Using Straggler Replication to Reduce Latency in Large-scale Parallel Computing

Da Wang [*]
Two Sigma Investments
New York, NY, USA
dawang@alum.mit.edu

Gauri Joshi [†]
EECS Dept., MIT
Cambridge, MA, 02139, USA
gauri@mit.edu

Gregory Wornell
EECS Dept., MIT
Cambridge, MA, 02139, USA
gww@mit.edu

ABSTRACT

In cloud computing jobs consisting of many tasks run in parallel, the tasks on the slowest machines (straggling tasks) become the bottleneck in the completion of the job. One way to combat the variability in machine response time is to add replicas of straggling tasks and wait for one copy to finish. Using the theory of extreme order statistics, we analyze how task replication reduces latency, and its impact on the cost of computing resources. We also propose a heuristic algorithm to search for the best replication strategies when it is difficult to model the empirical behavior of task execution time and use the proposed analysis techniques. Evaluation of the heuristic policies on Google Trace data shows a significant latency reduction compared to the replication strategy used in MapReduce.

Keywords

scheduling, parallel computation, task replication

1. INTRODUCTION

Applications such as Google search, Dropbox, Netflix need to perform enormous amounts of computing on the cloud. Recently, cloud computing is also being offered as a service by Amazon S3, Microsoft Azure etc. , where users can rent machines by the hour to run their computing jobs. The large-scale sharing of computing resources makes cloud computing flexible and scalable.

Cloud computing frameworks such as MapReduce and Hadoop employ massive parallelization to reduce latency. Large jobs are divided into hundreds of tasks that can be executed parallely on different machines. Several algorithms used in optimization and machine learning fall into the class of "embarrasingly parallel" computation, and can be easily divided into independent parallel tasks.

The execution time of a task on a machine is subject to stochastic variations due to co-hosting, virtualization and other hardware and network variations [7]. Thus, the key challenge in executing a job with a large number of tasks is the latency in waiting for the slowest tasks, or the "stragglers" to finish. As pointed out in [7, Table 1], the latency of executing many parallel tasks could be significantly larger (140 ms) than the median latency of a single task (1 ms).

In this work we provide a mathematical framework to analyze how replication of straggling tasks affects the latency, and the cost of computing resources.

1.1 Related prior work

The idea of replicating tasks in parallel computing has been recognized by system designers [6], and first adopted at a large scale via the "backup tasks" in MapReduce [5]. A line of systems work [2, 9] and references therein further developed this idea. While task replication has been studied in systems literature and also adopted in practice, there is not much work on careful mathematical analysis of replication strategies. Replication strategies are analyzed in [11], mainly for the single task case. In this paper we consider task replication for a job consisting of a large number of tasks, which corresponds closely to today's large-scale cloud computing frameworks. Note that the use of redundancy to reduce latency has also attracted attention in other contexts such as cloud storage and networking [8, 10].

1.2 Our contributions

To our knowledge, this work presents the first rigorous analysis of how the tail of the task execution time (heavy, light or exponential tail) affects the trade-off between the latency and the cost of computing resources. In particular for heavy tail distributions e.g. Pareto, we identify scenarios where the latency and computing cost can be reduced simultaneously. We also propose a heuristic algorithm to search for a good task replication policy when it is hard to use the proposed analysis techniques for the empirical distribution of task execution time.

2. PROBLEM FORMULATION

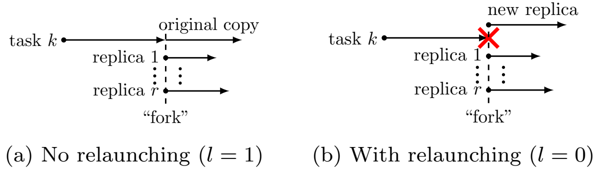
2.1 System Model

Consider a job consisting of n *parallel tasks*, where n is large. We assume the execution time of each task on a machine is independent and identically distributed (i.i.d.) according to F_X, where F_X denotes the cumulative distribution function (CDF) of random variable X. The distribution F_X accounts for the variability in the machine response due to factors such as congestion, queueing, virtualization, and competing jobs being run on the same machines. Modeling all existing jobs as exogenous factors could be suboptimal from a system designer's view, but is reasonable from the view of a user who is renting machines from a cloud com-

[*]Da Wang was affiliated with EECS Dept. MIT when this research was conducted.

[†]Da Wang and Gauri Joshi contributed equally to this work.

Copyright is held by author/owner(s).



(a) No relaunching ($l = 1$) (b) With relaunching ($l = 0$)

Figure 1: Single-fork policies with and without relaunching.

puting service and has no control over other jobs sharing the resources. We also assume that the number of machines the user rents from the cloud is much larger than the number of tasks in the given job, such that each new task (or new replica) is assigned to a new machine.

In the rest of the paper we use $X_{j:n}$ to denote the j-th smallest of the n random variables $X_1, X_2, \cdots, X_n \sim F_X$.

2.2 Scheduling Policy

A *scheduling policy* or *scheduler* assigns tasks to different machines, possibly at different time instants. We assume that the scheduler receives instantaneous feedback notifying it when a machine finishes its assigned task. But there is *no intermediate feedback* indicating the status of processing of a task. When the scheduler receives notification that at least one replica of each of the n tasks has finished, it *kills all* the residual running replicas. We focus on a class of policies called single-fork policies, defined as follows.

Definition 1 (Single-fork Scheduling policy). *A* single-fork scheduling policy $\pi_{\text{SF}}(p, r, l)$ *launches all n tasks at time 0. It waits until $(1-p)n$ tasks finish. For the pn straggling tasks, it chooses one of the following two actions:*

- ***replicate without relaunching** ($l = 0$): launch r new replicas;*
- ***replicate with relaunching** ($l = 1$): kill the original copy and launch $r + 1$ new replicas.*

When the earliest replica of a task finishes, all the other replicas are terminated.

We use l to denote the number of original replicas of each task remaining after the forking point. Hence $l = 0$ when the original replica is killed and restarted, and $l = 1$ otherwise. Note that for both relaunching ($l = 0$) or no relaunching ($l = 1$), there are a total of $r + 1$ replicas running after the forking point. The effect of r and l on the replication of straggling tasks is illustrated in Fig. 1.

For simplicity of notation we assume that p is such that pn is an integer. We note that $p = 0$ corresponds to running n tasks in parallel and waiting for all to finish, which is the baseline case without any replication or relaunching.

Remark 1 (Backup tasks in MapReduce). *The idea of 'backup' tasks used in Google's MapReduce [5], corresponds to a single-fork policy with $r = 1$ and $l = 1$. The value of p is tuned dynamically and hence not specified in [5].*

2.3 Performance metrics

We now define the metrics of latency and the computing cost. In later sections we analyze their trade-off and provide insights on when and how replication is useful.

Definition 2 (Expected Latency). *The expected latency $\mathbb{E}[T]$ is the expected value of T, the time taken for at least one replica of each of the n tasks to finish.*

Definition 3 (Expected Cost). *The expected computing cost $\mathbb{E}[C]$ is the sum of the running times of all machines, normalized by n, the number of tasks in the job. The running time is the time from when the task is launched on a machine, until it finishes, or is killed by the scheduler.*

For a user of a cloud computing service such as the Amazon Web Service (AWS), which charges the user by time and number of machines used, the money paid by the user to rent the machines is proportional to $\mathbb{E}[C]$.

3. SINGLE-FORK POLICY ANALYSIS

In this section we analyze the trade-off between the performance metrics $\mathbb{E}[T]$ and $\mathbb{E}[C]$, and develop insights into choosing the best single fork policy $\pi_{\text{SF}}(p, r, l)$. We observe that the tail behavior (heavy, light or exponential) of F_X is the key factor in characterizing the latency-cost trade-off. All proofs are omitted due to space limitations, but can be found in the extended version [12] available online.

3.1 Performance characterization

First let us define random variable Y, the residual time after forking when the earliest replica of a straggling task finishes. Its distribution F_Y can be expressed in terms of F_X as given by Lemma 1 below.

Lemma 1 (Residual Straggler Execution Time). *As $n \to \infty$, the tail distribution $\bar{F}_Y$ of the residual execution time (after the forking point) of each of the pn straggling tasks is*

$$\bar{F}_Y(y) = \begin{cases} \bar{F}_X(y)^{r+1} & \text{if } l = 0, \\ \frac{1}{p}\bar{F}_X(y)^r \bar{F}_X\left(y + F_X^{-1}(1-p)\right) & \text{if } l = 1. \end{cases} \quad (1)$$

For example, for $r = 2$ and $l = 0$, the tail of distribution $\bar{F}_Y = \bar{F}_X^2$, because two identical replicas with distribution F_X are launched at the forking point. For a job with a large number of tasks n, the expected latency and cost can be expressed in terms of F_X, F_Y and the single-fork policy parameters p, r and l as given by Theorem 1 below.

Theorem 1 (Single-Fork Latency and Cost). *For a computing job with n tasks, and task execution time distribution F_X, the latency and cost metrics as $n \to \infty$ are*

$$\mathbb{E}[T] = F_X^{-1}(1-p) + \mathbb{E}[Y_{pn:pn}], \quad (2)$$

$$\mathbb{E}[C] = \int_0^{1-p} F_X^{-1}(h)dh + pF_X^{-1}(1-p) + (r+1)p \cdot \mathbb{E}[Y], \quad (3)$$

where $\mathbb{E}[Y_{pn:pn}]$ is the expected maximum of pn i.i.d. random variables drawn from F_Y.

We now give a sketch of the proof of Theorem 1. The latency T of a single fork policy $\pi_{\text{SF}}(p, r, l; n)$ can be decomposed into $T^{(1)}$, the time to execute the first $(1-p)n$ tasks, and $T^{(2)}$, the time to execute the pn straggling tasks. The expected value of $T^{(1)}$ is

$$\mathbb{E}[T^{(1)}] = \mathbb{E}[X_{(1-p)n:n}] \approx F_X^{-1}(1-p) \quad \text{for large } n, \quad (4)$$

where (4) follows from the Central Value Theorem (Theorem 10.3 in [3]) which states that the $((1-p)n)^{th}$ order statistic of n i.i.d random variables concentrates sharply around $F_X^{-1}(1-p)$ as $n \to \infty$.

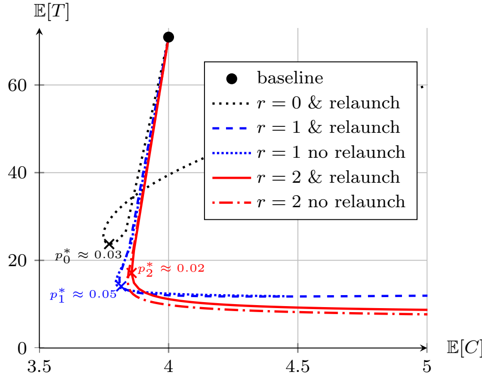


Figure 2: Expected Latency versus cost when F_X is $\mathsf{Pareto}\,(2,2)$, $n = 400$ tasks, and varying p from 0 to 1 along each curve.

The second part of the latency, $T^{(2)}$ is the maximum of the residual time Y for each of the pn straggling tasks finish. Its behavior for $n \to \infty$ is given by Lemma 2 below.

LEMMA 2. *The asymptotic behavior of* $\mathbb{E}[Y_{pn:pn}]$ *as* $n \to \infty$ *is given by*

$$\mathbb{E}[Y_{pn:pn}] = \begin{cases} \tilde{a}_{pn}\gamma_{\text{EM}} + \tilde{b}_{pn} & F_X \in \text{DA}\,(\Lambda)\,, \\ \tilde{a}_{pn}\Gamma\left(1 - \frac{1}{(r+1)\xi}\right) & F_X \in \text{DA}\,(\Phi_\xi)\,, \\ \tilde{b}_{pn} - \tilde{a}_{pn}\Gamma\left(1 + \frac{1}{((1-l)r+1)\xi}\right) & F_X \in \text{DA}\,(\Psi_\xi)\,. \end{cases}$$

where the terms $\tilde{a}_{pn}$ *and* $\tilde{b}_{pn}$ *can be determined using the Extreme Value Theorem (Theorem 1.1.3 in [4]).* $\text{DA}\,(\cdot)$ *is the domain of attraction* F_X*, and it can be one of the following: Gumbel (*$\text{DA}\,(\Lambda)$*), Frechet (*$\text{DA}\,(\Phi_\xi)$*), and Weibull (*$\text{DA}\,(\Psi_\xi)$*). The symbol* γ_{EM} *is the Euler-Mascheroni constant, and* $\Gamma(\cdot)$ *is the Gamma function.*

The domain of attraction of a distribution depends on its tail behavior (exponential, heavy or light). For example, exponentially decaying distributions belong to $\text{DA}\,(\Lambda)$ while heavy tailed distributions belong to $\text{DA}\,(\Phi_\xi)$.

A comparison of the latency and cost in Theorem 1 with simulation indicates that the metrics obtained from analytical calculation is very close to the actual performance when the number of tasks $n \geq 100$. Please see [12] for the plot comparing analysis and simulation.

3.2 Examples of the Effect of Tail Behavior

We now use two canonical distributions, the Pareto distribution (heavy, polynomially decaying tail) and the Shifted Exponential distribution (exponential tail), to demonstrate how the the tail of F_X affects the latency-cost trade-off.

Fig. 2 shows the latency versus the computing cost when F_X is $\mathsf{Pareto}\,(\alpha, x_m)$ with parameters $\alpha = 2$ and $x_m = 2$, and p varying from 0 to 1 along each curve. The black dot is the baseline case ($p = 0$) without replication where we simply wait for the original copies of all n tasks to finish. The baseline case is also equivalent to the policies with $r = 0, l = 1$ and any p. We observe that a small amount of replication (small p and r) can reduce latency significantly in comparison with the baseline case.

Intuition suggests that replicating earlier (larger p) and more (higher r) will increase the cost $\mathbb{E}[C]$. But Fig. 2 shows that this is not necessarily true, and that it is possible to reduce latency (from 70 to about 15 for $r = 1$ and $r = 2$ cases) with a decrease in cost! However this benefit diminishes as p and r increase above a certain threshold. For example, for the $r = 1$ and relaunch ($l = 0$) case, we can show that all policies with $p < p_1^* \approx 0.05$ are sub-optimal in both $\mathbb{E}[T]$ and $\mathbb{E}[C]$, where p_1^* is marked in Fig. 2. Similarly, for cases $r = 0$ and $r = 2$, the sub-optimal ranges are $[0, p_0^*]$ and $[0, p_2^*]$, with p_0^* and p_2^* as shown in Fig. 2.

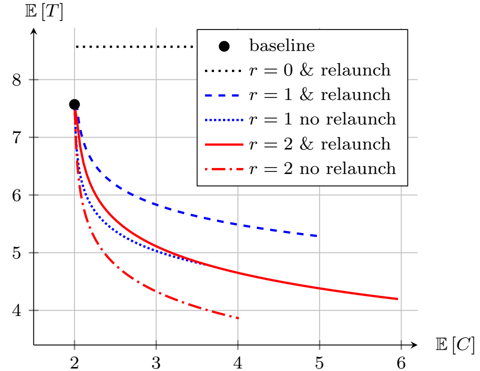


Figure 3: Expected Latency versus cost when F_X is shifted exponential $\mathsf{SExp}\,(1,1)$, $n = 400$ tasks, and varying p from 0 to 1 along each curve.

Next we consider that F_X follows the shifted exponential distribution (denoted by $\mathsf{SExp}\,(\Delta, \mu)$), which is a pure exponential with rate μ, with a constant additive shift Δ. Fig. 3 shows the latency-cost trade-off for $n = 400$ with $F_X = \mathsf{SExp}\,(1,1)$. Unlike Fig. 2 there is no range of p for which both latency and cost decrease (or increase) simultaneously. For any p and r, no relaunching gives lower latency than relaunching because the tail is not heavy, and thus it is better to not kill the original straggling replica.

4. HEURISTIC ALGORITHM

In certain practical systems it may be difficult to fit a well-known distribution to the empirical behavior of the task execution time, thus making the latency-cost analysis using the framework presented in Section 3 hard. In this section we present an algorithm that uses traces of task execution times to search for the best single-fork policy $\pi_{\text{SF}}\,(p, r, l)$, and evaluate it using Google trace data [1].

4.1 Estimation of Latency and Cost Metrics

To find the estimates $\tilde{T}$ and $\tilde{C}$ of the latency and cost for a given policy $\pi_{\text{SF}}\,(p, r, l)$, we first construct the empirical CDF $\hat{F}_X$ from experimental traces of start and finish times of tasks on a large cluster of machines. By drawing samples from $\hat{F}_X$ we can simulate the running of n tasks of a job, and the replicas of the pn straggling tasks and find T and C for that job. We repeat this procedure for m jobs and set $\tilde{T}$ and $\tilde{C}$ to the means of the m samples.

4.2 Heuristic Search for the Best Policy

We present a heuristic algorithm to search for the single-fork policy that minimizes the objective function $J \triangleq \tilde{T} + \mu\tilde{C}$ where μ represents the relative priority for minimizing $\tilde{C}$.

For a given p, we first find the optimal r and l and then perform gradient descent on p. This is repeated for k iterations. To optimize r and l we keep increasing r by 1 until J decreases. For each r, we set l to the value (0 or 1) which gives a smaller J. Note from (2) and (3) that $\mathbb{E}[T]$ and $\mathbb{E}[C]$

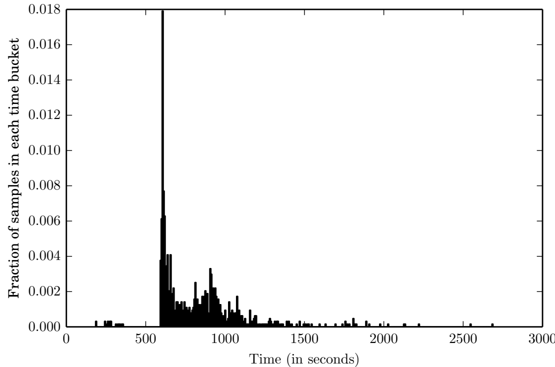


Figure 4: Normalized histogram of the task execution times for a Google cluster job with $n = 1017$ tasks.

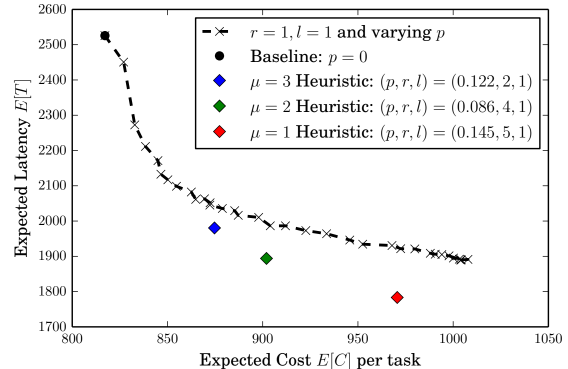


Figure 5: For a Google cluster job with 1017 tasks, heuristic policies give a better $\mathbb{E}[T] - \mathbb{E}[C]$ trade-off than the $r = 1$ and $l = 1$ (parameters of back-up tasks in MapReduce) case.

are convex in r, but not in p and l. Thus, the gradient descent may not converge to the optimal policy.

The policy found by this heuristic can be used for future jobs with similar task execution time statistics. This is true in several applications, where the same computation is performed repeatedly with different parameters.

4.3 Demonstration using Google Traces

The Google Trace data [1] gives timestamps of events such as SCHEDULE, EVICT, FINISH, FAIL, KILL etc. for each of the tasks of computing jobs that are run on Google's machine clusters. We consider the difference between the SCHEDULE and FINISH timestamps as the task execution time, and construct the empirical distribution $\hat{F}_X$.

We consider a large Google cluster job with $n = 1017$ tasks. Its normalized histogram plotted in Fig. 4 shows heavy-tail behavior of task execution time. We run the heuristic search on this empirical $\hat{F}_X$, with parameters $m = 500$ jobs used to estimate $\hat{T}$ and $\hat{C}$ and $k = 25$ iterations of gradient descent. The latency-cost trade-offs of the heuristic policies found by the algorithm are shown in Fig. 5. The plot also shows the estimated latency-cost trade-off for $r = 1$ and $l = 1$ as p varies from 0 to 1, which are the parameters of the back-up tasks option in MapReduce as described in Remark 1. Adding redundancy, that is $r \geq 1$ significantly reduces latency for a small cost, in comparison with the baseline case ($p = 0$). The heuristic algorithm finds policies with $r > 1$ that give lower latency for the same cost, as compared to the policies with $r = 1$ and $l = 1$. Also note that the latency reduction is more when μ is smaller, that is, the priority given to minimizing the cost is lower.

5. CONCLUDING REMARKS

In this paper we present a mathematical framework to analyze how replication of the slowest tasks in a job (the stragglers) affects the latency and the computing cost. We characterize the latency-cost trade-off for a set of replication strategies called single-fork policies. We also propose a heuristic algorithm to find the best scheduling policy. Experiments on Google Trace data show that the policies found by the algorithm can give a better latency-cost trade-off than the back-up tasks option used in MapReduce.

Although we focus on single-fork policies in this paper, the analysis can be generalized to multi-fork policies, where replicas of straggling tasks can be launched multiple times during the job execution. Another promising research direction is to develop an online algorithm that simultaneously learns the execution time distribution F_X and launches replicas, instead of estimating F_X using historical traces.

6. REFERENCES

[1] Google cluster data. http://code.google.com/p/googleclusterdata/.

[2] G. Ananthanarayanan, A. Ghodsi, and I. Stoica S. Shenker. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, pages 185–198, 2013.

[3] H. A. David and H. N. Nagaraja. *Order statistics.* John Wiley, Hoboken, N.J., 2003.

[4] L. de Haan and A. Ferreira. *Extreme value theory an introduction.* Springer, New York, 2006.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] G. Ghare and S. T. Leutenegger. Improving speedup and response times by replicating parallel programs on a SNOW. In *International conference on Job Scheduling Strategies for Parallel Processing*, pages 264–287, January 2005.

[7] J. Dean and L. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[8] G. Joshi, Y. Liu, and E. Soljanin. On the Delay-Storage Trade-off in Content Download from Coded Distributed Storage Systems. *IEEE JSAC*, pages 989 – 997, May 2014.

[9] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *ACM SOSP*, pages 69–84, 2013.

[10] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. *CoNEXT*, pages 283–294, 2013.

[11] D. Wang, G. Joshi, and G. Wornell. Efficient task replication for fast response times in parallel computation. *ACM Sigmetrics short paper*, June 2014.

[12] D. Wang, G. Joshi, and G. Wornell. Using straggler replication to reduce latency in large-scale parallel computing (extended version). *arXiv:1503.03128 [cs.dc]*, March 2015.