Improving Resource Utilization in MapReduce

Zhenhua Guo, Geoffrey Fox, Mo Zhou

School of Informatics and Computing

Indiana University Bloomington

{zhguo, gcf, mozhou}@cs.indiana.edu

*Abstract*— MapReduce has been adopted widely in both academia and industry to run data parallel applications that process large amounts of data. On each node, the parallelism of multi-core processors is explored by running multiple map/reduce tasks concurrently. Conceptually, each node hosts a number of map and reduce slots to which tasks can be assigned. So they limit the maximum number of tasks that can execute concurrently on each node. When all slots of a node are not used, some slots keep idle and therefore the corresponding resources are wasted. To improve resource utilization, we propose resource stealing which enables 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 create 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 (BASE) which evaluates the potential benefit of starting speculative tasks and eliminates the unnecessary launches of speculative tasks. We implement our proposed algorithms in Hadoop and conduct a series of experiments to show that our algorithms can significantly shorten job execution time and reduce the number of non-beneficial speculative tasks.

Keyword: MapReduce, scheduling, speculative execution

# Introduction

Data deluge has been witnessed in many science areas such as particle physics, astronomy, and biology. It requires a significant amount of compute capacity to process the data in addition to storage space. Message Passing Interface (MPI) [15] has been used widely in High Performance Computing (HPC) as a programming model. For data parallel applications, MapReduce [1] was proposed and has gained popularity in both academia and industry. It has been used in bioinformatics [6], machine learning [7], etc. Hadoop is a widely-used open source implementation of MapReduce and therefore our research target. Besides MapReduce, other data parallel systems including Dryad and Sector/Sphere have been developed with different features.

One significant difference between MapReduce and traditional HPC grid systems is the architecture. In traditional HPC architecture, storage nodes and compute nodes are separated and the scheduling strategy is to bring data to compute. In data-intensive computing, data movement incurs dramatic load on network and storage which may dominate the overall application execution. In MapReduce, each node takes the roles of both storage and computation, which enables the scheduler to bring compute to data and therefore is optimized for data-intensive applications.

In distributed systems, failure is the norm rather than the exception. Speculative execution is adopted by MapReduce. The master node keeps track of the progresses of all scheduled tasks of each job. When it finds a task that runs unusually slow compared with other tasks of the same job, a speculative task is created and launched to process the same input data. The best outcome is that a speculative task completes earlier than the corresponding speculated task, which implies that the execution of the speculative task is beneficial. Otherwise, the speculative task is unnecessary and does not benefit the overall job execution. So the number of non-beneficial speculative tasks should be minimized and therefore the amount of wasted resources is minimized. However, in Hadoop the progress rate is the only considered factor when a decision is made regarding whether to trigger speculative execution or not. For example, assume a job has two tasks *A* and *B*. Task *A* is 90% done but progresses slowly with rate 1. Task *B* progresses fast with rate 5. Obviously the progress rate of *A* is much lower than that of *B*. The master node decides to start a speculative task for *A* and it progresses with rate 5. By doing a little math, we can easily figure out that task *A* will complete sooner than the speculative task although *A* progresses slowly. The reason is that task *A* is close to completion when its speculative task is launched. 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 speculated tasks.

The rest of this paper is organized as follows. Related work is discussed in sector II. The details of our proposed resource stealing algorithm and BASE are discussed in sector III. The experiments we have conducted and their results are discussed in sector IV. Finally we conclude in sector V.

# Related Work

The term *speculative execution* has been used in different contexts. For example, at instruction level, branch predictors [12] guess which branch a conditional jump will go to and speculatively execute the corresponding instructions; and automatic I/O hint generation [13] 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 [11] redundantly executes some tasks on which other tasks critically depend to shorten overall execution time. 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) [2] 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 [8] 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 idle workstations to take control of idle workstations, supply them with work, and receive results. The motivation is to harness the otherwise wasted capacity of idle workstations. Our proposed resource stealing shares similar motivations but is applied at lower level across resources located on a single workstation.

In grid systems, batch scheduling has been used extensively. Usually, when a job is scheduled, a 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 [3] 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.

# Our Approaches

## Background

There are two levels of parallelism in MapReduce. Firstly, tasks run on more than one node concurrently. Secondly, each node can run multiple tasks concurrently to fully explore the processing capability of modern multi-core processors. To limit the number of concurrently running tasks on each node and therefore avoid intense resource contention, MapReduce makes each node host a configurable number of map and reduce slots where tasks can run. By default, each task can only be assigned to one slot and each slot can accept one task at most. A slot gets *occupied* when a task is assigned to it; and gets *released* when the task completes. There has been some work on how to tune Hadoop automatically [16, 17]. In this paper, we assume that the parameters are set optimally, which means optimal resource utilization is achieved when all slots are occupied. According to the trace of production clusters [18], resource utilization is way below 100% and varies across time. For instance, resource utilization during peak hours is significantly higher than that during off-peak hours. So the usual scenario of Hadoop systems is that some slots are idle while others are running map or reduce tasks. It implies that the capability of resources is not fully exploited to minimize execution time. Resource utilization is proportional to the number of occupied slots approximately. The more slots are occupied, the higher resource utilization becomes. The proportion of the resources that sit idle on a slave node is termed *residual resources* which can be utilized without incurring severe usage contention and degrading overall performance.

## Resource Stealing

Assume there are *N* nodes in a Hadoop system. For node *i,* the numbers of map and reduce slots are *MSi* and *RSi* respectively; and the numbers of occupied map and reduce slots are *OMSi* and *ORSi* respectively. Apparently *OMSi* and *ORSi* are also the numbers of map and reduce tasks running on node *i*. A node *i* is *map-underutilized* if it has idle map slots (i.e. *OMSi* is less than *MSi*) and *reduce-underutilized* if it has idle reduce slots (i.e. *ORSi* is less than *RSi*). We can consider that residual resources are reserved by Hadoop for prospective tasks that will be assigned to currently idle slots. One advantage of resource reservation is that whenever a new task is assigned it can immediately start to run without worrying about resource availability. However, an obvious drawback is the waste of resource because residual resources are left unused until new tasks are assigned to use them.

Instead of wasting residual resources, for each node we make running tasks (if any) expand resource usage dynamically to utilize residual resources so that all resources are fully used. We term it *resource stealing* which means running tasks temporarily steal resources reserved for prospective tasks. Resource stealing is not known to the job scheduler. From the perspective of job scheduler, the idle slots are still idle and new tasks can be assigned to them. However, if all resources of a node have been fully utilized by using resource stealing, to run 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 a portion of stolen resources proportionally. In this way, resource stealing does not violate the assumption that is critical to efficient Hadoop task scheduling. To summarize, the overall philosophy is to steal residual resources if corresponding map/reduce slots are idle, and return them back whenever new tasks are launched to fill the idle slots. Resource stealing can be used in combination with any Hadoop scheduler directly such as fair scheduler and capacity scheduler. Resource stealing is applied with the up-to-date information of task execution and system status periodically so that it can adapt to changes in real time.

Resource stealing shares similar idea with backfilling [3]. Backfilling moves small jobs ahead to fill holes under the condition that they will not delay the execution of large jobs in front significantly. So backfilled jobs are “formally” assigned to compute resources, which may influence the scheduling of subsequent jobs. In contrast, resource stealing makes opportunistic use of idle resources without influencing the task/job scheduling. Resource stealing can be applied to backfilling by preempting backfilled jobs if they do not complete when the required resources of jobs at the front of wait queue become available.

## Allocation Policies of Residual Resources

Given residual resources and the number of running tasks on a node, the next issue is how to distributed residual resources among running tasks, e.g. which tasks should get more and which tasks should get less. We come up with several strategies summarized in table I.

1. Resource Allocation Policies

|  |  |
| --- | --- |
| Strategy | Description |
| Even | Evenly allocate residual resource to tasks |
| First-Come-Most | The task that starts earliest is given residual resource. |
| Shortest-Time-Left-Most | The task that will complete soonest is given residual resource. |
| Longest-Time-Left-Most | The task that will complete latest is given residual resource. |
| Speculative-Task-Most | Speculative tasks are given residual resource. |
| Laggard-Task-Most | Straggler tasks are given residual resource. |

**Even**: This strategy equally divides residual resources among running tasks.

**First-Come-Most** (FCM): This strategy orders running tasks by start time. The task with the earliest start time is given residual resources. The heuristic is to respect the execution order and make tasks complete in FIFO order with best efforts.

**Shortest-Time-Left-Most** (STLM): In this strategy, the remaining execution time of tasks is estimated, where different mechanisms can be plugged in. Here we adopt the same mechanism used in [2] which assumes each task progresses at a constant rate throughout time and predicts the time left based on progress rate and current progress (shown in (1) and (2)). The task with the shortest time left is given residual resources. The heuristic is to make short tasks complete as soon as possible and therefore release resources that can be allocated to long-running tasks.

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

 

 

**Speculative-Task-Most** (STM): In Hadoop, fault tolerance is supported by adopting speculative execution mechanism. Hadoop constantly monitors the progress of all running tasks. If it detects that a task runs abnormally slow compared with peer tasks from the same job, it starts a backup task to duplicate the processing being done by the slow task and hopes the new task will complete sooner. So speculative execution aims to mitigate the impact of stragglers on overall job execution. The basic idea of STM strategy is that speculative tasks are given more resources than other 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 strategies can be applied. If there are multiple speculative tasks running on a node, residual resources are allocated to them evenly. Algorithm sketch is shown below.

Algorithm skeleton

**Input**: Statuses of running tasks on a node

**Output**: resource assignment to tasks

**Algorithm**:
 TS 🡨 The set of running tasks on a node

 ST 🡨 ∅ # speculative tasks

 for T in TS:

if T is a speculative task:

 ST = ST ∪ T

 if ST is not empty:

 allocate residual resource to tasks in ST

else

 fall back to other allocation policies

**Laggard-Task-Most**: In this approach, we do not distinguish between regular tasks and speculative tasks. Instead, for each job we use estimated time left of all its scheduled tasks (including both regular and speculative tasks) to calculate the *fastness* of a running task *T* using (3). It reflects the order of task completion for each job; and a task with small *fastness* will complete later than a task with large *fastness*.

*Fastness* of a task cannot be computed locally by a slave node because it requires the information of other tasks belonging to the same job. Job tracker manages all jobs and 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 job tracker in heartbeat messages; and job tracker calculates *fastness* of those tasks and return it to slave nodes. 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.* Laggards are given most of the 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, completion of any task renders the work done by other duplicate tasks to be wasted. Secondly, if the slowness of speculated tasks is caused by intermittent and temporary resource contention, it is highly likely that they still complete sooner than their speculative tasks, which subverts the motivation of speculative execution. Thirdly, sometimes speculative execution deteriorates performance rather than improve it [2]. LTM reduces the possibility that speculative execution is triggered by proactively allocating more resources to *laggards* whenever possible and therefore accelerating the 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 started. In that case, the tasks requiring the most computation progress slower than other tasks and therefore are the laggards. LTM speeds up the execution of laggards by assigning more resources, and therefore mitigates the load unbalancing problem. 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.

Algorithm skeleton

**Input**: Statuses of running tasks on a node

**Output**: resource assignment to tasks

**Algorithm**:
 TS 🡨 The set of running tasks on a node

 TS 🡨 sort TS by fastness in ascending order

 Laggards 🡨 ∅

 for T in TS:

if fastness(T) < SlowTaskThreshold:

 Laggards = Laggards ∪ T

allocate most resource to tasks in Laggards

## The BASE Scheduler

Speculative execution is not a simple matter of running redundant tasks for sufficiently slow tasks. To make speculative execution effective, two issues need to be addressed: a) detect the tasks that run slow; b) 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 longest remaining execution time to speculate. One drawback that was observed in our experiments is it does not consider the possibility that the speculative tasks will complete before the speculated tasks. In our tests, a large portion of speculative tasks were killed before their completion because the speculated tasks actually completed earlier than them. In other words, those speculative tasks were not beneficial at all and resulted in the waste of resource. So we propose our algorithm BASE for Benefit Aware Speculative Execution. The basic idea is a speculative task is launched only when its completion time is estimated to be earlier than the speculated task. The estimation of the remaining execution time of a running task has been discussed above. Now we need a way to estimate the execution time of prospective speculative tasks. It depends upon two factors: 1) progress rates of other tasks belonging to 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 [2]. Given a slow task *T* of job *J* and a slave node *N*, following algorithm sketch solves the problem whether a speculative task for *T* should be launched on *N*.

1. If some tasks belonging to *J* are running or have run on node N, the mean of their progress rates is calculated and used as the progress rate of the speculative task.
2. Otherwise, progress rates of all scheduled tasks of job *J* are gathered and normalized against the reference baseline. Then we randomly pick a progress rate. Because the chosen progress rate is against the reference, we de-normalize it against the specification of node *N*. The estimated progress rate follows the distribution of real progress rates of scheduled tasks.
3. No matter which of 1) and 2) is applied, the estimated progress rate has been calculated so far. The execution time is estimated by 1 / progress rate. If it is shorter than the remaining execution time of *T*, a speculative task is launched on *N*. Otherwise, do not run speculative task on *N*.

## Implementation

Our implementation is optimized for compute-intensive applications and therefore processors and cores are the critical resources. Multithreading technique is adopted to explore the parallel processing capability of multi-core processors. In Hadoop, each task is run in a separate process to isolate its execution environment. Within each task process, one thread is started up to process data by running user-provided map/reduce implementations. In our improvement, we change Hadoop framework to start multiple threads to concurrently process the data within the process where a task runs. Fig. 1 shows an example. There are two slave nodes each of which has four cores. Each node has four slots among which two slots are idle. For node *A*, Slots *A1* and *A2* are busy; and slots *A3* and *A4* are idle. In Hadoop, each task only runs one thread even if there are lightly-utilized cores (shown in Fig. 1(a)). Instead of wasting resources, we create one extra thread for both *A1* and *A2* (shown in Fig. 1(b)). As a result, we have four threads each of which can be scheduled to an individual core. For each task, *thread manager* periodically adjusts the number of threads dynamically based on the latest status of job execution, so resource stealing is adaptive. It requires some extra information (e.g. whether a task is speculated, whether a task is straggler) that is not sent to slave nodes in Hadoop. We piggyback heartbeat messages between slave nodes and master node to carry the additional information. The interval between consecutive thread manager runs is the same as that of heartbeat message exchange.

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 allocation policy of residual resource.



(a) Native Hadoop



(b) Resource stealing

1. Implementation

# Experiment

We conducted extensive experiments to evaluate our proposed algorithms. Instead of directly measuring resource utilization (e.g. CPU usage), we mainly measure user-perceivable job execution time which indirectly reflects the improvement or deterioration of resource utilization.

## Scheduling of Map-only Jobs

We deployed a Hadoop system on FutureGrid Hotel cluster. The system comprised one master node and twenty slave nodes which were homogeneous in terms of both hardware and software. There are 7 cores and 20GB memory on each node. According to the best practice that the number of slots should be between 1x and 2x the number of cores, each node is 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 is set to 128MB.

We ran *grep* without reduce phase to eliminate the impact of shuffle and merging and exactly measure the effectiveness of resource stealing for map-only jobs. A large portion of MapReduce jobs (over 70% [4]) are map-only jobs. In our tests, each map task processed 128MB 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 [1]. We ran multiple *grep* jobs with the number of map tasks varied. We set the number of map tasks to 35, 70, 105 and 126 which give map slot utilization ratios 25%, 50%, 75% and 90%. We did not test utilization ratio 100% because some slots are reserved by Hadoop to run administrative and maintenance tasks.

**Grep without BASE**: We ran *grep* without BASE and show job execution time in Fig. 2(a). Firstly, we observe that the execution time of jobs without resource stealing is not significantly influenced by the utilization ratio. This implies that processing 4.375GB, 8.75GB and 13.125GB data takes roughly the same amount of time, which demonstrates the inefficiency of Hadoop that resource cannot be fully utilized. The execution time does increase slowly with increased utilization. What is not shown in the figure is when utilization ratio becomes close to 100%, execution time increases sharply by 20%. The reason is that some map tasks became stragglers but job tracker could not start up speculative tasks immediately because all slots were being occupied. Towards the end of job execution, more and more slots were released where speculative tasks could run. In addition, running more tasks concurrently on each node incurred higher contention of resource usage, so that the execution time of each task is increased while the overall throughput is increased. Secondly, comparing the results of running *grep* with and without resource stealing, we can clearly observe that the lower the utilization ratio is, the more resource stealing outperforms Hadoop native implementation. The speed improvement is 64%, 32%, 13%, and 6% respectively for strategy Even. So the performance benefit of resource stealing is negatively related to the utilization ratio, which matches our expectation well. We also calculated the processing time per gigabyte by diving job execution time by data size. Increasing utilization ratio can drastically improve the efficiency for native Hadoop implementation, while it approximately keeps invariant for resource stealing. Thirdly, different allocation strategies 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 while progress-aware policy LTM works better. When utilization ratio gets relative high (e.g. 75%, 90%), the performance difference becomes smaller.

There is a little “luck” here. When the system was underloaded, map tasks were not concentrated on a small number of nodes and left other nodes totally idle. Instead, 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 the gain of resource stealing is not significant if the resources of a node are fully loaded already. The “luck” here is not coincident and is caused by the block distribution and scheduling strategies in Hadoop. In our setup, all nodes were on the same rack and block replicas were randomly placed on nodes. Data locality aware scheduling in Hadoop co-locates compute and data with best efforts. As a result, tasks were evenly scheduled to all nodes.

**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 utilization ratio. The difference between the execution time with and without BASE is calculated and shown in Fig. 2(c). Overall, BASE shortens execution time.

We also summarized the number of “wasted” speculative tasks and computed the difference for the cases where BASE is disabled and enabled. Fig. 2(d) shows the percent of the decrease of the number of wasted speculative tasks. BASE can drastically eliminate the launches of “wasted” speculative tasks. For utilization ratios 75% and 90%, almost all unneeded speculative tasks are removed.



 (a) Job execution time w/o BASE (b) Job execution time w/ BASE (c) Time improvement of BASE (d) Improv. of wasted spec. tasks w/ BASE

1. Run map-only *grep* in a homogeneous environment



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

1. Run map-only *grep* with straggler nodes. Two straggler nodes for (a) and (b). Four straggler nodes for (c) and (d).

Combining Fig. 2(c) and Fig. 2(d), we conclude that BASE reduces the number of wasted speculative tasks significantly while giving shorter execution time. Because a fewer number of speculative tasks are launched, the saved resources can be allocated to normal tasks to speed up the execution. It also indicates that the estimation of execution time is relatively accurate for *grep* so that BASE rarely removes the runs of beneficial speculative tasks.

## Scheduling of Map-only Jobs with Straggler Nodes

In this experiment, we ran background processes on some slave nodes to generate background load to slow down task execution and simulate stragglers caused by faulty nodes and indeterministic process hang. 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. In addition, one IO-intensive thread was run which read data continuously from disk in a sequential manner with Linux direct I/O enabled. The background load significantly slowed down the nodes without rendering them thoroughly unresponsive. If speculative execution is disabled, Hadoop performs drastically worse. So we only consider the case where speculative execution is enabled. We ran *grep* jobs that utilize 75% of all map slots and therefore 8.75GB data was processed total by 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 strategy 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. It 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 case with two stragglers, because more map tasks were slowed down. Resource stealing is still effective to speed up job execution. The performance disparity of different resource allocation strategies 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, we observe that 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.

## Scheduling of Reduce-mostly Jobs

In above experiments, map-only jobs were used. In this test, we will evaluate the effectiveness of our approaches for reduce-mostly jobs for which the execution of reduce phase dominates the overall job execution time. We still use *grep* as the test application. Each job had 20 map tasks each of which processed 128MB data. Each job was configured to run 10 reduce tasks. Operations in reduce tasks are run repeatedly to make reduce phase dominate overall execution. Each job runs for 5 minutes approximately. For resource stealing, only strategy Even is compared below because it is simple and performs among the best based on experiment results above.

We compared job execution time and the number of wasted speculative tasks and present results in Fig. 4. BASE reduces job execution time marginally, but reduces the number of wasted speculative tasks from 10 to 1 when native Hadoop is used. Fig. 4(b) also shows resource stealing thoroughly eliminated wasted speculative tasks, which results in the unnecessity to run tests against the configuration “resource stealing plus BASE”. Because of the drastic reduction of resource waste, more useful tasks can be run concurrently and therefore the efficiency of resource usage is improved. This demonstrates the effectiveness of BASE. Fig. 4(a) shows resource stealing significantly 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 threads (remember the number of reduce slots is 7 on each node) to run in parallel, which optimally should give 7x speedup over native Hadoop. In reality, we only got 4x speedup because of additional overhead. Hadoop provides an iterator style interface for underlying (key, values) sets. Reduce threads compete for the same input stream and only one thread can read from the stream at any time. To alleviate the contention, each thread locks the input stream, copies next (key, values) tuple to its local memory buffer, unlocks the input stream and processes the data in local buffer without interfering with other threads. The drawback of this approach is that it 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) Num. of wasted spec. tasks

1. Run reduce-mostly *grep*

## Experiments with Other Workload

In above experiments, we evaluated compute-intensive applications. We also ran jobs of other types to understand the implication of our approaches for different workload.

### Network-Intensive Workload: We wrote a distributed MapReduce-version web crawler mr-wc. Its input is a set of URLs of the webpages to download. Application mr-wc does not have reduce phase; and map tasks downloads web pages and saves them into HDFS. Obviously, network is the most critical resource for mr-wc. Lemur project published a data set of unique URLs [14]. We use a small portion of it as the input of mr-wc. The same testbed as above was used to run mr-wc jobs. In our tests, each map task downloads 400 web pages and the number of map tasks was set to 35, 70, 105 and 126 for different runs. As the total number of map slots is 140, the utilization ratio was 25%, 50%, 75% and 90% respectively. Fig. 5(a) shows the execution time of mr-wc. For native Hadoop, the execution time of mr-wc is not significantly impacted by the utilization, which implies spare resources cannot be utilized. In contrast, resource stealing expands the resource usage of running tasks by creating more threads to concurrently download webpages. Resource stealing dramatically shortens execution time by 61%, 45%, 21% and 23% respectively. The degree of improvement decreases with the increasing utilization approximately.

For above tests, speculative execution was disabled because our additional tests show speculative execution deteriorates performance in some cases. The efficiency of webpage crawling depends heavily on the response time of the servers where webpages are hosted. Our tests show the response time can range from milliseconds to seconds. Setting a low connection timeout can mitigate the issue, but incurs another issue that more webpages are skipped. Under this circumstance, running speculative tasks is not helpful because the efficiency variation of map tasks is not caused by the system itself.

### IO-Intensive Workload: Application wordcount counts the number of word occurrences. Each map task simply tokenizes the input text and emits an intermediate key/value pair for each word occurrence. Reduce tasks add up the occurrence count for each unique word and produce the final output. The computation carried out by wordcount is light and data IO is critical. So wordcount is an IO-intensive application. In our tests, each map task processed 128MB text, and again the number of map tasks was varied. Fig. 5(b) shows the result. As the size of input data 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 synchronization overhead among threads for IO-intensive applications because each map operation runs for a short period of time and synchronization becomes the performance barrier. As a result, the overhead outweighs the benefit of higher concurrency brought up by resource stealing for *wordcount*.

## ~~Summary~~

~~Strategy LTM performs the best for most of the cases. while strategy Even performs surprisingly well and even outperforms LTM for some tests. We argue it is caused by the dynamic and load-balanced scheduling in Hadoop. We cannot simply generalize it to jobs of arbitrary types. If map tasks of a job run much longer than that in our tests, LTM is expected to outperform Even more significantly. To evaluate the effectiveness of our approaches is among the future work. BASE can significantly reduce the amount of wasted resources by eliminating most of non-beneficial speculative tasks.~~

 

 (a) Network-intensive app: *mr-wc* (b) IO-intensive app: *wordcount*

1. Experiments with other workload

# Conclusion

The overall 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 are 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. We have implemented our proposed algorithms in Hadoop and conducted experiments to evaluate the 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.

##### Acknowledgment

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

##### Reference

1. J. Dean and S. Ghemawat. MapReduce: Simpliﬁed Data Processing on Large Clusters. In Communications of the ACM, 51 (1): 107-113, 2008
2. M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in Proceedings of the 8th USENIX conference on Operating systems design and implementation, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29-42.
3. A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529-543, Jun. 2001.
4. S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production MapReduce cluster," in Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, May 2010, pp. 94-103.
5. T. Studt. There's a Multicore in Your Future http://www.tectrends.com/tectrends/article/00158249.html
6. Ekanayake, J.; Gunarathne, T.; Qiu, J.; , "Cloud Technologies for Bioinformatics Applications," Parallel and Distributed Systems, IEEE Transactions on , vol.22, no.6, pp.998-1011, June 2011
7. C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in NIPS 2006
8. R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in Proceedings of the 35th Annual Symposium on Foundations of Computer Science. Washington, DC, USA: IEEE Computer Society, 1994, pp. 356-368.
9. H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in Proceedings of the 2007 ACM SIGMOD international conference on Management of data, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 1029-1040.
10. S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg, "On optimal strategies for Cycle-Stealing in networks of workstations," IEEE Trans. Comput., vol. 46, pp. 545-557, May 1997
11. I. Ahmad and Y. K. Kwok, "A new approach to scheduling parallel programs using task duplication," in Proceedings of the 1994 International Conference on Parallel Processing - Volume 02, ser. ICPP '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 47-51.
12. L. Gwennap. New algorithm improves branch prediction. Microprocessor Reports, March 27 1995.
13. F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in Proceedings of the third symposium on Operating systems design and implementation, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 1-14.
14. http://lemurproject.org/clueweb09.php/
15. W. Gropp, E. Lusk, and A. Skjellum, Using MPI: portable parallel programming with the message-passing interface. Cambridge, MA, USA: MIT Press, 1994.
16. K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in Proceedings of the 2009 conference on Hot topics in cloud computing, ser. HotCloud'09.
17. H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics," In Proc. of the Fifth Biennial Conf. on Innovative Data Systems Research (CIDR '11), January 2011.
18. L. A. Barroso and U. Hölzle, "The case for Energy-Proportional computing," Computer, vol. 40, pp. 33-37, Dec. 2007