used. In our tests, each map task downloaded 400 web pages and the number of map tasks was set to 35, 70, 105 and 126 for different runs, so the system workload was 25%, 50%, 75% and 90% respectively.

Fig. 5(a) shows the execution time. For native Hadoop, the execution time of mr-wc is not significantly impacted by the workload, which implies spare resources cannot be utilized. In contrast, resource stealing expands the usable resources of running tasks by creating more threads to concurrently download webpages. Resource stealing dramatically shortens execution time by 61%, 45%, 21% and 23% respectively.

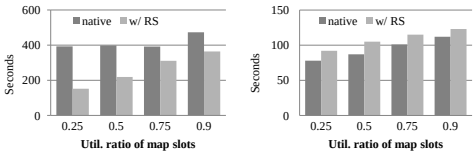
For above tests, speculative execution was disabled because our additional tests showed it deteriorates performance mostly. The efficiency of webpage crawling depends heavily on the response time of the servers where webpages are hosted, which ranges from milliseconds to seconds. Under this circumstance, running speculative tasks is not helpful because the efficiency variation of tasks is not caused by the system itself.

IO-Intensive Workload: *Wordcount*, which counts the number of word occurrences, is a typical IO-intensive application and used in this set of tests where each map task processed 128MB text and the number of map tasks was varied. Fig. 5(b) shows the result. As the number of map tasks increases, job execution time increases as well and the processing throughput (the amount of processed data per unit of time) is improved. Resource stealing slightly degrades rather than improves performance. Within map tasks, each map operation processes one line of text and is invoked repeatedly. Although resource stealing enables Hadoop to start multiple threads to run map operations in parallel, these threads share the same underlying input reader and output writer (to comply with current Hadoop design). This incurs significant overhead among threads for IO-intensive applications because the computation time of each map operation is short and synchronization becomes the performance barrier. As a result, the overhead outweighs the benefit of higher concurrency brought by resource stealing.



(a) Job execution time (b) Num. of non-beneficial spec. tasks

Fig. 4: Reduce-mostly modified grep



(a) mr-wc (b) wordcount

Fig. 5: Experiment with other workload

5 Conclusion

The goal of our work is to improve resource utilization in MapReduce. We present resource stealing to dynamically re-allocate idle resources to running tasks with the promise that they will be handed back whenever they are required by newly assigned tasks. It can be applied in conjunction with existing job schedulers smoothly because of its transparency to central job scheduling. In addition, we have analyzed the mechanism adopted by Hadoop to trigger speculative execution, discussed its inefficiency and proposed Benefit Aware Speculative Execution which starts speculative tasks based on the estimated benefit. Our conducted

experiments demonstrate their effectiveness. Resource stealing yields dramatic performance improvement for compute-intensive and network-intensive applications and BASE effectively eliminates a large portion of unnecessary runs of speculative tasks. For IO-intensive applications, we observed slight performance degradation caused by intensive contention of input reading and output writing. In future, we will investigate lock-free data structures and make resource stealing benefit IO-intensive applications as well.

References

1. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface. MIT Press, Cambridge, MA, USA (1994)
2. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of OSDI'04. (2004) 137–150
3. Qiu, X., Ekanayake, J., Beason, S., Gunarathne, T., Fox, G., Barga, R., Gannon, D.: Cloud technologies for bioinformatics applications. In: Proceedings of MTAGS'09, New York, NY, USA, ACM (2009)
4. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y.Y., Bradski, G., Ng, A.Y., Olukotun, K. NIPS
5. Gwennap, L.: New algorithm improves branch prediction. Microprocessor Report **9**(4) (1995) 17–21
6. Chang, F., Gibson, G.A.: Automatic I/O hint generation through speculative execution. In: Proceedings of OSDI'99, Berkeley, CA, USA (1999)
7. Ahmad, I., Kwok, Y.K. In: Proceedings of ICPP'94, Washington, DC, USA
8. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: Proceedings of OSDI'08, Berkeley, CA, USA (2008) 29–42
9. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: Proceedings of FOCS'94, Washington, DC, USA (1994) 356–368
10. Bhatt, S.N., Chung, F.R.K., Leighton, F.T., Rosenberg, A.L.: On Optimal Strategies for Cycle-Stealing in Networks of Workstations. IEEE Trans. Comput. **46** (May 1997) 545–557
11. Guo, Z., Pierce, M., Fox, G., Zhou, M.: Automatic Task Re-organization in MapReduce. In: Proceedings of CLUSTR'11, Washington, DC, USA (2011) 335–343
12. Mu'alem, A.W., Feitelson, D.G.: Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6) (June 2001) 529–543
13. Kambatla, K., Pathak, A., Pucha, H.: Towards optimizing hadoop provisioning in the cloud. In: Proceedings of HotCloud'09, Berkeley, CA, USA (2009)
14. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F., Babu, S.: Starfish: A self-tuning system for big data analytics. In: Proceedings of CIDR'11, Asilomar, California, USA. (2011)
15. Barroso, L.A., Hölzle, U.: The Case for Energy-Proportional Computing. Computer **40** (December 2007) 33–37
16. Kavulya, S., Tan, J., Gandhi, R., Narasimhan, P.: An Analysis of Traces from a Production MapReduce Cluster. In: Proceedings of CCGRID'10, Washington, DC, USA (May 2010) 94–103
17. Clueweb09: http://lemurproject.org/clueweb09.php/