2.1 stragglers in jobs

2.1.1 magnitude of stragglers and their impact

A job has different phases that execute the same type of tasks in parallel. Based on the progress rates of tasks, we can identify stragglers. Progress rates are expected to be similar to IO and compute operations when we do not have stragglers. It is used instead of task’s duration to remain agnostic to skews in work assignment among tasks.

---------->>>>>>> insert figure here

The slowdown ratio was measured for each phase to find a comparison with stragglers. Slowdown ration is the ratio of progress rate of the median task to the slowest task. ``The negative impact of the stragglers increases as the slowdown ratio increases \cite{anan:effective}.’’ Figure 1 gives more detail on this. The slowest task is up to 8 times slower than the median task in the job for LATE whereas it is 7 times slower for Mantri. Another noticeable thing for two cases (LATE and Mantri) is that speculation technique is not effective in mitigating stragglers in small jobs, but it is effective for large jobs. That is why it is not that beneficial since large amount of jobs (almost 80\%) consist of small (≤ 10) tasks and they dominate the production traces of Facebook and Bing.

2.1.2 Blacklisting is insufficient

A solution for mitigating stragglers is to black-list machine that will give leverage to stragglers. For this case, a straggler is classified as tasks that have progress rate lower than half of the median progress rate among tasks that take place. A time window of 5 minutes and 1 hour have been tried to analyze the performance and determine whether this can be used as a potential solution. ``The best case eliminates only 12% of the stragglers and improves the average completion time by only 8.4% (in the Bing trace, 11% of stragglers are eliminated leading to an improvement of 6.2%) \cite{anan:effective}.’’ Even though there is a small improvement, it will not be helpful in large production traces.

2.2 Heavy tail in Job Sizes

------>>>>>>> insert figure 2 from anan:effective (heavy-tail and power law)

Small interactive jobs dominate the cluster and have stringent latency demands. ``In the Facebook and Bing traces, jobs with ≤ 10 tasks account for 82% and 61% of all the jobs, respectively \cite{anan:effective}.’’ On the other hand, they are the ones that are most affected by stragglers. As job sizes have a {\em heavy$-$tail} distribution, we can clone small jobs using few extra resources. Figure 2a shows, 90% of the smallest jobs consume only 6% and 11% of the total cluster resources in the Facebook and Bing clusters, respectively. Indeed, the distribution of resources consumed by jobs follows a power law (Figure 2b). In fact, at any point in time, the small jobs do not use more than 2% of the overall cluster resources.

They also offer potential to speed up these jobs by using few extra resources. ``For instance, cloning each of the smallest 90\% of the jobs three times increases overall utilization by merely 3\% \cite{anan:effective}.’’ Google released traces from their cluster job scheduler that schedules a mixed workload of MapReduce batch jobs, interactive queries, and long-running services where 92\% of the jobs account for only 2\% of the overall resources.

3 Cloning of Parallel Jobs

Dolly suggests launching multiple clones of a job and consider the result of the first clone that finishes. Cloning is a better option than speculation because i) it does not have to wait and observe a task before acting and ii) it is free from the risk of speculating wrong tasks or missing the stragglers.

3.1 Granularity of Cloning

Granularity of cloning plays a crucial role in achieving efficiency. There are two types of cloning possible – job-level cloning and task-level cloning. For job-level cloning, multiple clones of the entire job are launched for every job submitted to the cluster. It is appealing due to its simplicity and ease of implementation. Another option is to clone at the granularity of individual tasks and launch multiple clone of each task. A clone group is formed with different clones of the same task. We will then consider the clone from a clone group that finishes first. That is why task-level cloning requires internal changes to the execution engine of the framework. ``As a result of the finer granularity, for the same number of clones, task-level cloning provides better probabilistic guarantees for eliminating stragglers compared to job-level cloning \cite{anan:effective}.’’

=====>>>>>>> figure 3 from anan:effective

Figure 3 gives a better illustration of this idea. We can see that task-level cloning gains more per clone and the probability of the job straggling drops off faster. But storage is a big issue for this design because it creates multiple clones of the same data and they need to be stored on disks. As a result, task-level cloning in Dolly is preferred over job-level cloning.

4.1 Two Opposite Strategies

A fundamental challenge of cloning is the potential contention it creates in reading data. To take care of this case, two pure strategies at opposite ends are suggested: Contention-Avoidance Cloning (CAC) and Contention Cloning (CC). CAC completely avoids contention by assigning each upstream clone, as it finishes, to a new downstream task clone. This avoid contention because it guarantees that every upstream task clone only transfers data to a single clone per downstream clone group. On the other hand, CC ignores the extra contention caused and assumes that the first finished upstream clone in every clone group can sustain transferring its intermediate output to all downstream task clones. None of these clones are handicapped as all the downstream clones start at the same time. Also, only one of the upstream clones in a clone group need to be a non-straggler for a job to succeed without straggling.

4.2 Comparison between CAC and CC

========>>>>>>>>> insert figure 5 from anan:effective

Figure 5 compares the probability of a job straggling with CAC and CC for different job sizes. Part a has 10 upstream and downstream jobs whereas part b has 20. ``With three clones per task, the probability of the job straggling increases by over 10\% and 30\% with CAC compared to CC \cite:{anan:effective}.’’ The gap between the two diminishes for higher numbers of clones. For this case, CAC is more likely to have jobs that will be stragglers compared to CC.

=========🡺>>>>>>>> insert figure 6 from anan:effective

But CC is not free from downsides also. Figure 6 shows the slowdown of transfers in each bin of jobs. Transfers of jobs in the first two bins slow down by 32\% and 39\% at median, third quartile values are 50\%. Transfers of large jobs are less hurt because tasks of large jobs are often not cloned because of lack of cloning budget. Overall, we see that contention cause significant slowdown of transfers and are worth avoiding.

4.4 Delay Assignment

None of the downsides of CAC and CC can be ignored because they both have some negative impacts that are crucial in determining stragglers. A deficiency with both CAC and CC is that they do not distinguish stragglers from tasks that have normal variations in their progress. Keeping these in mind, a hybrid approach called delay assignment was proposed that first waits to assign the early upstream clones (like CAC), and thereafter proceeds without waiting for any remaining stragglers (like CC).

The setting of wait time plays an important role in delay assignment’s performance. The objective is to minimize the expected duration of a downstream task, which is the minimum of the durations of its clones. They were able to pick a wait duration that minimized the completion time by following these three steps:

1. Calculate the clone’s expected duration for reading each upstream output using T\_C and T\_E.
2. Use read durations of all clones of a task to estimate the overall duration of the task.
3. Find the delay that minimizes the task’s duration.

GRASS

2.1 Approximation Jobs

Approximation jobs are explored across two dimensions – {\em deadline-bound} jobs and {\em error-bound} jobs. {\em Deadline-bound} jobs strive to maximize the accuracy of their result within a specified time limit whereas {\em error-bound} jobs strive to minimize the time taken to reach a specified error limit in the result.

Approximation jobs require scheduler to prioritize the appropriate subset of their tasks depending on the deadline or error bound. It is important because of cluster heterogeneities and multi-waved jobs (number of tasks is greater than available compute slots). According to the previous literatures, the trend of multi-waved jobs increases with smaller tasks.

2.2 Challenges

The main challenge in prioritizing tasks of approximation jobs is presence of stragglers. The widely adopted technique to handle them is speculation. This is a reactive technique that spawns speculative copies for tasks that are considered to be stragglers. The earliest among the original and speculative copies is picked while the rest are killed. While scheduling a speculative copy makes the task finish faster and thereby increases accuracy, they also compete for compute slots with the unscheduled tasks.

Therefore, our problem is to dynamically prioritize tasks based on the deadline/error-bound while choosing between speculative copies for stragglers and unscheduled tasks.

3.1.1 Deadline-Bound Jobs

In the absence of speculation, a different policy is considered known as Shortest Job First (SJF) that schedules the task with the smallest processing time. In many cases, it can be proven to minimize the number of incomplete tasks, thus maximizing the number of tasks completed. ``Thus, without speculation, SJF finishes the most tasks before the deadline \cite:{anan:grass}.’’

If speculation is allowed, a neutral approach would be to allow the currently running tasks to be placed in the queue, and to choose the task with the smallest size. If the chosen task has a copy currently running, we check that the speculative copy being considered provides a benefit. So, the next task to run is still chosen according to SJF, only now speculative copies are also considered. This policy is called {\em Greedy Speculative (GS) scheduling}, because it picks the next task to schedule greedily (the one that finishes the quickest), and thus improve the accuracy the earliest *at* present.

To account the opportunity cost of scheduling a speculative copy, {\em Resource Aware Speculative (RAS) scheduling} speculates only if it saves both time and resources. Thus, the sum of the resources used by the speculative and original copies, when running simultaneously, must be less than letting just the original copy finish.

======🡺>>>>>>>> insert figure 1 from anan:grass

Even though figure 1 shows better performance for RAS, it is not uniformly better than GS. In particular, RAS’s cautious approach can backfire if it over-estimates the opportunity cost. Looking at the figure, if the deadline of the job were reduced from 6 time units to 3 time units instead, GS performs better than RAS. At the end of 3 time units, GS has led to three completed tasks while RAS has little to show for its resource gains by speculating T1. From this, we can see that the value of deadline and the number of waves are two important factors in determining scheduling technique.

3.1.2 Error-Bound Jobs

The goal of error-bound hobs is to minimize the makespan of the tasks needed to achieve the error limit. Thus, instead of SJF, Longest Job First (LJF) is more preferred.

t\_rem = remaining duration of a running task

t\_new = duration of a new copy

===========🡺>>>>>>>>>>insert figure 2 from anan:grass

Figure 2 presents an illustration of GS and RAS for an error-bound job with 6 tasks and 3 compute slots. The t\_rem and t\_new values are at 5 time units. GS decides to launch a copy of T3 as it has the highest t\_rem. RAS conservatively avoids doing so. Consequently, when the error limit is high (say, 40\%) GS is quicker, but RAS is better when the limit decreases (to, say, 20\%).

There are three guidelines that need to be followed before building speculation algorithm GRASS. They are:

Guideline 1: During the early waves of a job, speculation is only valuable if task durations are extremely heavy tailed

Guideline 2: During the final wave of a job, speculate aggressively to fully utilize the allotted capacity

Guideline 3: For jobs that require more than two waves, RAS is near-optimal. GS is near-optimal for jobs that require fewer than two waves

4 GRASS Speculation Algorithm

We can build GRASS keeping the guidelines as baseline. To achieve those solutions, there are two possibilities we have to look into: deadline-bound jobs and error-bound jobs. For deadline-bound jobs, we should switch from RAS to GS when the time to the deadline is sufficient for at most two waves of tasks. One the other hand, for error-bound tasks, we should switch when the number of (unique) scheduled tasks needed to satisfy the error-bound makes up to two waves.

But there are some challenges when implementing these ideas. First, identifying the final two waves of tasks is difficult in practice because tasks are not scheduled at explicit wave boundaries but rather when slots open up. In addition, the wave-width of jobs varies considerably depending on cluster utilization, so does task durations. In light of these difficulties, we interpret the guideline as follows: RAS is better when the deadline is loose, or the error limit is low; otherwise GS performs better.

4.1 Learning the Switching Point

An ideal approach would accumulate enough samples of job performance based on switching to GS at different points. For deadline-bound jobs, this is decided by remaining time to the deadline. For error-bound jobs, this is decided by the number of tasks to complete towards meeting the error.

In an ideal situation, an incoming job starts with RAS and periodically compares samples of jobs smaller than its size during its execution to check if it is better to switch to GS. It checks by using its remaining work at any point and continues with RAS until the optimal switching point turns out to be at present. This process is performed periodically during the job’s execution.

But the size of job alone is insufficient to calculate the optimal switching point. That’s why it is augmented with the number of waves, which is approximated using current cluster utilization. Another measure that plays an important role in deciding optimal switching point is the estimation accuracy of t\_rem and t\_new. As a result, GRASS obtains samples of job performance with both GS and RAS across values of deadline/error-bound, estimation accuracy of t\_rem and t\_new, and cluster utilization. These three factors are used collectively to decide when (and if) to switch from RAS to GS.

4.2 Generating Samples