Automatic Task Re-organization in MapReduce

Zhenhua Guo[1], Marlon Pierce[2], Geoffrey Fox[3], Mo Zhou[4]

School of Informatics and Computing, Indiana University, Bloomington, IN, 47405 U.S.

{[1]zhguo, [2]mpierce, [3]gcf, [4]mozhou}@cs.indiana.edu

Abstract*—*MapReduce is increasingly considered as a useful parallel programming model for large-scale data processing. It exploits parallelism among execution of primitive operations. Hadoop is an open source implementation of MapReduce that has been used in both academic research and industry production. However, its implementation strategy that one map task processes one data block limits the degree of concurrency and degrades performance because of inability to fully utilize available resources. In addition, its assumption that task execution time in each phase does not vary much does not always hold, which makes speculative execution useless. In this paper, we present mechanisms to dynamically split and consolidate tasks to cope with load balancing and break through the concurrency limit resulting from fixed task granularity. For single-job system, two algorithms are proposed for circumstances where prior knowledge is known and unknown. For multi-job case, we propose a modified shortest-job-first strategy, which minimizes job turnaround time theoretically when combined with task splitting. We compared the effectiveness of our approach to the default task scheduling strategy using both synthesized and trace-based workloads. Simulation results show that our approach improves performance significantly.

Keywords: MapReduce, Bag-of-Divisible-Tasks, Task Splitting, Load Balancing

I. INTRODUCTION

MapReduce [1] has gained popularity as a programming model for large-scale data processing in both academia [2] and industry, because of scalability, fault tolerance and ease of use. In contrast to the traditional parallel programming models, e.g. MPI and workflow, where end users take the responsibility of decomposing a job into multiple tasks, in the MapReduce model, the framework itself takes the burden of the job decomposition. The MapReduce model is based on data parallelization [3] which focuses on parallelization of data rather than operations applied to data. In MapReduce model, input data is modeled as key-value pairs. Two primitive operations (*map* and *reduce*) are provided. Each *map* operation operates on a key-value pair and may produce some key-value pairs. Different *map* operations are independent. The reduce operation takes output of map operations as input and produces final results.

On the implementation side, *tasks* are schedulable entities and map operations must be organized as tasks for execution. The model itself does not impose any constraint on how map operations are grouped into tasks. Theoretically, map operations of a job can be grouped arbitrarily without affecting *correctness*. However, it affects *efficiency* of execution. To maximize performance, load unbalancing should be avoided and tradeoff between concurrency and management overhead must be considered.

Hadoop provides an open source implementation of MapReduce. In addition, a distributed file system - Hadoop File System (HDFS) is provided which derives from Google File System. HDFS chunks files into equally sized data blocks. The default strategy of map operation organization in Hadoop is that each map task processes key-value pairs contained in one block. The size of key-value pairs may vary so that number of key-value pairs stored in different blocks may differ. This simple and intuitive implementation strategy has several drawbacks we are targeting to solve. Firstly, it limits the degree of concurrency that can be achieved. Number of map tasks is fixed given input data size, input format and block size. This imposes a limit on how concurrent the processing can be, because even if number of available resources is larger than that of map tasks, not all available resources can be utilized. Secondly, Hadoop assumes that map tasks of a job require the same amount of work. This assumption may not hold for several reasons. Firstly, the nature of the map operation may result in computation time skew even if map tasks process the same amount of data. In addition, each task may process data of different sizes if user-defined input format is used. Lastly, map tasks may slow down because of process hang, software bug, software mis-configuration, and system fluctuation. In clusters, the underlying hardware may be heterogeneous and the time taken to run a map task may be drastically different depending on the capacity of the node the task is dispatched to.

Task execution time skew is observed in real studies. In study of parallel BLAST, one task takes more than 18 hours to complete while other tasks take 30 minutes to complete on average [4]. Hadoop bugs may prohibit spawning of speculative tasks even if some tasks run much longer than expected.

Cluster resource usage varies depending on workload characteristics. Usually severs are neither completely idle nor fully loaded. A study [5] done by Google shows that server utilization is between 10% and 50% most of the time based on profiling result of 5000 servers during a six-month period. As a result, the scheduling algorithm should fully exploit parallelism to utilize available resources to reduce job execution time. The above two drawbacks prohibit Hadoop from making full use of available resources even if they are idle under some circumstances.

In this paper, we mitigate the drawbacks described above by dynamically splitting map tasks according to resource availability. Our goal is to minimize average job turnaround time which is defined as the time between job submission and job completion. It is a metric that directly reflects how the user perceives the performance of a system, compared with throughput that measures performance from the perspective of system owner. Analysis of collected data from real Hadoop clusters shows that most of Hadoop jobs are map-only [6]. So in our study, we only consider map-only jobs. After discussing related work, we come up with Bag-of-Divisible-Tasks model and propose two new processing steps - *task consolidation* and *task splitting* which dynamically modify tasks. Then algorithms are proposed for single-job scenario where prior knowledge is known and unknown. After that, multi-job scheduling is investigated and scheduling algorithms are proposed integrating Shortest-Job-First strategy and task splitting. Then extensive simulation experiments are conducted and performance is compared. Finally we summarize and conclude our work.

II. Related Work

Traditional task scheduler utilizes task graphs which captures data flow and dependency among tasks to make scheduling decisions. Each schedule has both spatial and temporal aspect, which means it decides when to start a task and on which node to start it. Traditional scheduling algorithms such as list scheduling and clustering scheduling take task graphs as input and map tasks to nodes [7]. The task graph itself is not adjusted to improve performance. Bag-of-Tasks [8,9] simplifies task graph by assuming that tasks of each application are independent, which is motivated by prior efforts such as SETI@home [10] and parameter sweep applications[11]. Infrastructures (e.g. Condor [12] and BOINC[13]) haven been developed for both computing grids and more distributed and heterogeneous architectures (e.g. desktop grids). Traditional task scheduling research takes the strategy that once tasks start running, they are not modified dynamically. Our work is complementary in that during run time tasks can be split and consolidated as needed to improve performance.

There has been substantial research on load balancing which tries to balance resource usage in clusters [14]. Pre-emptive process migration supports dynamically migrating of processes from overloaded nodes to lightly-loaded nodes. It's possible that the whole system is well balanced while some nodes are idle (e.g. when the number of task processes is less than that of nodes). In that case, traditional load balancing algorithms cannot utilize idle nodes while our solution can split running tasks and dispatch spawned tasks to idle nodes.

Hadoop supports speculative execution to cope with situation where some tasks in a job become laggard compared with other tasks. The assumption of speculative execution is that the execution time of map tasks does not differ much, which makes it possible for Hadoop to predict map task execution time without any prior knowledge. When Hadoop detects that a task runs longer than expected, it starts a duplicate task to process the same data. Whenever any task completes, other duplicate tasks are killed. This can improve fault tolerance and mitigate performance degradation. However the performance gain is obtained at the expense of duplicate processing of some data and more resource usage compared with process migration. In addition speculative execution triggered by uneven map task execution caused by nature of map operation does not benefit at all, because duplicate task cannot shorten the run time either. Our work is complementary to task speculation in that task splitting and task duplication can be used together to deal with long running tasks resulting from either nature of map operation or system failure.

Divisible load theory [15] tries to solve the problem that how load received at one node (called *originator node*) can be distributed to other nodes in a system so that processing time is minimized. By and large, load is assumed to be arbitrarily partitionable, which has root in early sensor network research. Initially all input data is stored on originator node and during run time it is distributed to other nodes. It assumes that computation time per unit of data is known and task execution time is linear with amount of processed data. Our work tries to minimize job turnaround time instead of job execution time. In addition, our work enables load to be dynamically adjusted across nodes even if no prior knowledge is known.

III. Preliminary

Resource Model In Hadoop, each slave/worker node hosts a fixed number of *map slots*, which determines maximum number of map tasks the node can run simultaneously. If it is too small, resources cannot be fully utilized. If it is too big, severe resource use contention may happen and overhead is increased. For either case, performance is not optimal. We assume the number of map slots per node is perfectly turned, while how to tune it is out of our scope.

Task Model We propose *Bag-of-Divisible-Tasks*, derived from *Bag-of-Tasks*, as our task model. We use *Atomic Processing Unit* (APU) to represent a segment of processing that cannot be parallelized. Then we call a nonempty set of APU a *divisible task* such that it could be divided into sub-divisible-task(s) (or sub-task for short). Each job is modeled as a bag of *independent* divisible tasks. And from now on, we use divisible task and task interchangeably if no confusion under context. APUs may be heterogeneous in that data size and processing time varies. Given a set of independent APUs derived from problem domain, how to organize them into tasks has significant impact on performance. The optimal solution depends on both characteristics of APU and real-time system load. If tasks are too coarse-grained and therefore too large, load unbalancing is likely to happen because of large variation of task execution time. If tasks are too fine-grained and therefore too small, overhead and actual processing time get comparable and latency becomes significant.

In MapReduce, each map operation is considered as an APU. The limitation of default Hadoop implementation results from fixed granularity of map tasks driven by data blocks. Job turnaround time is affected by not only data size but also other factors, such as system fluctuation and hardware heterogeneity. We propose *task splitting* and *task consolidation* to mitigate load unbalancing and fully utilize available resources. Task splitting is a process that a task is split to spawn new tasks. Meanwhile input data is also split accordingly so that each newly spawned task processes part of it. Given a task *T*, if it is split to spawn *m* new tasks $\{T_1, T_2, \ldots, T_m\}$, following two equations hold where $UI(T)$ is unprocessed input data of task *T*. The processing that has been done by a task is not re-done after it is split.

$$UI(T) = UI(T_1) \cup UI(T_2) \cup \cdots \cup UI(T_m) \quad (1)$$

$$\forall i,j \; 1 \le i < j \le m \;\; UI(T_i) \cap UI(T_j) = \emptyset \quad (2)$$

Task consolidation is the inverse process, by which multiple tasks are merged into one task. Formally, if a set of tasks $\{T_1, T_2, \ldots, T_n\}$ are merged into a single task *T*, following equation holds.

$$UI(T_1) \cup UI(T_2) \cup \cdots \cup UI(T_n) = UI(T) \quad (3)$$

Task consolidation and split can be used to adjust task organization to adapt system environment changes. They make the scheduling more flexible and robust. If tasks are split too aggressively, overhead of splitting and task management may outweigh benefit of higher concurrency. So being splittable does not mean task splitting is beneficial. Based on the fact that tasks usually run much longer than APU, we make a simplification that APU is arbitrarily small.

A. *Split Tasks Waiting in Queue*

In this section, we give examples about how to split tasks that are waiting in queue. Running tasks are considered in next section. All map tasks in queue, which may be from different jobs, are considered when task splitting decision is made. If there are no available map slots, no map task in the queue is split.

If the number of available map slots is smaller than that of map tasks in queue, one possible strategy is to consolidate map tasks so that all of them can be dispatched immediately. The data to be processed is the same no matter whether map tasks are consolidated or not. Overall overhead of map task start-up and teardown is different because there are less tasks after consolidation. Another potential drawback brought up by consolidation is loss of data locality. The more map tasks are consolidated, the smaller the possibility becomes that input blocks of all consolidated tasks are located on the same node. As a result, amount of data transferred from remote nodes increases. So optimal decision relies on the tradeoff between task overhead and data transfer. Plot (b) in Fig. 1 shows an example. Three map tasks T_1, T_2 and T_3 are waiting and two nodes are available. So we can schedule two map tasks at most immediately. If we consolidate two map tasks, all map tasks can be scheduled to run immediately. In the plot, map task T_2 and T_3 are consolidated into map task T_{2-3} which is dispatched to node where block B_2 is stored. Block B_3 is remotely accessed by task T_{2-3}.

If the number of available map slots is larger than that of map tasks in queue, map tasks can be split to spawn new map tasks to fill all available map slots. Resultant benefits include better parallelism and load balancing. As number of map tasks increases, overall task start-up and teardown overhead increases as well. Another disadvantage is data locality may become worse. If a map task cannot be dispatched to the node where its input block is stored, none of its spawned map tasks after split can be dispatched to the node either if they are run immediately. Otherwise, one of the spawned map tasks is guaranteed to be able to be dispatched to that node while others may or may not be dispatched to it depending on map slot availability. Plot (c) in Fig. 1 shows an example of task splitting. Initially there are four available nodes and three map tasks T_1, T_2 and T_3. Task T_3 is split to task $T_{3.1}$ and task $T_{3.2}$ and all tasks are scheduled. Task $T_{3.1}$ and $T_{3.2}$ share the same input block B_3 but process different portions. Compared with the situation that split is not applied, task $T_{3.2}$ needs to access B_3 remotely but all nodes are utilized. One way to mitigate the data locality problem is data replication. When there are multiple copies of a block, the possibility is larger that data-local scheduling is achievable after task spit. One extreme case is each block is replicated on all nodes so that data locality becomes less significant.

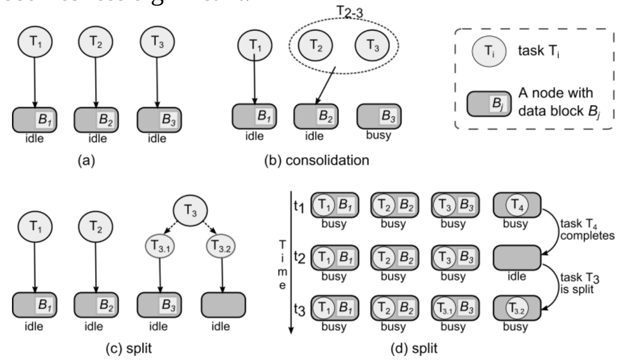

Figure 1. Task splitting and task consolidation. Arrows are scheduling decisions. Each node has one map slot and block B_i is input of task T_i.

B. *Split Running Tasks*

Besides tasks waiting in queue, running tasks can also be split dynamically to improve performance. When tasks are scheduled and running, computation time skew of tasks may slow down the progress of the whole job. Task splitting can be applied dynamically during task execution to offload some processing to other available map slots. Plot (d) in Fig. 1 shows an example. At time t_1, four tasks are running. At time t_2, task T_4 completes and the slot originally occupied by task T_4 becomes available while the other three tasks are still running. Task T_3 is chosen to spawn a new task $T_{3.2}$ which is scheduled to the available slot released by completed task T_4. Again, all nodes are utilized but task $T_{3.2}$ accesses its input data B_3 remotely.

C. *Summary*

The previous two algorithms are combined together to adjust all unfinished tasks (waiting tasks + running tasks), which achieves continuous optimization during whole lifetime of jobs.

Task consolidation reduces the number of tasks to manage and schedule, which is highly beneficial if task management overhead is high and task start-up and teardown overhead is comparable to the actual execution time. We assume task execution time is significant longer than task start-up and teardown time. If this does not hold, blocks can be enlarged to increase task granularity.

Task splitting is beneficial when loss of data locality does not impose critical performance degradation. When data are replicated on every node, the data access time is approximate no matter where a task is dispatched if data access contention (e.g. multiple tasks access different data on the same node) is not severe. If data access contention is severe, number of map slots on each node can be tuned appropriately to achieve optimal tradeoff between concurrency and resource use contention, so that data access does not affect scheduling much. This conclusion also holds when jobs are CPU-intensive and the data access cost is negligible. In other words, if the ratio of computation to data access is large, the computation factor is critical and other factors, such as disk I/O and network I/O, can be ignored. We focus on CPU-intensive jobs in the following discussions.

IV. Single-Job Task Scheduling

First, we consider the task scheduling problem when only one job is running at most at any time. In the next section, multi-job case is discussed. The following algorithm shows how task splitting is hooked into the task scheduling process.

Algorithm skeleton

```
while isRunning = true:
  split_tasks();
  schedule_tasks();
```

In the beginning of each scheduling iteration, task splitting is applied if needed. This step makes tradeoffs between concurrency and overhead. Then an existing task scheduling strategy (e.g. Hadoop's data locality based scheduling) is used to schedule tasks. So task splitting can be seamlessly integrated with existing schedulers. We focus on the task splitting process and present our proposed solutions when prior knowledge about workload is known and unknown. We summarize issues shown below that need to be solved to address the problem.

a) When to trigger task splitting
b) Which tasks should be split and how many new tasks to spawn; and
c) How to split

A. *Task Splitting without Prior Knowledge*

When no prior knowledge is known about execution time, a strategy we term Aggressive Splitting (AS) is proposed.

1) When to trigger task splitting: The goal of task splitting is to shorten average job turnaround time by utilizing as many nodes as possible. Assume the scheduler is invoked at time t, task splitting decision is made if following inequality is satisfied

$$N_{miq}(t) + N_{run}(t) < N_{ams} \quad (4)$$

where $N_{miq}(t)$, $N_{run}(t)$ and N_{ams} are the number of map tasks in queue at time *t*, the number of running map tasks at time *t* and the number of all map slots respectively. That means there are idle map slots even if all tasks in queue are scheduled immediately. In this case, the default scheduling strategy cannot use all idle slots. So the task splitting process should be initiated. Otherwise, it does not make sense to split tasks because there are no idle slots where newly spawned tasks can run. This will not make long-running tasks become stragglers because our task splitting process is invoked continuously and long-running tasks will become candidates of split target whenever there are idle slots.

2) Which tasks should be split and how many new tasks to spawn: We evenly distribute available map slots to unfinished tasks. Without prior knowledge, what we do is divide the number of idle map slots by the number of unfinished tasks to calculate how many new tasks to spawn for each task on average. Then tasks are split one by one until no map slots are idle. The algorithm skeleton is shown below.

Algorithm skeleton

```
UTS:set ← unfinished tasks
IMS:int ← number of idle map slots
MST:int ← $\lceil |UTS| / IMS \rceil$
for each task $T$ in UTS:
  if IMS ≤ 0: break
  if IMS < MST:
    NS ← split(T, IMS)
  else
    NS ← split(T, MST)
  IMS ← IMS - NS
```

Function $split(T, N)$ splits task T to spawn N new tasks. Depending on map slot availability, split policy and overhead, the actual number of spawned tasks may be smaller than N. The actual number is returned from the function call so that following code can update number of available map slots accordingly. Implementation of *split* is described in next section.

3) How to split: Given a task and maximum number of new tasks it may spawn, this section solves the problem how to split. Firstly, the number of new tasks is adjusted so that it does not exceed number of available map slots. Data block is logically split to equally sized sub-blocks. We consider the task processing one sub-block is atomic and not splittable. So it specifies smallest granularity of spawned tasks. For task T, total number of sub-blocks, the number of sub-blocks that have been processed or are being processed, and number of new tasks to spawn are denoted by $TS(T)$, $PS(T)$ and $NT(T)$ respectively. Since we don't have prior knowledgeof map execution time, we blindly spawn new tasks so that each one processes the same amount of data.

$$sub\ blocks\ per\ task\ = \frac{TS(T) - PS(T)}{NT(T) + 1} \quad (5)$$

The remaining work is evenly divided among the task being split and newly spawned tasks. The principle is to make them all complete simultaneously if map operation execution time is heterogeneous theoretically. To avoid inefficiency caused by spawning small tasks, a threshold is set to prevent small tasks being split. Optimal threshold depends on workload and map operation characteristics. It is our future work to make the threshold automatically tuned.

4) Complexity: The whole task list is scanned at most once, so time complexity is $O(n)$ with regard to number of tasks.

B. *Task Splitting with Prior Knowledge*

Now we assume that prior knowledge about task execution time is known. By prior knowledge, we mean that Estimate Remaining Execution Time (ERET) is known or predictable. ERET indicates how long a task will run before completion approximately. We propose Aggressive Split with Prior Knowledge (ASPK) to optimize job turnaround time.

1) When to trigger task splitting: The same algorithm from last section can be reused here.

2) Which tasks should be split and how many new tasks to spawn: Ways to split tasks are not unique. Number of task splits done during whole lifetime of a job should be as small as possible without degrading performance. Fig. 2 demonstrates different ways to split tasks to achieve the same turnaround time. Graph (a) shows a scenario where there are two running tasks - T_1, T_2, one idle slot and no waiting tasks. ERET of T_1 and T_2 is $2t$ and t respectively. If overhead and data locality are negligible, we definitely should split tasks to fill the idle slot. We can split task T_1 to spawn a new task and both will run for period t before completion, which is demonstrated in (b). At time t all tasks complete. Another way shown in (c) is to split task T_2 to spawn a new task and both will run for period $t/2$. At time $t/2$, two slots become idle and task T_1 is split to spawn two new tasks each of which runs for $(2t - t/2)/3 = t/2$. In both cases, the final job turnaround time is t. However number of spawned tasks is different. In (b), one task is spawned while in (c) three tasks are spawned. More task splits incur higher probability to degrade performance and destabilize system. In the example, (b) is preferred to (c).

Tasks that complete last determine when a job finishes. For jobs with tasks that have highly varied execution time, the scenario should be avoided that few long tasks last much long after other short jobs complete. When long running tasks exist, to split tasks with small ERET generates smaller tasks, which doesn't affect job turnaround time. So our heuristics is that tasks with large ERET should be split first so that they do not become "stragglers".

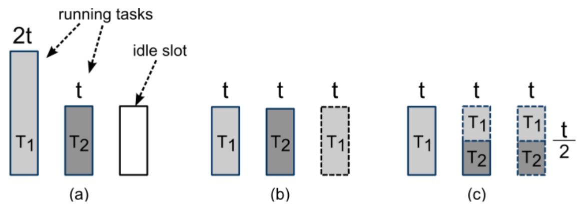

Figure 2. Different ways to split tasks (Processing time is the same). Dashed boxes represent newly spawned tasks.

Firstly, tasks with small ERET are filtered because to split a task that will end very soon does not provide much benefit. In addition, task filtering is an optimization step that reduces num of map tasks considered by following steps for faster processing. Secondly remaining tasks are sorted by ERET in descending order. After that, tasks are clustered into $\{C_1, C_2, \dots, C_m\}$ according to ERET so that tasks with similar ERET belong to the same cluster. Each cluster C has several pieces of information including task list ($C.TS$), number of tasks ($C.Count$), sum of ERET ($C.ERET$) and average of ERET ($C.AE$). We go through task clusters one by one to evaluate whether task splitting is beneficial. Initially, we only consider tasks in cluster C_1. Tasks in C_1 are split to fill all idle slots, and average task execution time T_1 is calculated. If T_1 is larger than $C_2.AE$, it doesn't benefit to split tasks contained in following clusters and estimated execution time of newly spawned tasks is set to $C_2.AE$. If T_1 is significantly smaller than $C_2.AE$, spawned tasks are small compared with tasks in C_2. So we consider tasks from both C_1 and C_2 for split. Time T_2 is calculated and compared with $C_3.AE$. If T_2 much smaller, we consider C_1, C_2 and C_3. This process is repeated until T_i is larger than or comparable to $C_{i+1}.AE$ or all clusters are included. The algorithm skeleton is shown below.

Algorithm skeleton

```
IMS ← number of idle map slots
UTS ← unfinished tasks
FTS ← filterTasks (UTS)
STS ← sortByERET (FTS)
{$C_1$,$C_2$,…,$C_m$} ← clusterTasks (STS)
sumERET ← 0, count ← IMS
for cluster $C_i$, 1≤i≤m:
  sumERET += $C_i$.ERET
  count += $C_i$.Count
  avgERET = sumERET / count
  if i = m: break
  if avgERET << $C_{i+1}$.AE:
    continue
  else
    break
```

Filtering Ideally, how tasks are filtered should depend on the ERET of unfinished tasks. A pre-set threshold is not flexible enough to capture task characteristics. Instead, we calculate the *optimal remaining job execution time* (ORJET) by assuming that tasks are split to use all available nodes. Total ERET is gained by summing ERET of all unfinished tasks. It is divided by number of all nodes (including both occupied and idle slots) to get ORJET. ORJET measures optimally how long a job will run before completion. Then ERET of each task is compared with ORJET. If task ERET is significant smaller than ORJET, it is filtered out. Towards end of job execution, ORJET becomes increasingly small because running tasks are close to completion and more slots are released. In this situation, task splitting is not beneficial because overhead of task splitting outweighs potential gain of higher concurrency. So we filter out tasks that are close to completion without affecting overall performance. Thus the filtering process is adaptive to workloads of different types.

Clustering Task clustering algorithm is designed to group tasks with similar ERET and separate tasks with significantly different ERET. Existing clustering algorithms, such as K-means, Expectation-Maximization and agglomerative hierarchical clustering, from the machine learning community can be used without modification. Considering that scheduling routine is called frequently and its performance is critical to the whole system, we favor simple linear algorithms. Tasks being clustered have been ordered by ERET, which guarantees that tasks belonging to the same cluster are consecutive in the task list. Our current algorithm requires that the task list is scanned once by moving a "cursor" from beginning to end. A running list is maintained to contain tasks that are before the "cursor" and belong to current cluster. If ERET of the task pointed by cursor is much smaller than the average ERET of the current cluster, then the current cluster is added to cluster set and a new cluster is created which initially only contained the task pointed by cursor. This guarantees maximal ERET of tasks within a cluster is significantly different than average ERET of tasks within previous cluster.

3) How to split: The way to split tasks can be optimized if we also have prior knowledge about mean task execution time, network throughput, disk I/O throughput, etc. For task T, disk I/O cost, network I/O cost, and computation cost are denoted by $DIO(T)$, $NIO(T)$ and$Com(T)$ respectively. So total time is $DIO(T) + NIO(T) + Com(T)$, if these operations don't overlap. Task being split is denoted by T_{cur}, and newly spawned tasks are $\{T_{cur}^1, T_{cur}^2, \ldots, T_{cur}^N\}$. Ideally, following equation should be satisfied to make tasks complete simultaneously after split.

$$\begin{aligned} &DIO(T_{cur}^1) + NIO(T_{cur}^1) + Com(T_{cur}^1) \\ &= \cdots\cdots \\ &= DIO(T_{cur}^N) + NIO(T_{cur}^N) + Com(T_{cur}^N) \\ &= DIO(T_{cur}) + NIO(T_{cur}) + Com(T_{cur}) \end{aligned}$$

Because we assume $DIO(T)$ and $NIO(T)$ are negligible compared to $Com(T)$, the above equation is converted to

$$Com(T_{cur}^1) = Com(T_{cur}^2) = \cdots = Com(T_{cur}^{\mathrm{N}}) = om(T_{cur})$$

So unfinished work of task T is evenly distributed to T and newly spawned tasks after split.

4) Complexity: In ASPK, complexity of sorting is $O(n\,log\,n)$ and that of other operations is not greater than $O(n)$.So overall complexity is $O(n\,log\,n)$. However, sorting can be further optimized considering that in each iteration, except the first one, tasks are mostly ordered.

C. Fault Tolerance

Our proposed algorithms do not handle fault tolerance directly. Task splitting is not enough to cope with situations where some tasks stall or fail due to hardware failure, severe system fluctuation or hanging process. We integrate speculative execution to solve the problem. Whenever the system detects failure, duplicate tasks are created automatically to replace failed tasks. Now we have a complete solution which can speed single-job execution by splitting relatively long tasks and speculatively re-execute failed tasks.

V. MULTI-JOB OPTIMIZATION

We put multi-job scheduling into the context of classic queuing theory. We adopted *M/G/s* model [16]. Jobs arrive according to a homogeneous Poisson process. Job execution time is independent and may follow generic distributions. Also there is more than one server in the system. One difference from the classic model is that a job may use multiple servers during its execution and the execution time depends on the number of used nodes. We propose *Greedy Task Splitting* (GTS) which minimizes run time of each job by splitting tasks to occupy all map slots and making tasks of last round complete simultaneously. Because each job uses all available nodes, following jobs cannot execute until current running job completes. In other words, queue time of some jobs is increased compared with non-GTS scheduling. As a result, change of job turnaround time depends on both decrease of job execution time and possible increase of job queue time. We will show that GTS gives optimal job turnaround time. Recall that we assume tasks are arbitrarily splittable.

A. Optimality of Greedy Task Splitting

Fig. 2 shows two examples of execution arrangement of a job *J*. In (a), job *J* starts at *S(J)* and completes at *F(J)*. It uses all resources during the execution. In (b), the processing is grouped to four segments - 1, 2, 3 and 4. Now we formulate the scheduling model. *C* denotes capacity of a certain type of resource in the system. *n* denotes number of jobs to run. S_i ($1 \le i \le n$) denotes total resource requirement of job *i*. Resource usage function $r(t,i)$ represents amount of resource consumed by job *i* at time *t*. Constraints are:

$$t \ge 0 \tag{6}$$

$$\forall t, \sum_{i=1}^{n} r(t,i) \le C \tag{7}$$

$$\forall\ 1 \le i \le n \sum_{t=0}^{+\infty} r(t,i) \ge S_i (or \int_{t=0}^{+\infty} r(t,i)\mathrm{dt} \ge S_i) \tag{8}$$

and objective function is

$$\min(\sum_{i=1}^{n} max_t\{r(t,i) \ne 0\}) \tag{9}$$

Inequality (7) means that at any moment, resource consumed by all jobs must not be more than capacity. Inequality (8) means that the sum of resource consumption by any job across time is not less than requirement of the job. The ideal case that actual resource consumption is equal to resource requirement, which means no overhead is incurred. In the objective function, $max_t\{r(t,i) \ne 0\}$ is turnaround time for job *i*. So our goal is to minimize overall job turnaround time.

Firstly we will show that once a job starts running, it should complete as soon as possible by using all available resources. Secondly we will convert this problem to *n/1* (*n* jobs/1 machine) scheduling problem solved in [17].

Given a job J, its start time $S(J)$ and its completion time $F(J)$, Fig. 3 shows possible strategies of execution arrangements. Execution arrangement of J affects completion time of other jobs. One fact is start time of job *J* does not matter when $F(J)$ is fixed. Intuitively, all parts of execution of Job J should be placed as close to $F(J)$ as possible. In plot (b) execution of job J is interspersed along time axis. Execution arrangement demonstrated in plot (b) can be converted to that demonstrated in plot (a) by interchanging interspersed execution segments of job J (e.g. marked by 1, 2 and 3 in the plot) and execution segments of other jobs falling into the continuous area S. After the interchange, completion time of those affected jobs either does not change or becomes earlier because their changed execution segments starts earlier. This interchange process can be iterated until each job utilizes all resources during its execution (see Fig. 3 for an example). In each iteration, only one job is considered. The whole process makes overall turnaround time monotonically decrease regardless of order of jobs picked during iterations.

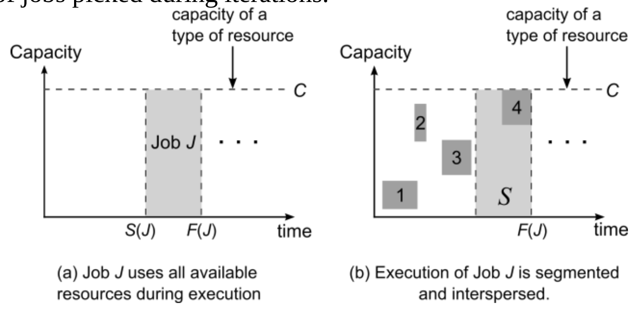
Capacity
capacity of a type of resource
C
Job J
S(J)
F(J)
time
(a) Job J uses all available resources during execution
Capacity
capacity of a type of resource
C
1
2
3
4
S
F(J)
time
(b) Execution of Job J is segmented and interspersed.

Figure 3. Different ways to arrange execution of a job.

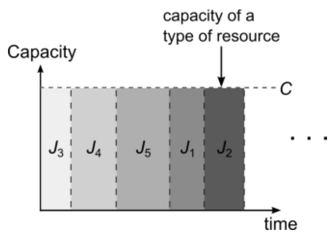

Figure 4. Multiple scheduled jobs (Each uses all resources for execution)

However, different job execution orders may result in different overall turnaround time. The next question is how to determine job execution order so that objective function is minimized. Because at any moment only one job consumes all resources, we can view the whole system as a single big virtual node. This problem becomes the *n/1* problem (n jobs / 1 machine) solved in [17]. Shortest-job-first strategy gives overall optimal turnaround time. So jobs should be executed in ascending order of execution time.

B. Multi-Job Scheduling

Given a number of jobs to run, the algorithm skeleton of *Shortest Job First Scheduling* (SJFS) is shown below. *Serial Execution Time* (SET) represents how long a job runs serially.

Algorithm skeleton of SJFS

```
order jobs by SET in ascending order
schedule jobs in turn
```

If we know serial execution time of all jobs that are to be run, we can apply SJFS directly. However, in real systems, it is hard, if not impossible, to know all jobs to run ahead. Jobs are submitted dynamically by end users or batch scripts. To cope with the uncertainty, we use *Non Overlapped Periodic Shortest Job First Scheduling* (NOPSJFS) in which SJFS is run periodically. Let I be interval that SJFS is called. So scheduling decision is made at time $0, I, 2I, \ldots$ Let J_t be set of jobs that are submitted at or earlier than time t. At time $n \cdot I$, SJFS is applied to the job set $J_{n \cdot I} - J_{(n-1) \cdot I}$. So jobs that are scheduled at time $n \cdot I$ only include those submitted between time $(n-1) \cdot I$ and $n \cdot I$. Jobs submitted prior than time $(n-1) \cdot I$ are not considered at all even if some of them are still waiting in the queue. This strategy makes each job scheduled just once and jobs scheduled during different period do not overlap. But unexpected system fluctuation exists and prior knowledge of SET may be inaccurate. So assumptions made when a job is scheduled may be rendered useless by the time it is dispatched to run. *Overlapped Shortest Job First Scheduling* (OSJFS) is proposed in which all jobs are considered that have been submitted but not completed yet. To avoid starvation of long jobs, an aging factor is associated with each job which measures how long a job has been waiting in the queue. Priority is positively correlated to aging factor. So the longer a job has waited, the higher its priority becomes.

VI. EXPERIMENT

We conduct experiments using the MapReduce simulator mrsim [18] which is built on top of an event-driven framework. Table 1 shows the configuration of simulated system. Data is placed randomly on nodes. Each node hosts only 1 map slot. We will assess *effectiveness* of our approaches. So hardware configuration affects absolute job turnaround time, but it does not affect comparison between our strategies and default strategy.

TABLE I. CONFIGURATION OF TEST ENVIRONMENT

Number of nodes	64	Disk I/O - read	40MB/s
Processor frequency	500MHz	Disk I/O - write	20MB/s
Map slots per node	1	Network	1Gbps

Several distributions are used to model execution time of map operations - Gaussian distribution and step functions abstracted from real workload trace. Firstly, we set up tests to show that our approach improves performance in single job environment.

A. Single-Job

In this set of tests, we investigate the effect of variation of map task execution time on performance. We design a micro-benchmark to measure performance improvement of task splitting. Based on number of all map slots and that of map tasks, two cases are considered. When the number of map tasks is smaller than that of available map slots, the default strategy cannot utilize all resources.

In first test, we compose a job whose input data has 32 blocks each of which is 64MB. The cluster has 64 nodes. We assume that task execution time follows Gaussian distribution with negative values cut off. Mean is fixed and variance is varied which is an indicator of variation of execution time of map tasks. Baseline distribution is uniform distribution with mean μ and coefficient of variance (CV) is zero by definition. We let Gaussian distributions have the same mean and change variance to $(k \cdot u)^2\ (1 \leq k \leq 10)$. So CV is between 1 and 10. Job turnaround time is shown in plot (a) in Fig. 5. One observation is that job turnaround time increases as CV increases. That results from cut-off of negative values sampled from tested distributions. So the mean of sampled values is no longer μ and it increases slightly with CV. Both AS and ASPK improves performance significantly and performance gain increases with CV. AS incurs larger variation compared with ASPK. When CV is small, the difference between AS and ASPK is not significant. As CV becomes large, ASPK performs significantly better than AS. When CV is 10, ASPK improves AS by 50%.

Now, we increase number of map tasks of a job to 200 to make it significantly larger than number of map slots. Test environment is the same as previous test. Plot (b) shows results. Distributions of task execution time are the same as in previous test. Default scheduling has embedded support for load balancing. Whenever a map slot becomes available, it dispatches a waiting task to it. Because execution time of map tasks is sampled from the same distribution, the sum of task execution time for different nodes follows the same distribution as well. In other words, mixture of long and short tasks dispatched to nodes naturally makes the load balanced during early lifetime of the job. In the early phase of job execution, all map slots are occupied so that task splitting does not benefit. Towards the end of execution, all tasks are either running or completed. Any released map slot cannot be utilized because there is no waiting task. Then task splitting improves performance by rebalancing load. Considering task splitting is mostly applied near job completion, it may not benefit much. Test result shows that even in that situation, AS and ASPK improves performance by 50% at most. Also as CV becomes large, ASPK increasingly outperforms AS.

Besides synthesized workload, workload data collected in real clusters is also used. Concretely, we use cluster data published by Google [19]. It is analyzed in [20] to extract characteristics of jobs and tasks. One observation made in the paper is that task execution time for three types of jobs is bimodal. Around 75% of map tasks are short, running for approximately 5 minutes. Around 20% of map tasks are long, running for approximately 360 minutes. Execution time of the remaining 5% the map tasks is between 5 minutes and 360 minutes. This distribution is used to model task execution time in this test. We measured both job turnaround time and completion time variation. Fig. 6 shows the results. AS and ASPK shorten job turnaround time by 20% - 30%. ASPK performs slightly better than AS by reducing job turnaround time by 5% - 10%. Standard deviation of slot completion time is shown in plot (b). For default scheduling, the value is 8521 seconds which indicates that the last round of map task execution results in severe load unbalancing. ASPK achieves the smallest standard deviation around 9 seconds, so that its histogram is almost invisible in the plot. This result is surprisingly good considering that the job runs for tens of thousands of seconds. For AS, standard deviation is around 570 seconds. To figure out whether best performance of ASPK is achieved by splitting much more tasks than AS, number of spawned tasks is measured. Plot (c) shows that ASPK even has smaller number of spawn tasks. So ASPK achieves shortest job turnaround time and smallest variation of slot completion time by spawning fewer tasks. This means when prior knowledge is known additional optimization done in ASPK is effective.

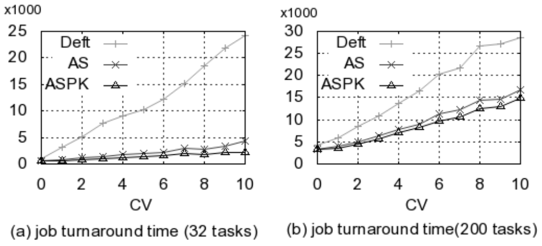

Figure 5. Single-Job test results (Gaussian distribution is used)

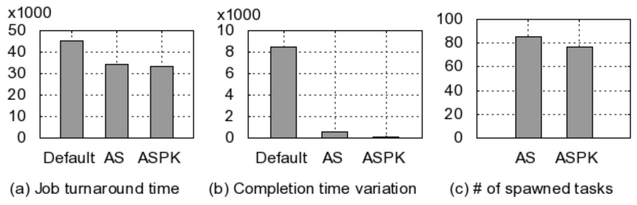

Figure 6. Single-Job test results (Real profiled distribution is used)

Above tests demonstrate that task splitting strategy improves performance significantly and the degree of improvement is related to characteristics of map tasks.

B. Multiple jobs

As *M/G/s* model is adopted for multi-job scenario, inter-arrival time of jobs follows exponential distribution. We generate a workload to have 100 jobs each of which is synthesized according to Google cluster data. We measure average job turnaround time with and without SJF policy applied. If interarrival time is longer than job execution time, on average one job is running at most at any time. Single-Job scheduling can be used directly. So we set mean of interarrival time to be much shorter than average job execution time.

In this test, all jobs have the same number of map tasks, which is equal to the number of all map slots, so that each job can occupy all map slots. Execution time of tasks belonging to a job is the same. 75% of jobs are short, 20% of jobs are long and 5% of jobs are medium. 100 jobs are generated. Task splitting in this test does not benefit much because all map tasks of a job complete almost simultaneously and load unbalancing occurs rarely. Results are shown in Fig. 7. Non-SJF scheduling and SJF scheduling have comparable makespan. SJF decreases average job turnaround time by 63%.

Then we tested the case where different jobs have the same serial execution time. Obviously SJF strategy does not make sense because all jobs are equally long. So we ignore

SJF and evaluate task splitting strategies. Task execution time of each job follow the same distribution extracted from Google cluster data. 100 jobs are generated and all slots are used at any time except near completion. Fig. 8 shows that both job turnaround time and makespan are shortened by 5% - 10%. One well-known fact is that if a system is fully loaded, it is harder to make optimization compared with the situation where a system is partially loaded. Our test results show that even if the system is fully loaded and SJF is useless, task splitting still benefits. Considering that study in Google shows CPU utilization ratio is between 20% and 50% for their production clusters, we believe task splitting will give more improvement in real clusters than in this test.

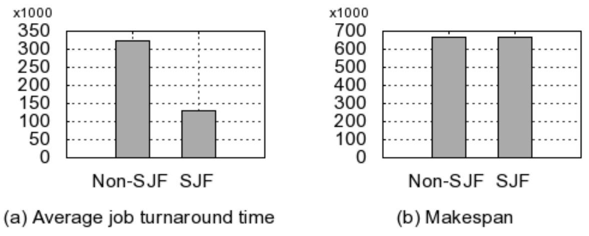

Figure 7. Multi-Job test results (task execution time is the same for a job)

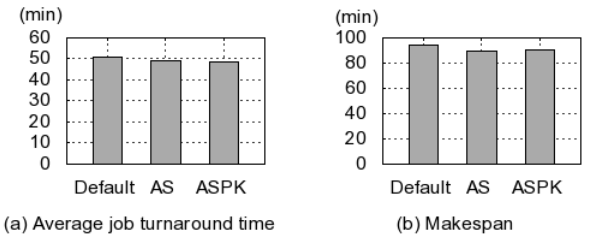

Figure 8. Multi-Job test results (job execution time is the same)

VII. CONCLUSIONS

In this paper, we examined strategies for optimizing job turnaround time in MapReduce. Firstly, we analyzed the MapReduce model and its Hadoop implementation, and found that the way map operations are organized into tasks in Hadoop has several drawbacks, such as limit of concurrency, task completion time skew and load unbalancing. Then we proposed task splitting, which is a process to split unfinished tasks to fill idle map slots, to tackle those problems. For single-job scheduling, Aggressive Scheduling (AS) and Aggressive Scheduling with Prior Knowledge (ASPK) were proposed for cases where prior knowledge is known and unknown respectively. For multi-job scheduling, we proved that combination of Shortest-Job-First strategy and task splitting mechanism gives optimal average job turnaround time if tasks are arbitrarily splittable. Overlapped Shortest-Job-First Scheduling (OSJFS) was proposed which invokes basic short-job-first scheduling algorithm periodically and schedules all waiting jobs. We also conducted extensive experiments to show that our proposed algorithms improve performance significantly compared with default strategy. One thing we may explore in the future is how task splitting and consolidation can benefit IO intensive applications. Tradeoffs between data access concurrency and data locality should be considered to achieve optimal performance.

ACKNOWLEDGMENT

This work is funded in part by the Pervasive Technology Institute. We would like to thank Lizhe Wang for in-depth discussions and feedbacks about our work.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM 51, 1 (January 2008),p107-113

[2] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga and D. Gannon, "Cloud Technologies for Bioinformatics Applications," 2nd MTAGS, SC2009

[3] A. Grama, A. Gupta, G. Karypis, and V. Kumar, Introduction to Parallel Computing, 2nd edition, Pearson AddisonWesley, London, UK, 2003

[4] W. Lu, J. Jackson, J. Ekanayake, R. S. Barga, and N. Araujo, "Performing Large Science Experiments on Azure: Pitfalls and Solutions," in Proc. CloudCom'10, 2010, p209-217

[5] L. A. Barroso and U. H. Olzle, "The case for energy-proportional computing," Computer, 40(12), 2007.

[6] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in Proc. CCGRID '10, 2010. p. 94-103

[7] O. Sinnen, Task Scheduling for Parallel Systems, Wiley 2007

[8] M. Adler, Y. Gong, and A. L. Rosenberg. "Optimal sharing of bags of tasks in heterogeneous clusters," In Proc SPAA'03 2003. p1-10.

[9] C. Weng and X. Lu, "Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid," FGCS, Vol. 21, no. 1, pp. 271–280, 2005.

[10] SETI@home, http://setiathome.ssl.berkeley.edu

[11] H. Casanova and F. Berman, "Parameter sweeps on the grid with APST," in Grid Computing: making the global infrastructure a reality, F. Berman, G. Fox, and T. Hey, Eds. Wiley, 2003

[12] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - A hunter of idle workstations," In Proc. ICDCS'88 1988, pp. 104–111.

[13] BONIC http://boinc.berkeley.edu

[14] X. Zhang, Y. Qu and L. Xiao, "Improving Distributed Workload Performance by Sharing both CPU and Memory Resources," in Proc. ICDCS'00, 2000, p.233-241

[15] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems," Cluster Computing Vol. 6, no. 1, Jan. 2003, p7-17

[16] D. G. Kendall, "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain," The Ann. Math Stat. Vol. 24, No. 3, Sep. 1953, pp. 338-354

[17] R. W. Conway, W. L. Maxwell, and L. W. Miller, "Theory of Scheduling," Addison Wesley, 1967.

[18] S. Hammoud, M. Li, Y. Liu, N. K. Alham, and Z. Liu, "MRSim: A discrete event based MapReduce simulator," FSKD 2010, p2993-2997

[19] http://code.google.com/p/googleclusterdata/

[20] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "Analysis and Lessons from a Publicly Available Google Cluster Trace," University of California, Berkeley, CA, Tech. Rep. 2010