Rook Coding for Batch Matrix Multiplication

Pedro Soto, Xiaodi Fan, Angel Saldivia, Jun Li, *Member, IEEE*

***Abstract*—Matrix multiplication is a fundamental building block in various distributed computing algorithms. In order to multiply large matrices, it is common practice to distribute the computation into multiple tasks running on different nodes. In order to tolerate stragglers among such nodes, various coding schemes have been proposed by adding additional coded tasks. However, most existing coding schemes for matrix multiplication are constructed for only one matrix multiplication, while batch matrix multiplication is common in large-scale distributed computing workloads. In this paper, we propose Rook Coding (RC), a novel polynomial-based coding framework for computing the multiplication of n pairs of matrices in batch. Designed to achieve lower encoding time in practice, we construct RC as polynomials of much simpler forms than existing coding schemes for batch matrix multiplication, achieving a recovery threshold of $O(n^{\log_2 3})$. Compared to existing coding schemes, RC achieves a lower encoding complexity in practice, because of its simpler forms in the encoding polynomials. Through extensive experiments, we show that RC can save the time of the whole job thanks to its low overhead of encoding.**

***Index Terms*—distributed computing, batch matrix multiplication, straggler mitigation, coded computing**

I. Introduction

A. Background

RECENT advances in large-scale distributed computing have demonstrated success in various applications, such as machine learning and data analytics. With the massive sizes of modern datasets, it has become inevitable to run large-scale computing jobs in a distributed infrastructure, by distributing the computation into multiple tasks running in parallel on a large number of nodes.

However, it is well known that nodes in a distributed infrastructure are typically built with commodity hardware and are subject to various faulty behaviors [2]–[4]. Therefore, when the computation is distributed onto multiple nodes, its progress can be significantly affected by the tasks running on slow or failed nodes [5], which we call *stragglers*. The adversarial effects of stragglers can be mitigated by launching redundant tasks in advance. A naive application of this principle is to replicate each task on multiple nodes. For example, if we run each task on three nodes, the results of any two tasks affected by stragglers can be simply disregarded while all the other tasks can continue without being delayed. This naive method, however, will significantly increase the consumption of resources, including computing, communication, and storage, with only a limited number of stragglers tolerable. Specifically, in order to tolerate any r stragglers, we have to replicate each task on $r + 1$ nodes.

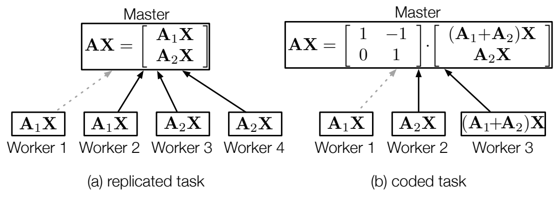

Master
$\mathbf{AX} = \begin{bmatrix} \mathbf{A}_1\mathbf{X} \\ \mathbf{A}_2\mathbf{X} \end{bmatrix}$
$\mathbf{A}_1\mathbf{X}$
Worker 1
$\mathbf{A}_1\mathbf{X}$
Worker 2
$\mathbf{A}_2\mathbf{X}$
Worker 3
$\mathbf{A}_2\mathbf{X}$
Worker 4
(a) replicated task
Master
$\mathbf{AX} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} (\mathbf{A}_1+\mathbf{A}_2)\mathbf{X} \\ \mathbf{A}_2\mathbf{X} \end{bmatrix}$
$\mathbf{A}_1\mathbf{X}$
Worker 1
$\mathbf{A}_2\mathbf{X}$
Worker 2
$(\mathbf{A}_1+\mathbf{A}_2)\mathbf{X}$
Worker 3
(b) coded task

Fig. 1. Examples of distributed matrix multiplications with additional workers (running replicated or coded tasks) to tolerate one single straggler, represented with a gray dotted arrow.

On the other hand, it has been demonstrated that we can tolerate the same number of stragglers with fewer tasks if we run *coded* tasks as redundant tasks. Fig. 1 illustrates an example of distributed matrix multiplication with replicated and coded tasks. We calculate $\mathbf{AX}$ on four worker nodes in Fig. 1a. The matrix $\mathbf{A}$ is split into two submatrices, $\mathbf{A}_1$ and $\mathbf{A}_2$, and then $\mathbf{AX}$ can be obtained from the results of two tasks, *i.e.*, $\mathbf{A}_1\mathbf{X}$ and $\mathbf{A}_2\mathbf{X}$. In Fig. 1a, the two tasks are replicated on two workers, respectively. Therefore, any single straggler among the total four workers can be tolerated without affecting the overall performance. In Fig. 1b, however, a third worker executes a coded task $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$, which can be used to recover $\mathbf{A}_1\mathbf{X}$ and $\mathbf{A}_2\mathbf{X}$ if $\mathbf{A}_1\mathbf{X}$ or $\mathbf{A}_2\mathbf{X}$ runs on a straggler. Therefore, compared to replicating the two tasks in Fig. 1a, coded matrix multiplication in Fig. 1b can save the number of additional workers by 50% and tolerate the same number of stragglers.

Although straggler-free coding for distributed computing, especially coded matrix multiplication (*e.g.,* [3], [6]–[8]), has attracted a significant amount of research attention, most existing coding schemes focus on the code construction for one single matrix multiplication so far. In this paper, we consider batch matrix multiplication, a more general scenario where multiple matrix multiplications need to be computed in parallel. Although we can launch multiple distributed jobs for each multiplication in parallel, each job will then need to have its own coded task to tolerate stragglers. It will naturally require significantly more tasks as we don't know which job will have stragglers. Therefore, we aim to design a coding scheme in this paper such that only one job needs to be launched for batch matrix multiplication where stragglers can be flexibly tolerated.

B. Related Work

There has been a surge of interest recently on the mitigation of stragglers in distributed computing, which runs a large number of parallel tasks on different nodes. It is well known

P. Soto and X. Fan are with the Graduate Center of the City University of New York, New York, NY. A. Saldivia is with the School of Computing and Information Sciences, Florida International University, Miami, FL. J. Li is with the Queens College and the Graduate Center of the City University of New York, New York, NY.

This paper was presented in part at the 2020 IEEE International Symposium on Information Theory [1].

that stragglers are common in distributed computing with a large number of nodes, and stragglers can add significant long-tail latency to the overall performance, even though there are only a small number of tasks affected by stragglers [3], [4]. Conventionally stragglers are tolerated by replicating each task on multiple nodes [9]–[13], such that a task affected by a straggler can be simply disregarded. However, replication incurs a significant resource overhead as all tasks need to be replicated. Compared to replication, coding-based techniques have been proposed to tolerate the same number of stragglers with lower resource overhead (*e.g.*, [3], [4]).

One of the critical applications of coded distributed computing is distributed matrix multiplication, as matrix multiplication is a common operation in various machine learning models and data analytics algorithms. In Lee *et al.*'s pioneering paper [3], Maximum Distance Separable (MDS) codes are applied in matrix-vector multiplication of the form $\mathbf{A}x$ where the matrix $\mathbf{A}$ is large and hence will be partitioned vertically along the rows and encoded with MDS codes. This direction was continued by Dutta *et al.* [6] and Yu *et al.* [7] who considered a more general problem of matrix-matrix multiplication, *i.e.*, $\mathbf{A} \cdot \mathbf{B}$. The two input matrices $\mathbf{A}$ and $\mathbf{B}$ are partitioned along their rows and columns, respectively, or the other way round. The proof of the optimality for such configurations have been given by Yu *et al.* [7], [8], where the authors have also devised entangled polynomial codes that allow partitioning the input matrices in a more general way, such that $\mathbf{A}$ and $\mathbf{B}$ can both be arbitrarily partitioned by their rows and columns.

In this paper, we consider a more general problem of distributed coded computing for batch matrix multiplication, *i.e.*, the computation of all matrix multiplications is completed in one round. A related problem was investigated by Krishnan *et al.* [14] where the computation is complete with additional rounds such that coding can be applied across time. However, multiple rounds of computation may incur a significant amount of communication overhead. Therefore, in this paper, we focus batch matrix multiplication, *i.e.*, $\mathbf{A}_i\mathbf{B}_i$, $i = 1, \ldots, n$. Workers perform multiplication of matrices encoded from the $2n$ input matrices, and the master can directly decode results received from a certain number of workers to obtain the results of n matrix multiplications.

Existing works for batch matrix multiplication have been constructed based on polynomial evaluations. In other words, any worker computes the multiplication of two polynomials $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, where the values of x differ on different workers. The coefficients of the polynomials are matrices encoded from the input matrices. For example, Lagrange Coded Computing (LCC) [15] constructs $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ as Lagrange polynomials. As $\tilde{\mathbf{A}}(x) \cdot \tilde{\mathbf{B}}(x)$ are still a polynomial, its coefficients can be interpolated from results received from different workers with different values of x. Another example is Cross Subspace Alignment (CSA) codes [16]. CSA codes construct $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ such that their multiplication facilitates a Cauchy-Vandermonde structure, where desired components ($\mathbf{A}_i\mathbf{B}_i, i = 1, \ldots, n$) correspond to the Cauchy part and other interference components are aligned into the Vandermonde part. Both LCC and CSA codes achieve an optimal *recovery threshold* of $2n-1$, *i.e.*, $2n-1$ workers are needed to decode the results. However, as we argue below, it does not necessarily minimize the completion time in practice.

To evaluate $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, LCC and CSA codes rely on fast algorithms for polynomial interpolation and/or evaluation with the $O(n \log^2 n)$ complexity [17]–[21]. However, the encoding based on such algorithms has to be centralized, *i.e*, input matrices will be encoded into all tasks on one single node. Instead of constructing complex polynomials for the centralized fast algorithm, we propose rook polynomial coding (RC), another polynomial-based coding scheme for batch matrix multiplication in this paper. RC is constructed as polynomials of much simpler forms, making it faster for encoding in practice, especially distributed encoding. Trading off the recovery threshold for a lower encoding complexity, RC can eventually achieve a lower completion time as demonstrated by experiments on AWS.

There is no partitioning of input matrices in the aforementioned LCC and CSA codes. In other words, each worker has the same computational load as multiplying $\mathbf{A}_i\mathbf{B}_i$, $\forall i = 1, \ldots, n$. To support partitioning, Yu *et al.* integrated LCC codes with entangled polynomial codes [22]. Moreover, Chen *et al.* extended CSA codes into Generalized Cross Subspace Alignment (GCSA) codes with noise alignment [23]. In this paper, we focus on the construction of RC without matrix partitioning. We aim to integrate RC with entangled polynomial codes to support matrix partitioning in our future work.

C. Contributions

To tolerate stragglers altogether, there exist coding schemes constructed based on polynomial evaluations, *e.g.*, LCC codes [15] and CSA codes [16]. Although they can tolerate stragglers among multiple matrix multiplications, we find that their costs of encoding can be overwhelmingly high. Although fast algorithms based on multi-point polynomial evaluations can be incorporated into the algorithm of encoding [17]–[21], they outperform the traditional algorithms (*e.g.*, Horner's method) only when n is very large [24]. In practice, we find that this number can be larger than 1000. Moreover, the fast algorithms require centralized computing, making it impossible to speed up encoding by distributed computing.

In this paper, we propose Rook Coding (RC) for batch matrix multiplication that is designed towards low overhead during encoding in practice. Instead of constructing complex polynomials fit for fast algorithms of multi-point polynomial evaluations, we construct RC as polynomials with very simple forms, which meanwhile allow for an easy distributed implementation of encoding with low overhead. We gradually update the constructions of RC by improving the recovery threshold step-by-step, and eventually achieve an $O(n^{\log_2 3})$ recovery threshold. By running experiments on Amazon EC2, we demonstrate that RC achieves a much lower time of encoding in both centralized and distributed manners. Thanks to the time saved during encoding, RC achieves lower job completion time than other coding schemes for batch matrix multiplication, even though it has a higher recovery threshold.

II. Motivating Examples

We start with a toy example to demonstrate the advantages of our coding framework for batch matrix multiplication. Assume that we need to compute the results of two matrix multiplications, *i.e.*, $\mathbf{A}_1\mathbf{B}_1$ and $\mathbf{A}_2\mathbf{B}_2$, where $\mathbf{A}_1$ and $\mathbf{A}_2$ are of the same size, and $\mathbf{B}_1$ and $\mathbf{B}_2$ are of the same size, too. If the two multiplications are computed as two jobs, we can replicate their sole tasks on $r+1$ nodes, such that any r stragglers can be tolerated. In other words, we need to have $2(r+1)$ tasks to tolerate any r stragglers and complete the two matrix multiplications, since the replicated tasks for one job cannot be used in the other job.

A naive way to add coded tasks for the two jobs is to embed the two matrix multiplications into one larger job as $\hat{\mathbf{A}} \cdot \hat{\mathbf{B}} \triangleq \begin{bmatrix}\mathbf{A}_1 \\ \mathbf{A}_2\end{bmatrix} \cdot \begin{bmatrix}\mathbf{B}_1 & \mathbf{B}_2\end{bmatrix} = \begin{bmatrix}\mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 \\ \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2\end{bmatrix}$. In this way, the result of $\mathbf{A}_1\mathbf{B}_1$ and $\mathbf{A}_2\mathbf{B}_2$ can be obtained as submatrices of $\hat{\mathbf{A}}\hat{\mathbf{B}}$. However, the complexity of $\hat{\mathbf{A}}\hat{\mathbf{B}}$ becomes four times as large as $\mathbf{A}_i\mathbf{B}_i$. In order to generate coded tasks with the same complexity as $\mathbf{A}_i\mathbf{B}_i$, we can apply polynomial codes [7], a polynomial-based coding scheme for matrix multiplication, to the job of $\hat{\mathbf{A}}\hat{\mathbf{B}}$, by encoding $\hat{\mathbf{A}}$ as $\tilde{\mathbf{A}}(x) = \mathbf{A}_1 x^0 + \mathbf{A}_2 x^2$ and $\hat{\mathbf{B}}$ as $\tilde{\mathbf{B}}(x) = \mathbf{B}_1 x^0 + \mathbf{B}_2 x^1$, and then the sizes of $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ equal those of $\mathbf{A}_i$ and $\mathbf{B}_i$, respectively. A coded task can then be generated as a polynomial of $\tilde{\mathbf{C}}(x) \triangleq \tilde{\mathbf{A}}(x)\tilde{\mathbf{B}}(x)$, *i.e.*, $\tilde{\mathbf{C}}(x) = \mathbf{A}_1\mathbf{B}_1 x^0 + \mathbf{A}_1\mathbf{B}_2 x^1 + \mathbf{A}_2\mathbf{B}_1 x^2 + \mathbf{A}_2\mathbf{B}_2 x^3$. Given any 4 coded tasks $\tilde{\mathbf{C}}(x)$ with different values of x, the coefficients of this polynomial can be solved with interpolation or Reed-Solomon decoding. In other words, the recovery threshold is 4, and we can tolerate r stragglers with a total of $4+r$ tasks. With n matrix multiplications, it is easy to infer that the recovery threshold is n^2.

In this paper, we propose a coding framework that requires significantly fewer tasks than replication and polynomial codes, tolerate the same number of stragglers with the same complexity in the coded tasks. Given the two matrix multiplications above, $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ can be generated differently as $\tilde{\mathbf{A}}(x) = \mathbf{A}_1 x^0 + \mathbf{A}_2 x^1$, and $\tilde{\mathbf{B}}(x) = \mathbf{B}_1 x^0 + \mathbf{B}_2 x^1$, respectively. Hence, $\tilde{\mathbf{C}}(x) = \mathbf{A}_1\mathbf{B}_1 x^0 + (\mathbf{A}_1\mathbf{B}_2 + \mathbf{A}_2\mathbf{B}_1)x^1 + \mathbf{A}_2\mathbf{B}_2 x^2$. In this way, we only need the results of any three coded tasks (with different values of x), and the recovery threshold becomes 3. Although when $n = 2$ the recovery threshold can only be saved by 25%, we demonstrate in the rest of this paper that the recovery threshold can eventually be saved from $O(n^2)$ to $O(n^{\log_2 3})$.

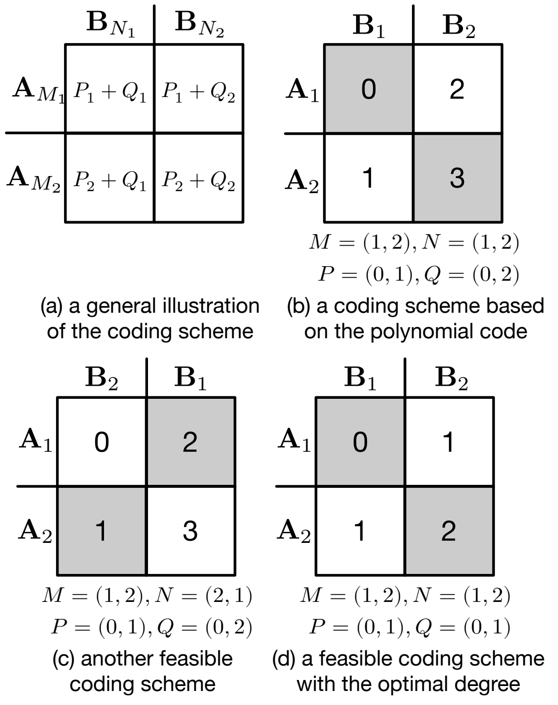
The polynomials in RC are also simpler compared to other coding schemes for batch matrix multiplication. For example, with LCC codes, $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ are encoded as $-\mathbf{A}_1(x-2) + \mathbf{A}_2(x-1)$ and $-\mathbf{B}_1(x-2) + \mathbf{B}_2(x-1)$, or equivalently $(-\mathbf{A}_1 + \mathbf{A}_2)x + (2\mathbf{A}_1 - \mathbf{A}_2)$ and $(-\mathbf{B}_1 + \mathbf{B}_2)x + (2\mathbf{B}_1 - \mathbf{B}_2)$, respectively. By comparing the coefficients, we can see that RC only uses one of the input matrices as the coefficient of each term in $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, allowing it to save the time of encoding in practice.

III. Coding Framework

Given n matrix multiplications, *i.e.*, $\mathbf{A}_1\mathbf{B}_1$, $\mathbf{A}_2\mathbf{B}_2$, ..., and $\mathbf{A}_n\mathbf{B}_n$, we assume that $\mathbf{A}_1, \ldots, \mathbf{A}_n$ are of the same size, and $\mathbf{B}_1, \ldots, \mathbf{B}_n$ also have the same sizes. In our coding framework, the parameters of the coding scheme can be described by four vectors with n elements, M, N, P, and Q. In particular, M and N are permutations of $\{1, \ldots, n\}$. The values in P and Q can be arbitrary integers, which will be used as the exponents in the polynomial. The n matrix multiplications can be encoded into coded tasks which multiply $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, where $\tilde{\mathbf{A}}(x) = \sum_{i=1}^{n} \mathbf{A}_{M_i} x^{P_i}$ and $\tilde{\mathbf{B}}(x) = \sum_{i=1}^{n} \mathbf{B}_{N_i} x^{Q_i}$. Compared to other polynomials such as the Lagrange interpolation polynomial, our encoding polynomials have much simpler forms since all input matrices appear directly as a coefficient, making it ideal for distributed encoding where each worker will encode the input matrices to get polynomials in its own task only.

Given $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, the result of a coded task will be $\tilde{\mathbf{C}}(x) = \tilde{\mathbf{A}}(x)\tilde{\mathbf{B}}(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{A}_{M_i}\mathbf{B}_{N_j} x^{P_i+Q_j}$, which is still a polynomial of x. The result in each task can be considered as an evaluation of $\tilde{\mathbf{C}}(x)$ if x in each task is unique. We can then interpolate $\tilde{\mathbf{C}}(x)$ if the number of results received from different workers is no less than the recovery threshold. With appropriate choices of M, N, P, and Q, we can find $\mathbf{A}_i\mathbf{B}_i$ from the coefficients of $\tilde{\mathbf{C}}(x)$. In other words, their corresponding exponents of x should be unique.

To illustrate the code scheme, in this paper, we use an $n \times n$ table to depict choices of M, N, P, Q, as shown in Fig. 2a. In this table, the entry in the i-th row and the j-th column is filled with $P_i + Q_j$, the exponent of $\mathbf{A}_{M_i}\mathbf{B}_{N_j}$. We also place $\mathbf{A}_{M_i}$ and $\mathbf{B}_{N_i}$, $i = 1, \ldots, n$, as the head of each row and each column, respectively. We demonstrate three examples of feasible coding schemes under our coding framework in Fig. 2b-Fig. 2d.



$\mathbf{B}_{N_1}$ $\mathbf{B}_{N_2}$
$\mathbf{A}_{M_1}$ P_1+Q_1 P_1+Q_2
$\mathbf{A}_{M_2}$ P_2+Q_1 P_2+Q_2
(a) a general illustration of the coding scheme
$\mathbf{B}_1$ $\mathbf{B}_2$
$\mathbf{A}_1$ 0 2
$\mathbf{A}_2$ 1 3
$M=(1,2), N=(1,2)$
$P=(0,1), Q=(0,2)$
(b) a coding scheme based on the polynomial code
$\mathbf{B}_2$ $\mathbf{B}_1$
$\mathbf{A}_1$ 0 2
$\mathbf{A}_2$ 1 3
$M=(1,2), N=(2,1)$
$P=(0,1), Q=(0,2)$
(c) another feasible coding scheme
$\mathbf{B}_1$ $\mathbf{B}_2$
$\mathbf{A}_1$ 0 1
$\mathbf{A}_2$ 1 2
$M=(1,2), N=(1,2)$
$P=(0,1), Q=(0,1)$
(d) a feasible coding scheme with the optimal degree

Fig. 2. The illustrations of the coding schemes achieved under the coding framework with $n = 2$.

In Fig. 2b, a naive choice of parameters in the coding framework is to let $M = N = (1, \ldots, n)$, $P = (0, \ldots, n-1)$, and $Q = (0, n, \ldots, (n-1)n)$, corresponding to the example of polynomial codes in Sec. II. Hence, we have $\tilde{\mathbf{A}}(x) = \sum_{i=1}^{n} \mathbf{A}_i x^{i-1}$ and $\tilde{\mathbf{B}}(x) = \sum_{i=1}^{n} \mathbf{B}_i x^{(i-1)n}$. Therefore, in $\tilde{\mathbf{C}}(x)$, there are n^2 terms whose coefficients are $\mathbf{A}_i\mathbf{B}_j$, $1 \leq i, j \leq n$. As shown in Fig. 2b, the exponents of the four terms in $\tilde{\mathbf{C}}(x)$ ranges between 0 and 3. Hence, we need to have the results of any 4 tasks to obtain the results of $\mathbf{A}_1\mathbf{B}_1$ and $\mathbf{A}_2\mathbf{B}_2$, as the coefficients of x^0 and x^3. In other words, the recovery threshold in Fig. 2b is 4. We can see from Fig. 2b that the polynomial code can be seen as a special and non-optimal scheme that is feasible in our framework.

In Fig. 2c, we present another possible way to construct the coding scheme where $N = (2, 1)$. Hence, we can see that the exponents of $\mathbf{A}_1\mathbf{B}_1$ and $\mathbf{A}_2\mathbf{B}_2$ are placed in the counter diagonal, as highlighted in the table. We also highlight the entries of $\mathbf{A}_1\mathbf{B}_1$ and $\mathbf{A}_2\mathbf{B}_2$ in the other examples. As M and N can be any permutations of $\{1, \ldots, n\}$, the pattern of highlighted entries can be more flexible in our coding framework. However, there should be one and only one highlighted entry in each row or each column.[1]

Furthermore, we demonstrate an optimal coding scheme for $n = 2$ in Fig. 2d, which minimizes the number of exponents, and hence the recovery threshold. To prove its optimality, we consider the number of exponents needed in the table. The highlighted entries must have unique exponents, and the other two entries also need to have at least one more exponent. Hence, there need to be at least three exponents, proving the optimality of the coding scheme illustrated in Fig. 2d. This scheme also corresponds to the example we demonstrated in Sec. II. We can see that in a feasible coding scheme, the exponents in highlighted entries in the corresponding table must be unique, while the exponents in other entries can coincide, helping to achieve lower recovery thresholds.

IV. Constructing RC with Fixed P

A. Scopes of Parameters

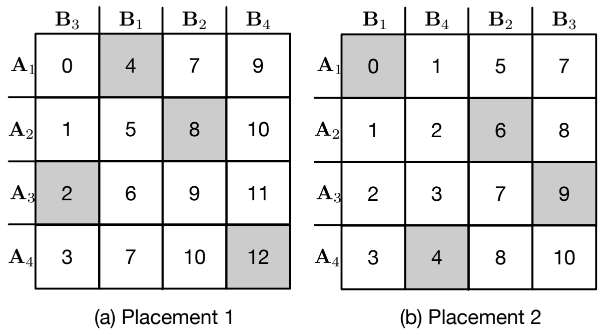
In this paper, we make multiple attempts to construct RC, with the recovery threshold improved step by step. We present the first attempt in this section. To make it convenient for the code construction, we first narrow down the scopes of the parameters. Without loss of generality, we assume that elements in P and Q are non-decreasing, *i.e.*, $P_1 \leq \ldots \leq P_n$ and $Q_1 \leq \ldots \leq Q_n$. In fact, to make the exponent of $\mathbf{A}_i\mathbf{B}_i$ unique, the elements in P and Q should be strictly increasing. Otherwise, if there exist two distinct integers j_1 and j_2 such that $Q_{j_1} = Q_{j_2}$, we have $P_i + Q_{j_1} = P_i + Q_{j_2}$ for any integer $i \in [1, \ldots, n]$. Considering i such that $M_i = N_{j_1}$ (as M and N are both permutations of $\{1, \ldots, n\}$), the exponent of $\mathbf{A}_{M_i}\mathbf{B}_{N_{j_1}}$ equals that of $\mathbf{A}_{M_i}\mathbf{B}_{N_{j_2}}$. In other words, $\mathbf{A}_{M_i}\mathbf{B}_{N_{j_1}}$ cannot be obtained after decoding, which is the result of one of the n matrix multiplications. Therefore, we have $P_1 < \ldots < P_n$ and $Q_1 < \ldots < Q_n$.

[1]The name of rook coding is inspired by this requirement, which is similar to placing non-attacking rooks in the chessboard. However, it is not equivalent to the rook polynomial, because we do not need to find the number of the placements of rooks in the construction, but to minimize the degree of $\tilde{\mathbf{C}}(x)$.

Without loss of generality, we can also assume that $P_0 = Q_0 = 0$, or we can easily get an equivalent coding scheme by subtracting P_0 (and Q_0) from all elements in P (and Q).

In this section, we consider a special case of P where $P = (0, \ldots, n-1)$.[2] We construct RC with this condition and prove the optimality of the construction in this special case. We extend the construction of RC to the general values of P in Sec. V.

Given a placement of highlighted entries in the table, there can be multiple possible choices of M and N that lead to the same placement. For example, in Fig. 3a, $M = (1, 2, 3, 4)$ and $N = (3, 1, 2, 4)$, where we place corresponding $\mathbf{A}_{M_i}$ and $\mathbf{B}_{N_j}$ as the title of each row and each column, respectively. If we switch $\mathbf{A}_{i_1}$ with $\mathbf{A}_{j_1}$ and meanwhile $\mathbf{B}_{i_2}$ with $\mathbf{B}_{j_2}$, if $i_1 = i_2$ and $j_1 = j_2$, the highlighted entries will remain unchanged. After such a switch, the new coded tasks will remain equivalent as the original coded tasks, only having entries in M and N switched. For example, the same entries will be highlighted if $M = (2, 1, 3, 4)$ and $N = (3, 2, 1, 4)$. Hence, we can assume, without loss of generality, that $M = (1, 2, 3, 4)$, so that the coding scheme will only depend on the value of N.



	$\mathbf{B}_3$	$\mathbf{B}_1$	$\mathbf{B}_2$	$\mathbf{B}_4$
$\mathbf{A}_1$	0	4	7	9
$\mathbf{A}_2$	1	5	8	10
$\mathbf{A}_3$	2	6	9	11
$\mathbf{A}_4$	3	7	10	12

(a) Placement 1

	$\mathbf{B}_1$	$\mathbf{B}_4$	$\mathbf{B}_2$	$\mathbf{B}_3$
$\mathbf{A}_1$	0	1	5	7
$\mathbf{A}_2$	1	2	6	8
$\mathbf{A}_3$	2	3	7	9
$\mathbf{A}_4$	3	4	8	10

(b) Placement 2

Fig. 3. Two placements of highlighted entries and their corresponding optimal coding schemes (with respect to their highlighted entries).

Now we have fixed the values in P and M. In the rest of this section, we will construct RC by finding the best values in Q and N that optimize the recovery threshold.

B. Achieving the Optimal Degree of $\tilde{\mathbf{C}}(x)$ with $P = (0, \ldots, n-1)$

Besides having P and M fixed, we first construct a coding scheme with the optimal degree of $\tilde{\mathbf{C}}(x)$ from a given placement of highlighted entries, *i.e.*, the values in M, N, and P are all fixed. In Alg. 1, we propose an algorithm that constructs such a coding scheme. The optimal coding scheme can then be found in two steps: 1) finding the optimal placement of highlighted entries; and 2) finding the values of Q that achieve the optimal degree in $\tilde{\mathbf{C}}(x)$. The second step can be solved by Alg. 1, with an $O(n)$ time complexity. We prove the optimality of Alg. 1 with Theorem 1, and leave the first step in Sec. IV-C.

Theorem 1. *Given highlighted entries,* i.e.*, $M = (1, \ldots, n)$, $P = (0, \ldots, n-1)$, and N as an arbitrary permutation of*

[2]It is equivalent to having $Q = (0, \ldots, n-1)$ and then choosing the optimal P. In this paper, we simply choose the value of P first and then optimize the value of Q.

Algorithm 1 The optimal values of Q with a given placement of highlighted entries.

```
Input: $N$ (with $M$ fixed, the placement of highlighted entries only depends on $N$
Output: $Q$
1: $Q_1 = 0$
2: for $i \leftarrow 2$ to $n$ do
3:     $Q_i \leftarrow Q_{i-1} + \max\{N_{i-1}, n - N_i + 1\}$
4: end for
```

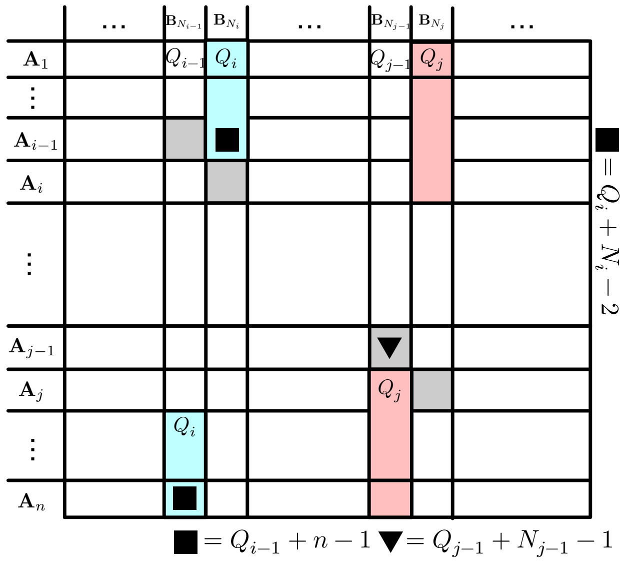


Fig. 4. Two placements of highlighted entries and their corresponding optimal coding schemes.

$\{1, \ldots, n\}$*, Alg. 1 finds the optimal values in* Q *that minimize the recovery threshold.*

Proof. The intuition of Alg. 1 is making the overlaps of exponents, *i.e.*, exponents shared by multiple entries in the $n \times n$ table as those in Fig. 3, as large as possible. Note that the overlaps may only occur between two neighboring columns since each column has a highlighted entry with a unique exponent. Given $P = (0, \ldots, n-1)$, we also know that each integer between 0 and $P_n + Q_n$ will be used as an exponent with Q given by Alg. 1, and thus the recovery threshold equals the degree of the polynomial plus one.

As shown in Fig. 3, within two neighboring columns, the overlapped exponents go up (down) from the bottom (top) entry until reaching a highlighted entry. For example, in Fig. 3a the two entries at the bottom in the third column share the same exponents with the top two entries in the last column. We can also find exponents 1, 2, and 3 shared among the first two columns in Fig. 3b, as well as 7 and 8. In the i-th column, the highlighted entry is in the N_i-th row since $M_{N_i} = N_i$. Hence, there are $N_i - 1$ entries above it and $n - N_i$ entries below it. To determine the value of Q_i, we consider if the number of entries above it is greater or less than the number of entries below the highlighted entry in the $(i-1)$-th column, and we illustrate such two cases in Fig. 4.

In order to make overlaps of exponents as large as possible, if the number of entries above the highlighted entry in the i-th column is less than that of entries below the highlighted entry in the $(i-1)$-th column, *i.e.*, $N_i - 1 < n - N_{i-1}$, there can be at most $N_i - 1$ entries with the same exponents as the entries at the bottom in the $(i-1)$-th column. Since the last exponent in the $(i-1)$-th column is $Q_{i-1} + n - 1$, the first exponent in the i-th column should be $Q_{i-1} + n - 1 - (N_i - 2) = Q_{i-1} + n - N_i + 1$, which also equals Q_i as $P_1 = 0$. On the other hand, if the number of entries below the highlighted entry in the $(j-1)$-th column is less than the number of entries above the highlighted entry in the j-th column, the first exponent in the j-th column should be at least greater than the exponent of the highlighted entry in the $(j-1)$-th column, which equals $Q_{j-1} + N_{j-1} - 1$. In other words, Q_j should be $Q_{j-1} + N_{j-1}$.

Since $N_{i-1} \geq n - N_i + 1$ if $n - N_{i-1} \leq N_i - 1$, we can further simplify the two cases above as $Q_i = Q_{i-1} + \max\{N_{i-1}, n - N_i + 1\}$, and the degree of $\tilde{\mathbf{C}}(x)$ is $\sum_{i=2}^{n} \max\{N_{i-1}, n - N_i + 1\} + n - 1$. □

We can see from Alg. 1 that all integers between 0 and the degree of $\tilde{\mathbf{C}}(x)$ appear at least once as the exponents in $\tilde{\mathbf{C}}(x)$, otherwise there must exist i such that Q_i can be reduced to use the missing integer as the exponent. Hence, the recovery threshold should also be $\sum_{i=2}^{n} \max\{N_{i-1}, n - N_i + 1\} + n$. We can also compute the recovery threshold as $P_n + Q_n + 1$ since values in both P and Q are both strictly increasing.

C. Optimal Placement of Highlighted Entries

As Alg. 1 minimizes the degree of $\tilde{\mathbf{C}}(x)$ given a placement of highlighted entries, we now discuss how to find the optimal placement of highlighted entries (when $P = (0, \ldots, n-1)$). Applying Alg. 1 to two different placements with $n = 4$ in Fig. 3a and Fig. 3b, we can see that different placements of highlighted entries can lead to different degrees of $\tilde{\mathbf{C}}(x)$.

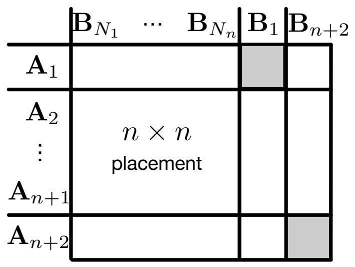


Fig. 5. The illustration of the algorithm to find the optimal placement of highlighted entries.

We now propose a placement of highlighted entries which can be proved to achieve the optimal degree in $\tilde{\mathbf{C}}(x)$. The placement can be obtained by induction. When $n = 1$, there is one and only one possible placement which is the only entry itself, as shown in Fig. 6a. When $n = 2$, Q has two permutations, leading to two patterns of highlighted entries which we can find in Fig. 2c and Fig. 2d. We can see that the placement in Fig. 2c does not have any overlapped exponent, and hence the placement in Fig. 2d is optimal.

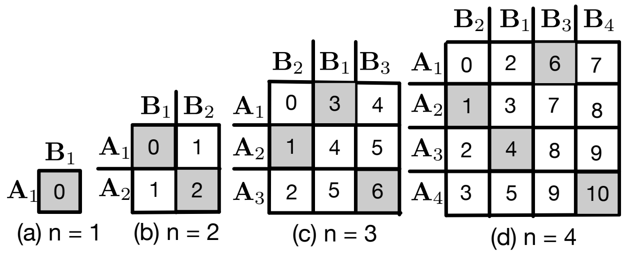


Fig. 6. Examples of the optimal placements of highlighted entries with $n = 1, 2, 3, 4$.

We now construct the placement with $n+2$ from a placement constructed from the $n \times n$ table. As shown in Fig. 5, we first construct a placement for the $n \times n$ table and place it between the second and the $(n+1)$-th row and between the first and the n-th column. We then highlight the top entry in the $(n+1)$-th column and the bottom entry in the $(n+2)$-th column. In Fig. 6c and Fig. 6d, we show the two placements with $n = 3$ and $n = 4$ constructed from the placement in Fig. 6a and Fig. 6b, respectively. With M fixed, N can also be determined after getting the optimal placement, and then we apply Alg. 1 to get the exponents in the table and hence obtain the value of Q. We summarize this recursive construction in Alg. 2. Although the complexity seems to be $O(n^2)$ with the recursion, it can be easily reduced to $O(n)$ by equivalently getting $N = (\dots, 2, n-1, 1, n)$. Therefore, the overall time complexity is still linear.

Algorithm 2 The optimal placement of highlighted entries with $P = (0, \dots, n-1)$.

```
Output: $N$
1: if $n = 1$ then
2:     $N = (1)$
3: else if $n = 2$ then
4:     $N = (1, 2)$
5: else
6:     construct $N$ for the placement with $n-2$, and denote it as $(N_1, N_2, \dots, N_{n-2})$
7:     $N = (N_1 + 1, N_2 + 1, \dots, N_{n-2} + 1, 1, n)$
8: end if
```

As a preparation to prove the optimality of the RC constructed by Alg. 1 and Alg. 2, we first analyze the properties of our code construction in Sec. IV-D. The proof of the optimality will be given in Sec. IV-E.

D. Analysis

In Sec. IV-B, we have demonstrated that the recovery threshold of RC constructed with Alg. 1 and Alg. 2 is one more than the degree of $\tilde{\mathbf{C}}(x)$. To analyze the degree of $\tilde{\mathbf{C}}(x)$, we first count the number of unhighlighted entries with unique exponents as $u(n)$. In particular, we use $u_0(n)$ to denote the value of $u(n)$ if the placement of highlighted entries is constructed by Alg. 2.

When $n = 1$ and $n = 2$, we can directly get $u_0(1) = u_0(2) = 0$ from Fig. 6a and Fig. 6b.

As for other values of n (when $n > 2$), we can get the value of $u_0(n)$ recursively. With the construction in Fig. 5, we can see that the right-bottom entry is always highlighted. In the two rightmost columns, all unhighlighted entries share the same exponents as those in the other row. For example, in Fig. 6d, the exponents of unhighlighted entries in the rightmost two columns are both 7, 8, and 9. Moreover, given an $(n+2) \times (n+2)$ table, in the n columns on the left, entries on the top row and the bottom row are always unhighlighted. Except the top entry in the first column and the bottom entry in the n-th column, the top entry in the i-th column can share the same exponent with the bottom entry in the $(i-1)$-th column, $i = 2, \dots, n$. Therefore, there are only two additional unhighlighted entries with unique exponents, *i.e.*, whose exponents are not shared by other exponents, *i.e.*, $u_0(n+2) = u_0(n) + 2$. For general values of n, we thus have

$$u_0(n) = \begin{cases} n-1 & n \text{ is odd;} \\ n-2 & n \text{ is even.} \end{cases}$$

Among a total of n^2 entries, there are n highlighted entries and $u_0(n)$ unhighlighted entries with unique exponents. Then the number of unhighlighted entries with shared exponents is $n^2 - n - u_0(n)$. As each exponent can be shared by at most two entries (since both P and Q are strictly increasing), the total degree of $\tilde{\mathbf{C}}(x)$ is $\frac{n^2 - n - u_0(n)}{2} + n + u_0(n) - 1 = \frac{n^2 + n + u_0(n)}{2} - 1$, which equals $\frac{n^2}{2} + n - \frac{3}{2}$ if n is odd or $\frac{n^2}{2} + n - 2$ if n is even. We use $r_0(n)$ to denote the recovery threshold of RC constructed with Alg. 1 and Alg. 2, and then $r_0(n)$ equals $\frac{n^2}{2} + n - \frac{1}{2}$ (or $\frac{n^2}{2} + n - 1$) if n is odd (or even). We can further simplify it as $r_0(n) = \left\lfloor \frac{(n+1)^2}{2} \right\rfloor - 1$.

E. Optimality

We now prove that the construction of RC given by Alg. 1 and Alg. 2 is optimal when $P = (0, \dots, n-1)$. Equivalently, we can prove that $u_0(n)$ is optimal, *i.e.*, $u_0(n) = u_{\text{opt}}(n)$.

We first prove a lemma which characterizes the unhighlighted entries with unique exponents.

Lemma 1. *In the $n \times n$ table ($n > 2$) under the coding framework of RC, the number of unhighlighted entries with unique exponents between two highlighted entries in two neighboring columns is no more than half of the total number of unhighlighted entries with unique exponents in the whole table when $P = (0, \dots, n-1)$.*

Proof. Assume that the index of the highlighted entry in the j-th column is N_j, $1 \le j \le n$. Additionally, we set $N_0 = n$ and $N_{n+1} = 1$. Then the number of unhighlighted entries with unique exponents is $|(n - N_j) - (N_{j+1} - 1)| = |(n+1) - (N_j + N_{j+1})|$. We then have $|(n+1) - (N_j + N_{j+1})|$ as the number of unhighlighted entries with unique exponents in the j-th column, then the total number of unhighlighted entries with unique exponents is $\sum_{j=0}^{n} |(n+1) - (N_j + N_{j+1})|$, and we aim to prove that $\forall j = 0, \dots, n$, $|(n+1) - (N_j + N_{j+1})| \le \frac{1}{2} \sum_{j=0}^{n} |(n+1) - (N_j + N_{j+1})|$.

We know that $\sum_{j=1}^{n} N_j = \frac{(n+1)n}{2}$, and then we have $\sum_{i=0}^{n} (N_j + N_{j+1}) = (n+1)^2$. Hence,

$\sum_{j=0}^{n}((n+1)-(N_j+N_{j+1})) = 0$. Hence, we can divide the $n+1$ terms above into three parts: $J_1 = \{j|(n+1)-(N_j+N_{j+1}) > 0\}$, $J_2 = \{j|(n+1)-(N_j+N_{j+1}) < 0\}$, and $J_3 = \{j|(n+1)-(N_j+N_{j+1}) = 0\}$. If $j \in J_1$ or $j \in J_2$, $|(n+1)-(N_j+N_{j+1})| \leq \sum_{j\in J_1}|(n+1)-(N_j+N_{j+1})| = \sum_{j\in J_2}|(n+1)-(N_j+N_{j+1})| = \frac{1}{2}\sum_{j=0}^{n}|(n+1)-(N_j+N_{j+1})|$. If $j \in J_3$, then $|(n+1)-(N_j+N_{j+1})| = 0$ which is also no more than $\frac{1}{2}\sum_{j=0}^{n}|(n+1)-(N_j+N_{j+1})|$. $\square$

Next, we consider $u(n+2)$ in an $(n+2)\times(n+2)$ table, with an arbitrary and legal placement of highlighted entries. Assuming $N_{j_1} = 1$ and $N_{j_2} = n+2$, we can remove the two rows (the first and the last row) and the two columns (the j_1-th and j_2-th columns) associated with the two highlighted entries, then we can get an $n\times n$ table with a legal placement of highlighted entries.

Lemma 2. $u(n+2) \geq u_{opt}(n)+2$.

Proof. Given the j_1-th column and the j_2-th column, we consider two cases by checking if the two columns are neighbors, *i.e.*, if $|j_1-j_2| = 1$.

Case 1: We first assume that such two columns are not neighbors, *i.e.*, $|j_1-j_2| > 1$. In this case, we can consider these two columns individually, *i.e.*, their exponents will not coincide with each other. Therefore, we only need to prove a less difficult statement: after removing the j_1-th column or the j_2-th column, $u(n+2) \geq u_{\text{opt}}(n+1)+1$. If it is proved, we can immediately have $u(n+2) \geq u_{\text{opt}}(n+1)+2$.

We first consider the case of removing the $(j-1)$-th column. If $1 < j_1 < n+2$, before removing the two rows and two columns, between the highlighted entries in the (j_1-1)-th column and the (j_1+1)-th column, the number of unhighlighted entries with unique exponents is $(n+2-N_{j_1-1})+(n+1-N_{j_1+1}+1) = 2(n+2)-(N_{j_1-1}+N_{j_1+1})$. After the removal, the number of unhighlighted entries with unique exponents between the same two entries is $|(n+1-N_{j_1-1})-(N_{j_1+1}-2)| = |(n+3)-(N_{j_1-1}+N_{j_1+1})|$. By Lemma 1, $|(n+3)-(N_{j_1-1}+N_{j_1+1})| \leq \frac{u_0(n)}{2} = \frac{n-1}{2}$, otherwise the placement in the $n\times n$ table after the removal is not optimal. If so, we can directly prove $|(n+3)-(N_{j_1-1}+N_{j_1+1})| < 2(n+2)-(N_{j_1-1}+N_{j_1+1})$.

If $j_1 = 1$, we only need to count the number of unhighlighted entries with unique exponents between the first entry in the first column and the highlighted entry in the second column. Before the removal, the number is $(n+1)-(N_2-1) = n+2-N_2$. After the removal, the number becomes N_2-1, which is no more than $\frac{n-1}{2}$. Hence, $N_2-1 < n+2-N_2$.

If $j_1 = n+2$, then after removing the number of unhighlighted entries with unique exponents will be reduced, as all the unhighlighted entries in the j_1-th column are not be shared by any other entries anyway.

The case of removing the j_2-th column can be proved using the same technique above. Combining these two cases, we can prove that at least two unlighted entries with unique exponents are removed.

Case 2: If $|j_1-j_2| = 1$, then there can be two possibilities of their positions. If $j_1 > j_2$, after the removal, there will be at least two unhighlighted entries with unique exponents removed, *i.e.*, the entry in the last second row of the j_1-th column, and the entry in the second row of the j_2-th column.

If $j_1 < j_2$, all entries in these two columns will not coincide with any other entries in the table. If the two columns are on the left or on the right of the table, then one entry in the first row and one entry in the last row will also be removed that have unique exponents. For example, if the two columns are on the left, then the top entry in the third column and the bottom entry in the last column are also removed and originally have unique exponents. We can also find these two entries if the two columns are on the right. If these two columns are in the middle, the top entry in the first column and the bottom entry in the last column will be removed, whose exponents are unique.

Combining all the statements above, we have $u(n+2) \geq u_{\text{opt}}(n)+2$. $\square$

With Lemma 2, we can prove the optimality of Alg. 2.

Theorem 2. *When $P = (0,\ldots,n-1)$, Alg. 2 finds the best values in N that lead to the optimal placement of highlighted entries.*

Proof. By Lemma 2, we immediately have $u_{\text{opt}}(n+2) \geq u_{\text{opt}}(n)+2$. We have known that $u_0(1) = u_{\text{opt}}(1)$ and $u_0(2) = u_{\text{opt}}(2)$. We also know that $u_0(n+2) = u_0(n)+2$, and by induction we can prove that $u_0(n) = u_{\text{opt}}(n)$. As the number of highlighted entries is always n, minimizing the number of unhighlighted entries with unique exponents is equivalent to maximizing the number of unhighlighted entries with shared exponents, and hence minimizing the degree of $\tilde{\mathbf{C}}(x)$. $\square$

Note that the optimality is achieved under the assumption of $P = (0,\ldots,n-1)$. In Sec. V, we generalize the construction by waiving this requirement for P, which counter-intuitively reduces the recovery threshold.

V. Constructing RC with General P

In this section, we make a second attempt to construct RC, where values in P can be chosen arbitrarily. Although minimizing P to be $(0,\ldots,n-1)$ seems to minimize the degree of $\tilde{\mathbf{C}}(x)$, we find that the degree of $\tilde{\mathbf{C}}(x)$ can be even lower with general values in P as it creates more chances to share exponents.

A. A Toy Example

We show a toy example of the construction in Fig. 7a. Compared to the construction in Fig. 6d, the degree of $\tilde{\mathbf{C}}(x)$ is reduced from 10 to 8 by setting $P = (0,1,3,4)$. The reason for this better result is that the construction in Fig. 7a allows more overlaps among exponents. In the construction in Sec. IV, an exponent can only overlap with another one. In Fig. 7a, we can see that the exponent 4 appears 4 times.

The increase in overlaps in Fig. 7a is due to a recursive construction. In this example, we group the four matrices $\mathbf{A}_1,\ldots,\mathbf{A}_4$ into two groups $\{\mathbf{A}_1,\mathbf{A}_2\}$ and $\{\mathbf{A}_3,\mathbf{A}_4\}$, and

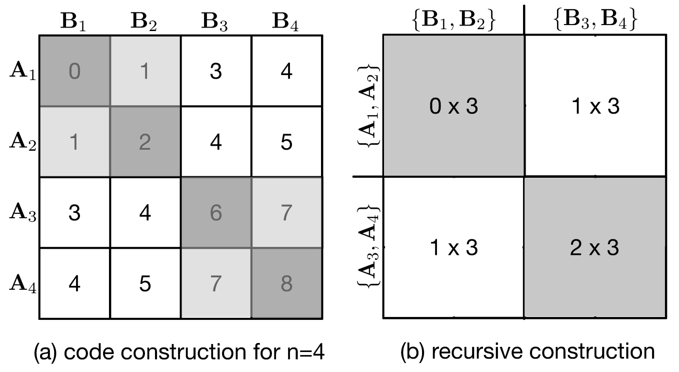

$\mathbf{B}_1$ $\mathbf{B}_2$ $\mathbf{B}_3$ $\mathbf{B}_4$
$\mathbf{A}_1$ 0 1 3 4
$\mathbf{A}_2$ 1 2 4 5
$\mathbf{A}_3$ 3 4 6 7
$\mathbf{A}_4$ 4 5 7 8
(a) code construction for n=4
$\{\mathbf{B}_1, \mathbf{B}_2\}$ $\{\mathbf{B}_3, \mathbf{B}_4\}$
$\{\mathbf{A}_1, \mathbf{A}_2\}$ 0 x 3 1 x 3
$\{\mathbf{A}_3, \mathbf{A}_4\}$ 1 x 3 2 x 3
(b) recursive construction

Fig. 7. A toy example of the construction of RC with general values in P.

also $\mathbf{B}_1, \ldots, \mathbf{B}_4$ into $\{\mathbf{B}_1, \mathbf{B}_2\}$ and $\{\mathbf{B}_3, \mathbf{B}_4\}$. We then construct the code in two steps. We first place the four groups in a 2×2 table, as shown in Fig. 7b. Each entry in such a table corresponds to two matrix multiplications. It is easy to see that the entry between $\{\mathbf{A}_1, \mathbf{A}_2\}$ and $\{\mathbf{B}_1, \mathbf{B}_2\}$ and that between $\{\mathbf{A}_3, \mathbf{A}_4\}$ and $\{\mathbf{B}_3, \mathbf{B}_4\}$ need to have unique exponents, and the other two entries can have shared exponents. Hence, we apply RC for such four entries with $n = 2$, as shown in Fig. 7b, with only one difference that the exponents need to be amplified for the next step.

In the second step, we construct RC again for each (outer) entry, with $n = 2$. Note that in each outer entry, there are 4 inner entries with 3 exponents. Hence, the exponents in the first step need to be multiplied by 3. Eventually, the exponents in each (inner) entry in Fig. 7a should be the sum of the corresponding exponents of the two steps, such that the exponents of $\mathbf{A}_i\mathbf{B}_i$, $i = 1, \ldots, 4$ can be both unique in each inner and outer entry. In this way, we can get $P = Q = (0, 1, 3, 4)$.

B. Code Construction

From Fig. 7, we can see that the construction is recursive. We assume that given n matrix multiplications, n is a composite number, *i.e.*, $n = pq$ where $n, p, q \in \mathbb{Z}^+$. For now, we assume both p and q are prime numbers. We then group $\mathbf{A}_1, \ldots, \mathbf{A}_n$ into p groups, where each group contains q matrices, *i.e.*, $\{\mathbf{A}_1, \ldots, \mathbf{A}_q\}$, …, $\{\mathbf{A}_{n-q+1}, \ldots, \mathbf{A}_n\}$. We can also do the same to $\mathbf{B}_1, \ldots, \mathbf{B}_n$. To avoid ambiguity, we rewrite P, Q, N as $P(n)$, $Q(n)$, $N(n)$ for RC constructed with n matrix multiplications.

We first construct an RC for $n = p$, for the outer entries between two groups of $\mathbf{A}_i$ and $\mathbf{B}_i$. In Fig. 8 we demonstrate the recursive construction with $p = 3$. We can see that the $p = 3$ groups of $\mathbf{A}_i$s and $\mathbf{B}_i$s are placed along the rows and columns of the 3×3 table, corresponding to the RC for $n = p = 3$. The only difference is that the exponents are multiplied by $r_0(q)$, the recovery threshold of RC for $n = q$, as there will be $r_0(q)$ exponents to be placed in each outer entry in the second step.

In the second step, we construct an RC for $n = q$ for each outer entry in the first step, and then we can determine the placement of $\mathbf{A}_i$s and $\mathbf{B}_i$s in each group. Note that all outer entries actually have the same placement as they are all RCs for $n = q$, so all outer entries in the same rows (columns) have the same placements of $\mathbf{A}_i$s ($\mathbf{B}_i$s). In particular, each inner entry of such RCs needs to be added with the exponents of its corresponding outer entry in the first step. In other words, each inner entry should be added with a value that equals $r_0(q)$ times the corresponding exponent in the first step, such that unique entries in each outer entry do not coincide with any unique entries in other outer entries.

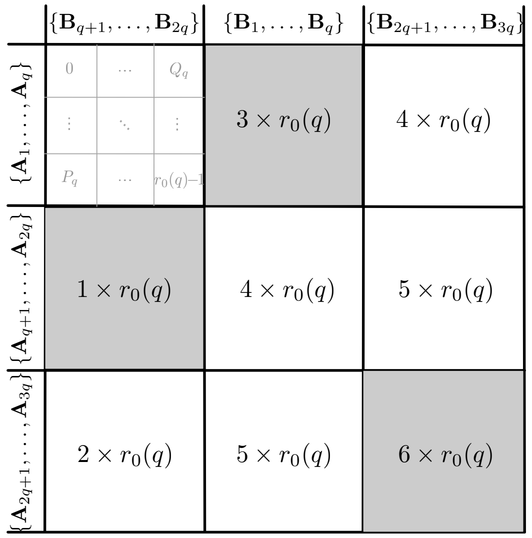

$\{\mathbf{B}_{q+1}, \ldots, \mathbf{B}_{2q}\}$ $\{\mathbf{B}_1, \ldots, \mathbf{B}_q\}$ $\{\mathbf{B}_{2q+1}, \ldots, \mathbf{B}_{3q}\}$
$\{\mathbf{A}_1, \ldots, \mathbf{A}_q\}$
0 $\cdots$ Q_q
$\vdots$ $\ddots$ $\vdots$
P_q $\cdots$ $r_0(q)-1$
$3 \times r_0(q)$
$4 \times r_0(q)$
$\{\mathbf{A}_{q+1}, \ldots, \mathbf{A}_{2q}\}$
$1 \times r_0(q)$
$4 \times r_0(q)$
$5 \times r_0(q)$
$\{\mathbf{A}_{2q+1}, \ldots, \mathbf{A}_{3q}\}$
$2 \times r_0(q)$
$5 \times r_0(q)$
$6 \times r_0(q)$

Fig. 8. An illustration of the construction of RC with $p = 3$. In this example q is also assumed to be prime.

Given the construction above, we can now formalize the algorithms to obtain $P(n)$, $Q(n)$, $N(n)$, $M(n)$ for $n = pq$. Given entry (i, j) which indicates the inner entry at the i-th row and the j-th column, by definition its exponent should be $P_i(n) + Q_j(n)$. For convenience, we choose i_1, i_2 such that $i = (i_1 - 1)q + i_2$ where $1 \le i_1 \le p$ and $1 \le i_2 \le q$, and similarly $j = (j_1 - 1)q + j_2$ where $1 \le j_1 \le p$ and $1 \le j_2 \le q$. In this way, (i_1, j_1) indicates the corresponding outer entry in the first step and (i_2, j_2) indicates corresponding inner entry in this outer entry. Therefore, we have $P_i(n) + Q_j(n) = (P_{i_1}(p) + Q_{j_1}(p)) \cdot r_0(q) + (P_{i_2}(q) + Q_{j_2}(q))$.

Since $P_0(n) = Q_0(n) = 0$, we have $P_i(n) = P_{i_1}(p) r_0(q) + P_{i_2}(q)$ and $Q_j(n) = Q_{j_1}(p) r_0(q) + Q_{j_2}(q)$.

Now we consider the placement of matrices, *i.e.*, $N(n)$. At the j-th column in the $n \times n$ table, it belongs to the j_1-th column of the p outer entries and j_2-th column in this outer entry. The group corresponding to this outer entry then should be $\{\mathbf{B}_{(N_{j_1}(p)-1)q+1}, \ldots, \mathbf{B}_{(N_{j_1}-1)(p)q+q}\}$. Hence, the matrix at the i-th column should be $N_{(N_{j_1}(p)-1)q+1+N_{j_2}(q)-1}$. In other words, $N_j(n) = (N_{j_1}(p) - 1)q + N_{j_2}(q)$.

Moreover, if p or q is still a composite number, we can recursively use the construction above until they are both prime numbers. We summarize the general code construction of RC in Alg. 3, where we use $r_1(n)$ to denote the recovery threshold of RC constructed in Alg. 3, and then we can replace $r_0(q)$ with $r_1(q)$ in the second step. We analyze the recovery threshold $r_1(n)$ in Sec. V-C (which is used in Line 11 of Alg. 3).

The complexity of Alg. 3 is $O(n)$. If n is a prime number, then the construction is the same as the special case in Sec. IV. Otherwise, we need to construct RC for $n = p$ and $n = q$ with $O(p)$ and $O(q)$ time complexities. Moreover, the complexity of code between Line 6 and Line 12 is $O(n)$. Therefore, the overall complexity when n is not a prime number is still $O(n)$.

Algorithm 3 The (second) construction of RC with general values in P.

```
Input: $n$
Output: $P(n), Q(n), N(n), r_1(n)$
 1: if $n$ is a prime number then
 2:     Obtain $P(n), Q(n), N(n)$ by Alg. 1 and Alg. 2
 3: else
 4:     Obtain $P(p), Q(p), N(p)$ from RC with $n = p$
 5:     Obtain $P(q), Q(q), N(q), r_1(q)$ from RC with $n = q$
 6:     for $i \leftarrow 1$ to $n$ do
 7:         Let $i = (i_1 - 1)q + i_2$ where $1 \leq i_1 \leq p$ and $1 \leq i_2 \leq q$
 8:         $P_i(n) = P_{i_1}(p)r_1(q) + P_{i_2}(q)$
 9:         $Q_i(n) = Q_{i_1}(p)r_1(q) + Q_{i_2}(q)$
10:         $N_i(n) = (N_{i_1}(p) - 1)q + N_{i_2}(q)$
11:         $r_1(n) = r_1(p)r_1(q)$
12:     end for
13: end if
```

C. Analysis

From Fig. 8, we can see that if the exponents in the RC for $n = q$ is consecutive, then the exponents for RC for $n = pq$, given by the construction in Alg. 3, will also be consecutive. This is because each outer entry in the first step separates from each other by $r_1(q)$ which is the number of exponents in each outer entry. As the number of exponents in each outer entry is $r_1(q)$, and the exponents in different outer entries do not overlap, the total number of exponents is $r_1(p)r_1(q)$. Since we will also recursively construct RC for $n = p$ or $n = q$ if p or q is still a composite number, we can also recursively get the recovery threshold of $r_1(n)$. If n can be factorized as $n = \prod_i p_i^{\alpha_i}$ where p_is are prime factors of n, then the recovery threshold of the construction $r_1(n)$ is $\prod_i r_0(p_i)^{\alpha_i}$.

We now analyze the recovery threshold by discussing some representative special cases. Obviously, if n is a prime number, then $r_1(n) = r_0(n) = \left\lfloor \frac{(n+1)^2}{2} \right\rfloor - 1$.

When n is not a prime number, the recovery threshold becomes

$$r_1(n) = \prod_i r_0(p_i)^{\alpha_i} = \prod_i O\left(\frac{p_i^2}{2}\right)^{\alpha_i}$$
$$= O\left(\frac{(\prod_i p_i^{\alpha_i})^2}{2^{\sum_i \alpha_i}}\right) = O\left(\frac{n^2}{2^{\sum_i \alpha_i}}\right).$$

From the equation above we can see that the recovery threshold can be minimized when $\sum_i \alpha_i$ is maximized, *i.e.*, when n is a power of 2. Specifically, if $n = 2^\alpha$, $r_1(n) = r_0(2)^\alpha = 3^{log_2 n} = O(n^{\log_2 3}) \approx O(n^{1.585})$.

Fig. 9 shows how $r_1(n)$ grows with n, when $2 \leq n \leq 128$. In order to make it easy to compare, we show $\log_n r_1(n)$ in

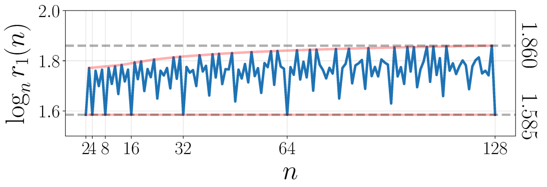


Fig. 9. The growth of the recovery threshold $r_1(n)$.

the Y-axis. We can see that $r_1(n)$ fluctuate between $O(n^{1.585})$ (when n is a power of 2) and $O(n^{1.860})$, meaning that for most values of n, their recovery threshold will be higher than $O(n^{\log_2 3})$. It is obvious that the lower bound of $\log_n r_1(n)$ is 1.585, as it is always reached when n is a power of 2. However, the upper bound keeps going up with n, and 1.860 is the reached when $n = 127$. Even worse, as $r_0(n) = O(n^2)$, we expect that $\log_n r_1(n) \rightarrow 2$ when $n \rightarrow \infty$ if n is a prime. Therefore, when n goes to infinity, the upper bound of $\log_n r_1(n)$ will go up to 2 eventually, and the gap between the upper and lower bound will be larger and larger. In the next section, we will further extend the construction to achieve an $O(n^{\log_2 3})$ recovery threshold for all valid values of n.

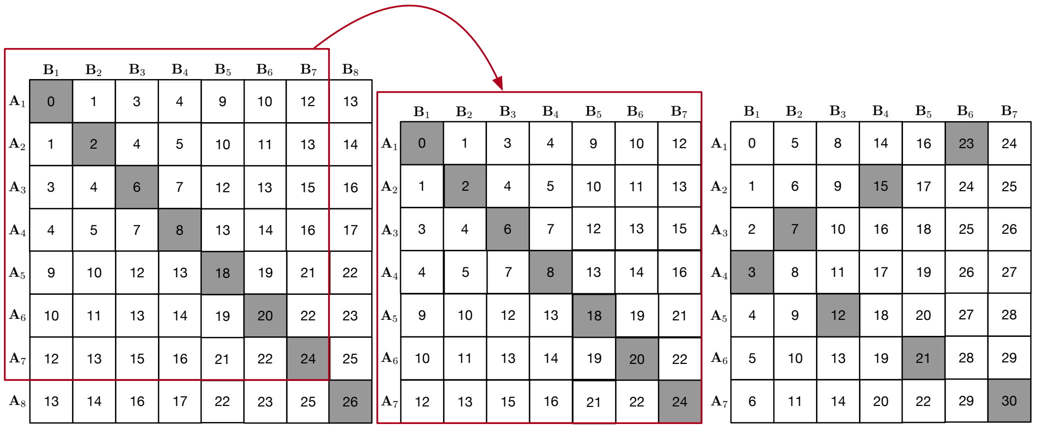
VI. Achieving an $O(n^{\log_2 3})$ Recovery Threshold for All n

A. Intuition and Construction

To achieve an $O(n^{\log_2 3})$ recovery threshold, we make a third attempt to construct RC, which is based on the second attempt in Sec. V. Hence, we first present an example of the RC given by this construction. In Fig. 10a, we present the RC constructed by Alg. 3 with $n = 8$, where $M = N = (1, 2, \ldots, 8)$, $P = Q = (0, 1, 3, 4, 9, 10, 12, 13)$. By taking the first seven entries from all the four parameters, we can construct RC with $n = 7$, such that $M = N = (1, 2, \ldots, 7)$, $P = Q = (0, 1, 3, 4, 10, 12)$. As shown in Fig. 10b, the degree of the corresponding polynomial is 25, which is even smaller than that of RC with $n = 7$ constructed by Alg. 3 (shown in Fig. 10c).

By Alg. 3, when $n = 2$ the values of M and N are sequential. Since we construct RC recursively when n is a power of 2, *i.e.*, $n = 2^i$ where i is a positive integer, entries in M and N will still be sequential. Therefore, if $n = 2^i$, we can arbitrarily take the first n_0 entries ($n_0 \leq n$) in M, N, P, and Q, to construct the RC for n_0 matrix multiplications.

Following this intuition, we give the code construction in Alg. 4. Since $2^i < 2n$, the complexity of Line 2 is $O(2n) = O(n)$. Since we then need to take a subset of $P(2^i)$, $Q(2^i)$, and $N(w^i)$, the overall complexity is also $O(n)$.



	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
A_1	0	1	3	4	9	10	12	13
A_2	1	2	4	5	10	11	13	14
A_3	3	4	6	7	12	13	15	16
A_4	4	5	7	8	13	14	16	17
A_5	9	10	12	13	18	19	21	22
A_6	10	11	13	14	19	20	22	23
A_7	12	13	15	16	21	22	24	25
A_8	13	14	16	17	22	23	25	26

(a) RC with n = 8 constructed by Alg. 3

	B_1	B_2	B_3	B_4	B_5	B_6	B_7
A_1	0	1	3	4	9	10	12
A_2	1	2	4	5	10	11	13
A_3	3	4	6	7	12	13	15
A_4	4	5	7	8	13	14	16
A_5	9	10	12	13	18	19	21
A_6	10	11	13	14	19	20	22
A_7	12	13	15	16	21	22	24

(b) RC with n = 7 constructed from (a)

	B_1	B_2	B_3	B_4	B_5	B_6	B_7
A_1	0	5	8	14	16	23	24
A_2	1	6	9	15	17	24	25
A_3	2	7	10	16	18	25	26
A_4	3	8	11	17	19	26	27
A_5	4	9	12	18	20	27	28
A_6	5	10	13	19	21	28	29
A_7	6	11	14	20	22	29	30

(c) RC with n = 7 constructed by Alg. 3

Fig. 10. Examples of RC constructed in the second and the third attempts.

Algorithm 4 The (third) construction of RC.

```
Input: $n$
Output: $P(n), Q(n), N(n)$
1: Find the smallest positive integer such that $2^{i-1} < n \le 2^i$.
2: Obtain $P(2^i), Q(2^i), N(2^i)$ from the RC constructed by Alg. 3
3: for $i \leftarrow 1$ to $n$ do
4:     $P_i(n) = P_i(2^i)$
5:     $Q_i(n) = Q_i(2^i)$
6:     $N_i(n) = N_i(2^i)$
7: end for
```

B. Analysis

Since from the construction of RC with $n = 2^i$ we can get the construction of RC for any $n < 2^i$, we only need to analyze the exponents of highlighted entries in the construction when $n = 2^i$. Let $r_2(n)$ denote the recovery threshold for RC constructed by Alg. 4. In particular, we do not consider if exponents in the corresponding polynomial are consecutive for now (which will be discussed in Sec. VI-C), *i.e.*, the coefficient of some term in the polynomial after decoding may be a zero matrix.

We analyze $r_2(n)$ recursively. If $n = 2^i$, we have $r_2(n) = r_1(n) = 3^{\log_2 n} = 3^i$, since the two constructions are the same in this case. Therefore, we have $r_2(1) = r_1(1) = 1$.

If $n > 1$, there exists $i \in \mathbb{Z}^+$ such that $2^{i-1} < n \le 2^i$. Based on Alg. 4, we will first construct RC with 2^i matrix multiplications from Alg. 3, and then the recovery threshold $r_2(n) = P_n(2^i) + Q_n(2^i) + 1$.

In Alg. 3, the RC with 2^i matrix multiplications should be recursively constructed from the RC with 2^{i-1} matrix multiplications. As $n = 2^{i-1} + (n - 2^{i-1})$, $P_n(2^i) = P_2(2)r_0(2^{i-1}) + P_{n-2^{i-1}}(2^{i-1})$ and $Q_n(2^i) = Q_2(2)r_0(2^{i-1}) + Q_{n-2^{i-1}}(2^{i-1})$. Therefore, $r_2(n) = r_0(2^{i-1})(P_2(2)+Q_2(2))+P_{n-2^{i-1}}(2^{i-1})+Q_{n-2^{i-1}}(2^{i-1})+1 = 2 \cdot r_2(2^{i-1}) + R_2(n - 2^{i-1}) = 2 \cdot 3^{i-1} + r_2(n - 2^{i-1})$.

Hence, we get Theorem 3 from the analysis above.

Theorem 3. *By the construction in Alg. 4, $r_2(n)$ satisfies the following equation:*

$$r_2(n) = \begin{cases} 1 & n = 1; \\ r_2(n - 2^{i-1}) + 2 \cdot 3^{i-1} & 2^{i-1} < n \le 2^i, i \in \mathbb{Z}^+. \end{cases}$$

From Theorem 3, we can also verify that when $n = 2^i$, $R_2(2^i) = r_2(2^{i-1}) + 2 \cdot 3^{i-1} = r_2(2^{i-2}) + 2 \cdot 3^{i-2} = \cdots = 1 + \sum_{j=0}^{i-1} 2 \cdot 3^j = 3^i = r_1(n)$. We analyze the scale of $r_2(n)$ in Theorem 4 below.

Theorem 4. $r_2(n) = O(n^{\log_2 3})$.

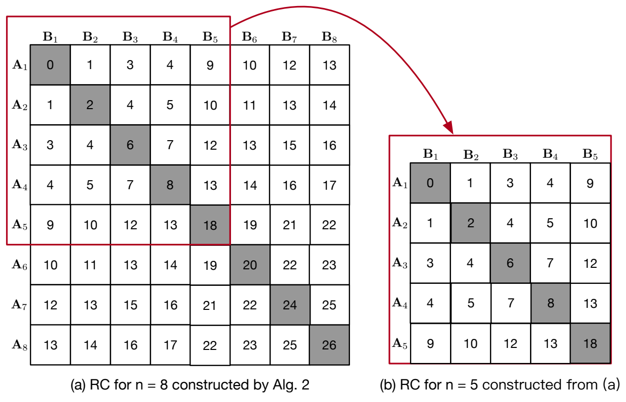
Proof. To prove this theorem, we first prove that $r_2(n)$ strictly increases with n. We prove it by proving that given $i \in \mathbb{Z}^+$, if the theorem is true for all $n \le 2^{i-1}$, it is also true for all $n \le 2^i$. Obviously it is true when $i = 1$, and we only need to consider the cases when $2^{i-1} < n \le 2^i$. If $2^{i-1} < n < 2^i$, we have $r_2(n - 2^{i-1}) < r_2(n + 1 - 2^{i-1})$. Therefore, by Theorem 3 we also have $r_2(n) < r_2(n + 1)$. If $n = 2^i$, $r_2(n+1) = r_2(2^i+1) = r_2(2^i+1-2^i)+2\cdot 3^i = r_2(1)+2\cdot 3^i > 3^i = r_2(2^i) = r_2(n)$.

As we only need to consider the cases where n is not a power of 2, we assume that i is the smallest integer such that $n < n' = 2^i$, $i \in \mathbb{Z}^+$, we have $n' < 2n$. Therefore, $r_2(n) < r_2(n') = r_1(n') = 3^{\log_2 n'} = (n')^{log_2 3} < (2n)^{\log_2 3} = 3n^{\log_2 3}$, *i.e.*, $r_2(n) = O(n^{\log_2 3})$. $\square$

Therefore, we eventually find a construction of RC with a linear complexity, and it achieves an $O(n^{\log_2 3})$ recovery threshold for all valid values of n.

C. Further Improvement

In fact, we may further improve the recovery threshold of RC constructed by Alg. 4 for many values of n, when exponents in $\tilde{\mathbf{C}}(x)$ are not consecutive. For example, when $n = 5$, we can construct RC with $r_2(5) = 19$, as shown in Fig. 11. However, since exponents 11, 14, 15, 16, and 17 are missing, we only need to have results of 14 tasks before decoding, *i.e.*, its actual recovery threshold is further reduced



(a) RC for n = 8 constructed by Alg. 2

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
A_1	0	1	3	4	9	10	12	13
A_2	1	2	4	5	10	11	13	14
A_3	3	4	6	7	12	13	15	16
A_4	4	5	7	8	13	14	16	17
A_5	9	10	12	13	18	19	21	22
A_6	10	11	13	14	19	20	22	23
A_7	12	13	15	16	21	22	24	25
A_8	13	14	16	17	22	23	25	26

(b) RC for n = 5 constructed from (a)

	B_1	B_2	B_3	B_4	B_5
A_1	0	1	3	4	9
A_2	1	2	4	5	10
A_3	3	4	6	7	12
A_4	4	5	7	8	13
A_5	9	10	12	13	18

Fig. 11. An example of non-consecutive exponents in $\tilde{\mathbf{C}}(x)$

from 19 to 14. Similarly, in Fig. 10b, we can also see that the exponents 17 and 23 are missing, further reducing the recovery threshold by 2. We use $r_3(n)$ to denote the recovery threshold after this improvement.

Lemma 3. *When $n = 2^i, i \in \mathbb{Z}$, exponents constructed by Alg. 4 are consecutive. Therefore, $r_3(n) = r_2(n)$.*

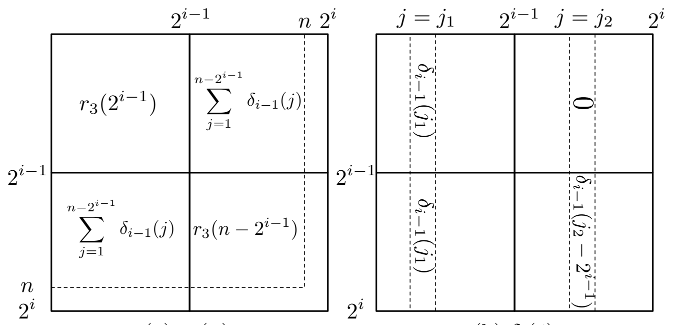
Proof. If $n = 1$, the statement can be trivially proved.

We now prove that if the statement is true for $n = 2^{i-1}$, it is also true for $n = 2^i$, $i \in \mathbb{Z}^+$. From Alg. 4, we know that the construction for $n = 2^i$ relies on the construction for $n = 2^{i-1}$, such that $p = 2^{i-1}$ and $q = 2$. Therefore, similar to Fig. 8, we can divide the table of exponents into 4 outer entries, each of which contains $2^{i-1} \times 2^{i-1}$ inner entries. From the assumption we know that exponents in each outer entry are consecutive. As the exponents of $n = 2$ are also consecutive, we can know that all exponents are consecutive. □

To know $r_3(n)$ in other cases, we need to count the number of exponents instead of the largest exponents, as some exponents may not appear in $\tilde{\mathbf{C}}(x)$. In Fig. 12a, we count the number of exponents in four parts. From the construction, we know that when $n = 2^i$, the upper right quarter is the same as the lower left quarter, and exponents in the three quarters are different and consecutive. Therefore, when $n \in (2^{i-1}, 2^i)$, $r_3(n)$ comes from three parts: $r_3(2^{i-1})$ from the upper left quarter, $r_3(n-2^i)$ from the lower right quarter, and $\sum_{j=1}^{n-2^{i-1}} \delta_{i-1}(j)$ from the upper right or the lower left quarter. In particular, $\delta_{i-1}(j)$ denotes the number of new exponents that do not appear in all the columns on the left of the j-th column, in the $2^{i-1} \times 2^{i-1}$ table, $j \in [1, 2^{i-1}]$, $i \geq 1$.

Obviously, if $j = 1$, all exponents are new exponents, and thus $\delta_i(1) = 2^i$. In other cases, we can see its recursion in Fig. 12b. By adding the j-th column, we can see that new exponents may come from two parts. If $j = j_1 < 2^{i-1}$, the two parts are the upper and lower left quarters. As exponents in these two quarters have no overlap, they can provide the same number of new exponents, *i.e.*, $\delta_{i-1}(j_1)$. If $j = j_2 > 2^{i-1}$, the two parts are the upper and lower right quarters. Note that the upper right quarter has the same exponents as the lower left quarter, so it offers no new exponents and the number of new exponents equals $\delta_{i-1}(j - 2^{i-1})$.

Summarizing the analysis above, we can get Theorem 5.



(a) $r_3(n)$ (b) $\delta_i(j)$

Fig. 12. Illustrations of the recursions of $r_3(n)$ and $\delta_i(j)$.

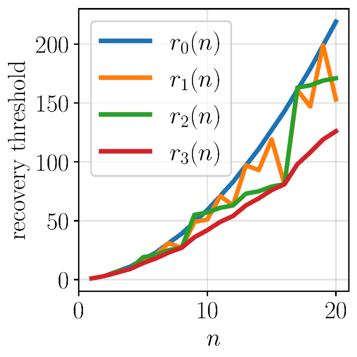


Fig. 13. The improvements of recovery thresholds.

Theorem 5. *When $n \in (2^{i-1}, 2^i), i \in \mathbb{Z}^+$, $r_3(n) = 3^{i-1} + r_3(n - 2^{i-1}) + \sum_{j=1}^{n-2^{i-1}} \delta_{i-1}(j)$, where*

$$\delta_i(j) = \begin{cases} 2^i & j = 1; \\ 2\delta_{i-1}(j) & j \in (1, 2^{i-1}]; \\ \delta_{i-1}(j - 2^{i-1}) & j \in (2^{i-1}, 2^i]. \end{cases}$$

As $r_3(2^i) = r_2(2^i)$, $i \in \mathbb{Z}^+$, the order of the recovery threshold will not change after removing missing exponents. However, the recovery threshold can still be saved significantly in many cases. As shown in Fig. 13, $r_3(n)$ can be saved by up to 60.1% lower than $r_2(n)$.

VII. Evaluation

We implement the code construction of RC with mpi4py, *i.e.*, MPI for python, which is a python packaging providing python bindings for MPI. The job of batch matrix multiplication will run in a master-worker architecture. We assume that the input matrices of the n matrix multiplications have been placed on each worker, and each worker will encode such input matrices into its own task. As the encoding is polynomial evaluations of $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ in RC, we simply use Horner's method to encode input matrices after the code is constructed. For example, we can evaluate $\tilde{\mathbf{A}}(x)$ as $\tilde{\mathbf{A}}(x) = \sum_{i=1}^{n} \mathbf{A}_{M_i} x^{P_i} = x(\mathbf{A}_{M_1} + x(\mathbf{A}_{M_2} + x(\mathbf{A}_{M_3} +$

$\cdots + x(\mathbf{A}_{M_{n-1}} + x\mathbf{A}_{M_n})\cdots)))$, and also $\tilde{\mathbf{B}}(x)$ similarly. Horner's method also allows us to easily encode A and B in a distributed or centralized manner, as different tasks can be encoded separately. After getting $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, each worker then computes the multiplication of $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$, where the value of x is uniquely chosen on each worker, and then upload the result to the master using mpi4py.MPI.Comm.send, which sends an end-to-end message synchronously. The master will continuously run mpi4py.MPI.Comm.Probe which tests if a new message arrives. Once it returns, it means one worker has finished its task and the result is received by the master, and then the master will save the result by calling mpi4py.MPI.Comm.recv. Once the number of finished tasks reaches the recovery threshold, the master will stop receiving new results (considering the rest of workers as stragglers), and decode the received results to obtain the results of batch matrix multiplication.

We run jobs of batch matrix multiplication on virtual machines hosted in Amazon EC2. The master runs on the virtual machine of type c4.4xlarge (Intel Xeon E5-2666 v3 (Haswell) processors with 16 vCPU, and 30 GB memory) and all workers run on the virtual machine of type c4.2xlarge (Intel Xeon E5-2666 v3 (Haswell) processors with 8 vCPU, and 15 GB memory). We don't add arbitrary delay to the task, since it has been reported that the performance of straggling virtual machines on EC2 be 5x slower than others [3], [4]. We run a job of n matrix multiplications where the sizes of input matrices for each multiplication are 2000×30000 and 30000×2000. We construct RC for $n = 3$, 4, 5, and 6, and the recovery thresholds of corresponding RC are 6, 9, 14, and 18, respectively. The number of workers in each job equals the recovery threshold plus 5 to tolerate up to 5 stragglers. We first run each job with distributed encoding where each worker just computes its own $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ during encoding, using Horner's method for RC, LCC, and CSA codes, and measure its performance in terms of the time of encoding and the completion time of the whole job. Each job is repeated 20 times and we report each data point below as the average.

As a comparison, we also implement existing coding schemes for batch matrix multiplication, including LCC codes [8] and CSA codes [16]. From Fig. 14, we can see that RC outperforms LCC and CSA codes significantly. Due to its simple polynomials, RC saves the encoding time by up to 34.6% compared to LCC codes, and by up to 38.5% compared to CSA codes. Moreover, as the algorithms for the code construction are $O(n)$, most time of encoding is spent evaluating polynomials. In fact, even if $n = 50$, the time of code construction is less than 10^{-4} seconds. Because of the low encoding time, even if RC can tolerate fewer stragglers than LCC and CSA codes (due to its higher recovery threshold which we will elaborate on in Fig. 16) and thus costs relatively more time in decoding, it still saves the completion time of the whole job by up to 27.3% compared to LCC codes and 30.6% compared to CSA codes.

As a comparison, we also run the centralized version of the encoding algorithms in all the jobs above, where the master encodes all coded tasks and distributes such tasks to workers. Surprisingly, we find that the fast encoding algorithms for LCC and CSA codes actually consume more time than the naive algorithm based on matrix multiplication for small numbers of n. To achieve lower encoding time, fast algorithms typically require more than 1000 matrix multiplications (*i.e.*, $n > 1000$) in our implementation. Even so, the encoding time is still significantly higher than the time of the whole job with distributed encoding, because of the large number of matrices to encode. Hence, we use a naive algorithm where the master compute all $\tilde{\mathbf{A}}(x)$ and $\tilde{\mathbf{B}}(x)$ with values of x of all tasks, by applying Horner's method again. This naive algorithm achieves even lower encoding time than the fast encoding algorithms when n is small. Note that the complexities of the naive algorithm for RC, LCC, and CSA codes are in the same order, which is quadratic to the recovery threshold. However, as we will see below, the encoding time of RC is still lower thanks to its simpler forms of polynomials.

We run centralized encoding for all the jobs with the same configurations. In Fig. 15a, we show the encoding time for RC, LCC and CSA codes. Compared to the job completion time in Fig. 14b, we can see that the time of centralized encoding is significantly higher, making it impractical for batch matrix multiplication. Nevertheless, with the naive encoding algorithm, RC achieves the lowest time of encoding again thanks to its simpler polynomials.

We present the recovery thresholds of the coding schemes that we used in the experiments above in Fig. 16. In exchange for a lower amount of time of encoding, RC requires higher recovery thresholds than all the other coding schemes. With the same number of workers, a higher recovery threshold reduces the number of stragglers tolerable in the experiments above, and thus may consume more time to complete the job. However, as we have seen in Fig. 14, the time to complete the whole job with RC is not compromised by the higher recovery threshold. Instead, its low time of encoding compensates for any additional time of computation.

We now consider more extreme cases with $n = 100, 200, 300$, and 400. In such cases, we expect that the recovery threshold of RC will become significantly higher than LCC and CSA codes, because its recovery threshold still grows with a higher order of magnitude than $O(n)$, as demonstrated in Fig. 17. However, we find that the time of distributed encoding of RC is not compromised by the higher recovery threshold. In Fig. 18, we reduce the sizes of $\mathbf{A}_i$ and $\mathbf{B}_i$ as 2000×1000 and 1000×2000, so that we can fit input matrices into the memory. We can see that RC still achieves the lowest encoding time, thanks to its simple polynomials.

VIII. Conclusion and Open Problems

Coded computing has been demonstrated to tolerate stragglers efficiently for distributed matrix multiplication. However, most existing coding schemes can only create coded tasks to tolerate stragglers within only one matrix multiplication. In this paper, we propose rook polynomial coding (RC), a coding framework for batch matrix multiplication, constructed towards saving the time of encoding in practice. We demonstrate that compared to existing schemes, RC can save the time of encoding and achieve lower completion time of the job.

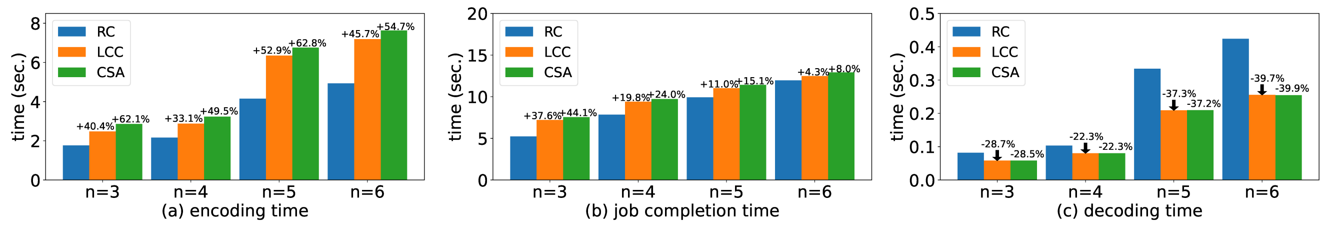


Fig. 14. Time of encoding and the whole job of n matrix multiplications with $n = 3, 4, 5,$ and 6.

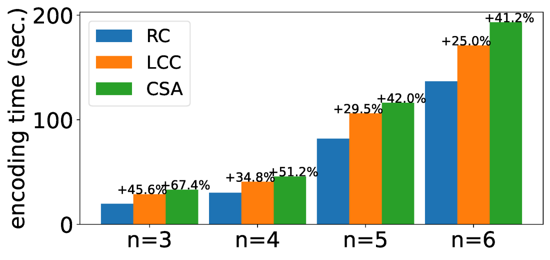


Fig. 15. The time of centralized encoding of RC, LCC, and CSA codes with $n = 3, 4, 5,$ and 6.

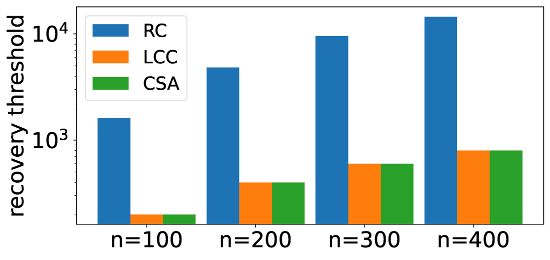


Fig. 17. Recovery thresholds of RC, LCC, and CSA codes with $n = 100, 200, 300,$ and 400.

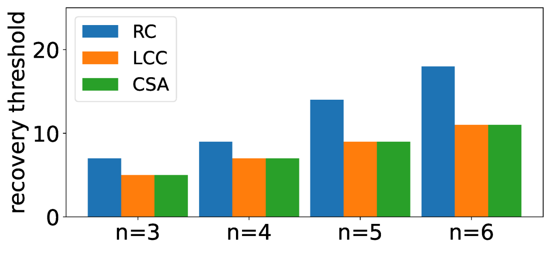


Fig. 16. Recovery thresholds of RC, LCC, and CSA codes with $n = 3, 4, 5,$ and 6.

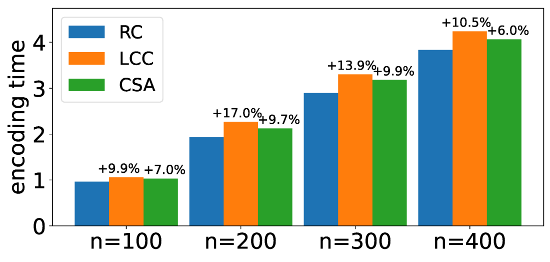


Fig. 18. The time of distributed encoding of RC, LCC, and CSA codes with $n = 100, 200, 300,$ and 400.

In this paper, we show that the recovery threshold of RC can be saved to $O(n^{\log_2 3})$. However, it is unknown if it is the optimal under the coding framework of RC (the coefficients of the polynomial are one of the input matrices). Obviously, the lower bound is $2n-1$ since P and Q are strictly increasing. In other words, it is no less than LCC and CSA codes, in exchange for simpler forms of its polynomials. However, what the optimal recovery threshold is and how to achieve it are still open problems, and we leave them as our future work. We will also make RC support matrix partitioning in the future work.

Acknowledgments

This paper is based upon work supported by the National Science Foundation under Grant No. CCF-2101388. A. Saldivia is supported by the NSF Research Experiences for Undergraduates (REU) supplement award.

References

[1] P. Soto, X. Fan, and J. Li, "Straggler-free Coding for Concurrent Matrix Multiplications," in *IEEE International Symposium on Information Theory (ISIT)*, 2020.
[2] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," in *USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2017.
[3] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding Up Distributed Machine Learning Using Codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.
[4] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *International Conference on Machine Learning (ICML)*, 2017, pp. 3368–3376.
[5] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[6] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the Optimal Recovery Threshold of Coded Matrix Multiplication," *IEEE Transactions on Information Theory*, vol. 66, no. 1, pp. 278–301, 2019.
[7] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," *Advances in Neural Information Processing Systems (NIPS)*, 2017.
[8] ——, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2022–2026.
[9] N. B. Shah, K. Lee, and K. Ramchandran, "When Do Redundant Requests Reduce Latency?" *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 715–722, 2016.
[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective

Straggler Mitigation: Attack of the Clones," in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 185–198.

[11] Z. Qiu and J. F. Pérez, "Evaluating Replication for Parallel Jobs: An Efficient Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2288–2302, 2016.

[12] D. Wang, G. Joshi, and G. Wornell, "Efficient Task Replication for Fast Response Times in Parallel Computation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 599–600, 2014.

[13] K. Lee, R. Pedarsani, and K. Ramchandran, "On Scheduling Redundant Requests with Cancellation Overheads," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1279–1290, 2017.

[14] M. N. Krishnan, E. Hosseini, and A. Khisti, "Coded Sequential Matrix Multiplication For Straggler Mitigation," in *Advances in Neural Information Processing Systems 33 (NeurIPS 2020*, 2020.

[15] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange Coded Computing: Optimal Design for Resiliency, Security, and Privacy," in *The 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2019, pp. 1215–1225.

[16] Z. Jia and S. A. Jafar, "Cross Subspace Alignment Codes for Coded Distributed Batch Computation," *arXiv:1909.13873*.

[17] A. Gerasoulis, M. D. Grigoriadis, and L. Sun, "A Fast Algorithm for Trummer's Problem," *SIAM journal on Scientific and Statistical Computing*, vol. 8, no. 1, pp. s135–s138, 1987.

[18] A. Gerasoulis, "A Fast Algorithm for the Multiplication of Generalized Hilbert Matrices with Vectors," *Mathematics of Computation*, vol. 50, no. 181, pp. 179–188, 1988.

[19] V. Pan, M. A. Tabanjeh, Z. Chen, E. Landowne, and A. Sadikou, "New Transformations of Cauchy Matrices and Trummer's Problem," *Computers & Mathematics with Applications*, vol. 35, no. 12, pp. 1–5, 1998.

[20] K. S. Kedlaya and C. Umans, "Fast Polynomial Factorization and Modular Composition," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1767–1802, 2011.

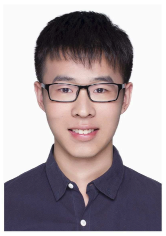
[21] V. Y. Pan, "Matrix Structures of Vandermonde and Cauchy Types and Polynomial and Rational Computations," in *Structured Matrices and Polynomials*. Springer, 2001, pp. 73–116.

[22] Q. Yu and A. S. Avestimehr, "Entangled Polynomial Codes for Secure, Private, and Batch Distributed Matrix Multiplication: Breaking the "Cubic" Barrier," in *IEEE International Symposium on Information Theory (ISIT)*, 2020.

[23] Z. Chen, Z. Jia, Z. Wang, and S. A. Jafar, "GCSA Codes with Noise Alignment for Secure Coded Multi-Party Batch Matrix Multiplication," in *IEEE International Symposium on Information Theory (ISIT)*, 2020.

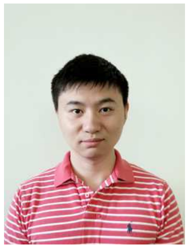
[24] J. Gauthier, "Fast Multipoint Evaluation On n Arbitrary Points," Master's thesis, Simon Fraser University, July 2007.

Pedro Soto is currently a Ph.D. student at the Graduate Center of the City University of New York. He received his B.S. degree in Mathematics at the Florida International University in 2016. His major research interest is the application of coding theory towards distributed computing, with a particular interest in the use of erasure codes towards fault tolerance and straggler mitigation in distributed matrix multiplication and distributed machine learning algorithms.



Xiaodi Fan received his B.S. degree in Communication Engineering at the Hangzhou Dianzi University in July 2018. He is currently a Ph.D. student at the Graduate Center of the City University of New York. His Ph.D. research interest is designing and deploying novel schemes of erasure coding to tolerate stragglers and leverage stragglers in large-scale distributed matrix multiplication and machine learning.

Angel Saldivia is an undergraduate student in the Department of Computer Science, School of Computing & Information Sciences, Florida International University. His current research interests are machine learning and robot vision.



Jun Li received his Ph.D. degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2017, and his B.S. and M.S. degrees from the School of Computer Science, Fudan University, China, in 2009 and 2012. He is currently an assistant professor at the Queens College and the Graduate Center, City University of New York. His research interests fall into the intersection between coding theory and distributed computing and systems.