Rateless Codes for Distributed Non-linear Computations

Ankur Mallick, Sophie Smith, Gauri Joshi
Carnegie Mellon University
Pittsburgh, PA 15213
{amallic1,sophiesm,gaurij}@andrew.cmu.edu

***Abstract*—Machine learning today involves massive distributed computations running on cloud servers, which are highly susceptible to slowdown or straggling. Recent work has demonstrated the effectiveness of erasure codes in mitigating such slowdown for linear computations, by adding redundant computations such that the entire computation can be recovered as long as a subset of nodes finish their assigned tasks. However, most machine learning algorithms typically involve non-linear computations that cannot be directly handled by these coded computing approaches. In this work, we propose a coded computing strategy for mitigating the effect of stragglers on non-linear distributed computations. Our strategy relies on the observation that many expensive non-linear functions can be decomposed into sums of cheap non-linear functions. We show that erasure codes, specifically *rateless* codes can be used to generate and compute random linear combinations of these functions at the nodes such that the original function can be computed as long as a subset of nodes return their computations. Simulations and experiments on AWS Lambda demonstrate the superiority of our approach over various uncoded baselines.**

A full version of this paper is accessible at [1]

I. Introduction

Due to the advent of expensive yet embarrassingly parallel machine learning models, and the availability of cheap computing resources, most large scale machine learning models are now implemented in distributed fashion across multiple nodes on the cloud [2]. This typically involves writing an expensive function as a sum of multiple cheap functions which can then be parallelized in a MapReduce [3] fashion wherein each of the cheap functions is computed by a distinct node (worker), while a central node (master) aggregates and adds the results of the worker computations. Examples include distributed gradient descent [4] and distributed kernel ridge regression [5].

Distributed computations on the cloud are commonly bottlenecked by slow or unreliable nodes, called stragglers [6], whose presence can delay the entire computation. While systems like MapReduce [3] relied on simple solutions like task replication to mitigate the effect of stragglers, a recent line of work [7], [8], [9] has shown that adding redundancy using erasure codes can provide much better resilience to stragglers. These works rely on the fact that linear computations, such as the matrix-vector product $\mathbf{Ax}$, can be split into sub computations $\mathbf{A}_1\mathbf{x}$ and $\mathbf{A}_2\mathbf{x}$ where $\mathbf{A} = [\mathbf{A}_1^T \ \ \mathbf{A}_2^T]^T$ and a redundant computation $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{x}$. Three worker nodes store matrices $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_1 + \mathbf{A}_2$ respectively, and each multiplies its matrix with $\mathbf{x}$. Results from any two workers suffice to recover $\mathbf{Ax}$, and thus the system can tolerate one straggler.

The *linearity* of computations like matrix-vector multiplication enables the addition of redundancy using linear codes at no extra cost. Thus, in the above example, once $\mathbf{A}_1+\mathbf{A}_2$ have been pre-computed, the cost of computing $\mathbf{A}_1\mathbf{x}$ and $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{x}$ is the same. On the other hand, most machine learning computations such as gradients of neural network parameters [2] or kernel functions between training and test points [10] are *non-linear*, and the cost of computing coded functions can be higher than that of computing uncoded functions thus undoing some the benefits of adding redundancy. To see this consider a computation of the form $F(\mathbf{x}) = f_1(\mathbf{x}) + f_2(\mathbf{x})$ where F, f_1 and f_2 are *non-linear* functions. A naive approach to encoding this computation across three nodes would be to have the nodes compute $f_1(\mathbf{x})$, $f_2(\mathbf{x})$, and $f_1(\mathbf{x}) + f_2(\mathbf{x})$ respectively. In this case the cost of computing $f_1(\mathbf{x}) + f_2(\mathbf{x})$ is twice that of the other two computations since the non-linearity precludes pre-computation of $f_1 + f_2$. This extra cost at the third node can potentially negate the speedup expected from having to wait for only two out of three nodes.

In this work we show that sparse rateless codes, specifically LT codes [11] can mitigate the slowdown due to stragglers when computing non-linear functions of the form $F(\mathbf{x}) = \sum_{i=1}^{m} f_i(\mathbf{x})$ in a distributed fashion *without* significantly increasing the cost of computing coded symbols. This is because the sparsity of the encoded symbols in LT codes limits the cost of computing them while for dense codes like Reed-Solomon codes, the cost of computing an encoded symbol can be as large as that of the entire computation, as we described above. Moreover LT codes also provide significantly faster decoding complexity of $\mathcal{O}(m \log m)$ unlike the $\mathcal{O}(m^3)$ decoding complexity of Reed-Solomon codes. The fast encoding and decoding allows our scheme to scal to really large values of m in emerging distributed settings like serverless computing [12] which are known to suffer from significant straggling [13]. Simulations, and experiments on AWS Lambda[14], Amazon's serverless computing framework, show that LT codes offer significant speedup over uncoded baselines for distributed computation of *non-linear* functions.

Related Work. While there is a long line of work on coded computing [7], [8], [9] (incomplete list), most works have been restricted to coding for linear computations. Among the few works that have looked at non-linear computations, [15]

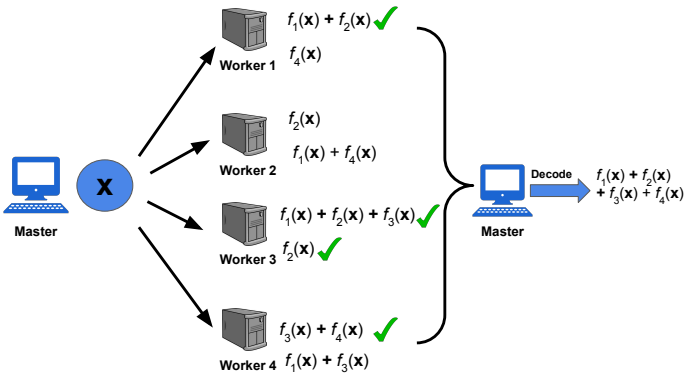


Fig. 1: An illustration of our rateless coded (LT-1) strategy for a system of 4 workers tasked with computing $F(\mathbf{x}) = \sum_{i-1}^{4} f_i(\mathbf{x})$. Tick marks indicate completed tasks which suffice to recover $F(\mathbf{x})$. Observe that Worker 2 (straggler) does not complete any tasks.

only considers polynomial functions, [16] only approximately recovers the original computation without any convergence guarantees, and [17] and follow-up works need knowledge of the number of stragglers for code design which is unrealistic in practice. On the other hand, we use rateless codes on *general* non-linear functions that can be expanded as a sum of cheaper non-linear functions. Our rateless coded approach can scale to a large number of nodes, without any prior knowledge of the number of stragglers, and provide efficient encoding and decoding, and guaranteed recovery of the original computation.

II. Preliminaries

A. Problem Setup

We consider the computation of an expensive *non-linear* function that can be expressed as the sum of m cheap *non-linear* functions as

$$F(\mathbf{x}) = \sum_{i=1}^{m} f_i(\mathbf{x}) \tag{1}$$

In machine learning, such computations include ensemble models like Random Forests [18] where f_i is the i^{th} decision tree in the ensemble, and kernel methods [10] where f_i is the kernel functions between the test point and the i^{th} training point. Outside machine learning such computations can arise in fields like statistics [19] and non-linear optics [20] where expensive non-linear functions are approximated by their Taylor series wherein f_i is the i^{th} Taylor series term. We assume that f_i's are inexpensive to compute as compared to $F(\mathbf{x})$. Since m is large, we wish to parallelize the computation to speed it up.

B. Baselines

We consider the following three baseline strategies for computing (1), which we will compare with our approach:

1) **All-at-One.** In this approach, the entire computation is performed at a single node. The node receives $\mathbf{x}$ as input, computes $f_i(\mathbf{x})$, $i = 1, \ldots, m$ and adds all the results to compute $F(\mathbf{x})$.
2) **Uncoded.** In this approach, the computation is parallelized across m nodes (workers). Worker i is tasked with computing $f_i(\mathbf{x})$. The process is coordinated by a central node (master) which communicates $\mathbf{x}$ to the workers, collects the results of worker computations, and computes the sum in (1). In this case the master needs to wait for *all* workers to complete their tasks.
3) **r-Replication.** In this approach, the computation is again parallelized across m workers but this time tasks are *replicated* across workers. The group of workers $r(j-1)+1, \ldots, rj$ each compute the *partial sum* $\sum_{i=r(j-1)+1}^{rj} f_i(\mathbf{x})$ for $j = 1, \ldots, m/r$. For eg. if $r = 2$, workers 1 and 2 compute $f_1(\mathbf{x}) + f_2(\mathbf{x})$, workers 3 and 4 compute $f_3(\mathbf{x}) + f_4(\mathbf{x})$ etc.. For each partial sum, the master only needs to wait for the *fastest* of the workers that it was assigned to. $F(\mathbf{x})$ can be computed once the master has collected all distinct partial sums.

If m is large then one would expect Uncoded to outperform All-at-One. However as we will see in our experiments in Section V, the requirement of waiting for all nodes in Uncoded can lead to *significant* slowdown in the presence of stragglers, to the extent that it is even slower than the All-at-One approach which only uses a single node. The $r-$Replication approach can mitigate the effect of stragglers since it only needs to wait for the fastest of the r workers for each partial sum. Systems like Mapreduce [3] use $r = 2$ i.e. each computation is assigned to 2 different worker nodes for added reliability. However as we will see in Section V, this approach is still bottlenecked by the slowest of the m/r groups computing distinct partial sums. Hence, in Section III, we propose a rateless coded strategy for computing (1) which can be designed such that we only need to wait for the fastest m/r workers *overall* to compute $F(\mathbf{x})$.

While we assume the number of workers to be equal to m, the number of terms in the sum (1), this can easily be generalized to consider $p < m$ worker nodes by considering each group of m/p terms in the sum to be a single function. We study the case with m workers because straggling depends on the tail of delay distribution at workers and hence is most severe when the number of workers is large [6]. Moreover emerging distributed computing frameworks like AWS Lambda [12], [14] *do* consider the setting where each worker computes a single function. Prior work [13] has shown the susceptibility of such frameworks to straggling while computing linear functions, and in Section V we show the presence of straggling, and the speedup with our coded scheme, for non-linear functions.

C. Evaluation Criterion

We will use latency (defined below) as a metric to compare the different schemes.

Definition 1 (Latency (T))**.** *The latency T is the time taken to complete the set of computations needed to recover $F(\mathbf{x})$.*

Observe that the set of computations needed to recover $F(\mathbf{x})$ varies across schemes, for example, all computations in the Uncoded strategy versus at least one instance of each partial sum in the $r-$Replication strategy.

III. Proposed Coding Strategy for Distributed Non-Linear Computations

We first introduce a general coded framework for speeding up distributed non-linear computations of the form (1), and then describe two rateless coded versions of the framework.

A. Coded Non-Linear Computing

Consider a $m_e \times m$ matrix $\mathbf{G_e}$ ($m_e > m$) such that any $m \times m$ submatrix of $\mathbf{G_e}$ is invertible, for eg. if the elements of $\mathbf{G_e}$ are Gaussian random variables. Now consider the computation $\mathbf{f_e} = \mathbf{G_e f}$ where $\mathbf{f}$ is a vector whose i^{th} element is $f_i(\mathbf{x})$. If each of the m workers computes m_e/m rows of the product $\mathbf{G_e f}$, then we only need to wait for m rows to be completed *in total*, across all workers. If $\mathbf{G}$ is the $m \times m$ submatrix corresponding to the completed rows, and $\mathbf{f}'$ is the vector of completed computations then we can recover $\mathbf{f}$ by solving the system of equations $\mathbf{f}' = \mathbf{G f}$. This is akin to the coded computing approach of [7] for linear computations.

However observe that if $\mathbf{G_e}$ is a dense matrix, as is the case with the MDS codes of [7], then each worker have to compute $f_i(\mathbf{x})$ for a large subset of the indices $i = 1, \ldots, m$ and then multiply the vector $\mathbf{f} = [f_1(\mathbf{x}) \ldots f_m(\mathbf{x})]^T$ with the corresponding row of $\mathbf{G_e}$. This would significantly increase the worker load as compared to the Uncoded or $r-$Replication approaches and thus may not offer much speedup. Also if $\mathbf{G_e}$ is dense then the submatrix $\mathbf{G}$ will also be dense and solving $\mathbf{f}' = \mathbf{G f}$ will be prohibitive ($\mathcal{O}(m^3)$ complexity). Hence we require sparse $\mathbf{G_e}$, and efficient decoding of $\mathbf{f}$ from worker computations, both of which are provided by *rateless codes*.

B. The first solution: LT-1

We now introduce our rateless coding strategy, illustrated in Fig. 1, which uses LT codes [11] for straggler mitigation in distributed computing of non-linear functions of the form (1).

Luby Transform (LT) codes proposed in [11] are a class of erasure codes that can generate a limitless number of encoded symbols from a finite set of source symbols. We apply LT codes to (1) by treating each function $f_i(\mathbf{x})$ as a source symbol. Each encoded symbol is the sum of d source symbols chosen uniformly at random from the set of functions. Thus if $\mathcal{S}_d \subseteq \{1, 2, \ldots m\}$ is the set of d row indices, the corresponding encoded function is

$$f^{(e)}(\mathbf{x}) = \sum_{i \in \mathcal{S}_d} f_i(\mathbf{x}) \tag{2}$$

The number of source symbols in each encoded symbol, or the degree d, is sampled from the Robust Soliton degree distribution [11], which ensures that encoded symbols are sparse linear combinations of source symbols ($\mathcal{O}(\log m)$ average degree).

A total of $m_e (m_e > m)$ encoded symbols are assigned to the m workers such that each worker is assigned m_e/m symbols of the form (2) (we assume m_e/m is an integer). To compute $F(\mathbf{x})$, the master sends $\mathbf{x}$ to the workers which compute encoded symbols. Since the degree d of each encoded symbol is a random variable, the amount of computation assigned to each worker may be different.

The master collects the encoded symbols until it has enough to be able to recover $F(\mathbf{x})$. One way to recover $F(\mathbf{x}) = \sum_{i=1}^{m} f_i(\mathbf{x})$ is to recover each $f_i(\mathbf{x})$, $i = 1, \ldots, m$ and then compute their sum. For this we use the the iterative peeling decoder [11]. Since encoded symbols are random sums of source symbols, the master may receive symbols such as $f_1(\mathbf{x}) + f_2(\mathbf{x}) + f_3(\mathbf{x})$, $f_2(\mathbf{x}) + f_4(\mathbf{x})$, $f_3(\mathbf{x})$, $f_4(\mathbf{x})$, and so on. Decoding is performed in an iterative fashion. In each iteration, the decoder finds a degree one encoded symbol, covers the corresponding source symbol, and subtracts it from all other encoded symbols connected to that source symbol. Decoding is possible once the master receives M' encoded symbols.

Since the encoding uses a random bipartite graph, the number of symbols, M', required to decode the m source symbols successfully is a random variable, which for the Robust Soliton degree distribution is $m + \mathcal{O}(\sqrt{m} \ln^2(m/\delta))$ with probability at least $1 - \delta$ [11]. Moreover, the complexity of the decoding process described above is $\mathcal{O}(m \log m)$ for LT codes (due to the careful design of the Robust Soliton distribution). Thus, while workers need to complete slightly more than m computations the decoding cost is *significantly* less than MDS codes and this combination of sparse degrees and fast decoding is the key reason for choosing LT codes in our setting.

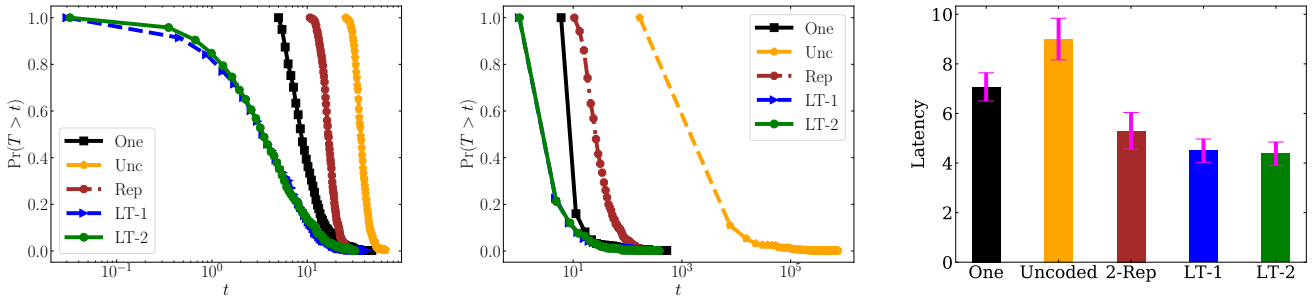
C. A general solution: LT-r

Observe that unlike prior coded computing works [7], [9] we *do not* need to recover all the source symbols $f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})$. Instead we just need to recover their sum $F(\mathbf{x})$. A natural question to ask in this case is whether we can reduce the cost (encoding or decoding or both) given that recovering all source symbols is a sufficient condition but not a necessary condition for recovering the sum. To answer this, we propose the following simple generalization of the LT-1 approach.

Consider splitting the expression for F in (1) into groups analogous to r-replication as (for $r = 2$)

$$F(\mathbf{x}) = (f_1(\mathbf{x}) + f_2(\mathbf{x})) + \ldots + (f_{m-1}(\mathbf{x}) + f_m(\mathbf{x})) \tag{3}$$

Each term within parantheses corresponds to a group of f_i's that in the case of $r-$replication are replicated at r nodes. In the LT-r case, instead of replicating these functions we *encode* them using an LT code over the groups. This will increase the degree of the symbols by a factor of r because, if $r = 2$, the encoded symbols obtained by adding the first two source symbols will now be $f_1(\mathbf{x}) + f_2(\mathbf{x}) + f_3(\mathbf{x}) + f_4(\mathbf{x})$ and so on. While a higher degree will lead to higher computation cost at the workers, the master will have to wait for much fewer coded symbols for successful decoding. As per the results in [11], the master will now need $M' = m/r + \mathcal{O}(\sqrt{m/r} \ln^2(m/r\delta))$ for successful decoding with probability at least $1 - \delta$ [11]. This corresponds to (approximately) the fastest m/r computations across *all* workers unlike $r-$Replication where we need to wait for at least one worker from each of the m/r groups computing distinct partial sums. Moreover the decoding cost ($\mathcal{O}((m/r)\log(m/r))$) will be lower than LT-1. In experiments in Section V we will see that LT-2 ($r = 2$) is indeed faster than 2-replication and also (slightly) faster than LT-1.



(a) $X_i \sim \text{Exp}(0.2)$ (b) $X_i \sim \text{Pareto}(1, 1)$ (c) AWS Lambda

Fig. 2: The tail probability of the latency is the highest for the Uncoded scheme for both Exp(0.2) and Pareto(1,1) distributions on initial delay X_i. Observe that One (All-at-One) where the entire computation is performed by a single node is faster than Uncoded in all cases, and also has a lighter tail than 2-Replication in the simulations ((a) and (b)). Both LT coded schemes, LT-1 and LT-2, have the lightest latency tail probability in the simulations and are the fastest in the AWS Lambda experiments. Both simulations and experiments use $m = 1000$.

IV. Theoretical Latency Analysis

Our main theoretical results involve comparing the latency of the proposed LT-coded strategies with the baseline strategies in Section II-B. We assume that worker j performs B_j computations in time Y_j where

$$Y_j = X_j + \tau B_j, \quad \text{for all } j = 1, \ldots, m \tag{4}$$

where X_j is a random variable that includes the network latency, initial setup time, and other random components, and τ is a constant which represents the time taken by any worker to perform a single computation of the form $f_i(\mathbf{x})$. Observe that in the All-at-One, $r-$replication and LT-coded cases the worker performs multiple such computations. When $X_j \sim \exp(\mu)$, the time taken by worker j to perform b computations is distributed as $\Pr(Y_j \leq t) = 1 - \exp(-\mu(t - \tau b))$. While following the shifted exponential delay model of [7], we believe this formulation is more realistic since shift is parameterized by the number of computations at workers and captures the fact that delay is higher at workers performing more computations.

Theorem 1 (Latency of the LT-r strategy)**.** *For $m_e \to \infty$, and assuming that $\mathbb{E}[M'] \approx m/r$ is the expected number of symbols needed for successful decoding, the expected latency for the LT-coded case with m workers and $X_i \sim \exp(\mu)$ for all workers $i = 1, \ldots, m$, is bounded as.*

$$\mathbb{E}[T_{LT}] \leq \tau(r+1)\bar{d}_{m/r} + \frac{1}{\mu}, \bar{d}_{m/r} \sim \mathcal{O}(\log(m/r)) \tag{5}$$

where $\bar{d}_{m/r}$ is the average degree of a LT code over m/r source symbols.

Setting $r = 1$ in (5) corresponds to the LT-1 scheme.

Remark 1. The need for a sparse code is illustrated by the average degree or $\bar{d}_{m/r}$ term in (5). For a dense code like an MDS code, $\bar{d}_{m/r}$ would be $\mathcal{O}(m/r)$ which would significantly increase the latency (while also increasing decoding complexity). This is owing to the non-linearity of the problem due to which the encoded functions cannot be pre-computed as in [7].

Proposition 1 (Latency of r-Replication)**.** *The expected latency for the $r-$Replication strategy with $X_i \sim \exp(\mu)$ for all workers $i = 1, \ldots, m$ is*

$$\mathbb{E}[T_{rep}] = \tau r + \frac{1}{r\mu} H_{m/r} \simeq \tau r + \frac{1}{r\mu} \log \frac{m}{r} \tag{6}$$

where $H_j = \sum_{v=1}^{j} 1/v$, the j^{th} Harmonic number.

Setting $r = 1$ in (6) corresponds to the Uncoded scheme.

Remark 2. Comparing (5) and (6) we can see that there is a $\log(m/r)$ in both terms which can potentially be large (for large m). However, in case of the replication scheme the term is multiplied by a term proportional to $1/\mu$, the average initial delay at the workers, while for the coded scheme the term is multiplied by a term proportional to τ, the time taken by any worker to perform a single computation of the form $f_i(\mathbf{x})$. We typically expect the average initial delay to be much larger than the time taken to perform a single computation, especially in systems with straggling nodes (the factor of $1/r$ in (6) typically will not affect this since r is usually chosen to be 2 or 3 [3]). Hence we expect the latency of the LT-r strategy to be much less than that of the $r-$Replication strategy.

Proposition 2 (Latency of All-at-One)**.** *The expected latency for the All-at-One strategy with a single worker whose initial delay is distributed as $X_1 \sim \exp(\mu)$ is*

$$\mathbb{E}[T_{One}] = \tau m + \frac{1}{\mu} \tag{7}$$

Remark 3. In situations where $1/\mu$ is large (X_i has a heavier tail) it is possible for the latency of All-at-One to be lower than that of Uncoded since the latter is adversely affected by having to wait for the slowest worker. We show this below, both through simulations and experiments on AWS Lambda, a real distributed computing framework, thus clearly illustrating the need for the coded schemes.

V. Evaluation

We simulate the baseline and LT-coded schemes under our delay model (4) for distributed non-linear computation with $m = 1000$ and $\tau = 0.005$. While we only included theoretical results for exponentially distributed X_j, in simulations we use both Exp(0.2) (Fig. 2a) and Pareto(1, 1) (Fig. 2b) distributions for X_j to demonstrate the generality of our scheme. The tail probabilities are calculated using 500 Monte-Carlo simulations. The results clearly illustrate the superiority of the LT-1 and LT-2 ($r = 2$) approaches over the baselines. Note that All-at-One (One) outperforms Uncoded since both our choices of X_j have a relatively heavy tail due to which the effect of the stragglers is especially pronounced in the uncoded case.

We also evaluated our approach on AWS Lambda [14], Amazon's Serverless Computing framework. Serverless Computing is especially appropriate for coded computing since it targets users without cloud computing expertise who are seeking a cheap and easy way to paraellelize and speed up their computations [12] and typically have no control over how tasks are allocated. There can be huge variability in the underlying infrastructure leading to significant straggling [13]. We observed that the straggling follows the trend indicated by (4) (random set-up time X_j, deterministic compute-time τ). We implement a distributed computation of the form (1) where f_i's are Gaussian kernels between $m = 1000$ randomly chosen 1000 dimensional vectors and an input vector $\mathbf{x}$. The computation is implemented over $m = 1000$ Lambda workers. Since it is unclear how to stop workers between computations, we let each worker perform 2 computations in the LT-1 and LT-2 schemes and then select the results from the fastest $M'/2$ workers overall where $M' = 1400$ and $M' = 783$ computations respectively (corresponding to 99% probability of successful decoding in each case). Results in Fig. 2c for Latency averaged over 20 trials show that LT-1 and LT-2 clearly outperforms baselines in this real-world setting. Additionally, average decoding time for LT-1 is around 55 ms while that of LT-2 is 29 ms thus confirming that LT-2 can be decoded faster.

VI. Conclusion

The huge data and computation requirement of modern machine learning has increased the reliance of machine learning algorithms on parallel and distributed computing. It has been amply demonstrated that naive distributed implementations are severely bottlenecked by straggling nodes. Coded computing offers a principled solution to this problem but has so far been limited to linear computations. In this work we take a step towards extending the benefits of coded computing to *non-linear* computations which constitute the bulk of machine learning computations. We demonstrate that rateless codes, specifically LT codes, due to their sparsity and fast decoding, can speed up distributed computations of expensive non-linear functions, that can be written as sums of cheap non-linear functions, in the presence of stragglers. In future work we plan to explore novel code designs, extend the framework to include other rateless codes [21], and apply our ideas to actual serverless machine learning applications [12].

VII. Acknowledgments

This project was supported in part by NSF CAREER Award CCF2045694, NSF CRII Award CCF1850029 and the Qualcomm Innovation fellowship.

References

[1] "Full Version," http://www.andrew.cmu.edu/user/gaurij/ISTC2021.pdf.
[2] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al., "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.
[3] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[4] Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2018, pp. 803–812.
[5] Yuchen Zhang, John Duchi, and Martin Wainwright, "Divide and conquer kernel ridge regression: A distributed algorithm with minimax optimal rates," *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 3299–3340, 2015.
[6] Jeffrey Dean and Luiz André Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[7] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, 2017.
[8] Qian Yu, Mohammad Maddah-Ali, and Salman Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," in *Advances in Neural Information Processing Systems*, 2017, pp. 4406–4416.
[9] Ankur Mallick, Malhar Chaudhari, Utsav Sheth, Ganesh Palanikumar, and Gauri Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–40, 2019.
[10] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola, "Kernel methods in machine learning," *The annals of statistics*, pp. 1171–1220, 2008.
[11] Michael Luby, "LT codes," in *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002, pp. 271–271.
[12] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al., "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.
[13] Vipul Gupta, Shusen Wang, Thomas Courtade, and Kannan Ramchandran, "Oversketch: Approximate matrix multiplication for the cloud," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 298–304.
[14] Amazon, "AWS Lambda," https://aws.amazon.com/lambda/, 2014.
[15] Qian Yu, Songze Li, Netanel Raviv, Seyed Mohammadreza Mousavi Kalan, Mahdi Soltanolkotabi, and Salman A Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1215–1225.
[16] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman, "Parity models: A general framework for coding-based resilience in ML inference," *CoRR*, vol. abs/1905.00863, 2019.
[17] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis, "Gradient coding: Avoiding stragglers in synchronous gradient descent," *stat*, vol. 1050, pp. 8, 2017.
[18] Leo Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[19] Edward F Vonesh, Hao Wang, and Dibyen Majumdar, "Generalized least squares, taylor series linearization and fisher's scoring in multivariate nonlinear regression," *Journal of the American Statistical Association*, vol. 96, no. 453, pp. 282–291, 2001.
[20] Alan Newell, *Nonlinear optics*, CRC Press, 2018.
[21] Amin Shokrollahi, "Raptor codes," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. SI, pp. 2551–2567, 2006.