Rook Polynomial 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 Polynomial Coding (RPC), a novel polynomial-based coding framework for batch matrix multiplication. Designed for decentralized encoding, we construct RPC to recover the result of batch matrix multiplication from a small number of tasks. We also extend RPC to allow partitioning input matrices to save the task complexity. Through extensive experiments, we show that RPC can enjoy much lower time of encoding and achieve lower completion time of the job compared to other coding schemes for batch matrix multiplication.

Index Terms

distributed computing, batch matrix multiplication, straggler mitigation, coded computing, decentralized encoding

I. Introduction

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]. For example, nodes may experience temporary performance degradation, due to load imbalance or resource congestion [3]. It is measured in an Amazon EC2 cluster that a virtual machine affected can have a performance degradation by up to 5 times [3], [4]. A node may even fail to complete a task due to hardware failures, network partition, or power failures. In a Facebook data center, it has been reported that up to more than 100 such failures can happen on a daily basis [5], [6]. Therefore, when the computation is distributed onto multiple nodes, its progress can be significantly affected by the tasks running on such slow or failed nodes, which we call *stragglers*.

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.

The first two authors contributed equally to this work.

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

The adversarial effects of stragglers can be mitigated by launching redundant tasks in advance. A naive application in this principle is replicating 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 further 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.

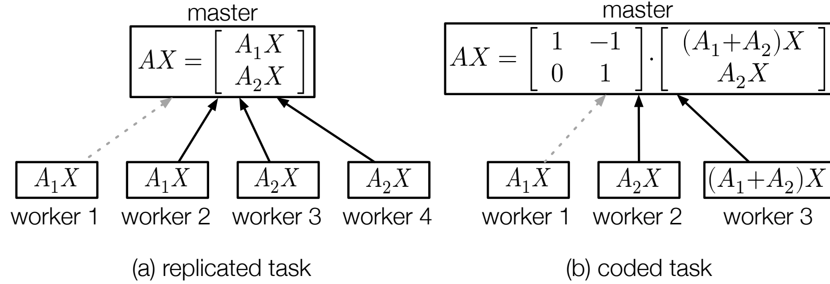


Fig. 1. Examples of distributed matrix multiplications with additional workers (running replicated or coded tasks) to tolerate one single straggler, represented with a gray dotted arrow.

On the other hand, it has been demonstrated that we can tolerate the same number of stragglers with much 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 AX on four worker nodes in Fig. 1a. The matrix A is split into two submatrices, A_1 and A_2, and then AX can be obtained from the results of two tasks, *i.e.*, A_1X and A_2X. In Fig. 1a, such 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 $(A_1 + A_2)X$, which can be decoded to recover A_1X or A_2X if any other task 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 significant attention of research has been attracted to straggler-free coding for distributed computing (*i.e.*, coded computing), especially coded matrix multiplication (*e.g.,* [3], [7]–[9]), most existing coding schemes have been focusing 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 altogether. Conventionally, we can launch multiple distributed jobs for each multiplication. However, additional tasks must be added into each job to tolerate its own stragglers, *i.e.*, a coded task can only tolerate a straggler inside the same job.

In this paper, we propose Rook Polynomial Coding (RPC), a novel coding framework for distributed batch matrix multiplication where the results of multiple matrix multiplications can be obtained concurrently with one job only. In this coding framework, additional coded tasks can be used to tolerate any stragglers. Based on polynomials, we first construct a special case of RPC with a proof of its optimality, and then propose two general constructions based on the special case. With generalized constructions, we save the *recovery threshold*, *i.e.*, the number of tasks required for recovering the results of n matrix multiplications from $O(n^2)$ to $O(n^{\log_2 3})$. We further integrate RPC with entangled polynomial code, a polynomial-based coding scheme for distributed matrix multiplication, so that RPC can also work with partitioned input matrices when

they are large.

Moreover, RPC is designed towards decentralized encoding so that each worker will only need to encode its own coded task. Compared to existing coding schemes for batch matrix multiplication, RPC is constructed with a much simpler form of polynomials, leading to the lowest complexity for decentralized encoding. With experiments running on AWS, we demonstrate that RPC achieves much lower time of encoding, and thus also achieves lower job completion time than other coding schemes for batch matrix multiplication.

The rest of this paper is organized as follows. We first discuss related works on coded matrix multiplication, especially coding for batch matrix multiplication in Sec. II, and then present motivating examples of RPC in Sec. III. We then present the general coding framework in Sec. IV. In Sec. V we consider a special case in the coding framework and construct RPC in this special case with a proof of its optimality. We then extend the construction into two general constructions that require fewer tasks to recover the whole job in Sec. VI and Sec. VII. In Sec. VIII we integrate RPC with entangled polynomial codes to allow matrix partitioning. We present the implementation of RPC and its evaluation results in Amazon EC2 in Sec. IX and conclude this paper in Sec. X.

II. Related Work

There has been a surge of interests recently on the mitigation of stragglers in distributed computing, which runs a large number of parallel tasks on different nodes. It is well known 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 [10]–[14], 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 which tolerate the same number of stragglers with much lower resource overhead (*e.g.*, [3], [4]).

One of the critical applications of coded distributed computing is the 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], MDS codes are applied in matrix-vector multiplication of the form Ax where the matrix A is large and hence will be partitioned vertically along the rows and encoded with MDS codes. This direction was continued by Yu *et al.* [8] and Dutta *et al.* [7] who considered a more general problem of matrix-matrix multiplication, *i.e.*. $A \cdot B$. The two input matrices A and B are partitioned along their rows and columns, respectively, or the other way round. The proof of the optimally for such configurations have been given by Yu *et al.* [8], [9], where the authors have also derived entangled polynomial codes that allow partitioning the input matrices in a more general way, *i.e.,* A and B can both be arbitrarily partitioned by their rows and columns.

In this paper, we consider a more general problem, *i.e.*, distributed coded computing for batch matrix multiplication. In other words, different from existing works above that consider one matrix multiplication only, this paper investigates coding 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.* [15] where the computation is complete with additional rounds such that coding can be applied across time. In this paper, we focus on the problem of batch matrix multiplication.

Existing works for batch matrix multiplication have been constructed as polynomials, either using the Lagrange polynomial [16], [17] or based on the Cauchy-Vandermonde matrix [18], [19]. Although fast algorithms (*e.g.,* [20]–[23]) exist for the encoding of such coding schemes, which achieve a low complexity, the encoding algorithms must be centralized, *i.e*, input matrices will be encoded into all tasks on one single node. As input matrices are large, it will still consume a significant amount of time for encoding. In this paper, we propose rook polynomial coding (RPC), another polynomial-based coding scheme for batch matrix multiplication. Different from existing works, RPC is constructed as a polynomial with a much simpler form, making it faster for decentralized encoding where each worker will encode input matrices for its own task only. Although the recovery threshold of RPC will be larger than existing coding schemes designed for centralized encoding, RPC can still achieve lower completion time thanks to its low encoding overhead.

It is worth mentioning that security and privacy can be supported in coded batch matrix multiplication, by adding additional matrix multiplications of random matrices [16]–[19], [24]. Similar designs for a single matrix multiplication have also been proposed by Nodehi and Maddah-Ali [25], [26]. Although RPC is not originally designed to support security and privacy, it can be extended following the same approach, by adding additional multiplications of randomly generated matrices. Hence, we focus on the coding scheme of RPC to tolerate stragglers in this paper.

III. Motivating Examples

We start with a toy example to demonstrate the advantages of our coding framework for batch matrix multiplication. Instead of applying coding individually in each matrix multiplication, we propose RPC in this paper which encodes all input matrices in batches. In this section, we discuss a case of RPC with two multiplications as a toy example. We will present the general construction of RPC in the rest of this paper.

Assume that we need to compute the results of two matrix multiplications, *i.e.*, A_1B_1 and A_2B_2, where A_1 and B_1 are of the same size with A_2 and B_2, respectively. 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

$$\hat{A} \cdot \hat{B} \triangleq \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \cdot \begin{bmatrix} B_1 & B_2 \end{bmatrix} = \begin{bmatrix} A_1B_1 & A_1B_2 \\ A_2B_1 & A_2B_2 \end{bmatrix}.$$

In this way, the result of A_1B_1 and A_2B_2 can be obtained as submatrices of $\hat{A}\hat{B}$. However, the complexity of $\hat{A}\hat{B}$ becomes four times as A_iB_i. In order to generate coded tasks with the same complexity as A_iB_i, we can apply polynomial codes [8], a polynomial-based coding scheme for matrix multiplication, to the job of $\hat{A}\hat{B}$, by encoding $\hat{A}$ as $\tilde{A}(x) = A_1x^0 + A_2x^2$ and $\hat{B}$ as $\tilde{B}(x) = B_1x^0 + B_2x^1$, and then the sizes of $\tilde{A}(x)$ and $\tilde{B}(x)$ equal those of A_i and B_i, respectively. A coded task can then be generated as a polynomial of $\tilde{C}(x) \triangleq \tilde{A}(x)\tilde{B}(x)$, *i.e.*,

$$\tilde{C}(x) = A_1B_1x^0 + A_1B_2x^1 + A_2B_1x^2 + A_2B_2x^3.$$

Given any 4 coded tasks $\tilde{C}(x)$ 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, $\tilde{A}(x)$ and $\tilde{B}(x)$ can be generated differently as $\tilde{A}(x) = A_1x^0 + A_2x^1$, and $\tilde{B}(x) = B_1x^0 + B_2x^1$, respectively. Hence, $\tilde{C}(x) = A_1B_1x^0 + (A_1B_2 + A_2B_1)x^1 + A_2B_2x^2$. 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 be saved from $O(n^2)$ to $O(n^{\log_2 3})$ as n scales.

Compared to other schemes that construct codes for batch matrix multiplication based on Lagrange interpolation polynomial or its variant, such as LCC codes [16] and CSA codes [18], our coding scheme enjoys much simpler form in $\tilde{A}(x)$ and $\tilde{B}(x)$, as the input matrices directly become the coefficients in the two polynomials. Although our coding scheme requires more tasks, we demonstrate that in practice the corresponding additional time will be compensated by the saving of encoding time. We also extend RPC to support batch matrix multiplication while allowing A_is and B_is being partitioned into submatrices.

IV. Coding Framework

Given n matrix multiplications, *i.e.*, A_1B_1, A_2B_2, …, and A_nB_n, we assume that $A_1, \ldots, A_n$ are of the same size, and $B_1, \ldots, B_n$ also have the same sizes. In this paper, we propose a polynomial-based coding scheme which requires a low number of tasks to tolerate any r stragglers with a low complexity for decentralized encoding. In other words, coded tasks are not encoded by one single node, *e.g.*, the master node, but individually by every workers. Hence, each worker only needs to run one single evaluation of $\tilde{A}(x)$ and $\tilde{B}(x)$, and existing fast algorithms [20]–[23] for polynomial evaluations cannot be applied for such decentralized encoding.

In particular, we only consider the case where the input matrices are not partitioned in this section, *i.e.*, the coding is only applied across multiplications, without partitioning each input matrix. Hence, the complexity of each coded task remains the same as that of the original matrix multiplications. In Sec. VIII, we discuss the extension of applying coding across multiple matrix multiplications with matrix partitioned simultaneously.

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 $\tilde{A}(x)$ and $\tilde{B}(x)$, where $\tilde{A}(x) = \sum_{i=1}^{n} A_{M_i} x^{P_i}$ and $\tilde{B}(x) = \sum_{i=1}^{n} B_{N_i} x^{Q_i}$. Compared to other polynomials such as the Lagrange interpolation polynomial, our encoding polynomials have a much simpler form since all input matrices appear directly as a coefficient, making it ideal for decentralized encoding.

Given $\tilde{A}(x)$ and $\tilde{B}(x)$, a coded task will then compute $\tilde{C}(x) = \tilde{A}(x)\tilde{B}(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} A_{M_i}B_{N_j}x^{P_i+Q_j}$, which is still a polynomial of x. The result in each task can be considered as a evaluation of $\tilde{C}(x)$ if x in each task is unique. We can

then interpolate $\tilde{C}(x)$ 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 A_iB_i from the coefficients of $\tilde{C}(x)$. In other words, their corresponding exponents of x should be unique.

To illustrate the code scheme, in this paper, we use a $n \times n$ table to depict a particular 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 $A_{M_i}B_{N_j}$. We also place A_{M_i} and B_{M_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.

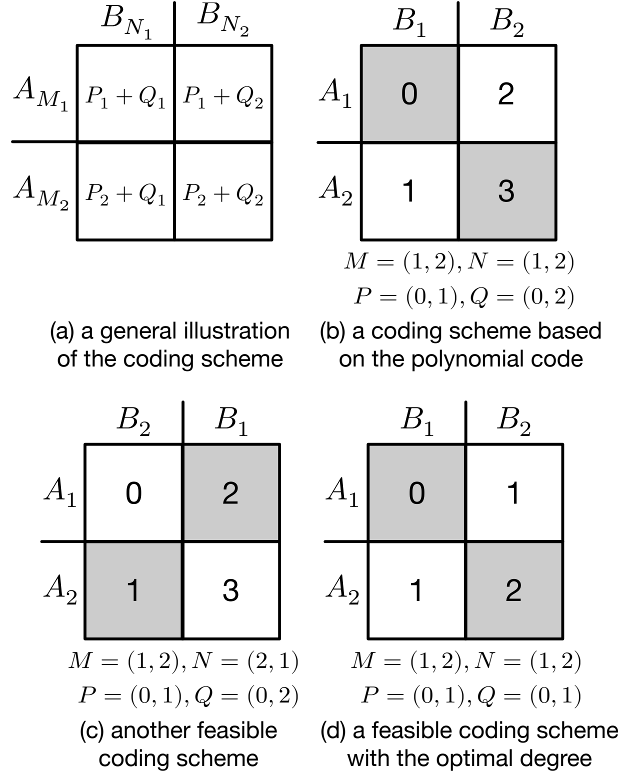


Fig. 2. The illustrations of the coding schemes achieved under the coding framework with $n = 2$.

As shown 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. III. In this way, we have $\tilde{A}(x) = \sum_{i=1}^{n} A_i x^{i-1}$ and $\tilde{B}(x) = \sum_{i=1}^{n} B_i x^{(i-1)n}$. Therefore, in $\tilde{C}(x)$, there are n^2 terms whose coefficients are A_iB_j, $1 \leq i, j \leq n$. As shown in Fig. 2b, the exponents of the four terms in $\tilde{C}(x)$ ranges between 0 and 3. Hence, we need to have the results of any 4 tasks to obtain the results of A_1B_1 and A_2B_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 A_1B_2 and A_2B_2 are placed in the counter diagonal, as highlighted in the table. We also highlight the entries of A_1B_1 and A_2B_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.

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. III. 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 which help to achieve lower recovery thresholds.

V. Rook Polynomial Coding: A Special Case

A. *Scopes of Parameters*

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 $A_i B_i$ 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 $A_{M_i} B_{N_{j_1}}$ equals that of $A_{M_i} B_{N_{j_2}}$. In other words, $A_{M_i} B_{N_{j_1}}$ 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 $Q_1 < \ldots < Q_n$.

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 where $P = (0, \ldots, n-1)$.[1] We construct RPC with this condition and prove the optimality of the construction in this special case. We extend the construction of RPC to the general values of P in Sec. VI and Sec. VII.

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 A_{M_i} and B_{N_j} as the title of each row and each column, respectively. If we switch A_{i_1} with A_{j_1} and meanwhile B_{i_2} with B_{j_2}, if $A_{i_1} = B_{j_1}$ and $A_{i_2} = B_{j_2}$, 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.

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

[1] 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.

	B_3	B_1	B_2	B_4
A_1	0	4	7	9
A_2	1	5	8	10
A_3	2	6	9	11
A_4	3	7	10	12

(a) Placement 1

	B_1	B_4	B_2	B_3
A_1	0	1	5	7
A_2	1	2	6	8
A_3	2	3	7	9
A_4	3	4	8	10

(b) Placement 2

Fig. 3. Two placements of highlighted entries and their corresponding optimal coding schemes.

B. Achieving the Optimal Degree of $\tilde{C}(x)$

Besides having P and M fixed, we first construct a coding scheme with the optimal degree of $\tilde{C}(x)$ from a given placement of highlighted entries, *i.e.*, the values in N has also been 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 $\tilde{C}(x)$. We now discuss the algorithm for the second step in Alg. 1 and leave the first step in Sec. V-C. We also prove the optimality of Alg. 1 with Theorem 1.

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 ← 2 to n do
3:     if n − N_{i−1} ≤ N_i − 1 then
4:         Q_i ← Q_{i−1} + N_{i−1}
5:     else
6:         Q_i ← Q_{i−1} + n − N_i + 1
7:     end if
8: end for
```

Theorem 1. *Given highlighted entries,* i.e.*,* $M = (1, \ldots, n)$*,* $P = (0, \ldots, n-1)$*, and* N *given as a permutation of* $\{1, \ldots, n\}$*, 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 as large as possible. 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 of 1, 2, and 3 shared in the same way 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 in the gap 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 gap at the top in the i-th column is smaller than the gap at the bottom of 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

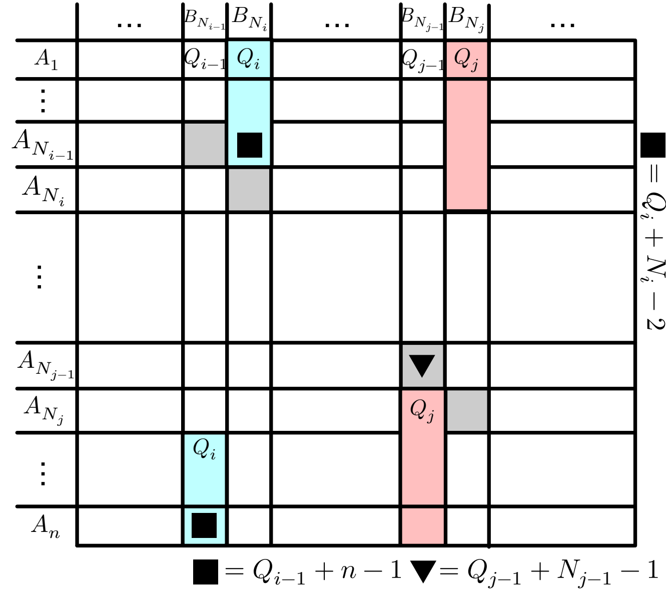


Fig. 4. Two placements of highlighted entries and their corresponding optimal coding schemes.

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 gap at the bottom in the $(j-1)$-th column is smaller than the gap at the top of 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 $Q_{j-1}+N_{j-1}$. $\square$

Since $N_{i-1} \geq n-N_i+1$ if $n-N_{i-1} \leq N_i-1$, we can further simplify Line 3–6 in Alg. 1 as $Q_i = Q_{i-1}+\max\{N_{i-1}, n-N_i+1\}$, and the degree of $\tilde{C}(x)$ is $\sum_{i=2}^{n} \max\{N_{i-1}, n-N_i+1\}+n-1$.

Moreover, we can see from Alg. 1 that all integers between 0 and the degree of $\tilde{C}(x)$ appear at least once as the exponents in $\tilde{C}(x)$, 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 $\tilde{C}(x)$ 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 $\tilde{C}(x)$.

We now propose a placement of highlighted entries which can be proved to achieve the optimal degree in $\tilde{C}(x)$. 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. 5. The illustration of the algorithm to find the optimal placement of highlighted entries.

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.

Algorithm 2 The algorithm that constructs the optimal placement of highlighted entries with $P = (0, \ldots, n - 1)$.

Output: N

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

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

D. Analysis

In Sec. V-B, we have demonstrated that the recovery threshold of RPC constructed with Alg. 1 and Alg. 2 is one more than the degree of $\tilde{C}(x)$. To analyze the degree of $\tilde{C}(x)$, we first count the number of unhighlighted entries with unique exponents as $u(n)$.

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

As for other values of n, we can get the value of $u(n)$ recursively. With the construction in Fig. 5, we can see that the right-bottom entry is always highlighted. In the two rightmost rows, 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, $i = 2, \ldots, n$. Therefore, there are only two additional unhighlighted entries with unique exponents, *i.e.*, $u(n+2) = u(n) + 2$. For general values of n, we thus have

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

Among a total of n^2 entries, there are n highlighted entries and $u(n)$ unhighlighted entries with unique exponents. Then the number of unhighlighted entries with shared exponents is $n^2 - n - u(n)$. As each exponent can be shared by at most two entries (since both P and Q are strictly increasing), the total degree of $\tilde{C}(x)$ is $\frac{n^2-n-u(n)}{2} + n + u(n) - 1 = \frac{n^2+n+u(n)}{2} - 1$, 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 $r(n)$ to denote the recovery threshold of RPC constructed with Alg. 1 and Alg. 2, and then $r(n)$ 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 $r(n) = \left\lfloor \frac{(n+1)^2}{2} \right\rfloor - 1$.

E. Optimality

We now prove that the construction of RPC given by Alg. 1 and Alg. 2 is optimal when $P = (0, \ldots, n-1)$. Equivalently, we can prove that $u(n)$ is optimal.

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

Lemma 1. *In a $n \times n$ table ($n > 2$), 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.*

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 $\forall j = 0, \ldots, n$, $|(n+1) - (N_j + N_{j+1})| \le \frac{1}{2} \sum_{j=0}^{n} |(n+1) - (N_j + N_{j+1})|$.

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 $j \in J_2$, $|(n+1) - (N_j + N_{j+1})| \le \sum_{j \in J_1} |(n+1) - (N_j + N_{j+1})| = \sum_{j \in J_2} |(n+1) - (N_j + N_{j+1})| = \frac{1}{2} \sum_{j=0}^{n} |(n+1) - (N_j + N_{j+1})|$. If $j \in J_3$, then $|(n+1) - (N_j + N_{j+1})| = 0$ which is also no more than $\frac{1}{2} \sum_{j=0}^{n} |(n+1) - (N_j + N_{j+1})|$. $\square$

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

Proof. To prove that the placement of highlighted entries constructed in Sec. V-C is optimal, we demonstrate that given any placement in an $(n+2) \times (n+2)$ table, the number of unhighlighted entries with unique exponents will be two more than that of an optimal placement in an $n \times n$ table.

Given a placement in an $(n+2) \times (n+2)$ table, we can always find two columns whose highlighted entries appear in the top row or in the bottom row. In other words, there exist j_1 and j_2 such that $N_{j_1} = 1$ and $N_{j_2} = n+2$. After removing the top and the bottom rows, as well as the j_1-th and the j_2-th columns, we can get a placement of the $n \times n$ table.

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. We first consider 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 $|(n+1-N_{j_1-1}) - (N_{j_1+1}-1)| = |(n+2) - (N_{j_1} + N_{j_2})|$. By Lemma 1, $|(n+2) - (N_{j_1-1} + N_{j_1+1})| \leq \frac{u(n)}{2} = \frac{n-1}{2}$, otherwise the placement in the $n \times n$ table after the removal is not optimal. If so, we have $|(n+2) - (N_{j_1} + N_{j_2})| < 2(n+2) - (N_{j_1} + N_{j_2})$.

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.

We now consider the j_2-th column. Similarly, if $1 < j_2 < n+2$, before the removal, the number of unhighlighted entries with unique exponents between the highlighted entries in the (j_2-1)-th and the (j_2+1)-th column is $[(n+1) - (n+2 - N_{j_2-1})] + (N_{j_2+1} - 1) = (N_{j_2-1} + N_{j_2+1}) - 2$. After the removal, the number still becomes $|(n+2) - (N_{j_1} + N_{j_2})|$, which is no more than $\frac{n-1}{2}$. Hence, $|(n+2) - (N_{j_1} + N_{j_2})| < (N_{j_2-1} + N_{j_2+1}) - 2$.

If $j_2 = 1$, then after removal the number of unhighlighted entries with unique exponents will be reduced, as all the unhighlighted entries in the j_2-th column are not be shared by any other entries anyway.

If $j_2 = n+2$, we only need to count the number of unhighlighted entries with unique exponents between and the highlighted entry in the $(n+1)$-th column and the last entry in the last 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 should be no more than $\frac{n-1}{2}$. Hence, we have $N_2 - 1 < n+2-N_2$.

Combine the two columns together, the number of unhighlighted entries with unique entries will be decreased by 2 at least after the removal, if the placement in the $n \times n$ table is optimal.

If $|j_1 - j_2| = 1$, then there can be two possibilities of their positions. Without loss of generality, we assume $j_1 < j_2$. If $N_{j_1} = n+2$, then 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 $N_{j_1} = 1$, then $N_{j_2} = n+2$. Hence, 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 $u(n+2) \geq u(n)+2$. By our analysis in Sec. V-D, the equality is achieved by Alg. 2 in Sec. V-C. □

Note that the optimality of the construction above is achieved under the assumption that $P = (0, \dots, n-1)$. Based on this construction, we propose two general constructions in Sec. VI and Sec. VII which do not have such a requirement for P and counter-intuitively reduce the recovery threshold from $O(n^2)$ to $O(n^{\log_2 3})$.

VI. First General Construction

We now extend the construction of RPC in Sec. V by removing the assumption of $P = (0, \dots, n-1)$. We propose two constructions that achieve similar recovery thresholds, yet either one of them may have a better performance with a specific value of n. In those general constructions, we allow choosing arbitrary values in P. Although minimizing P to be $(0, \dots, n-1)$ seems also minimizing the degree of $\tilde{C}(x)$, we find that the degree of $\tilde{C}(x)$ can be even lower with general values in P as it creates more chances to share exponents. We now present the first general construction in this section. The second general construction is based on the first general construction and will be presented in Sec. VII.

A. A Toy Example

It is counter-intuitive to have values in P larger than $n-1$, since it seemingly will only increase the corresponding degree of $\tilde{C}(x)$. However, as illustrated in Fig. 7a, the degree of $\tilde{C}(x)$ is reduced from 10 to 8 with $P = (0, 1, 3, 4)$, compared to the original construction in Fig. 6d. The reason for this better result is that the construction in Fig. 7a allows more overlaps among exponents. In the construction in Sec. V, 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 $A_1, \dots, A_4$ into two groups $\{A_1, A_2\}$ and $\{A_3, A_4\}$, and also $B_1, \dots, B_4$ into $\{B_1, B_2\}$ and $\{B_3, B_4\}$. We then construct the code in two steps. We first place the four groups in a 2×2 table, as shown in Fig. 7b. It is easy to see that the intersection of $\{A_1, A_2\}$ and $\{B_1, B_2\}$ and that of $\{A_3, A_4\}$ and $\{B_3, B_4\}$ need to have unique exponents and the other two intersections can have shared exponents. Hence, we apply RPC for such four groups 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 RPC again for each intersection, with $n = 2$. Note that in each intersection, there are 3 different exponents, and the first and last exponents need to be unique. Hence, the exponents in the first step need to be

	B_1	B_2	B_3	B_4
A_1	0	1	3	4
A_2	1	2	4	5
A_3	3	4	6	7
A_4	4	5	7	8

(a) code construction for n=4

	$\{B_1, B_2\}$	$\{B_3, B_4\}$
$\{A_1, A_2\}$	0 x 3	1 x 3
$\{A_3, A_4\}$	1 x 3	2 x 3

(b) recursive construction

Fig. 7. An toy example of the first general construction of RPC.

multiplied by 3, and the exponents in each entry in Fig. 7a should be the sum of the corresponding exponents of the two steps, such that the exponents of A_iB_i, $i = 1, \ldots, 4$ can be both unique in each intersection and across different intersections. In this way, we can get $P = Q = (0, 1, 3, 4)$.

B. Code Construction

From Fig. 7, we can see that the first general 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 $A_1, \ldots, A_n$ into p groups, where each group contains q matrices, *i.e.*, $\{A_1, \ldots, A_q\}$, ..., $\{A_{n-q+1}, \ldots, A_n\}$. We can also do the same to $B_1, \ldots, B_n$. To avoid ambiguity, we rewrite P, Q, N as $P(n)$, $Q(n)$, $N(n)$ for RPC constructed with n matrix multiplications.

We first construct a RPC for $n = p$, where each entry is for the intersection of two groups of A_i and B_i. In Fig. 8 we demonstrate the recursive construction with $p = 3$. We can see that the $p = 3$ groups of A_is and B_is are placed along the rows and columns of the 3×3 table, corresponding to the RPC for $n = p = 3$. The only difference is that the exponents are multiplied by $r(q)$, the recovery threshold of RPC for $n = q$, as there will be $r(q)$ exponents to accommodate in each intersection in the second step.

In the second step, we construct a RPC for $n = q$ for each intersection in the first step, and then we can determine the placement of A_is and B_is in each group. Note that all intersections actually have the same placement as they are all RPCs for $n = q$, so all intersections in the same rows (columns) have the same placements of A_is (B_is). In particular, each entry of such RPCs needs to be added with the exponents of its corresponding intersection in the first step. In other words, each entry should be added with a value that equals $r(q)$ times the corresponding exponent in the first step, such that unique entries in each intersection do not coincide with any unique entries in other intersections.

Given the general 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 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 intersection in the first

	$\{B_{q+1},\dots,B_{2q}\}$	$\{B_1,\dots,B_q\}$	$\{B_{2q+1},\dots,B_{3q}\}$
$\{A_1,\dots,A_q\}$	0 ... Q_p / ⋮ ⋱ ⋮ / P_p ... r(q)-1	3 × r(q)	4 × r(q)
$\{A_{q+1},\dots,A_{2q}\}$	1 × r(q)	4 × r(q)	5 × r(q)
$\{A_{2q+1},\dots,A_{3q}\}$	2 × r(q)	5 × r(q)	6 × r(q)

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

step and (i_2, j_2) indicates corresponding entry in this intersection. Therefore, we have

$$P_i(n) + Q_j(n) = (P_{i_1}(p) + Q_{j_1}(p)) \cdot r(q) + (P_{i_2}(q) + Q_{j_2}(q)).$$

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

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 intersections and j_2-th column in this intersection. The group corresponding to this intersection then should be $\{B_{(N_{j_1}(p)-1)q+1}, \dots, B_{(N_{j_1}-1)(p)q+q}\}$. 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 use $R_1(n)$ to represent the recovery threshold of RPC constructed with the first general construction, and then we can replace $r(q)$ with $R_1(q)$ in the second step. We summarize the first general code construction of RDP in Alg. 3, and then analyze the recovery threshold $R_1(n)$ in Sec. VI-C (used at Line 11 in Alg. 3).

Algorithm 3 The first general construction of RPC.

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 an RPC with $n = p$
5: Obtain $P(q), Q(q), N(q), R_1(q)$ from an RPC with $n = q$
6: **for** $i \leftarrow 1$ to n **do**
7: Let $i = (i_1 - 1)q + i_2$ where $1 \le i_1 \le p$ and $1 \le i_2 \le q$
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 RPC for $n = q$ is consecutive, then the exponents for RPC for $n = pq$, given by the first general construction, will also be consecutive. This is because each intersection in the first step separates from each other by $R_1(q)$ which is the number of exponents in each intersection. As the number of exponents in each intersection is $R_1(q)$, and the exponents in different intersections do not overlap, the total number of exponents is $R_1(p)R_1(q)$. Since we will also recursively construct RPC 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 general construction $R_1(n)$ is $\prod_i r(p_i)^{\alpha_i}$.

We now analyze the recovery threshold by discussing some representative special cases. Obviously, if n is a prime number, then $R_1(n) = r(n) = \left\lfloor \frac{(n+1)^2}{2} \right\rfloor - 1$.

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

$$R_1(n) = \prod_i r(p_i)^{\alpha_i} = \prod_i O\left(\frac{p_i^2}{2}\right)^{\alpha_i} = O\left(\frac{(\prod_i p_i^{\alpha_i})^2}{2^{\sum_i \alpha_i}}\right) = O\left(\frac{n^2}{2^{\sum_i \alpha_i}}\right).$$

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, when $n = 2^\alpha$, $R_1(n) = r(2)^\alpha = 3^{log_2 n} = O(n^{\log_2 3}) \approx O(n^{1.585})$.

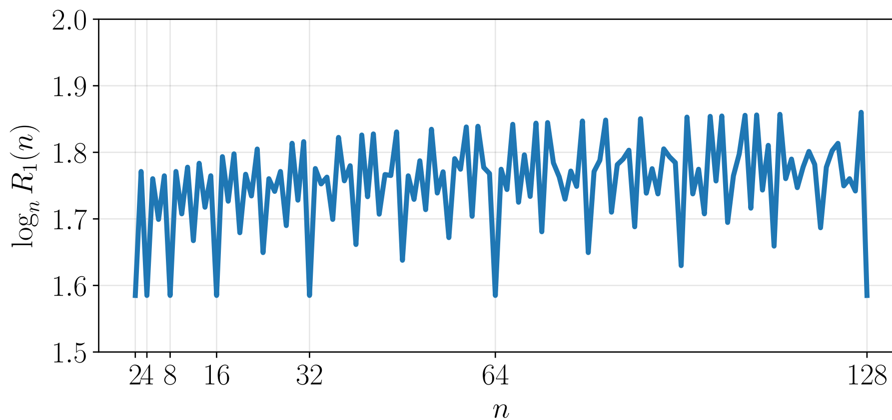


Fig. 9. The growth of the recovery threshold $R_1(n)$.

We show in Fig. 9 how $R_1(n)$ grows with n, when $2 \le n \le 128$. In order to make it easy to compare, we show $\log_n R_1(n)$ in the Y-axis. We can see that $R_1(n)$ grows between $O(n^{1.585})$ (when n is a power of 2) and $O(n^{1.875})$. We expect that $\log_n R_1(n) \to 2$ when $n \to \infty$ if n is a prime.

In order to make the recovery threshold be $O(n^{\log_2 3})$, we can apply an easy trick that adds dummy matrix multiplications in order to increase the number of matrix multiplications to a power of 2. If i is the smallest integer such that $n \le n' = 2^i$, $i \in \mathbb{Z}^+$, we have $n' < 2n$. Therefore, the recovery threshold is less than or equal to $(2n)^{\log_2 3} = 3n^{\log_2 3}$, *i.e.*, the recovery threshold will always be $O(n^{\log_2 3})$. Sometimes this trick can help to save the recovery threshold. When $n = 7$, we can save the recovery threshold from 31 to 27 by adding one dummy matrix multiplication. However, in many cases, the actual recovery

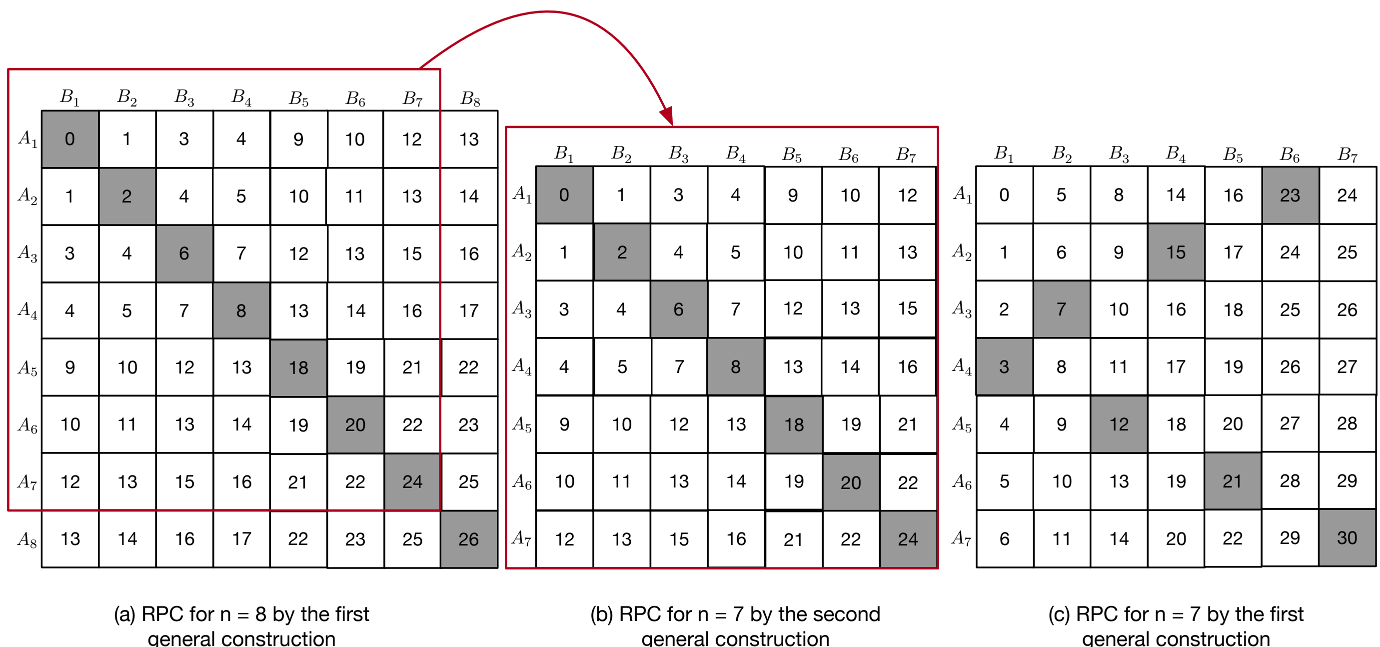
threshold may be increased after adding dummy matrix multiplications. For example, when $n = 3$, adding one dummy matrix multiplication will make the recovery threshold be increased from 7 to 9.

VII. Second General Construction

We now present the second general construction for RPC. From Fig. 9, we can see that in the first general construction the recovery threshold does not monotonically increase with n. In other words, if additional matrix multiplications are added for some n, it is possible that we can save the number of workers. In this section, we present another construction that not only makes the recovery threshold strictly monotonically increase with n, but also achieves a better recovery threshold for some values of n.

A. Intuition and Construction

The second general construction of RPC is, in fact, based on the first general construction, and hence we first present an example of the RPC given by the second general construction. In Fig. 10a we first present a construction of RPC using the first general construction for $n = 8$, 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 an RPC for $n = 7$, such that $M = N = (1, 2, \ldots, 7)$, $P = Q = (0, 1, 3, 4, 10, 12)$. As shown in Fig. 10b, the recovery threshold of this construction is 25. This recovery threshold is even smaller than that of RPC for $n = 7$ using the first general construction (as shown in Fig. 10c).



(a) RPC for n = 8 by the first general construction

(b) RPC for n = 7 by the second general construction

(c) RPC for n = 7 by the first general construction

Fig. 10. Examples of RPC constructed with the first and second general construction.

In the first general construction, when $n = 2$ the values of M and N are sequential. Since we construct RPC recursively when n is a power of 2, *i.e.*, $n = 2^i$ where i is an 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 RPC for n_0 matrix multiplications.

Following this intuition, we give the second general code construction in Alg. 4.

Algorithm 4 The second general construction of RPC.

Input: n
Output: $P(n), Q(n), N(n)$
1: Find the smallest positive integer such that $2^{i-1} < n \leq 2^i$.
2: Obtain $P(2^i), Q(2^i), N(2^i)$ from an RPC constructed with the first general construction
3: **for** $i \leftarrow 1$ to n **do**
4: $\quad P_i(n) = P_i(2^i)$
5: $\quad Q_i(n) = Q_i(2^i)$
6: $\quad N_i(n) = N_i(2^i)$
7: **end for**

B. Analysis

Since from the first general construction of RPC with $n = 2^i$ we can get the RPC of the second general construction with any $n_0 \leq n$, we only need to analyze the exponents of highlighted entries in the first general construction when $n = 2^i$. Let $R_2(n)$ denote the recovery threshold for RPC constructed with the second general construction. We have $R_2(n) = R_1(n) = 3^{\log_2 n} = 3^i$ if $n = 2^i$, since the two constructions are the same in this case. As for the general value of n, we prove the following theorem.

Theorem 3. *Using the second general construction, $R_2(n)$ satisfies the following equation:*

$$R_2(n) = \begin{cases} 1 & n = 1; \\ R_2(n - 2^{i-1}) + 2 \cdot 3^{i-1} & 2^{i-1} < n \leq 2^i, i \in \mathbb{Z}^+. \end{cases}$$

Proof. If $n = 1$, we have $R_2(1) = R_1(1) = 1$. Otherwise, there exits $i \in \mathbb{Z}^+$ such that $2^{i-1} < n \leq 2^i$. Based on the second general construction, we will first construct an RPC with $n = 2^i$ using the first general construction, and then the recovery threshold $R_2(n) = P_n(2^i) + Q_n(2^i) + 1$.

In the first general construction, an RPC with $n = 2^i$ should be recursively constructed from the RPC with $n = 2^{i-1}$. From Alg. 3 we have $n = 2^{i-1} + (n - 2^{i-1})$, and then $P_n(2^i) = P_2(2)r(2^{i-1}) + P_{n-2^{i-1}}(2^{i-1})$ and $Q_n(2^i) = Q_2(2)r(2^{i-1}) + Q_{n-2^{i-1}}(2^{i-1})$. Therefore, $R_2(n) = r(2^{i-1})(P_2(2) + Q_2(2)) + P_{n-2^{i-1}}(2^{i-1}) + Q_{n-2^{i-1}}(2^{i-1}) + 1 = 2 \cdot R_2(2^{i-1}) + R_2(n - 2^{i-1}) = 2 \cdot 3^{i-1} + R_2(n - 2^{i-1})$.

□

Based on Theorem 3, we further prove that $R_2(n)$ increase strictly monotonically with n.

Theorem 4. $\forall n \in \mathbb{Z}^+, R_2(n) < R_2(n+1)$.

Proof. We use induction to prove this theorem. First, if $n = 1$, we have $R_2(1) = 1 < 3 = R_2(2)$.

We now prove that given $i \in \mathbb{Z}^+$, if the theorem is true for all $n \leq 2^{i-1}$, it is also true for all $n \leq 2^i$. Obviously we only need to consider the cases when $2^{i-1} < n \leq 2^i$.

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 $n = 2^i$, $R_2(n+1) = R_2(2^i+1) = R_2(2^i+1-2^i) + 2 \cdot 3^i = R_2(1) + 2 \cdot 3^i > 3^i = R_2(2^i) = R_2(n)$. $\square$

Since $R_2(n)$ is strictly monotonic and $R_2(2^i) = R_1(2^i)$, the recovery threshold of the second general construction is still $O(n^{\log_2 3})$. However, in many cases, the second general construction can achieve a lower recovery threshold than the first general construction. For example, when $n = 7$, $R_1(n) = 31$ and $R_2(n) = 25$. Fig. 11 illustrates the differences of the recovery thresholds between the two constructions. In practice, given a specific value of n, we may simply try both two constructions and choose the construction with a lower recovery threshold.

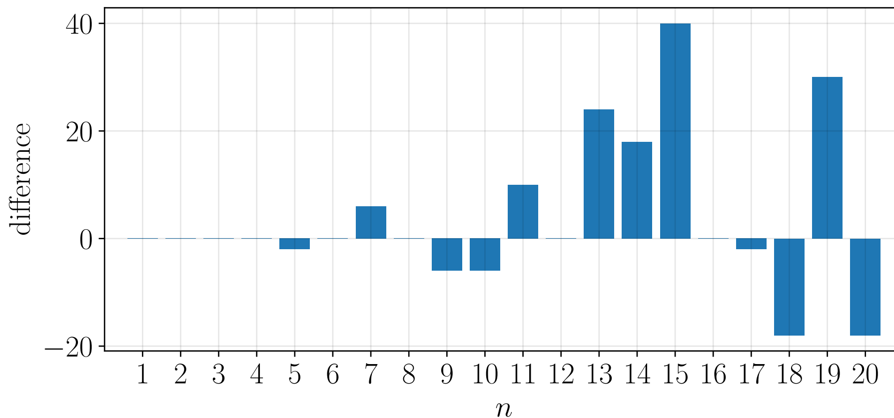


Fig. 11. The difference of the recovery threshold between the first and second general constructions. Data entries are positive if the recovery threshold of the first general construction is greater.

VIII. Rook Polynomial Coding with Matrix Partitioning

In this section, we extend our coding framework to apply coding on batch matrix multiplication and matrix partitioning at the same time. In the code constructions of RPC we have presented so far, each entry in the table is associated with only one exponent, whose corresponding coefficient is a multiplication between A_i and B_j, $1 \leq i, j \leq n$. To further support matrix partitioning, we extend the representation of the $n \times n$ table where each entry can be associated with multiple exponents, which comes from the coding for distributed computing of the single matrix multiplication.

For the single large-scale matrix multiplication, various works (*e.g.,* [7]–[9], [19]) have applied coding on the matrix partitioned into submatrices. We choose to couple entangled polynomial codes [9] into our coding framework, which is one of the state-of-the-art coding schemes for distributed matrix multiplication based on the polynomial. Given a matrix multiplication $A \cdot B$, we partition A and B into fh and hg submatrices, respectively. In other words,

$$A = \begin{bmatrix} A(1,1) & \dots & A(1,h) \\ \vdots & \ddots & \vdots \\ A(f,1) & \dots & A(f,h) \end{bmatrix}, \text{ and } B = \begin{bmatrix} B(1,1) & \dots & B(1,g) \\ \vdots & \ddots & \vdots \\ B(h,1) & \dots & B(h,g) \end{bmatrix}.$$

Hence

$$AB = \begin{bmatrix} \sum_{i=1}^{h} A(1,i)B(i,1) & \dots & \sum_{i=1}^{h} A(1,i)B(i,h) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{h} A(f,i)B(i,1) & \dots & \sum_{i=1}^{h} A(f,i)B(i,h) \end{bmatrix}.$$

By an entangled polynomial code,[2] we encode A and B into $\ddot{A}(x) = \sum_{i=1}^{f}\sum_{j=1}^{h} A(i,j)x^{j-1+(i-1)gh}$ and $\ddot{B}(x) = \sum_{i=1}^{g}\sum_{j=1}^{h} B(j,i)x^{(h+1-j)+(i-1)h}$. Hence,

$$\ddot{A}(x)\ddot{B}(x) = \sum_{i=1}^{f}\sum_{j=1}^{g}\sum_{l=2}^{2h}\sum_{m=\max\{1,l-h\}}^{\min\{h,l-1\}} A(i,m)B(h-l+m+1,j)x^{h(i-1+g(j-1))+l-2}.$$

In particular, when $l = h+1$, the fg corresponding coefficients become $\sum_{m=1}^{h} A_{i,m}B_{m,j}$, $1 \le i \le f$, $1 \le j \le g$. Hence, by decoding the coefficients of $\ddot{C}(x) \triangleq \ddot{A}(x)\ddot{B}(x)$, we can obtain the fg submatrices in AB and hence complete the matrix multiplication. As the degree of $\ddot{C}(x)$ is $fgh+h-2$, its coefficients can be obtained from any $fgh+h-1$ coded tasks with different values of x.

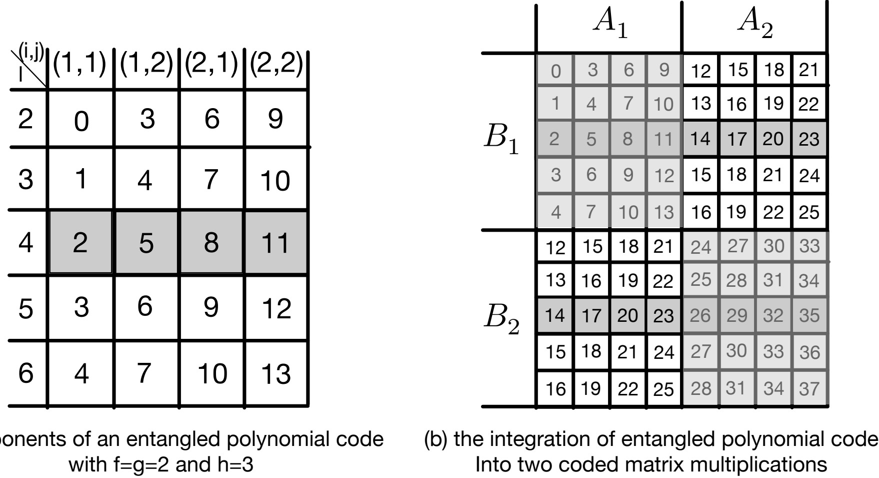


Fig. 12. The illustration of the exponents in an entangled polynomial code and its integration with two matrix multiplications

Similar to our coding framework, we represent the exponents in the entangled polynomial code in a $fg \times (2l-1)$ table. We can see in Fig. 12a that the exponents with $l = h+1$ are desired which need to be unique and other exponents can be shared, and thus we highlight the corresponding entries. We can then replace each entry of our coding framework with such an illustrative table of entangled polynomial code, leading to a two-level hierarchy of highlighted entries. The lower level corresponds to the entangled polynomial codes. We can see in Fig. 12b that each entry now contains 5×4 cells that correspond to the entries in Fig. 12a. The upper level, on the other hand, corresponds to RPC where the two diagonal entries in the table are also highlighted, *i.e.*, the two other entries can share the same exponents.

Formally, the coding scheme with the entangled polynomial code integrated can be defined as follows. We assume that the n matrix multiplications are split in the same way, where A_i is split into fh submatrices and B_i is split into hg submatrices,

[2]More detailed knowledge of entangled polynomial codes can be found in [9].

$i = 1, \ldots, n$. In an entangled polynomial code, the last exponent highlighted is $fgh - 1$. We then let $y = x^{fgh}$, so that all desired exponents can remain unique. The coded task can then be written as

$$\tilde{A}(x) = \sum_{l=1}^{n} \ddot{A}_{M_l}(x) y^{P_l} = \sum_{l=1}^{n} \sum_{i=1}^{f} \sum_{j=1}^{h} A_{M_l}(i, j) x^{j-1+(i-1)gh+fghP_l}$$

and

$$\tilde{B}(x) = \sum_{l=1}^{n} \ddot{B}_{N_l}(x) y^{Q_l} = \sum_{l=1}^{n} \sum_{i=1}^{g} \sum_{j=1}^{h} B_{N_l}(j, i) x^{(h+1-j)+(i-1)h+fghQ_l}.$$

Then we have $\tilde{A}(x)\tilde{B}(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} \ddot{A}_{M_i}(x) \ddot{B}_{N_j}(x) y^{P_i+Q_j}$. The largest exponent of x is achieved when $i = j = n$, which is $(R(n) - 1)fgh + (fgh + h - 2) = R(n)fgh + h - 2$, where $R(n)$ can be either $R_1(n)$ or $R_2(n)$ depending on which construction is used. In other words, the recovery threshold of RPC coupled with the entangled polynomial code is $R(n)fgh + h - 1$.

IX. Evaluation

We implement the code constructions of RPC with mpi4py. 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 $\tilde{A}(x)$ and $\tilde{B}(x)$ in RPC, we simply use Horner's method to encode input matrices after the code is constructed. Each worker then computes the multiplication of $\tilde{A}(x)$ and $\tilde{B}(x)$, where the value of x is uniquely chosen on each worker, and then upload the result to the master using MPI.send. The master will continuously polls (using MPI.Probe) to check if there is one worker which has finished one task. 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.

As a comparison, we also implement existing coding schemes for the batch matrix multiplication, including LCC [9], CSA [18], entangled polynomial (EP) [17], and GCSA [19] codes. LCC and CSA codes are designed for the coding across matrix multiplications, and EP[3] and GCSA codes are designed to also support matrix partitioning. To make the comparisons fair, we also apply RPC without or with matrix partitioning correspondingly in the experiments below.

We first run jobs of batch matrix multiplication without matrix partitioning on virtual machines hosted in Amazon EC2. The master runs on the virtual machine of type c4.4xlarge and all workers run on the virtual machine of type c4.2xlarge. We run a job of n matrix multiplications where the sizes of input matrices for each multiplication are 2000×30000 and 30000×2000. We run the job with 12, 14, 22, and 26 workers when n is 3, 4, 5, and 6, respectively. We 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.

From Fig. 13, we can see that RPC outperforms LCC and CSA codes significantly. Due to its simple polynomials, RPC saves the encoding time by up to 31.1% compared to LCC codes, and by up to 39.9% compared to CSA codes. Because of the

[3]EP codes are originally proposed for a single matrix multiplication [9] which is used in Sec. VIII. In [17], Yu and Avestimehr extend EP codes (without changing its name) for batch matrix multiplication, which we will compare with RPC in this section.

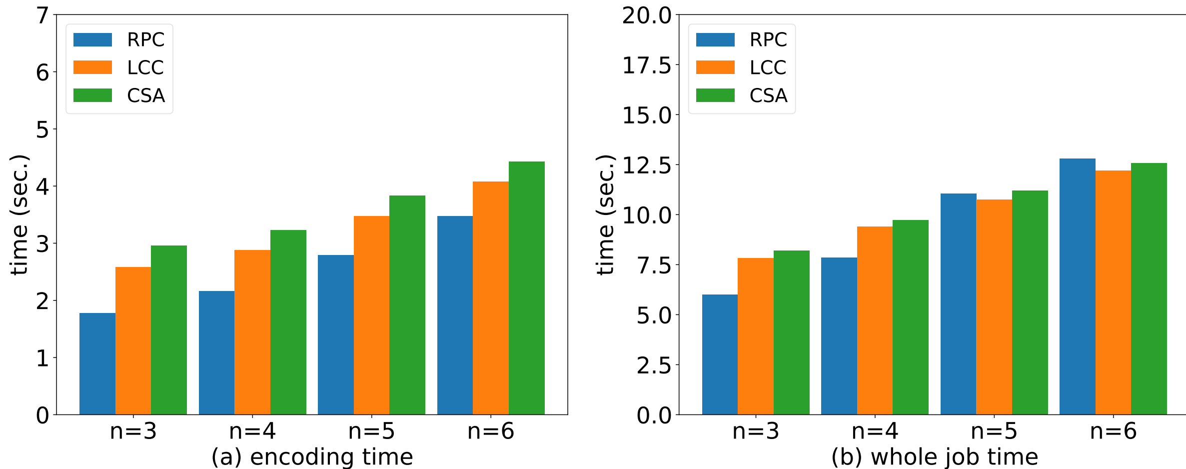


Fig. 13. Time of encoding and the whole job of n matrix multiplications without matrix partitioning.

low encoding time, even if RPC can tolerate fewer stragglers than LCC and CSA codes (due to its higher recovery threshold which we will elaborate on in Fig. 16), it still achieves lower completion time of the whole job when $n = 3$ (by 26.7%) and $n = 4$ (by 19.4%). Even with higher values of n, RPC still achieves completion time at the same level as LCC and CSA codes.

We also illustrate the encoding time and the completion time of the whole job for batch matrix multiplication with matrix partitioning in Fig. 14. We now run a job of n matrix multiplications with matrix partitioning where $f = h = 2$ and $g = 1$. The sizes of input matrices in each multiplication are 2000×45000 and 45000×1500. As the input matrices can be split in half, we now run the job on less powerful workers of type t2.xlarge in Amazon EC2. When $n = 3$ and $n = 4$, we run the job on 34 and 42 workers, respectively.

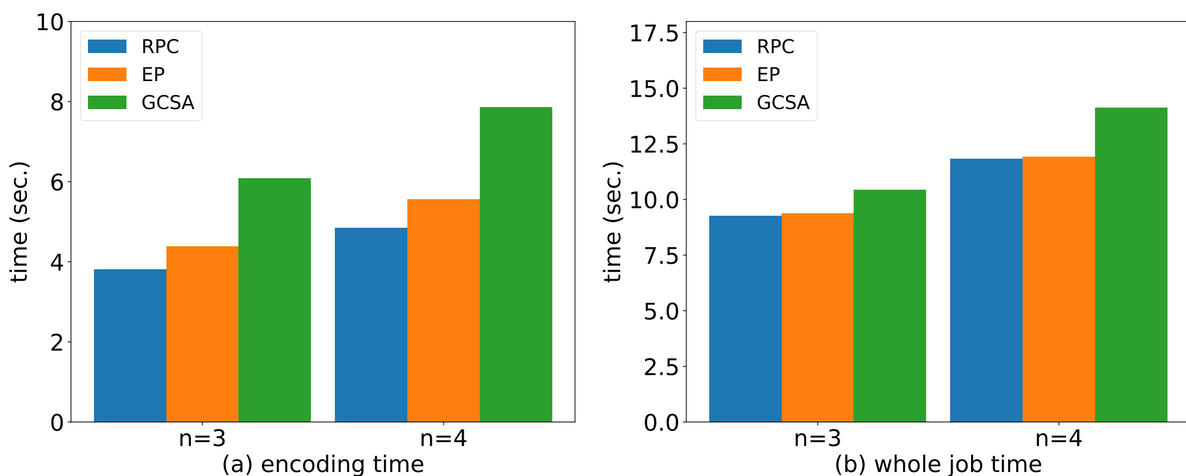


Fig. 14. Time of encoding and the whole job of n matrix multiplication with matrix partitioning.

In Fig. 14, we also see lower encoding time is achieved by RPC than EP codes and GCSA codes. RPC saves encoding time by up to 13.1% compared to EP codes, and by up to 38.3% compared to GCSA codes. As for the completion time of

the whole job, RPC achieves very similar performance as LCC codes, while the difference between RPC and LCC codes is limited within 1.4%. They are both faster than GCSA codes by up to 16.1%, mainly because of their low encoding time.

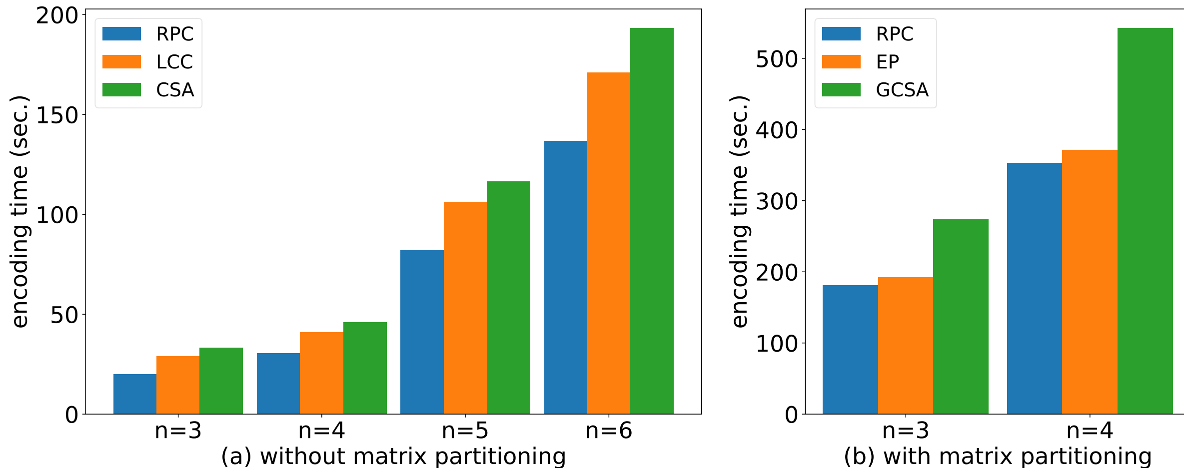


Fig. 15. The time of centralized encoding of LCC, CSA, EP, and GCSA codes.

As a comparison, we also run centralized version of the encoding algorithms in all the jobs above, where the master will encode all coded tasks and distribute such tasks to workers. Surprisingly, we find that the fast encoding algorithms for LCC, CSA, EP, and GCSA 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 decentralized encoding, because of the large number of matrices to encode. Hence, we use the naive algorithm in the centralized encoding in Fig. 15, which achieves even lower encoding time than the fast encoding algorithms when n is small.

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

We finally take a look at the recovery thresholds of the coding schemes we used in the experiments above and present the data in Fig. 16. In exchange for lower time in decentralized encoding, RPC 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 can see in Fig. 13 and Fig. 14, the time to complete the whole job with RPC is not compromised by the higher recovery threshold. Instead, its low time of encoding compensates any additional time of computation. The completion time of the whole job with RPC is at least at the same level as other coding schemes, and even better in many cases.

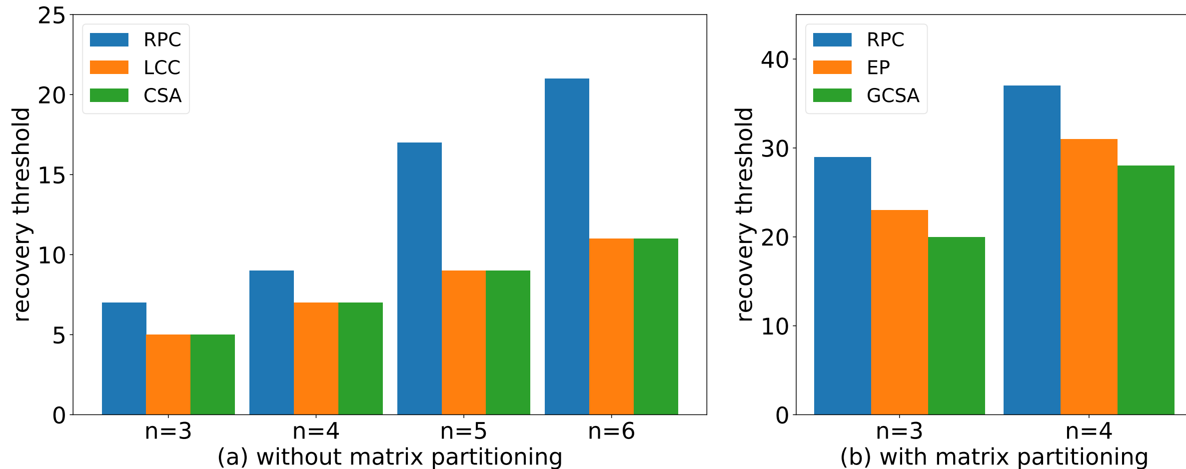


Fig. 16. Recovery thresholds for coding schemes used in Fig. 13 and Fig. 14.

X. Conclusion

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 (RPC), a coding framework for batch matrix multiplication, constructed towards saving the time of decentralized encoding. We demonstrate that compared to existing schemes, RPC can save the time of encoding and achieve lower completion time of the job.

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] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[6] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.

[7] 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.

[8] 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.

[9] ——, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2022–2026.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and A. S. Avestimehr, "Lagrange Coded Computing: Optimal Design for Resiliency, Security, and Privacy," in *Workshop on Systems for ML and Open Source Software at NeurIPS 2018*, 2018.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] J. Zhu and X. Tang, "Secure Batch Matrix Multiplication From Grouping Lagrange Encoding," *IEEE Communications Letters*, 2020.

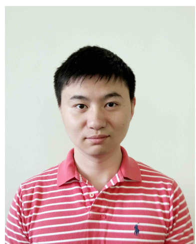
[25] H. A. Nodehi and M. A. Maddah-Ali, "Secure Coded Multi-Party Computation for Massive Matrix Operations," *arXiv:1908.04255 [cs.IT]*, 2019.

[26] ——, "Limited-Sharing Multi-Party Computation for Massive Matrix Operations," in *IEEE International Symposium on Information Theory (ISIT)*, 2018.

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.