Rateless Codes for Distributed Computations with Sparse Compressed Matrices

Ankur Mallick, Gauri Joshi
Carnegie Mellon University
Email: {amallic1,gaurij}@andrew.cmu.edu

***Abstract*—Unpredictable slowdown of worker nodes, or node straggling, is a major bottleneck when performing large matrix computations such as matrix-vector multiplication in a distributed fashion. For sparse matrices, the problem is compounded by irregularities in the distribution of non-zero elements, which leads to an imbalance in the computation load at different nodes. To mitigate the effect of stragglers we use *rateless* codes that generate redundant linear combinations of the matrix rows (or columns) and distribute them across workers. This coding scheme utilizes all partial work done by worker nodes, and eliminates tail latency. We also propose a balanced row-allocation strategy for allocating rows of a sparse matrix to workers that ensures that equal amount of non-zero matrix entries are assigned to each worker. The entire scheme is designed to work with compressed, memory-efficient formats like CSR/CSC that are used to store sparse matrices in practice. Theoretical analysis and simulations show that our balanced rateless coding strategy achieves significantly lower overall latency than conventional sparse matrix-vector multiplication strategies.**

***Index Terms*—rateless codes, coded computing, sparse matrix computations, straggler mitigation**

I. Introduction

Matrix-vector multiplication is ubiquitous in scientific computing and Big Data analysis, as numerous applications in signal processing (For eg. FFT), machine learning (For eg. PCA), and graph analytics (For eg. Page Rank) involve linear computations that can be expressed as the multiplication of a large matrix by a vector. Such large computations are usually distributed across multiple computing nodes to store and process the data in an efficient and scalable fashion. However, distributing the computation over a large number of nodes makes it susceptible to unpredictable node slowdown/failure [1]. Specifically, a few slow nodes, called *stragglers* delay the computation even after the remaining nodes have completed their assigned tasks.

In distributed computing systems like MapReduce [2] and Spark [3], the problem of stragglers is countered by back-up tasks. A more recent line of work [4], [5], called coded computing, applies erasure coding, specifically maximum distance separable (MDS) codes, to mitigate the effect of stragglers – a (p, k) MDS code allows recovery of the matrix-vector product from the computations of the fastest k out of p workers. A major drawback of MDS codes is that they discard the partial computations performed by the $(p-k)$ slow workers entirely. In [6] we address this issue by proposing a rateless coded strategy (based on LT codes [7]) that can efficiently utilize the computations performed by *all* workers including the stragglers, and seamlessly adapt to varying node speeds.

Another drawback of coded computation strategies is that they are agnostic to the entries of the matrix in question. In practice, large matrices are often extremely sparse and have a irregular distribution of non-zero entries. Conventional methods that equally split rows/columns across workers can cause an imbalance in the computation load at different workers. Moreover, coding adds redundant and dense rows/columns to the matrix, further increasing the computation load. Some recent works [8], [9] try to limit the density of the coded rows, but do not utilize the partial work performed by stragglers.

Moreover, sparse matrices are generally stored in a compressed format such as compressed sparse row (CSR) and compressed sparse column (CSC). Performing computations directly in these compressed formats can save the cost of preprocessing of the matrix for the purpose of encoding/decoding. Sparse distributed computation solutions that split the non-zero entries equally, and can operate in these compressed formats have been proposed in the high-performance computing community [10], [11]. But these are uncoded strategies and thus are not naturally robust to straggling worker nodes.

To the best of our knowledge, this work is the first to combine a rateless coding scheme with a balanced task allocation strategy in order to perform straggler-resilient and efficient sparse matrix computations. Our balanced rateless coding strategy utilizes all partial work done by stragglers and also provide robustness to uneven distribution of non-zero elements in sparse matrices. Moreover, our coding strategy allows computations directly in the CSR and CSC matrix formats as we describe in Section III and Section V below. Latency analysis in Section IV shows a large improvement over uncoded and MDS coding strategies. Proofs of the theoretical results are presented in Appendix D of the full version [12].

II. Preliminaries

A. System Model

We consider the problem of multiplying a sparse $m \times n$ matrix $\mathbf{A}$ with a $n \times 1$ vector $\mathbf{x}$ using p worker nodes and a master node. The worker nodes can only communicate with the master, and cannot directly communicate with other workers. The goal is to compute the result $\mathbf{b} = \mathbf{A}\mathbf{x}$ in a distributed fashion and mitigate the effect of unpredictable node slowdown or straggling. While we consider the setting where a single vector $\mathbf{x}$ is being multiplied with $\mathbf{A}$, the proposed straggler mitigation scheme is also applicable to the more general setting where the matrix $\mathbf{A}$ is fixed (representing the system) while a stream of vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots$, (representing inputs/tasks) arrive and need to be multiplied with $\mathbf{A}$.

B. Compressed Sparse Matrices

The Compressed Sparse Row (CSR) format of storing sparse matrices is a memory-efficient row-oriented approach for storing sparse matrices. Instead of storing the matrix $\mathbf{A}$ as a 2-dimensional array, it just stores an array $\mathbf{A}$.data containing all non-zero elements, the corresponding column indices ($\mathbf{A}$.indices), and the offsets ($\mathbf{A}$.offsets) corresponding to the starting position of each row of $\mathbf{A}$ in $\mathbf{A}$.data. For example, consider the following sparse matrix.

$$\mathbf{A} = \begin{bmatrix} 1 & 5 & 7 & -1 \\ 2 & 4 & 0 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}.$$

In the CSR format it is represented by the following 3 arrays.

$$\begin{aligned} \mathbf{A}.\text{data} &= [1, 5, 7, -1, 2, 4, 6], \\ \mathbf{A}.\text{indices} &= [0, 1, 2, 3, 0, 1, 0], \\ \mathbf{A}.\text{offsets} &= [0, 4, 6, 7]. \end{aligned}$$

This can potentially reduce the storage cost from $\mathcal{O}(mn)$ to $\mathcal{O}(m)$ for matrices which are highly sparse. A similar format, Compressed Sparse Column (CSC) is used when the data is to be stored in column major format by storing the row indices and the starting position of each column along with the non-zero elements of the matrix.

In addition to efficient storage, the CSR/CSC formats enable efficient computation of the matrix-vector product $\mathbf{Ax}$ since only the non-zero elements of each row/column of $\mathbf{A}$ are multiplied with the corresponding elements of $\mathbf{x}$ instead of multiplying the entire row/column. This is a huge saving for highly sparse matrices and efficient routines exist for computing the matrix-vector product in this fashion [10], [11], [13]. For the rest of this paper we assume that $\mathbf{A}$ is stored in the CSR/CSC format and propose rateless coding schemes that can perform encoding, computations and decoding directly in these formats.

C. LT Codes and Systematic LT Codes

Rateless codes, also referred to as fountain codes, can be used to generate a limitless number of encoded symbols from a finite set of m input symbols. In this work, we employ Luby Transform (LT) codes (proposed in [7]) and their systematic version (proposed in [14]), which ensure decoding from $m(1+\epsilon)$ for small ϵ, and a low decoding complexity $O(\log m)$ per symbol. In classic LT codes, each encoded symbol is the sum of d input symbols chosen uniformly at random from the set of input symbols. The number of input symbols in each encoded symbol, or the degree d, is chosen according to the Robust Soliton degree distribution, the details of which are described in [7]. Decoding is performed by applying the belief propagation or *peeling* decoder, which involves iteratively mapping a degree 1 encoded symbol to the corresponding input symbol, and then eliminating it from any other encoded symbols [7], [14].

Systematic LT codes [14], are constructed first applying the *LT decoding algorithm* to the m input symbols to generate a set of m *pre-input symbols*. The parity symbols are generated by applying the LT encoding process to these pre-input symbols. The overall set of input symbols and parity symbols follow the same properties as the encoded symbols of a general LT code (since they can be generated by applying LT encoding to the same set of pre-input symbols) and together form the systematic LT code. The original m pre-input symbols can be recovered from any set of $m(1+\epsilon)$ systematic and parity symbols, where the overhead $\epsilon \to 0$ as $m \to \infty$. If the $m(1+\epsilon)$ received symbols include all the systematic symbols then decoding is unnecessary. Otherwise, LT encoding is applied to the decoded pre-input symbols to obtain the missing systematic symbols.

III. Rateless Coding for the CSR Format

In this section, we use rateless codes, specifically LT and systematic LT codes, to mitigate the effect of stragglers in computing the matrix-vector product $\mathbf{b} = \mathbf{Ax}$, where $\mathbf{A}$ is a $m \times n$ sparse matrix stored in the CSR format. Systematic LT codes preserve the sparsity of original matrix rows, but result in denser encoded (parity) rows than classic LT codes. Both provide seamless adaptation to fluctuations in node speeds (straggling), and allow computations to be performed directly in the CSR format. We also propose a *balanced row allocation scheme* that balances the number of non-zero elements assigned to each worker. In what follows we describe the encoding, task allocation, and decoding for systematic LT codes and illustrate it in Figure 1. The uncoded and LT coded approaches can be viewed as two extremes of using systematic LT codes. In the uncoded case we only have the input (systematic) rows, while in the LT coded case we only have the encoded (parity) rows.

A. Encoding and Balanced Row Allocation

The $m \times n$ matrix $\mathbf{A}$ is encoded using systematic LT codes (Section II-C), by treating the m rows as source symbols, to generate an $m_e \times n$ encoded matrix $\mathbf{A}^{(\mathbf{e})}$ where $m_e = \alpha m$ ($\alpha \geq 1$). The first m out of m_e rows of the encoded matrix $\mathbf{A}^{(\mathbf{e})}$ are the same as the corresponding rows of $\mathbf{A}$ while the remaining rows correspond to parity symbols of the systematic code i.e. $\mathbf{A}^{(\mathbf{e})} = [\mathbf{A}^T, \mathbf{A_c}^T]^T$ where $\mathbf{A_c}$ has size $(\alpha - 1)m \times n$. Parity symbols are expected to be slightly denser than the systematic symbols (density quantified concretely in Section IV).

Since the straggling at the workers is unknown, we would like to allocate equal amount of computations to all the workers (to balance the load) and let the LT encoding of the rows handle the straggling. To ensure that each worker performs equal amount of computations in the absence of straggling we divide non-zero elements of both $\mathbf{A}$ and $\mathbf{A_c}$ (almost) equally among workers according to the following policy. We describe the row allocation policy for $\mathbf{A}$ below and note that the same policy is used to divide rows of $\mathbf{A_c}$ among workers.

Definition 1 (Balanced Row Allocation)**.** *If row j of $\mathbf{A}$ has S_j non-zero elements and $\bar{S} = (1/p)\sum_j S_j$, then $\mathbf{A}$ is split*

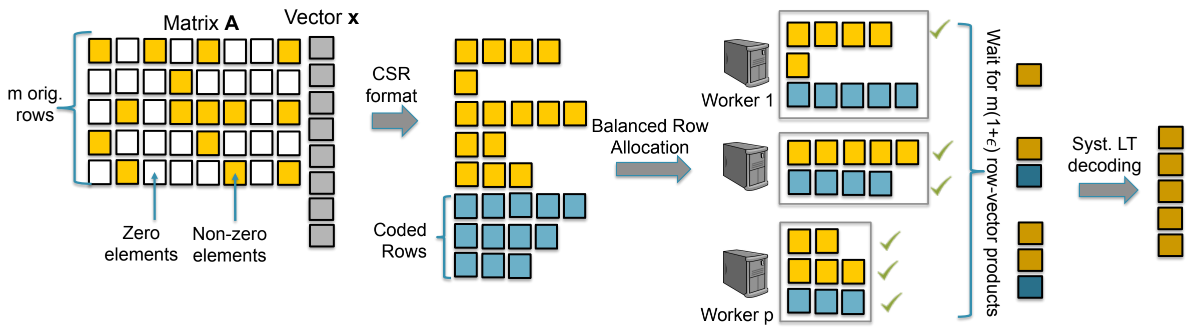


Fig. 1: Rows of $\mathbf{A}$ in the CSR format are encoded by adding systematic LT coded rows. The systematic rows (yellow) are allocated to workers by balancing the number on non-zero elements at each worker, while the parity rows (blue) are split equally across workers.

along its rows and worker i is assigned rows J_i^0 to J_i^f where $J_1^0 = 1,\ J_i^0 = J_{i-1}^f + 1,\ i = 2, \dots, p$ and

$$J_i^f = \min\left(\min\left\{l : \sum_{j=J_i^0}^{l} S_j \geq \bar{S}\right\}, m - p + i\right)$$

$$\text{for } i \in [1, p-1],$$

$$J_p^f = m.$$

Assigning exactly $\bar{S}$ non-zero elements to each worker would ensure perfect load balancing but may split individual matrix rows across multiple workers causing problems with storing and managing data in the efficient CSR format. Our row allocation policy ensures that each worker is assigned at least 1 matrix row (the outer 'min') and the total number of non-zero elements at a worker is as close as possible to $\bar{S}$ without splitting rows across multiple workers (the inner 'min'). The following lemma quantifies the worker loads under our policy.

Lemma 1. *If row j of $\mathbf{A}$ has S_j non-zero elements, then under our row allocation policy the number of non-zero elements at worker i, N_i is $N_i \in [\bar{S}, \bar{S} + S_{\max}]$ where $S_{\max} = \max_j S_j$.*

B. Computation and Decoding

To multiply $\mathbf{A}$ with a vector $\mathbf{x}$, the master sends $\mathbf{x}$ to the workers. Each worker multiplies $\mathbf{x}$ with each row of $\mathbf{A}^{(\mathbf{e})}$ stored in its memory and returns the product (a scalar) to the master. The master collects row-vector products of the form $\mathbf{a}_{\mathbf{e},j}\mathbf{x}$ (elements of $\mathbf{b}_\mathbf{e} = \mathbf{A}^{(\mathbf{e})}\mathbf{x}$) from workers until it has enough elements to be able to recover $\mathbf{b}$. Each worker first computes the dot product of the systematic rows assigned to it with $\mathbf{x}$, and then that of the parity rows. If a worker node completes all the row-vector products assigned to it before the master is able to decode $\mathbf{b} = \mathbf{A}\mathbf{x}$, it will remain idle, while the master collects more row-vector products from other workers. Alternately, to minimize communication, the worker may only send progress updates to the master node indicating the number of row-product computation tasks it has completed, and send the products only upon request by the master.

The master can recover $\mathbf{b}$ from the encoded symbols in $\mathbf{b}_\mathbf{e}$ using the iterative peeling decoder [7], [14]. In each iteration, the decoder finds a degree one received symbol, covers the corresponding source symbol, and subtracts the symbol from all other encoded symbols connected to that source symbols. For Systematic LT codes, the received symbols (elements of $\mathbf{b}_\mathbf{e}$) are dot products of the rows of $\mathbf{A}$ or $\mathbf{A}_\mathbf{c}$ with $\mathbf{x}$. Since the rows of $\mathbf{A}$ and $\mathbf{A}_\mathbf{c}$ are obtained by applying LT encoding to the pre-input symbols $\tilde{\mathbf{a}}_1, \dots, \tilde{\mathbf{a}}_m$, as described in Section II-C, the decoder recovers dot-products of the form $\tilde{\mathbf{a}}_1\mathbf{x}$. LT encoding can be applied to these symbols to recover the elements of $\mathbf{b}$.

Since encoding uses a random bipartite graph, the number of symbols required to decode the m source symbols is a random variable M'. Analysis in [7] for the Robust Soliton distribution shows that $m_d = \mathbb{E}[M'] = m(1+\epsilon)$, $\epsilon \to 0$ as $m \to \infty$.

All operations in the encoding, task allocation, computations, and decoding steps are performed on the rows of $\mathbf{A}$ or $\mathbf{A}^{(\mathbf{e})}$. Since $\mathbf{A}$ and $\mathbf{A}^{(\mathbf{e})}$ are stored in the CSR format which allows fast row access as described in [10], [11], therefore our strategy can be efficiently implemented in existing sparse matrix vector multiplication frameworks that use this format.

We can also obtain $\mathbf{A}^{(\mathbf{e})}$ from $\mathbf{A}$ using a $(\alpha m, m)$ MDS code on the rows of $\mathbf{A}$. This allows us to use the partial work of the stragglers since $\mathbf{b}$ is recoverable using MDS decoding on any m elements of $\mathbf{b}_\mathbf{e}$. However MDS codes have a prohibitively large encoding and decoding complexity ($\mathcal{O}(m^2)$, and $\mathcal{O}(m^3)$). LT codes have encoding and decoding complexity of $\mathcal{O}(m \ln m)$ which scales much more efficiently for large m due to which they are practically applicable in this setting.

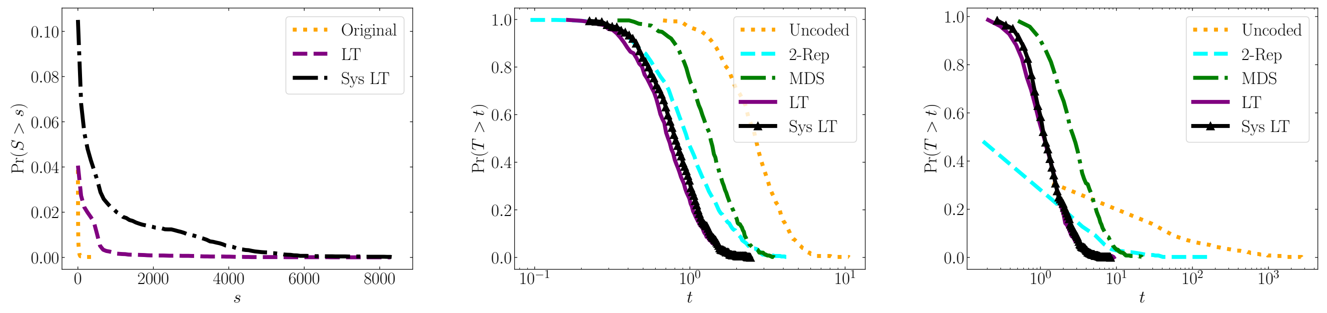
IV. Analysis of Latency

We compare latency of the uncoded and rateless coded strategies for distributed sparse matrix-vector multiplication, under the balanced row allocation scheme (Definition 1).

A. Delay Model

Definition 2 (Latency (T))**.** *The latency T is the time required by the system to complete enough number of computations so that $\mathbf{b} = \mathbf{A}\mathbf{x}$ can be successfully decoded from the worker computations aggregated in $\mathbf{b}_\mathbf{e}$.*

The dot product of the j^{th} matrix row (containing S_j non-zero elements) with vector $\mathbf{x}$ involves S_j scalar multiplications and additions. We refer to each such operation of multiplying two numbers and adding the product to the portion of the



(a) Non-zeros after encoding (b) Latency ($X \sim$ Exponential with rate $\mu = 2.0$) (c) Latency ($X \sim$ Pareto(1, 1))

Fig. 2: LT and Systematic LT encoding slightly increases the density of rows (Figure 2a). However, the latency tail is heaviest for the uncoded scheme and lightest for LT and Systematic LT schemes (Figure 2b and Figure 2c). Results are obtained from 500 Monte-Carlo Simulations with a 10000×10000 matrix $\mathbf{A}$, $p = 10$ worker nodes, $\tau = 5 \times 10^{-7}$, a $(10, 8)$ MDS Code, and $\alpha = 2.0$ for LT and Systematic LT codes.

row-vector product up to that point as a single *computation*. The time taken by worker i to perform N_i computations is the random variable Y_i, which is given by

$$Y_i = X_i + \tau N_i, \quad \text{for all } i = 1, \ldots, p, \tag{1}$$

where X_i is a random variable that includes the network latency, initial setup time, and other random variations in computation time, and τ is a constant shift which is the time taken by any worker to perform a single computation. For eg., when $X_i \sim \exp(\mu)$, the time in which worker i performs b computations is distributed as $\Pr(Y_i \leq t) = 1 - \exp(-\mu(t - \tau b))$.

B. Latency Bounds for Uncoded and Rateless Coded Schemes

We present bounds on the expected latency (proofs are given in the full version) for uncoded and LT coded distributed sparse matrix vector multiplication. In both cases, the matrix to be multiplied ($\mathbf{A}$ for uncoded, and $\mathbf{A}^{(\mathbf{e})}$ for LT coded) is distributed among p worker nodes using our balanced row allocation scheme (Definition 1). We expect the latency for the systematic LT coded case to lie somewhere between these two extremes. Exact analysis for systematic LT codes is hard due to randomness in the number of systematic and parity rows at each worker, and is left for future work.

Theorem 1 (Latency of Uncoded Strategy)**.** *The expected latency of the uncoded strategy* $\mathbb{E}[T_{unc}]$ *is bounded as*

$$\mathbb{E}[X_{p:p}] + \tau\bar{S} \leq \mathbb{E}[T_{unc}] \leq \mathbb{E}[X_{p:p}] + \tau(\bar{S} + S_{\max}), \tag{2}$$

where $X_{p:p} = \max_{i \in [1,p]} X_i$ *and* $S_{\max} = \max_j S_j$.

As expected, the uncoded strategy is bottlenecked by the slowest worker (with setup time corresponding to $X_{p:p}$).

To characterize the latency of the LT Coded strategy we first observe that encoding increases the density of matrix rows because rows of $\mathbf{A}^{(\mathbf{e})}$ are obtained by adding d rows of $\mathbf{A}$. We assume that $\mathbf{A}$ only contains non-negative entries (as is the case for sparse matrices like graph adjacency matrices), so that non-zero elements do not sum to zero upon encoding, and that positions of non-zero entries in each row of $\mathbf{A}$ are chosen uniformly at random.

Theorem 2 (Density of Encoded Rows)**.** *The number of non-zeros,* $S^{(e)}$ *in any row of the encoded matrix* $\mathbf{A}^{(\mathbf{e})}$ *satisfies*

$$\Pr(S^{(e)} \geq s) \leq \sum_{d=1}^{m} \Omega_d \exp(-2\epsilon_{s,d}^2 n), \tag{3}$$

where $\epsilon_{s,d} = (\bar{s_n})^d - s_n$, $\bar{s_n} = \sum_s s_n \rho_s$, $s_n = 1 - s/n$, Ω_d *is the probability of generating a degree* d *encoded symbol, and* ρ_s *is the probability that a row of* $\mathbf{A}$ *has* s *non-zeros.*

Thus, the probability of a row of $\mathbf{A}^{(\mathbf{e})}$ having a large number of non-zeros is upper bounded by a term that decays exponentially. The exponent $\epsilon_{s,d}$ captures the deviation of s_n, the fraction of zeros in a row of $\mathbf{A}^{(\mathbf{e})}$, from $\bar{s_n}$, the expected fraction of zeros in a row of $\mathbf{A}$. In Figure 2a, we simulate the distribution of non-zero entries in the rows of $\mathbf{A}^{(\mathbf{e})}$ for both LT and Systematic LT coded strategies to illustrate this exponential decay. Here $\mathbf{A}$ is a 10000×10000 matrix with $\Pr[S_i = s] \propto s^{-2.5}$, $1 \leq s \leq 10000$ (Power law distribution).

Theorem 3 (Latency of LT Strategy)**.** *For large* m_e *i.e.* $\alpha = m_e/m \to \infty$, *the expected latency for the LT-coded case with* p *workers has the following upper and lower bounds.*

$$\mathbb{E}[T_{LT}] \leq \frac{\tau m_d \mathbb{E}[S^{(e)}]}{p} + \tau + \mathbb{E}[X], \tag{4}$$

$$\mathbb{E}[T_{LT}] \geq \frac{\tau m_d \mathbb{E}[S^{(e)}]}{p} + \mathbb{E}[X_{1:p}], \tag{5}$$

where $m_d = m(1 + \epsilon)$, *the expected number of symbols necessary for successful decoding.*

Latency of the rateless coded strategy is bounded by $\mathbb{E}[X]$ as opposed to $\mathbb{E}[X_{p:p}]$ for the uncoded case. A small penalty is paid in the other term due to the ϵ decoding overhead (in m_d), and the slight increase in sparsity after encoding (in $\mathbb{E}[S^{(e)}]$). However when setup time, X_i, dominates computation time (as happens in straggling) we can expect to benefit from coding.

We simulate the tail of the latency under different coded computing strategies in Figure 2b and Figure 2c for an exponential and Pareto distribution on the initial delay X

respectively. In addition to the uncoded, LT coded, and Systematic LT coded strategies, we also compare against the (p, k) encoded strategy [4] and a 2-replication strategy that is often used in real distributed computing systems, [2], [3]. In the 2-replication strategy, $\mathbf{A}$ is equally distributed across $p/2$ workers, and another copy is run of the remaining $p/2$ workers. The master uses the result returned by the faster worker for each submatrix. As expected, the LT and Systematic LT coded strategies have a lighter latency tail than the other strategies. The difference is especially stark when X follows a Pareto Distribution. This is because the Pareto distribution has a much heavier tail than the exponential distribution and thus represents more severe straggling. We also observe that the LT coded scheme appears to perform better than the systematic LT coded scheme. This is probably because the encoded matrix $\mathbf{A}^{(\mathbf{e})}$ generated by LT coding is generally sparser than its systematic LT coded counterpart (Figure 2a) due to which the row-vector products are computed faster in the LT coded case. However, the benefit of using systematic codes is that they can be easily built on top of existing uncoded systems since they only require addition of parity rows to $\mathbf{A}$, and in situations where there is little or no straggling, their performance is better than that of the LT codes, as the systematic rows have lower density than the LT coded rows and decoding is not required.

V. Rateless Coding for the CSC Format

Large matrices are often stored in the compressed sparse column (CSC) format, which is an alternative to the CSR format described in Section II-B. Row-based encoding strategies would involve significant encoding and decoding overhead for matrices in the CSC format. A naive strategy to compute the matrix-vector product $\mathbf{Ax}$ in a distributed manner when $\mathbf{A}$ is in the CSC format is to assign a subset of the columns $\mathbf{c}_1, \mathbf{c}_2, \dots \mathbf{c}_n$ to each worker. Each worker then computes the element-wise (or outer) product $\mathbf{c}_j \otimes x_j$ product for its assigned columns, and the master computes $\mathbf{b} = \mathbf{Ax} = \sum_{j=1}^{n} \mathbf{c}_j \otimes x_j$. However, this uncoded column-based strategy is susceptible to tail latency due to one or more straggling nodes.

Thus, in order to combat straggling nodes, we aim to design methods to encode the matrix along this compressed columns $\mathbf{c}_1, \mathbf{c}_2, \dots \mathbf{c}_n$. Similar to the CSR case, we propose using rateless codes for matrices in the CSC format. Each worker now stores a subset of the n columns of the matrix $\mathbf{A}$ and returns partial sums of the form $\sum_{j \in D} \mathbf{c}_j \otimes x_j$. The set $\mathcal{D}$ is determined by the code construction, which can be a systematic or a regular LT code. Unlike the coding strategy in Section II-B, the sums of the elements are computed after multiplication with the elements of the vector $\mathbf{x}$. Thus, this column-based strategy needs to store uncoded columns and uses more storage per worker node than CSR-format coding. The master will use the partial sums to determine $\mathbf{b} = \mathbf{Ax} = \sum_{j=1}^{n} \mathbf{c}_j \otimes x_j$. The latency analysis, which is similar to that presented in Section IV, is omitted due to space constraints.

Interestingly, since gradient codes proposed in [15] are specifically designed to recover the sum of source symbols, they can also be used for CSC-format matrix-vector multiplication. In the future, we plan to explore other sum-recovery codes, going beyond state-of-the-art gradient codes.

VI. Concluding Remarks

In this work we introduce the use of rateless codes to compute the product of a sparse matrix $\mathbf{A}$ with a vector $\mathbf{x}$ using a system of unreliable nodes. We combine a load balancing scheme with rateless coding to achieve near-perfect load balancing that overcomes heterogenity in the distribution of non-zero elements in the matrix as well as slowdown or straggling among worker nodes. Our theoretical analysis and simulations show the superiority of our approach to prior theoretical and system level solution, which are not designed to handle both sparsity in the data and straggling. Another advantage of our work is its applicability to matrices stored in sparse compressed formats like CSR and CSC due to which it can be easily incorporated into existing matrix-vector multiplication frameworks. Future directions include conducting experiments on existing distributed computing infrastructure, and exploring low-density alternatives [16] to LT codes.

References

[1] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[2] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Comm. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[4] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” *IEEE Transactions on Information Theory*, 2017.

[5] S. Dutta, V. Cadambe, and P. Grover, “Short-dot: Computing large linear transforms distributedly using coded short dot products,” in *Advances In Neural Information Processing Systems*, 2016, pp. 2100–2108.

[6] A. Mallick, M. Chaudhari, and G. Joshi, “Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication,” *arXiv preprint arXiv:1804.10331*, 2018.

[7] M. Luby, “LT Codes,” in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 2002, pp. 271–280.

[8] Y. Yang, M. Chaudhari, P. Grover, and S. Kar, “Coded iterative computing using substitute decoding,” *arXiv preprint arXiv:1805.06046*, 2018.

[9] S. Wang, J. Liu, and N. Shroff, “Coded sparse matrix multiplication,” *arXiv preprint arXiv:1802.03430*, 2018.

[10] J. L. Greathouse and M. Daga, “Efficient sparse matrix-vector multiplication on gpus using the csr storage format,” in *Proceedings of ACM/IEEE Supercomputing (SC)*. IEEE Press, 2014, pp. 769–780.

[11] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication,” in *Proceedings of ACM/IEEE Supercomputing (SC)*, Nov. 2016, pp. 678–689.

[12] A. Mallick and G. Joshi, “Rateless Codes for Distributed Computations with Sparse Compressed Matrices,” *http://www.andrew.cmu.edu/user/gaurij/rateless_coded_sparse.pdf*, 2019.

[13] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, “Fast sparse matrix-vector multiplication on gpus for graph applications,” in *Proceedings of ACM/IEEE Supercomputing (SC)*, 2014, pp. 781–792.

[14] A. Shokrollahi, “Raptor codes,” *IEEE transactions on information theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[15] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, “Gradient coding: Avoiding stragglers in distributed learning,” in *Proceedings of the International Conference on Machine Learning (ICML)*, vol. 70, 06–11 Aug 2017, pp. 3368–3376.

[16] G. Joshi and E. Soljanin, “Round-robin overlapping generations coding for fast content download,” in *IEEE International Symposium on Information Theory (ISIT)*, Jul. 2013, pp. 2740–2744.

[17] A. Wald, “On cumulative sums of random variables,” *The Annals of Mathematical Statistics*, vol. 15, no. 3, pp. 283–296, 1944.

[18] W. Hoeffding, “Probability inequalities for sums of bounded random variables,” *Journal of the American statistical association*, vol. 58, no. 301, pp. 13–30, 1963.

Appendix

A. CSR Matrix-Vector Multiplication

Multiplying a sparse matrix $\mathbf{A}$ with a vector $\mathbf{x}$ in the CSR format, involves only multiplying the non-zero elements of each row of $\mathbf{A}$ with the corresponding entries of $\mathbf{x}$. The exact algorithm is given in Algorithm 1.

Algorithm 1: CSR Matrix-Vector Multiplication

```
Input  : CSR Matrix $\mathbf{A}$ and input vector $\mathbf{x}$
Output : Product $\mathbf{b} = \mathbf{A}\mathbf{x}$
1 for $0 \leq i < m$ do
2     start = $\mathbf{A}$.offsets[i]
3     end = $\mathbf{A}$.offsets[i+1]
4     $\mathbf{b}[i] =< \mathbf{A}.\text{data}[\text{start} : \text{end}], \mathbf{x}[\mathbf{A}.\text{indices}[\text{start} : \text{end}]] >$
5 end
```

B. Systematic LT Codes

Systematic LT encoding involves applying LT decoding to the m input symbols $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m$ to generate a set of m pre-input symbols $\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2, \dots, \tilde{\mathbf{a}}_m$. LT encoding is then applied to the pre-input symbols to generate parity symbols. The exact algorithm is given in Algorithm 2 and is illustrated in Figure 4a

Algorithm 2: Systematic LT Encoder

```
Input  : $m$ input symbols $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m$
Output : $m_e$ encoded symbols $\mathbf{a}_{e_1}, \mathbf{a}_{e_2}, \dots, \mathbf{a}_{e_{m_e}}$ with $\mathbf{a}_{\mathbf{e}_{i_j}} = \mathbf{a}_j$, $1 \leq j \leq m$
1 Draw $m(1+\epsilon)$ vectors $\mathbf{v}_1, \dots, \mathbf{v}_{m(1+\epsilon)}$ from $\Omega(x)$ on $\mathbb{F}_2^m$. Let $\mathbf{E}$ be the matrix formed by $\mathbf{v}_1, \dots, \mathbf{v}_{m(1+\epsilon)}$ as rows.
2 Set $c = 0$ and $G = \mathbf{E}$
3 while $c$ < $m$ do
4     Identify a row of degree 1 in $\mathbf{E}$. If no such row exists raise an error.
5     If a row of degee 1 exists, set $i_c$ equal to the index of the row in $\mathbf{E}$. Delete the column of $G$ corresponding to the
        position of the non-zero element of this row.
6 end
7 Set $\mathbf{E_S}$ equal to the rows $i_1, \dots, i_m$ of $\mathbf{E}$
8 Apply the BP decoder on $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_{\mathbf{e}m}$ with $\mathbf{E_S}$ representing the encoding graph to get pre-input symbols $\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2, \dots, \tilde{\mathbf{a}}_m$
9 Generate $m_e$ encoded symbols $\mathbf{a}_{\mathbf{e}1}, \mathbf{a}_{\mathbf{e}2}, \dots, \mathbf{a}_{\mathbf{e}m}$ by performing LT encoding on $\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2, \dots, \tilde{\mathbf{a}}_m$. Out of the first $m(1+\epsilon)$
    encoded symbols, $\mathbf{a}_{\mathbf{e}_{i_j}} = \mathbf{a}_j$, $1 \leq j \leq m$. All remaining symbols are parity symbols.
```

C. Additional Simulations

To further study the effect of our balanced row allocation policy (Definition 1) we simulated the multiplication of a 10000×10000 sparse matrix $\mathbf{A}$ with a 10000×1 vector $\mathbf{x}$, over $p = 10$ worker nodes using the uncoded and LT coded ($\alpha = m_e/m = 2.0$) strategies. The number of non-zeros in row j of $\mathbf{A}$ follows a Power law distribution ($\Pr[S_j = s] \propto s^{-2.5}$, $1 \leq s \leq 10000$). We assumed the time taken to perform a single computation (as per our delay model in (1)) is $\tau = 5 \times 10^{-7}$ and that the initial delay X_i at worker i follows a Pareto (1,1) distribution (for all $i = 1, \dots, p$). The results are shown in Figure 3. Uncoded distributed sparse matrix vector multiplication clearly benefits from the balanced task allocation policy as seen by the much lighter latency tail of the balanced version (Uncoded (Bal)). However the difference is much less stark in the LT coded case where both the balanced and unbalanced versions have almost a similar latency tail (that is much lighter than the uncoded cases). We hypothesize that since the encoding involves adding a random subset of the rows of $\mathbf{A}$ to generate a row of $\mathbf{A}^{(\mathbf{e})}$, the variability in the number of non-zeros in the rows of $\mathbf{A}^{(\mathbf{e})}$ is much lower than the variability in the number of non-zeros in the rows of $\mathbf{A}$. Due to this, balanced row allocation in the coded case does not lead to a large difference in latency. We leave theoretical/experimental analysis to justify this hypothesis for a future work.

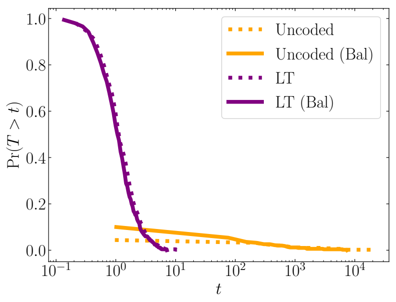


Fig. 3: Comparison of uncoded and Systematic LT coded strategies with and without balanced row allocation (Bal)

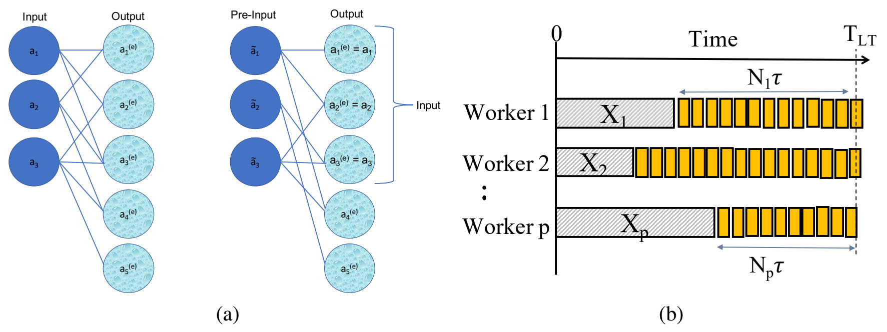


Fig. 4: (a) Comparison between LT encoding (left) and Systematic LT encoding (right). (b) Worker i has a random exponential initial delay X_i, after which it completes N_i computations (multiplications and additions), denoted by the small rectangles, taking time τ per computation. The latency T_{LT} is the time to complete S' computations in total.

D. Proof of Theorems

1) Proof of Theorem 1:

Proof. In the uncoded strategy every worker needs to complete *all* computations assigned to it and the latency is equal to the time taken by the slowest worker to complete its assigned task.

Since the sub-matrix $\mathbf{A}_i$ assigned to worker i has N_i non-zero elements, worker i performs N_i computations. As per our model, the time taken by worker i to perform N_i computations is given by

$$Y_i = X_i + \tau N_i, \quad \text{for } i = 1, \dots, p. \tag{6}$$

and the overall latency is the time taken by the slowest worker which is $Y_{p:p} = \max_i Y_i$. Therefore

$$Y_{p:p} = \max_i (X_i + \tau N_i) \tag{7}$$

From Lemma 1 we know that $N_i \in [\bar{S}, \bar{S} + S_{\max}]$. Thus

$$\max_i (X_i + \tau N_i) \geq \max_i (X_i + \tau \bar{S}) \tag{8}$$

$$\max_i (X_i + \tau N_i) \leq \max_i (X_i + \tau(\bar{S} + S_{\max})) \tag{9}$$

The result of the theorem follows from the fact that the second part in both the upper and the lower bounds ((8), (9)) is independent of i and $X_{p:p} = \max_i X_i$ and hence we can take the expectation over both the lower and the upper bound. □

2) Proof of Theorem 2:

Proof. We assume that the the j^{th} row of the encoded matrix $\mathbf{A}^{(\mathbf{e})}$ is generated by drawing D_j rows of the original matrix $\mathbf{A}$ uniformly at random ($\Pr[D_j = d] = \Omega_d$) and then adding them. Thus,

$$\Pr[S_j^{(e)} \geq s] = \sum_d \Pr[S_j^{(e)} \geq s | D_j = d] \Pr[D_j = d] \tag{10}$$

$$= \sum_d \Pr[S_j^{(e)} \geq s | D_j = d] \Omega_d \tag{11}$$

For a given degree 'd', the following lemma bounds the probability of the number of non-zeros in an encoded row obtained by adding 'd' original rows,

Lemma 2. *Given 'd' non-negative vectors $\mathbf{a}_1, \dots, \mathbf{a}_d$ each of length n, where vector $\mathbf{a}_i$ has S_i non-zeros ($\Pr[S_i = s] = \rho_s$), the sum $\mathbf{a}^{(e)} = \mathbf{a}_1 + \dots + \mathbf{a}_d$ has $S^{(e)}$ non-zeros where*

$$\Pr[S^{(e)} \geq s] \leq \exp(-2\epsilon_{s,d}^2 n) \tag{12}$$

where $\epsilon_{s,d} = (\bar{s_n})^d - s_n$, $\bar{s_n} = \sum_s s_n \rho_s$, $s_n = 1 - s/n$.

Since the rows of $\mathbf{A}$ are chosen uniformly at random to obtain each row of $\mathbf{A}^{(\mathbf{e})}$, we can substitute (12) in place of $\Pr[S_j^{(e)} \geq s | D_j = d]$ in (11) to obtain the final result. □

3) Proof of Theorem 3:

Proof. As per our model, the time taken by worker i to perform N_i computations is given by

$$Y_i = X_i + \tau N_i, \quad \text{for } i = 1, \dots, p. \tag{13}$$

The latency T_{LT} is the earliest time when the workers collectively complete M' row-vector products, as illustrated in Figure 4b. Let S' be the (random) number of non-zeros in the M' encoded row-vector products. Thus S' is the total number of computations completed by the workers in time T_{LT}.

We note that, in this case it is not necessary that each worker has completed at least 1 computation. Specifically, if $T_{\text{LT}}^{(\infty)} - X_i \leq \tau$ for any i then it means that worker i has not performed even a single computation in the time that the system as a whole has completed S' computations (owing to the large initial delay X_i). Therefore we define

$$\mathcal{W}_{\text{LT}} := \{i : T_{\text{LT}} - X_i \geq \tau\} \tag{14}$$

Here $\mathcal{W}_{\text{LT}}$ is the set of workers for which $N_i > 0$. Thus

$$\begin{aligned} T_{\text{LT}} &= \max_{i \in \mathcal{W}_{\text{LT}}} Y_i = \max_{i \in \mathcal{W}_{\text{LT}}} (X_i + \tau N_i), & (15)\\ &\geq \min_{i \in \{1, \dots p\}} X_i + \tau \max_{i \in \mathcal{W}_{\text{LT}}} N_i, & (16)\\ &\geq X_{1:p} + \tau \frac{S'}{p}, & (17) \end{aligned}$$

where to obtain (16), we replace each X_i in (15) by $\min_{i \in [1, \dots p]} X_i$ and then we can bring it outside the maximum. To obtain (17), we observe that in order for the p workers to collectively finish S' computations, the maximum number of computations completed by a worker has to be at least S'/p. Taking expectation on both sides of (17) gives

$$\begin{aligned} \mathbb{E}[T_{\text{LT}}] &\geq \mathbb{E}[X_{1:p}] + \tau \frac{\mathbb{E}[S']}{p}, & (18)\\ &= \mathbb{E}[X_{1:p}] + \frac{\tau}{p} \mathbb{E}[\sum_{j=1}^{M'} S_j^{(e)}], & (19)\\ &= \mathbb{E}[X_{1:p}] + \frac{\tau}{p} \mathbb{E}[M'] \mathbb{E}[S^{(e)}], & (20)\\ &= \mathbb{E}[X_{1:p}] + \frac{\tau m_d}{p} \mathbb{E}[S^{(e)}] & (21) \end{aligned}$$

where $S_j^{(e)}$ is the number of non-zero entries in row j of $\mathbf{A}^{(\mathbf{e})}$ and S' is the total number of non-zero entries in the M' completed row-vector products. Since $S_j^{(e)}$ are all i.i.d and independent of M', according to Wald's identity [17], $\mathbb{E}[\sum_{j=1}^{M'} S_j^{(e)}] = \mathbb{E}[M'] \mathbb{E}[S^{(e)}]$ in (19).

To derive the upper bound, we note that

$$T_{\text{LT}} \leq X_i + \tau(N_i + 1), \quad \text{for all } i = 1, \dots, p \tag{22}$$

This is because at time T_{LT} each of the workers $1, \dots, p$, have completed $N_1, \dots, N_p$ computations respectively, but they may have partially completed the next computation. The 1 added to each N_i accounts for this edge effect, which is also illustrated in Figure 4b. Summing over all i on both sides, we get

$$\begin{aligned} \sum_{i=1}^{p} T_{\text{LT}} &\leq \sum_{i=1}^{p} X_i + \sum_{i=1}^{p} \tau (N_i + 1) & (23)\\ p T_{\text{LT}} &\leq \sum_{i=1}^{p} X_i + \tau (S' + p) & (24)\\ T_{\text{LT}} &\leq \frac{1}{p} \sum_{i=1}^{p} X_i + \tau \left(\frac{S'}{p} + 1 \right) & (25) \end{aligned}$$

Likewise for the upper bound we can compute expectation on both sides of (25) to get,

$$\mathbb{E}[T_{\mathrm{LT}}] \leq \frac{1}{p}\sum_{i=1}^{p}\mathbb{E}[X_i] + \tau\left(\frac{\mathbb{E}[S']}{p}+1\right) \tag{26}$$

$$= \mathbb{E}[X] + \tau + \frac{\tau}{p}\mathbb{E}[\sum_{j=1}^{M'} S_j^{(e)}] \tag{27}$$

$$= \mathbb{E}[X] + \tau + \frac{\tau m_d \mathbb{E}[S^{(e)}]}{p} \tag{28}$$

where (28) once again follows from Wald's identity. □

E. Proof of Lemmas

1) Proof of Lemma 1: Recall the balanced row allocation policy for a $m \times n$ matrix $\mathbf{A}$ over p workers.

Definition 3 (Balanced Row Allocation)**.** *Let $\bar{S} = (1/p)\sum_j S_j$, where row j of $\mathbf{A}$ has S_j non-zero elements. Then $\mathbf{A}$ is split along its rows and worker i is assigned rows J_i^0 to J_i^f as per the following policy.*

$$J_1^0 = 0$$
$$J_i^0 = J_{i-1}^f + 1, \quad i = 2, \ldots, p$$
$$J_i^f = \min_l \sum_{j=J_i^0}^{l} S_j \geq \bar{S}, \quad i = 1, \ldots, p-1$$
$$J_p^f = m$$

Let, $\Delta_i = \sum_{j=J_i^0}^{J_i^f} S_j - \bar{S}$. This denotes the number of *additional* elements allocated to worker i to avoid dividing a row of $\mathbf{A}$ over multiple workers. Thus the total number of elements allocated to worker i is,

$$N_i = \sum_{j=J_i^0}^{J_i^f} S_j = \bar{S} + \Delta_i \tag{29}$$

$$\leq \bar{S} + \max_j S_j \tag{30}$$

where (30) follows from the fact that $\Delta_i \leq \max_j S_j$ (since Δ_i is less than the number of entries in row J_i^f of $\mathbf{A}$ which is upper bounded by $\max_j S_j$). This proves the upper bound in the lemma.

Moreover since for every worker i,

$$J_i^f = \min_l \sum_{j=J_i^0}^{l} S_j \geq \bar{S} \tag{31}$$

Therefore,

$$N_i = \sum_{j=J_i^0}^{J_i^f} S_j \geq \bar{S} \tag{32}$$

This proves the lower bound in the lemma.

2) Proof of Lemma 2: Define indicator variables $V_j = \mathbb{1}\{\mathbf{a}_j^{(e)} \neq 0\}$ and $U_{ij} = \mathbb{1}\{\mathbf{a}_{ij} \neq 0\}$ for $i = 1, \ldots, d$ and $j = 1, \ldots, n$. Thus U_{ij} indicates whether the j^{th} element of $\mathbf{a}_i$ is non-zero and V_j indicates whether the j^{th} element of the sum $\mathbf{a}^{(e)} = \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_d$ is non-zero. Since $\mathbf{a}_i$ has S_i non-zero elements whose positions are assumed to be chosen uniformly at random, therefore

$$\Pr(U_{ij} = 1) = \sum_{s=1}^{n} \Pr(U_{ij} = 1 | S_i = s)\Pr(S_i = s) \tag{33}$$

It is assumed that $\Pr(S_i = s) = \rho_s$. Also,

$$\Pr(U_{ij} = 1 | S_i = s) = \frac{\binom{n-1}{s-1}}{\binom{n}{s}} = \frac{s}{n} \tag{34}$$

This is because $\Pr(U_{ij} = 1 | S_i = s)$ is the ratio of the number of n length vectors with 's' non-zeros and a non-zero element at position j (and the position of the other non-zero entries chosen uniformly at random), to the total number of n length vectors with 's' non-zeros (with the position of the other non-zero entries chosen uniformly at random). Thus,

$$\Pr(U_{ij} = 1) = \sum_{s=1}^{n} \frac{s}{n} \rho_s = \bar{\rho} \tag{35}$$

Since $\mathbf{a}_j^{(e)} = \sum_{i=1}^{d} \mathbf{a}_{ij}$, therefore $\mathbf{a}_j^{(e)} = 0$ only when $\mathbf{a}_{ij} = 0$, $i = 1, \ldots, d$ ($\mathbf{a}_i$'s are assumed to be non-negative). Thus,

$$\Pr(V_j = 0) = \prod_{i} \Pr(U_{ij} = 0) = (1 - \bar{\rho})^d \tag{36}$$

since U_{ij}'s are i.i.d for all i with $\Pr(U_{ij} = 0) = 1 - \Pr(U_{ij} = 1)$.

Define,

$$\bar{s_n} = 1 - \bar{\rho} = \sum_{s=1}^{n} \left(1 - \frac{s}{n}\right) \rho_s \tag{37}$$

Thus, $\Pr(U_{ij} = 0) = 1 - \Pr(U_{ij} = 1) = \bar{s_n}$ and $\Pr V_j = 0 = (\bar{s_n})^d$.

Finally, the number of non-zero elements, $S^{(e)}$ in $\mathbf{a}^{(e)}$ is equal to the total number of positions at which $\mathbf{a}^{(e)}$ has non-zero entries. Thus,

$$S^{(e)} = V_1 + \ldots + V_n \tag{38}$$

Thus,

$$\Pr(S^{(e)} \geq s) = \Pr(\sum_{j=1}^{n} V_j \geq s) \tag{39}$$

Using Hoeffding's inequality [18] on the sum of i.i.d Bernoulli random variables $V_1, \ldots, V_n$, we can show that,

$$\Pr(\sum_{j=1}^{n} V_j \geq s) \leq \exp\left(-2n \left(\frac{s}{n} - (1 - (\bar{s_n})^d)\right)^2\right) \tag{40}$$

This is because $\Pr(V_j = 1) = 1 - \Pr(V_j = 0) = 1 - (\bar{s_n})^d$ for all $j = 1, \ldots, n$. The result follows by setting $s_n = 1 - s/n$ and $\epsilon_{s,d} = (\bar{s_n})^d - s_n$