Learning Effective Straggler Mitigation from Experience and Modeling

Mehmet Fatih Aktaş [1] **Emina Soljanin** [1]

Abstract

We address the problem of scheduling multi-task jobs with redundancy in a Master-Worker cluster, using a combination of Reinforcement Learning (RL) and mathematical modeling. We firstly use RL techniques to learn from realistic experience that *right amount* of redundancy shall be added into *small enough* jobs, and only when cluster operates under *low enough* load. Building on these principles, we derive a simple scheduling policy and present an approximate analysis of its performance. Specifically, we derive expressions to decide when and which jobs should be scheduled with how much redundancy. We show that policy that we derive in this way performs as good as the more complex policies that are devised by RL.

1. Introduction

Large scale compute systems experience random performance degradation, which lead to significant variability in task execution times (1; 2; 3; 4; 5; 6; 7). *Redundancy* has long been used in production systems to attain predictable performance in the presence of runtime variability (8; 2). The idea is to speculatively launch multiple copies for the same task and wait only for the fastest one to complete, hence avoid having to wait for the slow running tasks, or famously *stragglers*. Task replication has been shown to be effective for straggler mitigation in both practice (5; 9; 1; 10) and theory (11). *Erasure coding* implements a more general form of redundancy and has been shown to mitigate stragglers by incurring smaller redundant load on the system (12; 13; 14; 15). Coding techniques have been applied for straggler mitigation in common distributed linear computation (16; 17; 18) or iterative optimization algorithms (19**?** ; 20; 21; 22; 23; 24) that empower large scale ML.

Despite the plethora of papers devising new redundancy techniques, no clear guidelines could yet be found on how to schedule compute jobs with redundancy. Redundant tasks might exert additional load on the system, hence the guidance for introducing them is usually conservative, e.g., replicas are launched only for "short" tasks (1), or only for tasks that seem to straggle (3), or issued only to idle servers (25). However, even then important questions have yet to be addressed. Jobs arriving to a system consist of different number of tasks, request varying resource capacity and have random service times. How should we quantify the *demand* of a job? Which jobs are short enough to be scheduled with redundancy, and with how much redundancy? Excessive redundancy may aggravate the job slowdowns, or even drive system to instability. At what level of offered load does redundancy start to hurt performance?

[1]Department of ECE, Rutgers University, NJ, USA. Correspondence to: Mehmet Fatih Aktaş <mehmet.aktas@rutgers.edu>, Emina Soljanin <emina.soljanin@rutgers.edu>.

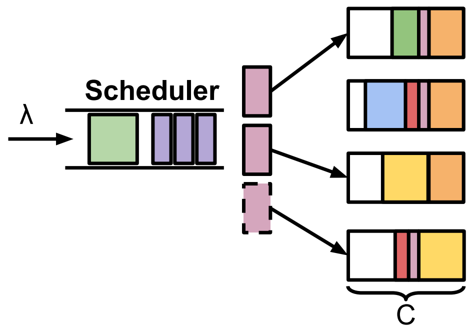

Figure 1: System model for scheduling with redundancy. A job of two tasks (solid) gets scheduled with a redundant task (dashed) such that any two of the three tasks is sufficient for its completion.

2. System Model

System architecture and job arrivals: We consider a Master-Worker compute cluster as implemented in Hadoop Yarn (26), Mesos (27) or Kubernetes (28) (see Fig. 1). The cluster consists of a single scheduler (master) managing N nodes (workers), each with capacity C. Compute jobs arrive as a Poisson process of rate λ, and each job consists of a random number (k) of tasks. Real compute jobs are collection of one or more usually identical tasks (29), so we assume tasks within the same job request equal (r) amount of capacity and have the same size (b), i.e., minimum service time. We assume k, r and b are independently sampled from random variables K, R and B for each arriving job.

Number of tasks K and task sizes B in real compute jobs exhibit heavy tail (30; 29). B is observed to be commonly distributed as Pareto in practice, hence we model $B \sim \mathrm{Pareto}$ with a minimum value $b_{\min}$ and tail index β. We model $K \sim$ Zipf with an exponent of 1 and a maximum value of $k_{\max}$. With our policy search using Deep-RL, we found that distribution of R is not a significant factor in finding a good scheduling policy, hence we set $R = 1$ to keep the discussion simpler throughout, although the study presented here easily extends to the case with random R.

Runtime variability: We model the runtime variability with a random variable S that is identically and independently distributed (i.i.d.) across different nodes and tasks. Once a task of size b starts execution, it samples a straggling factor s from S and takes $s \times b$ of time to complete. This model is adopted in (25) and shown to support the experimental evidence. To capture the significant variability that is observed in practice, we model S as a Pareto with minimum value 1 and tail index α.

Scheduling: Number of tasks and their requested capacity are known for each arriving job (as in Kubernetes, Yarn, Mesos), and task sizes are also assumed to be known here. Jobs wait in a first-in first-out queue and the job at the head of the queue gets dispatched as soon as enough resources become available in the cluster to fit all its tasks (both initial and redundant). We here focus on scheduling *MDS coded redundancy*. Scheduler decides how many redundant tasks to add into each job. When a job of k tasks is scheduled with $n - k$ MDS parity tasks, the job completes as soon as the fastest k of its tasks finish service, then the remaining $n - k$ tasks are immediately removed from service.

System configuration: We built a cluster simulator using SimPy (31) to implement our system model. Results presented in this paper are computed by setting the system parameters $N = 20$, $C = 10$, $k_{\max} = 10$, $b_{\min} = 10$, $\beta = 1.5$, $\alpha = 3$, and varying arrival rate λ to change the offered load on the cluster. Reported simulation results are computed by sampling from 30 different runs, each executed until the first 100,000 arriving jobs finish execution.

3. Learning how to schedule with redundancy

RL formulation: In the problem of scheduling, learning environment is the compute cluster and learning agent is the scheduler. Scheduler interacts with the cluster by adding redundancy to arriving jobs, with the aim of *minimizing job slowdowns*. Slowdown experienced by a job is the total time it spends in the system divided by its minimum service time.

We use Deep Q-learning since it is known to be data-efficient on Markovian environments such as our system. While scheduling a job, scheduler observes the load at every cluster node, and the number of tasks k, resource request r and task size b for the job. Scheduler decides (acts) on the number of redundant tasks to add into each job, and collects the negative of the slowdown experienced by the jobs as the reward for each scheduling decision.

What does Deep-RL learn? We evaluate our implementation of Deep-RL by running it with our cluster simulator under varying offered load ρ on the cluster. We found that learning a scheduling policy by only observing the *job demand* $k \times r \times b$ performs as good as the policy learned by observing k, r and b individually. We observed that Deep-RL devises a natural strategy; it learns to introduce gracefully less redundancy for larger values of job demand or ρ. Load on the cluster nodes assigned for the job's tasks did not turn out to influence the scheduling decision much.

4. Scheduling small jobs with redundancy

Building on the policy learned by Deep-RL, we propose $\mathrm{Redundant\text{-}small}$ policy that expands a job of k tasks into $\lceil rk \rceil$ tasks with redundancy only if the job's demand is $\leq d$.

Latency and Cost: We refer to the execution time of an arbitrary job as Latency, and the total resource-usage time it consumes throughout its execution as Cost. When scheduled with no redundancy, a job of k tasks each with a service time of b completes once its slowest task finishes, hence its Latency $\sim b \times S_{k:k}$[1] and its Cost $\sim k \times b \times S$. When scheduled together with $n-k$ redundant tasks, job will finish as soon as any k of its n tasks finish, hence its Latency $\sim b \times S_{n:k}$ and its Cost $\sim b \times \left(\sum_{i=1}^{k-1} S_{n:i} + (n-k)S_{n:k}\right)$ (14). A job will be scheduled with redundancy only if its demand $D = kB \leq d$. By the law of total expectation,

$$\begin{aligned}\mathbb{E}[X] = \mathbb{E}[X \mid D \leq d] \Pr\{D \leq d\} \\ + \mathbb{E}[X \mid D > d] \left(1 - \Pr\{D \leq d\}\right).\end{aligned} \tag{1}$$

where X is the placeholder for Latency or Cost, and

$$\begin{aligned}
\Pr\{D \leq d\} &= \Pr\{kB \leq d\} = \mathbb{E}_k\left[\Pr\{B \leq d/k\}\right], \\
\mathbb{E}[\text{Latency} \mid D > d] &= \mathbb{E}_k\left[\mathbb{E}[S_{k:k}]\, \mathbb{E}[B \mid B > d/k]\right], \\
\mathbb{E}[\text{Latency} \mid D \leq d] &= \mathbb{E}_k\left[\mathbb{E}[S_{n:k}]\, \mathbb{E}[B \mid B \leq d/k]\right], \\
\mathbb{E}[\text{Cost} \mid D > d] &= \mathbb{E}[S]\, \mathbb{E}_k\left[k\, \mathbb{E}[B \mid B > d/k]\right], \\
\mathbb{E}[\text{Cost} \mid D \leq d] &= \mathbb{E}_k\left[\mathbb{E}[C_{n,k}]\, \mathbb{E}[B \mid B \leq d/k]\right].
\end{aligned}$$

where $n = \lceil kr \rceil$, $\mathbb{E}_k$ denotes expectation with respect to random variable k, and $C_{n,k} = \sum_{i=1}^{k} S_{n:i} + (n-k)S_{n:k}$. Using the results given in (14), we find

$$\begin{aligned}
\mathbb{E}[S_{n:k}] &= \frac{\Gamma(n+1)}{\Gamma(n-k+1)} \frac{\Gamma(n-k+1-1/\alpha)}{\Gamma(n+1-1/\alpha)}, \\
\mathbb{E}[C_{n,k}] &= \frac{n}{\alpha - 1}\left(\alpha - (1 - k/n)\ \mathbb{E}[S_{n:k}]\right),
\end{aligned}$$

where the Gamma function $\Gamma(a) = \int_0^\infty u^{a-1} e^{-u} du$. Average load on the cluster is given by

$$\rho = \frac{\lambda}{NC} \mathbb{E}[\text{Cost}]. \tag{2}$$

$M/G/c$ approximation: Multi-server queueing systems can be analytically analyzed if each job takes up a fixed space in the system, e.g., as in $M/G/c$ queue. This is not the case in our system, but our idea is to make an approximation as follows. Suppose that each job is given a share of σ

[1] $X_{n:i}$ is ith smallest of n i.i.d. samples of random variable X.

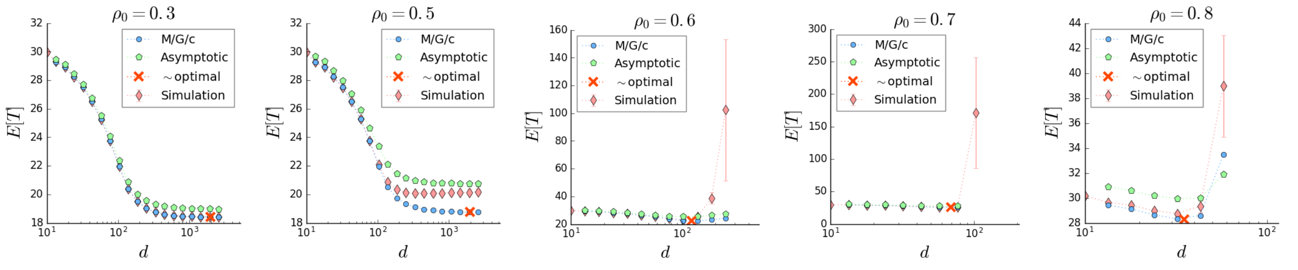

Figure 2: Average response time of Redundant-small with job expansion rate $r = 2$ under different values of offered load ρ_0. $M/G/c$ refers to the values estimated by the approximation given in Claim 1, and asymptotic refers to the same approximation at large scale limit.

capacity per unit time. Given that and the fact that total capacity available in the system is NC, it becomes natural to treat the system as an $M/G/c$ queue with NC/σ servers. On average, an arbitrary job consumes $\mathbb{E}[\text{Cost}]/\mathbb{E}[\text{Latency}]$ capacity per unit time. This is an unbiased estimator of σ and we use it to approximate σ. Then the number of servers c is approximated as $NC\ \mathbb{E}[\text{Latency}]/\mathbb{E}[\text{Cost}]$, which can be non-integer but we circumvent this by formulating the expressions to make them work with non-integer c. The most well known approximation on the average response time in $M/G/c$ queue is given by adjusting the average waiting time in $M/M/c$ queue as

$$\mathbb{E}[T_{M/G/c}] \approx \mathbb{E}[X] + \frac{\mathcal{C}^2+1}{2}\mathbb{E}[W_{M/M/c}], \qquad (3)$$

where X is the job service time in $M/G/c$ queue and $\mathcal{C}$ is the coefficient of variation of X (32). We know

$$\mathbb{E}[W_{M/M/c}] = \Pr\{\text{Queueing}\}\frac{\rho}{\lambda(1-\rho)},$$

where $\rho = \lambda\mathbb{E}[X]/c$ denotes the average load on a server. $\Pr\{\text{Queueing}\}$ denotes the probability that an arriving job waits in the queue before starting service and is given by

$$\Pr\{\text{Queueing}\} = \left(1 + (1-\rho)\frac{c!}{(c\rho)^c}\sum_{i=0}^{c-1}\frac{(c\rho)^i}{i!}\right)^{-1},$$

The exponential sum here can be written in terms of the incomplete Gamma function $\Gamma(a,x) = \int_x^\infty u^{a-1}e^{-u}du$, so

$$\Pr\{\text{Queueing}\} = \left(1 + (1-\rho)\frac{c\,e^{c\rho}}{(c\rho)^c}\Gamma(c,c\rho)\right)^{-1}. \qquad (4)$$

The form given above now is defined for non-integer c. At large scale limit, i.e., keeping ρ fixed while taking $c\rho \to \infty$,

$$\lim_{c\rho\to\infty}\Gamma(c,c\rho) = (c\rho)^{c-1}e^{-c\rho}.$$

Substituting this into (4),

$$\lim_{c\rho\to\infty}\Pr\{\text{Queueing}\} = (1 + (1-\rho)/\rho)^{-1} = \rho. \qquad (5)$$

Note that approximate expression in (3) requires the coefficient of variation $\mathcal{C}$ for the service time distribution. $\mathcal{C}$ is given for the approximate system as $\sqrt{\mathbb{E}[\text{Latency}^2]/\mathbb{E}[\text{Latency}]^2 - 1}$, where the second moment of Latency can be derived exactly the same way as we derived its first moment in (1).

Claim 1. *Average response time $\mathbb{E}[T]$ in the Master-Worker system described in Sec. 2 is approximated as*

$$\mathbb{E}[T] \approx \mathbb{E}[\textit{Latency}] + \frac{\mathbb{E}[\textit{Latency}^2]}{2\mathbb{E}[\textit{Latency}]^2}\Pr\{\text{Queueing}\}\frac{\rho}{\lambda(1-\rho)}, \qquad (6)$$

where $\mathbb{E}[\textit{Latency}]$ is given by (1) (and so is $\mathbb{E}[\textit{Latency}^2]$ similarly), ρ is given by (2), and $\Pr\{\text{Queueing}\}$ is given by (4) and by (5) at large scale limit.

Fig. 2 shows that average system response time estimated by our approximation follows the simulated values fairly closely. When the offered load ρ_0 (when no job is scheduled with redundancy) on the cluster is low ($\rho_0 \leq 0.5$), scheduling even very large jobs with redundancy (i.e., $d \to \infty$) increases system performance. When ρ_0 is high, redundancy improves performance until d reaches a threshold, beyond which increasing d hurts performance ($\rho_0 = 0.6$), or even drive the system to instability ($\rho_0 \geq 0.7$).

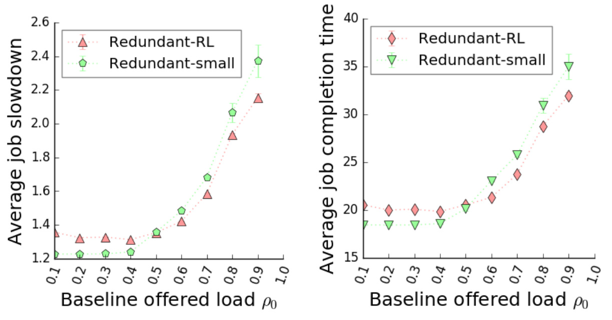

Figure 3: Redundant-RL vs. Redundant-small with $r = 2$ and d optimized using the approximation given in Claim 1.

Most importantly, our approximation allows accurately estimating the optimal d^* (shown with red-cross in Fig. 2). Fig. 3 shows that Redundant-small with approximately computed d^* performs as good as the more complex policies learned by Deep-RL. When offered load ρ_0 is low, d^* is set to a large value, while as ρ_0 gets higher, d^* is reduced gradually, mimicking the behavior learned by Deep-RL.

References

[1] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Osdi*, volume 10, page 24, 2010.

[5] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[6] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 7. ACM, 2013.

[7] Peter Garraghan, Xue Ouyang, Renyu Yang, David McKee, and Jie Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[9] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 283–294. ACM, 2013.

[10] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[11] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, and Samuel Zbarsky. Redundancy-d: The power of d choices for redundancy. *Operations Research*, 65(4):1078–1094, 2017.

[12] Gauri Joshi, Yanpei Liu, and Emina Soljanin. Coding for fast content download. In *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, pages 326–333. IEEE, 2012.

[13] Longbo Huang, Sameer Pawar, Hao Zhang, and Kannan Ramchandran. Codes can reduce queueing delay in data centers. In *Proceed. 2012 IEEE International Symposium on Information Theory (ISIT'12)*, pages 2766–2770.

[14] Mehmet Fatih Aktas, Pei Peng, and Emina Soljanin. Effective straggler mitigation: Which clones should attack and when? *SIGMETRICS Performance Evaluation Review*, 45(2):12–14, 2017.

[15] Mehmet Fatih Aktas, Pei Peng, and Emina Soljanin. Straggler mitigation by delayed relaunch of tasks. *SIGMETRICS Performance Evaluation Review*, 45(3):224–231, 2017.

[16] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *Advances In Neural Information Processing Systems*, pages 2092–2100, 2016.

[17] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover. Coded convolution for parallel and distributed computing within a deadline. In *Information Theory (ISIT), 2017 IEEE International Symposium on*, pages 2403–2407. IEEE, 2017.

[18] Qian Yu, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding. *arXiv preprint arXiv:1801.07487*, 2018.

[19] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 2017.

[20] Songze Li, Seyed Mohammadreza Mousavi Kalan, A Salman Avestimehr, and Mahdi Soltanolkotabi. Near-optimal straggler mitigation for distributed gradient methods. *arXiv preprint arXiv:1710.09990*, 2017.

[21] Netanel Raviv, Itzhak Tamo, Rashish Tandon, and Alexandros G Dimakis. Gradient coding from cyclic mds codes and expander graphs. *arXiv preprint arXiv:1707.03858*, 2017.

[22] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient coding: Avoiding stragglers in distributed learning. In *International Conference on Machine Learning*, pages 3368–3376, 2017.

[23] Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. Straggler mitigation in distributed optimization through data encoding. In *Advances in Neural Information Processing Systems*, pages 5434–5442, 2017.

[24] Wael Halbawi, Navid Azizan, Fariborz Salehi, and Babak Hassibi. Improving distributed gradient descent using reed-solomon codes. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 2027–2031. IEEE, 2018.

[25] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, and Benny Van Houdt. A better model for job redundancy: Decoupling server slowdown and job size. *IEEE/ACM Transactions on Networking*, 25(6):3353–3367, 2017.

[26] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[28] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10, 2016.

[29] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[30] Yanpei Chen, Archana Sulochana Ganapathi, Rean Griffith, and Randy H Katz. Analysis and lessons from a publicly available google cluster trace. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95*, 94, 2010.

[31] Norm Matloff. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2(2009), 2008.

[32] Noah Gans, Ger Koole, and Avishai Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing & Service Operations Management*, 5(2):79–141, 2003.