Improving Resource Utilization in MapReduce

Zhenhua Guo, Geoffrey Fox, Mo Zhou, Yang Ruan
School of Informatics and Computing
Indiana University
Bloomington, IN, USA
{*zhguo, gcf, mozhou, yangruan*}*@cs.indiana.edu*

***Abstract*—MapReduce has been adopted widely in both academia and industry to run large-scale data parallel applications. In MapReduce, each slave 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 unutilized. 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 unnecessary runs. We implement the proposed algorithms in Hadoop, and our experiments show that our algorithms can significantly shorten job execution time and reduce the number of non-beneficial speculative tasks.**

***Keywords*-MapReduce; scheduling; utilization; speculative execution**

I. 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 ever-growing quantity of data collected by modern instruments such as Large Hadron Collider and next-generation gene sequencers. Message Passing Interface (MPI) [1] has been used widely in High Performance Computing (HPC) as a programming model. In data-intensive computing, frequent data movement incurs significant load on network and storage which may dominate the overall application execution. Data affinity is implicitly integrated into task scheduling in recently proposed data parallel systems among which MapReduce is representative. MapReduce [2] has quickly gained popularity in both academia and industry. It been used in indexing [2], 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 [5] and Sector/Sphere [6] have been developed with different features.

In MapReduce, multiple tasks can run in parallel on each node to exploit the parallel processing capability of modern multi-core processors. To limit the degree of task concurrency and thus avoid intense resource contention, each slave node hosts a configurable number of map and reduce slots to which map and reduce tasks are scheduled for execution respectively. In Hadoop implementation, each slave node has 2 map slots and 2 reduce slots by default. A task slot gets *occupied* when a task is assigned to it, and gets *released* when the task completes. The hard partition of physical processing capability into virtual map and reduce slots has drawbacks. Firstly, it is not trivial to determine the optimal numbers of map and reduce slots. Secondly, it results in resource underutilization when there are not enough tasks to occupy all slots, which is mitigated by our proposed *resource stealing*.

In distributed systems, failure is the norm rather than the exception. *Speculative execution* is adopted by MapReduce 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. To maximize efficiency, the number of speculative tasks should be minimized which complete later than original tasks and do not speed up the overall job execution. In Hadoop, the default mechanism to trigger speculative execution only considers the progress rates of running tasks, and incurs the execution of many non-beneficial speculative tasks. We propose Benefit Aware Speculative Execution to solve it.

The rest of this paper is organized as follows. Related literatures are presented in section II. The design and implementation details of our proposed resource stealing algorithm and BASE are described in section III. Experimental evaluations are presented in section IV. Finally conclusions and future work are summarized in section V.

II. Related Work

The term speculative execution has been used in different contexts. For example, at instruction level, branch predictors guess which branch a conditional jump will go to and speculatively execute the corresponding instructions [7]. For distributed systems where communication overhead is sub-

stantial, task duplication redundantly executes some tasks on which other tasks critically depend [8]. 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) 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 [9]. Our BASE algorithm improves upon LATE to further maximize performance.

Work stealing enables idle processors to steal computational tasks from other processors and is more communication efficient than its work-sharing counterparts [10]. 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 enables busy nodes to take control of idle nodes, supply them with work, and receive results [11]. 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 [12]. Our proposed resource stealing is applied at a lower level to the resources located on individual nodes. Iterative MapReduce optimizes the execution of iterative applications by aggressively caching and reusing intermediate data across iterations [13]. Mantri monitors task execution and acts on outliers early using cause- and resource-aware techniques including restarting outliers, network-aware task placement, and replicating outputs of valuable tasks [14].

In grid systems, batch queuing 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 moves small jobs ahead to leapfrog big jobs in front to alleviate fragmentation and improve resource utilization [15]. Backfilling does not delay the first job or any job waiting in the queue depending on its aggressiveness. Batch queuing systems adopt reservation-based resource allocation mechanisms while resources are shared among jobs in MapReduce. In resource stealing, jobs are not reordered or moved in the queue and stealing is done at task level without impacting job scheduling at all, so it is a finer-grained and lower-level optimization of resource usage.

III. Our Approaches

A. Resource Stealing (RS)

Motivation: How to tune Hadoop parameters automatically has been studied in [16][17]. In this paper, we assume the number of task slots is set optimally so that ideal resource utilization is achieved when all slots are occupied. Resource utilization is proportional to the number of occupied slots approximately. Usually, the utilization of real data centers is low. For example, according to the real traces of production clusters, CPU utilization was 5%–10% in Yahoo's M45 cluster [18] and below 50% mostly in a multi-thousand node Google cluster [19]. The low utilization may be caused by several factors. There may not be sufficient workload to keep the whole cluster busy. Or most of the submitted jobs are disk- or IO-bound so that CPU is not the bottleneck. In addition, resource utilization varies across time periods. It implies idle slots exist in most large systems and the capability of resources can be further exploited to minimize job run 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 resource availability is guaranteed whenever a new task is assigned. However, a drawback is that residual resources are left unused until new tasks are assigned.

Resource stealing: 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 on a slave node, its running tasks temporarily steal resources reserved for prospective tasks so that residual resources are fully utilized. If a node is perfectly loaded after resource stealing is applied, to assign a new task obviously will overload it and degrade the performance of running tasks. Our solution is to shrink the resource usage of running tasks by making them relinquish stolen resources proportionally to new tasks. In this way, resource stealing does not violate the assumption made by Hadoop scheduler 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 occupy those slots. Resource stealing is applied on each slave node locally. From the perspective of the central task scheduler running on the master node, idle slots on slave nodes are still available 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 existing Hadoop scheduler directly such as fair scheduler and capability scheduler. Resource stealing runs 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.

B. Allocation Policies of Residual Resources

Given residual resources and the number of running tasks on a slave node, the next issue is how to distribute residual resources among running tasks, i.e. which tasks should get how much. The policies can range from simple to complex in their use of system state information. Complex policies

Table I
ALLOCATION POLICIES OF RESIDUAL RESOURCES

Policy	Description
Even	Evenly allocate residual resources to tasks.
First-Come-Most	The task that starts earliest is given residual resources.
Shortest-Time-Left-Most	The task that will complete soonest is given residual resources (on a per-node basis).
Longest-Time-Left-Most	The task that will complete latest is given residual resources (on a per-node basis).
Speculative-Tasks-Most	Speculative tasks are given residual resources.
Laggard-Tasks-Most	Straggler tasks are given residual resources.

have the potential to take full advantage of the processing capability of each slave 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 in Table I.

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 [9] which assumes each task progresses at a constant rate across time and predicts the time left based on prior 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. This policy is applied locally on individual slave nodes.

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

Speculative-Task-Most (STM): Speculative execution 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 job run time. Because speculative tasks obtain more resources, they can run faster and will not be "stragglers" any longer with high probability. If there are no speculative tasks on a node, it falls back to the regular case and any other policy can be applied. If there are multiple speculative tasks running on a node, residual resources are allocated to them evenly.

Laggard-Task-Most (LTM): This policy does not distinguish between regular 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 running tasks using (1) where T is a task, t is a time point, $N_l(t,T)$ is the number of tasks that are expected to complete later than T, and $N_r(t)$ is the number of running tasks. Fastness reflects the expected order of task completion for each job; and a task with small fastness will complete after a task with large fastness.

$$fastness(t,T) = \frac{N_l(t,T)}{N_r(t)} \tag{1}$$

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. The master node maintains the statuses of all tasks and thus is the ideal component to compute fastness. Each slave node reports the statuses (e.g. progress, failure) of its running tasks to the master node in heartbeat messages. After collecting the information of all the tasks of a job, the master node calculates the fastness of each running 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.

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 useless. 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 [9]. 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 run 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 substantially different amounts of computation to solve. Speculative execution is not helpful because the efficiency variation is not mainly caused by extrinsic factors (e.g. faulty nodes) and the run 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 with small fastness. LTM speeds up their execution by assigning more resources. By balancing the workload within each job, LTM reduces both job run time and the number of speculative tasks. The assignments of new tasks decrease the amount of residual

Table II
PROBABILITY OF ACHIEVING DATA LOCALITY

RF*	regular task	speculative task
1	p	$q \cdot p$
2	$1-q^2$	$(q^2+p)\cdot(1-q^2)$

Table III
ENERGY CONSUMPTION OF REGULAR AND SPECULATIVE TASKS (DATA ACCESSING)

RF*	regular task	speculative task
1	$p \cdot E_L + q \cdot E_{NL}$	$q \cdot p \cdot E_L + (1 - q \cdot p) \cdot E_{NL}$
2	$(1-q^2) \cdot E_L + q^2 \cdot E_{NL}$	$(q^2+p)(1-q^2) \cdot E_L + (1-(q^2+p)(1-q^2)) \cdot E_{NL}$

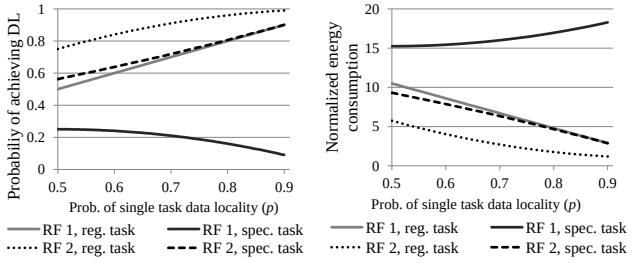
* RF: Replication factor

resources while the completion of running tasks increases the amount of residual resources. They both trigger the re-allocation of residual resources.

C. 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) detecting slow tasks; ii) choosing 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 much slower than average rate ($\frac{5+1}{2}$=2.5), the master node decides to start a speculative task A' which progresses with rate 5. By doing a little math, we can easily figure out that task A' will complete later than task A although task A' progresses much faster. 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 as the original tasks completed earlier. Those speculative tasks were not beneficial and their execution resulted in the inefficiency of resource usage, so they are termed *Non-Beneficial Speculative Tasks* (NBST).

One argument is that given idle resources in most clusters speculative execution should be applied aggressively to eliminate the impact of straggler tasks and thus accelerate job execution. From the perspective of pure performance, aggressive application of speculative execution can potentially reduce overall job run time. However this argument bears several pitfalls. For regular and speculative tasks, the probabilities of achieving data locality are different. Hadoop randomly places data blocks among nodes and thus eliminates hot spots for most workload in multi-user environments. For an individual job, let p denote the average probability that a task can achieve data locality. According to the trace shown in [20], we vary p between 0.5 and 0.9 and use q to denote $1-p$. Table II shows how likely a regular task and its speculative task achieve data locality for different replication factors, which is visualized in Fig. 1 (a). Obviously speculative tasks have lower probability to achieve data locality than regular tasks by up to 90% and 25% for replication factor 1 and 2 respectively. The degradation of data locality results in more data movement



(a) Data locality (b) Energy consumption

Figure 1. Comparison of regular and speculative tasks

and network traffic. Besides, energy consumption is also increased. Some studies show inter-node data movement requires much more energy than local memory accessing by 10 to 20 times [21][22]. Table III shows the expected energy consumption where E_L and E_{NL} are the amounts of energy consumed by the accessing of local and non-local data respectively. We set the ratio of E_{NL} to E_L to 20 and plot the results in Fig. 1 (b) from which we can see speculative tasks are much less efficient. Adopting a large replication factor can potentially mitigate the problem but requires more storage space and incurs higher data management overhead. Thus, it is cost prohibitive to blindly create an excessive number of speculative tasks without evaluating their potential benefit. What we want to achieve is to eliminate speculative execution as much as possible without sacrificing performance.

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: i) the progress rates of other tasks of the same job; ii) 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 [9]. Given a slow task T of job J and a slave node N_i, the following algorithm solves the problem whether a speculative task T' should be launched for T on N_i.

1) If some tasks belonging to job 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, computed based on hardware

processing power, is needed to eliminate the effect of hardware heterogeneity. 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' on N_i. We assume the scheduling order of tasks is stochastic approximately and thus the mean of the progress rates of scheduled tasks reflects the mathematical expectation of real progress rate, which is reasonable given Hadoop scheduling strategy.

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 $1/progress\ rate$. If the difference of the estimated execution time of T and T' is larger than a preset threshold, 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.

Our algorithm considers the heterogeneity of hardware capability of slave nodes. In step 2 above, the progress rate of T' is estimated based on that of all other tasks. For the less typical cases where the disparity of progress rates is mainly caused by node-specific failure or software stack, a more complex heuristic can be designed. For node N_i, a set of "similar" nodes would be selected for the estimation of the progress rate of T'. Node similarity would be calculated based on normalized task progress rates. The basic idea is the execution time of a task would be predicted based on the information of peer tasks running on only similar nodes. We plan to investigate it in more detail in future work.

D. Implementation

To utilize residual resources, the parallelism of running tasks needs to be increased. Multithreading technique has been adopted by us to fully exploit the capability of modern servers. In Hadoop, each task is run in a separate process to isolate its execution environment. Fig. 2 shows an example. There are two slave nodes each of which has 5 cores. One core is dedicated to running various Hadoop daemons. Fig. 2 (a) shows an instant state of a MapReduce system. Each node has 4 slots among which 2 slots are idle. For node A, slots A_1 and A_2 are occupied while slots A_3 and A_4 are idle. In Hadoop, each task process only runs one thread to process data even if there are lightly-utilized cores, IO subsystems or network resources. So the resources corresponding to slots A_3 and A_4 are left idle. Resource stealing starts multiple threads within a task process that process input data in parallel. In Fig. 2 (b), one extra thread (marked in black) is created for both A_1 and A_2, and there are four threads total after applying resource stealing on node A. Each thread can be scheduled to an individual core. *Task manager* periodically adjusts the number of threads based on the latest system state.

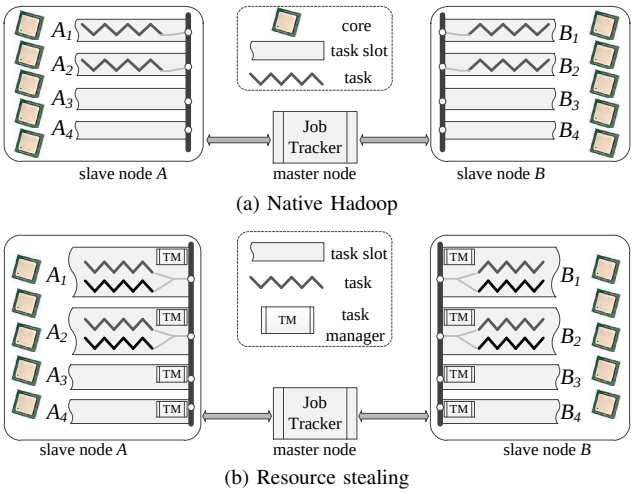


Figure 2. An example of native Hadoop scheduling and resource stealing

Resource stealing and BASE are transparent to end users. Regular MapReduce applications can be run directly without any modification. Additional configuration parameters are added to allow administrators 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.

Although our implementation is based on Hadoop, our algorithms are not specific to Hadoop and can be applied to other systems as well that adopt resource "partition" and speculative execution.

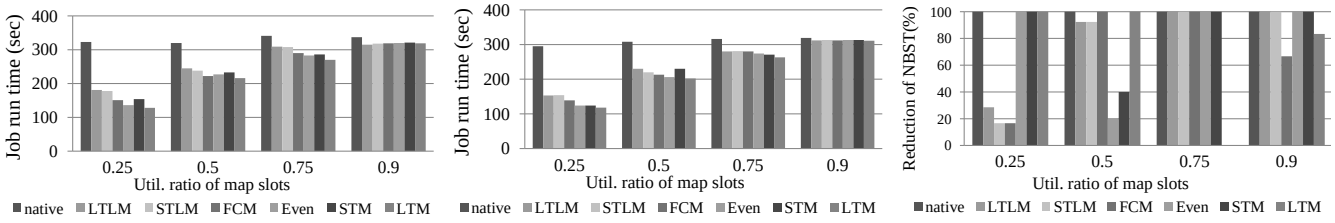
IV. Experiments

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. We adopt the trace-based workload used by other MapReduce papers [20][23][24] in addition to some real applications. Compute-, IO- and network-intensive workload is used in our tests below. The results are applicable to not only those tested applications per se but also other applications of the same types.

We ran experiments on FutureGrid Hotel cluster which is homogeneous in both hardware and software. A 33-node Hadoop system was deployed among which 1 node was a dedicated master node and the other 32 nodes were slave nodes. Each node has 8 cores, 23 GB memory, and 1 local hard drive. The network is Gigabit Ethernet. 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 224 map slots and 224 reduce slots total. HDFS block size was set to 258 MB because this improved performance.

A. Results for Map-Mostly Compute-Intensive Workload

According to the study in [18], a large portion of MapReduce jobs (over 71%) are map-only. In this section, we ran two map-mostly applications: *ts-map* and *PSA-SWG*.



(a) Job run time without BASE (b) Job run time with BASE (c) Reduction of NBST‡ with BASE(%)

Figure 3. Results for *ts-map* (‡NBST: Non-Beneficial Speculative Tasks)

1) ts-map: Firstly, we adopted the text extraction application included in Hive benchmark [23] which was adopted by Zaharia et al's delay scheduling as well [20]. The benchmark itself is based on Pavlo et al's benchmark that compares MapReduce and parallel SQL database management systems [24]. The text extraction job is IO-intensive and termed *ts*. We modified it by applying a compute-intensive User Defined Function (UDF) to each input record in addition to the original text extraction operation, which made it run much longer. This strategy was also used by delay scheduling [20]. This modified version of *ts* is compute-intensive and termed *ts-map* by us. It does not have a reduce phase so that we can eliminate the impact of shuffling and merging and exactly measure the effectiveness of resource stealing for map-only jobs. In our tests, only 0.01% of input records are extracted. Each map task processed 256 MB text data and was tuned to run approximately for 5 minutes to simulate the interactive job types in Google's MapReduce [2], which was also adopted by Zaharia et al. to test their scheduling algorithm LATE [9]. Multiple *ts-map* jobs were run sequentially with the size of input data varied between 14 GB, 28 GB, 42 GB, and 50.5 GB. The corresponding numbers of map tasks were 56, 112, 168, and 202 respectively, which yielded system utilization 25%, 50%, 75% and 90%.

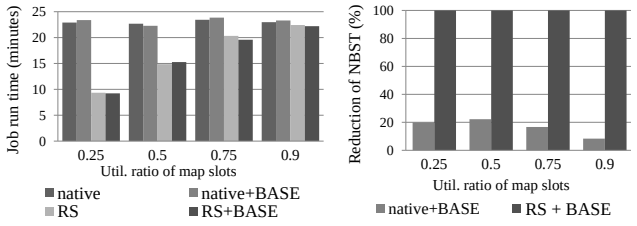
***ts-map* without BASE:** We ran *ts-map* without BASE and show job run time in Fig. 3 (a). Run time is not significantly influenced by workload for native Hadoop, which means that processing 14GB, 28GB, 42GB, and 50.5GB data takes similar amounts of time. The reason is resource usage is proportional to the number of tasks and residual resources are not utilized at all. Resource stealing shortens run time by 58%, 29%, 17%, and 5% 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 run 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, which implies that it benefits to have global knowledge of job execution. When the workload gets relative high (e.g. 90%), the performance difference becomes smaller.

In our setup, data blocks in HDFS were randomly placed on nodes with the default block placement strategy. Data locality aware scheduling in Hadoop co-locates compute and data with best efforts. As a result, map tasks were evenly distributed among all slave nodes approximately so that each node ran a similar number of tasks. That is beneficial to resource stealing because its gain is not substantial if the resources of a node are heavily loaded already (i.e. there are no resources to steal).

***ts-map* with BASE:** We ran the same tests as above except BASE was enabled and present results in Fig. 3 (b). The plot has similar characteristics to Fig. 3 (a) in that native Hadoop performs the worst and the performance superiority of resource stealing decreases with increasing sytem workload. By comparing 3 (b) and 3 (a), we observe that BASE slightly shortens run time. LTM again yields the best performance.

For the cases where BASE is disabled and enabled, we calculated the percent of non-beneficial speculative tasks eliminated by BASE and show the result in Fig. 3 (c). BASE drastically eliminates the launches of non-beneficial speculative tasks. For workload 75% and 90%, almost all of them are removed. LTM policy performs well constantly.

2) Pairwise Sequence Alignment(PSA) with Smith-Waterman-Gotoh Algorithm (SWG): SWG is a well known algorithm for performing local sequence alignment [25]. We ran our PSA-SWG implementation, which aligns all pairs of input sequences, in Hadoop with 16S rRNA sequences from the NCBI database. The number of processed sequences was set to 4676, 6608, 8064, and 8888. Input sequences were partitioned in a way that balances load and makes each job run for between 20 and 25 minutes with native Hadoop scheduling. Job run time is shown in Fig. 4 (a), from which similar observations as *ts-map* tests can be made. Resource stealing speeds up job execution by 2.45x, 1.53x, 1.15x, and 1.03x respectively. The impact of BASE on job run time is negligible. Fig. 4 (b) shows the percent of non-beneficial speculative tasks eliminated by BASE compared to native Hadoop without BASE. Applying BASE alone achieves 10%–20% improvement while applying BASE and resource stealing simultaneously eliminates all non-beneficial speculative tasks. With dynamic parallelism adjustment, resource



(a) Job run time (b) Reduction of NBST

Figure 4. Results for PSA-SWG

stealing can smooth out the intermittent fluctuation of task execution and thus make BASE able to predict run time more accurately.

We conclude that BASE reduces the number of non-beneficial speculative tasks significantly without sacrificing run time. Because a fewer number of speculative tasks are launched, the saved resources can be allocated to regular tasks for better efficiency. It also implies that the estimation of run time is approximately accurate so that BASE rarely removes the runs of beneficial speculative tasks.

B. Results for Compute-Intensive Workload with Stragglers

In this experiment, background load is generated to slow down some nodes and simulate straggler nodes. We developed a load generator that can generate user-specified load of computation, network and disk IO. We ran multiple CPU-hogging threads yielding high core utilization, and IO-intensive threads reading/writing data continuously from/to local disks. The background load significantly slowed down the nodes without rendering them thoroughly unresponsive. We ran *ts-map* jobs that utilize 75% of all map slots and thus 42GB data was processed total in each run.

Firstly, two slave nodes were slowed down and job run time is shown in Fig. 5 (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. 5(b) shows BASE can save runs of nearly all unnecessary speculative tasks, which implies the estimation of run time is accurate when only a small number of slave nodes are stragglers.

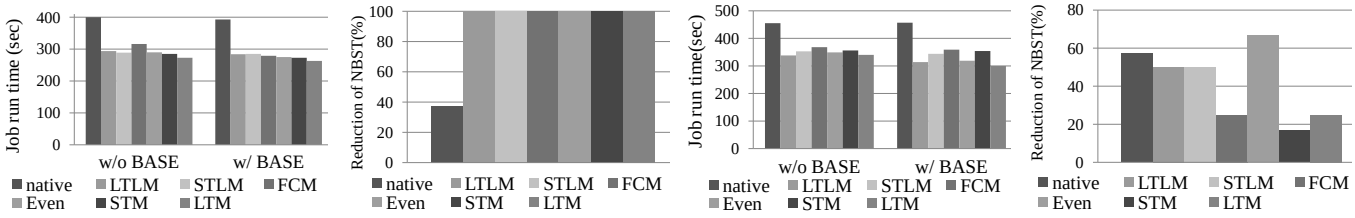
Secondly, four slave nodes were slowed down. Fig. 5 (c) shows job run time. The jobs ran longer compared with the previous test because more map tasks were impacted. Resource stealing is still effective to speed up job execution. Policies LTM, Even, and LTLM yield the best performance. Fig. 5 (d) shows BASE can eliminate 20%–50% of non-beneficial speculative tasks. Compared with the previous case, BASE becomes less effective. It indicates our estimation of run time gets inaccurate as more straggler nodes incur larger variation of task execution. To make the estimation of run time adaptive to the change of system state is part of our future work.

C. Results for Reduce-Mostly Jobs

The trace analyzed in [18] shows 9% jobs are reduce-mostly. In this test, we experimented with two reduce-mostly applications: *ts-reduce* and matrix multiplication.

ts-reduce: Firstly, we modified the original text extraction application by plugging a compute-intensive UDF in reduce operation. The new application is called *ts-reduce*. The numbers of map and reduce tasks were set according to the fact that most MapReduce jobs tend to have significantly lesser reduce tasks than map tasks. For resource stealing, only policy LTM was evaluated because it performs among the best based on the results above. Each job had 101 map tasks and the number of reduce tasks was varied between 32 and 64. Each job ran for 5-10 minutes approximately. Job run time is shown in Fig. 6 (a). Resource stealing substantially shortens job run time by 71% and 44% respectively. Resource stealing thoroughly eliminated non-beneficial speculative tasks. When each job contained 32 reduce tasks, they were well spread out so that each node ran one reduce task at most on average. For each reduce task, resource stealing created 6 new reduce tasks (note the number of reduce slots is 7 on each node) to run in parallel, which should yield 7x optimal speedup. In reality, we only got 4x speedup approximately because of additional overhead. Reduce threads contend for the same input stream and only one thread can read at any time. To alleviate the issue, 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 copying. In addition, reduce threads belonging to the same task contend for the same output stream as well. To investigate advanced techniques to mitigate contention further is among future work. We also measured the number of speculative tasks. The results are not shown because of space limit. BASE shortens job execution marginally, but reduces the number of non-beneficial speculative tasks by up to 90% compared to native Hadoop.

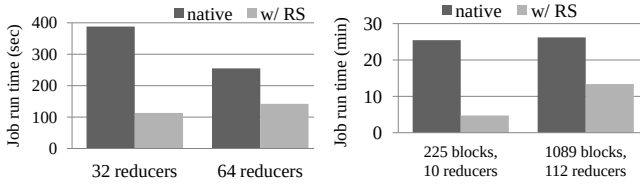
Matrix multiplication: Li et al. used matrix multiplication to evaluate Dryad in [26]. We implemented it in Hadoop using the simple three-loop approach and call it *mm*. Input matrices were stored in HDFS. Map tasks only split the input matrices into smaller blocks and thus do not run real computation. Each reduce task calculates a number of final output blocks. Most of the work is done by reduce tasks, which makes *mm* reduce-mostly. The distribution of real matrix multiplication operations among reduce tasks is controlled by a partitioner. Our initial tests showed the default hash partitioner yielded substantial workload skew and resulted in severe load imbalancing. So a custom partitioner has been implemented by us to make total work distributed more evenly. We ran two tests in which the size of each block was 750 x 750. In the first test, the number of reduce tasks was set to 10 and the size of each input matrix was 11.25k x 11.25k, so each matrix was split into 225 blocks ($(11.25k/750)^2 = 225$) and each *mm* run



(a) Job run time (b) Reduction of NBST w/ BASE(%) (c) Job run time (d) Reduction of NBST w/ BASE(%)

Figure 5. Results for *ts-map* with straggler nodes. There are two straggler nodes for (a) and (b), and four straggler nodes for (c) and (d)

read 20 GB data total from HDFS. In the second test, each job ran 112 reduce tasks and the size of each input matrix was 24.75k x 24.75k, so each matrix comprised 1089 blocks($(24.75k/750)^2 = 1089$) and each *mm* run read 210 GB data total. The total number of reduce slots was 224 in the system, so those two tests approximately achieved utilization 5% and 50% respectively. Fig. 6 (b) shows the average job run time. Resource stealing yielded 5.4x and 2x speedup respectively, which results from the opportunistic exploitation of the processor cores "reserved" for non-occupied reduce slots.



(a) modified text extraction - *ts-reduce* (b) matrix multiplication - *mm*

Figure 6. Results for reduce-mostly applications

D. Results for 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*, whose 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 containing approximately 500 million unique URLs [27]. We selected a portion of it randomly as the input of *mr-wc*. If most pages downloaded by each task are hosted by the same server, the variation of server response time can result in the severe skew of crawling tasks. To mitigate the issue, we shuffled the input URLs so that each task fetched pages from different domains and workload was better balanced. The same testbed as above was used. In our tests, each map task downloaded 2000 web pages and the total number of download pages was 112k, 224k, 336k and 404k, which means the number of map tasks was 56, 112, 168 and 202 respectively. So the system workload was 25%, 50%, 75% and 90% respectively. Fig. 7 shows job run time. For native Hadoop, the run time of *mr-wc* is not significantly impacted by the workload, which implies spare resources cannot be utilized. In contrast, resource stealing expands the used resources of running tasks by creating more threads to download webpages in parallel. Run time is shortened drastically by 66%, 37%, 24% and 15% 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.

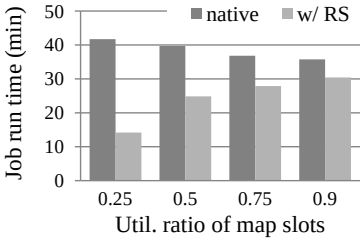
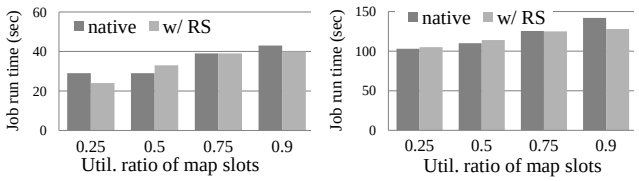


Figure 7. Results for network-intensive workload *mr-wc*

IO-Intensive Workload: In this test, applications *word-count* and text extraction *ts* are used. Application *ts* has been described above. *Wordcount* counts the number of word occurrences in input text. Again each map task processed 256MB text and the number of map tasks was varied. Fig. 8 shows the results. For both applications, as the number of map tasks increases, job run time increases as well and parallel efficiency deteriorates, which is caused by the increasing resource contention. However, with more data processed, the processing throughput (the amount of processed data per unit of time) is improved significantly, which results from higher task-level parallelism. Resource stealing performs only slightly better than native Hadoop scheduling, although resource stealing achieves higher parallelism within each task. It is caused by a limitation of Hadoop implementation. In map tasks, each map operation processes one line of text and is invoked repeatedly. Although resource stealing enables Hadoop to run more map operations in parallel, these threads share the same underlying input reader and output writer (to comply with Hadoop design). This incurs substantial contention among threads because the computation time of each map operation is short and synchronization is the performance barrier. As a result, the overhead is comparable to the benefit of higher concurrency brought by resource stealing. This inefficiency is not intrinsic to our approach and



(a) text extraction application *ts* (b) *wordcount*

Figure 8. Results for IO-intensive workload

mainly pertains to Hadoop design. Theoretically moderate increase of I/O parallelism can exploit the interleaving of computation and data I/O and improve overall throughput.

V. Conclusions

The goal of our work is to improve resource utilization in MapReduce. We present resource stealing to dynamically re-allocate residual resources to running tasks with the promise that they will be handed back whenever required by newly assigned tasks. It can be applied in conjunction with existing job schedulers smoothly because of its transparency to central Hadoop 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 substantial 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, performance improvement is not substantial, which is caused by accessing contention in Hadoop framework. In future, we will investigate lock-free data structures and algorithms to mitigate the issue. Currently we assume all jobs have the same priority and thus treat all tasks equally. We plan to integrate job prioritization to make our algorithms comply with the fair sharing policies.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 0910812.

References

[1] W. Gropp, et al. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1994.

[2] J. Dean, et al. "MapReduce: Simplified Data Processing on Large Clusters," in *Proc of OSDI '04*, 2004, pp. 137–150.

[3] X. Qiu, et al. "Cloud technologies for bioinformatics applications," in *Proc of MTCGS '09*. ACM, 2009.

[4] C. T. Chu, et al. "Map-reduce for machine learning on multicore," *NIPS '07*, vol. 19, p. 281, 2007.

[5] M. Isard, et al. "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007.

[6] Y. Gu, etal. "Sector and Sphere: the design and implementation of a high-performance data cloud." *PTRS - Series A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1897, pp. 2429–2445, 2009.

[7] L. Gwennap, "New algorithm improves branch prediction," *Microprocessor Report*, vol. 9, no. 4, pp. 17–21, 1995.

[8] I. Ahmad, et al. "A New Approach to Scheduling Parallel Programs Using Task Duplication," in *Proc of ICPP '94*. IEEE Computer Society, 1994, pp. 47–51.

[9] M. Zaharia, etal. "Improving MapReduce performance in heterogeneous environments," in *Proc of OSDI '08*. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.

[10] R. D. Blumofe, et al. "Scheduling multithreaded computations by work stealing," in *Proc of the 35th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 356–368.

[11] S. N. Bhatt, et al. "On Optimal Strategies for Cycle-Stealing in Networks of Workstations," *IEEE Trans. Comput.*, vol. 46, pp. 545–557, May 1997.

[12] Z. Guo, et al. "Automatic Task Re-organization in MapReduce," in *Proc of Cluster '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 335–343.

[13] J. Ekanayake, et al. "Twister: a runtime for iterative MapReduce," in *Proc of HPDC '10*. ACM, 2010, pp. 810–818.

[14] G. Ananthanarayanan, et al. "Reining in the outliers in map-reduce clusters using Mantri," in *Proc of OSDI '10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

[15] A. W. Mu'alem, et al. "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," *IEEE TPDS*, vol. 12, no. 6, pp. 529–543,2001.

[16] K. Kambatla, et al. "Towards optimizing hadoop provisioning in the cloud," in *Proc of HotCloud '09*.

[17] H. Herodotou, et al. "Starfish: A self-tuning system for big data analytics," in *Proc of CIDR'11*

[18] S. Kavulya, et al. "An Analysis of Traces from a Production MapReduce Cluster," in *Proc of CCGrid '10*. Washington, DC, USA: IEEE Computer Society, May 2010, pp. 94–103.

[19] L. A. Barroso, et al. "The Case for Energy-Proportional Computing," *Computer*, vol. 40, pp. 33–37, Dec. 2007.

[20] M. Zaharia, et al. "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc of EuroSys '10*. ACM, 2010, pp. 265–278.

[21] K. Bergman, et al. "Let there be light!: the future of memory systems is photonics and 3D stacking," in *Proc of MSPC '11*. New York, NY, USA: ACM, 2011, pp. 43–48.

[22] "Advanced Memory Systems for HPC," http://exascaleresearch.labworks.org/ascrOct2011/uploads/dataforms/PRES_SNL_AdvMem4HPC_111011.pdf

[23] "Hive performance benchmarks," https://issues.apache.org/jira/browse/HIVE-396

[24] A. Pavlo, et al. "A comparison of approaches to large-scale data analysis," in *Proc of SIGMOD '09*. New York, NY, USA: ACM, 2009, pp. 165–178.

[25] O. Gotoh, "An improved algorithm for matching biological sequences." *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, Dec. 1982.

[26] H. Li, et al. "Design patterns for scientific applications in DryadLINQ CTP," in *Proc of DataCloud '11*, ser. DataCloud-SC '11. ACM, 2011, pp. 61–70

[27] "Clueweb09," http://lemurproject.org/clueweb09.php/.