Sequence-Aware Coding for Leveraging Stragglers in Coded Matrix Multiplication

Xiaodi Fan*, Pedro Soto†, Yuchun Zou*, Xian Su*, Jun Li‡
*The Graduate Center, City University of New York
†University of Oxford
‡Queens College and the Graduate Center, City University of New York

***Abstract*—Matrix multiplication is a foundational building block in numerous artificial intelligence algorithms. With the fast-increasing sizes of workloads, it is common to split the job of matrix multiplication into multiple tasks and execute them on different servers in parallel. However, stragglers which perform significantly slower than other servers are inevitable in distributed computing. Running coded tasks can tolerate the same number of stragglers with much fewer servers compared to replicating tasks on multiple servers.**

Although stragglers only partially complete their tasks, they can also be utilized toward a faster completion of the job, by uploading the results of sub-tasks split from each task. Existing designs utilizing partially completed tasks assume that all sub-tasks have an equal probability of being incomplete and require all input data to generate coded sub-tasks. However, not any arbitrary placement of incomplete sub-tasks is valid when sub-tasks are executed sequentially. If we only consider valid placements of incomplete sub-tasks, each coded sub-tasks can be encoded from much less input data, thus significantly saving the encoding complexity. In this paper, we introduce a new coding scheme called Sequence-Aware Coding (SAC), which exploits only valid placements of incomplete sub-tasks to reduce the encoding complexity while still leveraging the results of sub-tasks on stragglers. Experiment results show that SAC can reduce job completion time by up to 80.3% thanks to its lower encoding and decoding time.

I. Introduction

Matrix multiplication is a foundational building block in numerous artificial intelligence algorithms. With the fast-increasing sizes of datasets and complexities of models, the size of matrix multiplication also grows quickly. Therefore, it is challenging for one server to perform matrix multiplication when the input matrices are massive. To solve this challenge, it is common practice to split the job of matrix multiplication into multiple tasks which multiply submatrices of input matrices, and execute them on different servers called *workers* in parallel.

However, it is well known that in distributed computing, some workers, known as *stragglers*, may perform significantly slower than others and become the bottleneck of the job due to various reasons such as hardware issues, resource contention, and load imbalance [1]–[4]. In Amazon EC2, the performance of stragglers can be 2-5x slower than regular workers [2], [3].

One common method to mitigate the effects of stragglers is to launch redundant tasks on additional workers [5]–[11]. However, we need to run replicated tasks on $r+1$ workers to tolerate any r stragglers. On the contrary, *coded tasks* [2], [12], [13] have been proposed to get the results by decoding the results from other workers except stragglers. Compared to replicated tasks, coded tasks can tolerate the same number of stragglers with fewer additional tasks.

In most existing coding schemes for matrix multiplication, results from stragglers are typically discarded. As a result, computational resources on stragglers are wasted. Therefore, there have been coding schemes and techniques proposed to leverage partially completed results from stragglers [10], [14]–[28]. The common idea of these works is further partitioning each task into *sub-tasks*. We hence call sub-tasks directly partitioned from an uncoded task *uncoded sub-tasks*, which can then be encoded into *coded sub-tasks* to tolerate stragglers. Each worker can then upload the results of completed sub-tasks instead of the whole task. However, existing coding schemes that exploit partial results from workers assume that all sub-tasks have an equal probability of being incomplete. Therefore, coded sub-tasks are encoded from all uncoded sub-tasks. Nevertheless, if the execution order of sub-tasks is fixed, each sub-tasks won't have an equal probability of being incomplete. In other words, many arbitrary placements of incomplete uncoded sub-tasks don't exist in practice.

In this paper, we propose Sequence-Aware Coding (SAC), which respects the temporal sequence in which sub-tasks are executed. Instead of using all uncoded sub-tasks to generate coded sub-tasks, SAC can adjust the encoding locality and encode each coded sub-task from a subset of all uncoded sub-tasks. Therefore, SAC can leverage partial results from stragglers with a much lower encoding complexity. We implement SAC for distributed matrix multiplication with mpi4py. Our experiments demonstrate that compared to global MDS codes (GLO) [15] and C^{3}LES [16], SAC can save job completion time by up to 80.3%.

II. Motivating Examples

We now show a toy example of matrix-vector multiplication Ax in Fig. 1. We assume that the job of Ax is originally divided into three tasks A_1x, A_2x, and A_3x, running in a master-worker architecture. To leverage resources from stragglers, the tasks are further partitioned into two uncoded sub-tasks A_i^1x and A_i^2x, $i=1,2,3$, and sub-tasks are placed at the top of each worker. The three coded sub-tasks P_i^1x, $i=1,2,3$, can be encoded from all uncoded sub-tasks or a subset of them, and we place them at the bottom of each worker. We assume

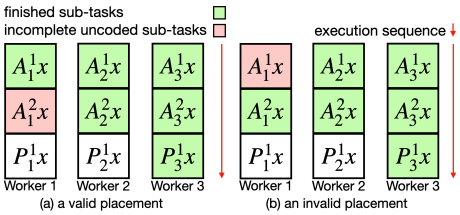

Fig. 1: Toy examples with a valid and an invalid placement of incomplete uncoded sub-tasks.

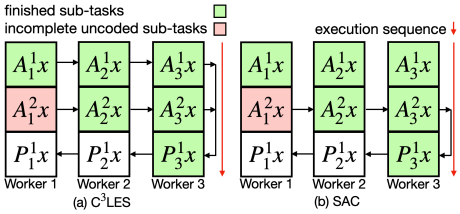

Fig. 2: A comparison between C^{3}LES and SAC. The black arrow shows how coded sub-tasks are encoded from uncoded sub-tasks. The red arrow shows the execution order of sub-tasks.

that sub-tasks on each worker are executed from top to bottom, unless terminated by the master.

Fig. 1a shows an example that the job is completed with one uncoded sub-task incomplete. Therefore, P_3^1 needs to be encoded from A_1^2 such that A_1^2x can be recovered by decoding. In fact, A_i^2, $i = 1, 2, 3$, should all be encoded into P_j^1, $j = 1, 2, 3$, as any one of them can be incomplete. However, as shown in Fig. 1b, it may not always be necessary to encode A_i^1 into P_j^1, $i, j = 1, 2, 3$, especially when the order of execution is determined. For example, if there is only one uncoded sub-task incomplete, the only possible placement is A_i^2x, $i = 1, 2, 3$. If A_i^1x is incomplete, $i = 1, 2, 3$, all of its consecutive sub-tasks must also be incomplete. Therefore, the placement of incomplete uncoded sub-tasks in Fig. 1b is invalid, and encoding A_i^1 into P_j^1 may not be necessary, $i, j = 1, 2, 3$.

Following the observation above, we can reduce the number of uncoded sub-tasks needed in encoding, while enabling the coded sub-tasks to recover the result from valid placements of incomplete uncoded sub-tasks only. We use Fig. 2 to illustrate our proposed coding scheme, SAC, and compare it with C^{3}LES in which P_i^1 is encoded from all uncoded sub-tasks, $i = 1, 2, 3$. Assuming that there is at most one uncoded sub-task incomplete, SAC and C^{3}LES can both recover Ax after receiving any six sub-tasks. However, SAC saves the encoding complexity by 50% as each coded sub-task is encoded from three uncoded sub-tasks only.

III. System Model

In this paper, we consider a distributed computing job with one master and w workers. The objective of the job is to compute Ax.[1] The job will be executed on w workers and Worker i executes the task that multiplies A_i with x, $i = 1, \ldots, w$. In order to exploit partial results of tasks, we further partition each of $A_1, \ldots, A_w$ into p sub-matrices, denoted as $A_i^1, A_i^2, \ldots, A_i^p$, $i = 1, \ldots, w$. We place $A_1^j, A_2^j, \ldots, A_w^j$ on the j^{th} row of uncoded sub-tasks, $j = 1, \ldots, p$. On each worker, we also generate t coded sub-tasks $P_i^1, P_i^2, \ldots, P_i^t$, $i = 1, \ldots, w$, and place them below uncoded sub-tasks. We similarly place $P_1^j, P_2^j, \ldots, P_w^j$, $j = 1, \ldots, t$, on the j^{th} row of coded sub-tasks. The rows of uncoded and coded sub-tasks of the job are illustrated in Fig. 3. We assume that each task starts from uncoded sub-tasks and continues to run coded sub-tasks afterwards. The result of each sub-task is sent to the master when it is complete. The execution can be terminated by the master when results received from the master suffice to decode. Specifically, we use (i, j) to denote the i^{th} sub-task performed by the j^{th} worker, such that (i, j) corresponds to $A_j^i x$ on Worker j if $i \le p$, and $P_j^{i-p}x$ otherwise.

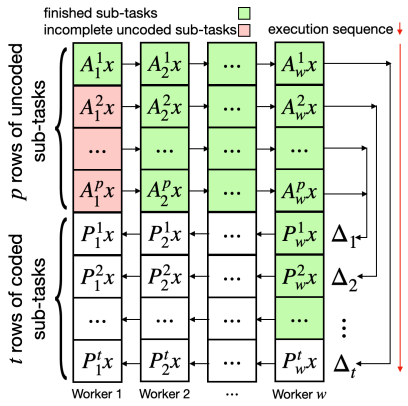

Fig. 3: The system model of SAC. The black arrow shows how coded sub-tasks are encoded from uncoded sub-tasks.

Since sub-tasks are executed sequentially, we then have the following observation.

Observation 1. *Worker j performs its computation in the order of $(1, j), (2, j), \ldots, (p + t, j)$. In particular, if (i, j) is completed, (i', j) must also be completed if $1 \le i' < i$.*

As the coded sub-tasks need to be encoded from uncoded sub-tasks, we define $\Delta = (\Delta_1, \ldots, \Delta_t)$ where Δ_i is the set of rows of uncoded sub-tasks where P_j^i, $j = 1, \ldots, w$, (*i.e.*, sub-tasks in the i^{th} row of coded sub-tasks) are encoded from, $1 \le i \le t$. Hence, we have $\Delta_i \subset \{1, 2, \ldots, p\}$. For example, if P_1^1 is encoded from the p^{th} uncoded sub-task row, *i.e.*, $A_1^p, \ldots, A_w^p$, we have $\Delta_1 = \{p\}$. If all coded sub-tasks are encoded from all uncoded sub-tasks, *e.g.,* in GLO and C^{3}LES, we have $\Delta_i = \{1, \ldots, p\}$

When the master receives enough sub-tasks for decoding, we assume that there are ϵ uncoded sub-tasks incomplete, which is also the number of received coded sub-tasks for

[1]We use matrix-vector multiplication for simplicity as only A needs to be encoded in this case. SAC can be extended to matrix-matrix multiplication.

decoding. We then define the *sequence property* below.

Definition 1. *(Sequence property.) We can recover ϵ incomplete uncoded sub-tasks in any valid placement using ϵ received coded sub-tasks in any valid placement. The master can receive more than ϵ coded sub-tasks, but only the first ϵ coded sub-tasks received can be used for decoding. A placement is valid if the order of sub-tasks does not violate Observation 1.*

Therefore, the objective of SAC is to construct Δ to achieve the sequence property while optimizing the encoding complexity by minimizing $\sum_{i=1}^{t} |\Delta_i|$. To achieve this objective, the parameters of SAC include the number of workers w, the number of rows of uncoded sub-tasks p, the number of rows of coded sub-tasks t, and the number of incomplete uncoded sub-tasks ϵ.

We introduce two methods to construct Δ in this paper. The first method (SAC-I) works for the general choice of (w, p, t, ϵ). The second method (SAC-II) can generate Δ when $p = \epsilon = w - 1$, $t \geq 2$, and $w \geq 1$, which may construct Δ with $\sum_{i=1}^{t} |\Delta_i|$ being lower.

With Δ given by SAC-I or SAC-II, we use Δ_i to generate P_j^i in the i^{th} row of coded sub-tasks, $1 \leq j \leq w$. Note that Δ_i is a set, and thus we let $q_i = |\Delta_i|$ and $\delta_{i,j}$ as the j^{th} entry in Δ_i. Coded sub-tasks in this row are then encoded with a systematic $(wq_i + w, wq_i)$ MDS code. Assume the MDS code has a generator matrix $G = (g_{i,j})_{(wq_i+w)\times wq_i}$ with an identity matrix in the top wq_i rows, we have $P_j^i = \sum_{l=1}^{q_i} \sum_{m=1}^{w} g_{j,(l-1)w+m} A_m^{\delta_{i,l}}$. Note that the generator matrix of MDS codes for different rows of coded sub-tasks should be constructed differently, which is not a challenge when entries in the input are real numbers.

SAC is constructed such that we can start decoding when the number of incomplete uncoded sub-tasks in rows covered by Δ is ϵ, the number of received coded sub-tasks is greater than or equal to ϵ, and all uncoded sub-tasks in the rows that are not covered by Δ are received. Since we have ϵ coded sub-tasks completed, the corresponding linear system has ϵ equations, in which we have ϵ coded sub-tasks on one side, and ϵ incomplete uncoded sub-tasks and wp - ϵ received uncoded sub-tasks on the other side. Since all generator matrices for each Δ_i are unique, this linear system is solvable.

We introduce the details of SAC-I and SAC-II in the rest of this paper.

IV. Constructing Δ with SAC-I

As there is a large number of combinations of (w, p, t, ϵ), it is challenging to find a general formula to describe the construction of the optimal Δ for all these combinations. For example, we manually find the optimal Δs for some examples of (w, p, t, ϵ) in Fig. 4. Therefore, we perform simulations to generate all valid placements of incomplete uncoded sub-tasks and complete coded sub-tasks, and test all possible designs of Δ for a range of (w, p, t, ϵ). Based on the empirical results, we develop a search-based algorithm to construct Δ for a general value of (w, p, t, ϵ).

(w, p, t, ϵ)	Δ_1	Δ_2
$(5, 3, 2, 3)$	$\{1, 2, 3\}$	$\{3\}$
$(5, 3, 2, 4)$	$\{2, 3\}$	$\{1, 3\}$
$(5, 3, 2, 5)$	$\{1, 2\}$	$\{3\}$
$(5, 4, 2, 4)$	$\{1, 2, 3, 4\}$	$\{1, 4\}$
$(5, 4, 2, 5)$	$\{1, 2, 3\}$	$\{4\}$
$(5, 4, 2, 6)$	$\{2, 3\}$	$\{1, 2, 4\}$

Fig. 4: The optimal Δs for some examples of (w, p, t, ϵ).

The value of (w, p, t, ϵ) is not entirely arbitrary, with the following requirement.

Theorem 1. *The values of w, p, t, and ϵ must satisfy $\lceil \frac{\epsilon}{p} \rceil + \lceil \frac{\epsilon}{t} \rceil \leq w$, $1 \leq w$, $1 \leq p$, $1 \leq t$, and $0 \leq \epsilon$.*

Proof. The ϵ incomplete uncoded sub-tasks can be placed on at least $\lceil \frac{\epsilon}{p} \rceil$ workers, and similarly the ϵ received coded sub-tasks can be placed on at least $\lceil \frac{\epsilon}{t} \rceil$ workers. A worker can either have incomplete uncoded sub-tasks or received coded sub-tasks before decoding, as uncoded sub-tasks are always executed beforehand. Therefore, the two numbers added should never be larger than w. □

By rewriting the inequality in Theorem 1, we get $0 \leq \epsilon \leq t(w - \lceil \frac{\epsilon}{p} \rceil)$. This reveals that when w, p, and t are fixed, the maximum value of ϵ is $t(w - \lceil \frac{\epsilon}{p} \rceil)$.

We use a list $L_{uncoded} = [n_1, n_2, n_3, \cdots, n_p]$ to describe how the ϵ incomplete uncoded sub-tasks are distributed on the w workers, where n_i denotes the number of incomplete uncoded sub-tasks in the i^{th} row of uncoded sub-tasks. Similarly, we maintain another list $L_{coded} = [m_1, m_2, m_3, \cdots, m_t]$, where m_i represents the number of received coded sub-tasks for decoding in the i^{th} row of coded sub-tasks. By Observation 1, if an uncoded sub-task A_j^i is incomplete, all the following uncoded sub-tasks on this worker must also be incomplete. Therefore, the number of incomplete uncoded sub-tasks in the i^{th} row of uncoded sub-tasks, $1 \leq i \leq p$, must be larger than or equal to that of its previous rows, *i.e.*, $0 \leq n_1 \leq n_2 \leq n_3 \cdots \leq n_p \leq \epsilon$. By Observation 1 again, if a coded sub-task P_j^i is received, all of its previous coded sub-tasks in this worker must have been received, *i.e.*, $0 \leq m_t \leq m_{t-1} \leq m_{t-2} \leq \cdots \leq m_1 \leq \epsilon$. In addition, $\sum_{i=1}^{t} m_i = \epsilon$ and $\sum_{i=1}^{p} n_i = \epsilon$ according to the sequence property.

The combinations that are hardest to cover are those where the incomplete uncoded sub-tasks are in the smallest number of workers, *i.e.*, $\lceil \frac{\epsilon}{p} \rceil$ workers, and the received coded sub-tasks are in the smallest number of workers, *i.e.*, $\lceil \frac{\epsilon}{t} \rceil$ workers. As long as we can cover the combinations that are hardest to cover, we can successfully cover all combinations.

Therefore, given (w, p, t, ϵ), we first generate $L_{uncoded}$ where all incomplete uncoded sub-tasks are in $\lceil \frac{\epsilon}{p} \rceil$ workers and all possible L_{coded} corresponding to each $L_{uncoded}$. We input the newly generated combinations of $L_{uncoded}$ and L_{coded} to Alg. 1. We name Δ generated in this step as Δ^1.

Alg. 1 uses $L_{uncoded}$ and L_{coded} as its input and generates one instance of Δ as its output. In Alg. 1 we always use the

sub-tasks from the rows of received coded sub-tasks with the highest row index to cover the incomplete uncoded sub-tasks with the lowest row index. If we use one coded sub-task in the i^{th} row of coded sub-tasks to cover one uncoded sub-task in the j^{th} row of uncoded sub-tasks, we add j to Δ_i.

Algorithm 1 Construction of Δ from $L_{uncoded}$ and L_{coded}

Input $L_{uncoded}$ and L_{coded}
Output Δ

$pointer$ = 1
keep increasing $pointer$ until $L_{uncoded}[pointer] \neq 0$
while $pointer \leq p$ **do**
  find the rows with received coded sub-tasks and put their row indexes to the array RowsWithReceivedCodedSubtasks in a descending order
  NumberOfUncodedSubtasksNeededToCover = $L_{uncoded}[pointer]$
  for i in RowsWithReceivedCodedSubtasks **do**
    if NumberOfUncodedSubtasksNeededToCover > $L_{coded}[i]$ **then**
      NumberOfUncodedSubtasksNeededToCover-= $L_{coded}[i]$
      $L_{coded}[i]$ = 0
      append $pointer$ to Δ_i
    else
      $L_{coded}[i]$ -= NumberOfUncodedSubtasksNeededToCover
      NumberOfUncodedSubtasksNeededToCover=0
      append $pointer$ to Δ_i
      $pointer$ += 1
      break
    end if
  end for
end while

We then generate another version of Δ as Δ^2. In this step, we generate L_{coded} such that all received coded sub-tasks are in $\lceil \frac{\epsilon}{t} \rceil$ workers and all possible $L_{uncoded}$ corresponding to each L_{coded}. We also input the newly generated combinations of $L_{uncoded}$ and L_{coded} to Alg. 1.

At last, we compute the union of Δ_i in Δ^1 and Δ_i in Δ^2, and get the final result of Δ.

V. Constructing Δ with SAC-II

SAC-I provides a construction of Δ for general values of (w, p, t, ϵ). However, there is still room to improve Δ's design with certain values of (w, p, t, ϵ). In this section, we present SAC-II, a different method to generate Δ. Using SAC-II. We can generate Δ when $p = \epsilon = w - 1$, $t \geq 2$, and $w \geq 1$. Different from SAC-I, we can theoretically prove that SAC-II is optimal when $p = \epsilon = w - 1$, $t = 2$, and $w \geq 3$.

A. Construction

In SAC-II, we simply let $\Delta_i = \{1, 2, 3, \cdots, p\}$, $i < t$, and $\Delta_t = \{p - \lfloor \frac{p}{t} \rfloor + 1, p - \lfloor \frac{p}{t} \rfloor + 2, p - \lfloor \frac{p}{t} \rfloor + 3, \cdots, p\}$, when $p = \epsilon = w - 1$, $t \geq 2$, and $w \geq 1$.

Theorem 2. *The design of SAC-II can achieve the sequence property.*

Proof. Since $0 \leq n_1 \cdots \leq n_p \leq \epsilon = p$, and $\sum_i^p n_i = p$, we can divide the rows of incomplete uncoded sub-tasks into t equal parts, and the sum of n_i for rows with higher indexes will be larger than those with lower indexes. Therefore, $n_1 + \cdots + n_{\lfloor \frac{p}{t} \rfloor} \leq \cdots \leq n_{p-3\lfloor \frac{p}{t} \rfloor + 1} + \cdots + n_{p-2\lfloor \frac{p}{t} \rfloor} \leq n_{p-2\lfloor \frac{p}{t} \rfloor + 1} + \cdots + n_{p-\lfloor \frac{p}{t} \rfloor} \leq n_{p-\lfloor \frac{p}{t} \rfloor + 1} + \cdots + n_p$. After that, we can get $p \leq (n_{p-\lfloor \frac{p}{t} \rfloor + 1} + \cdots + n_p)t$ and $\lfloor \frac{p}{t} \rfloor \leq n_{p-\lfloor \frac{p}{t} \rfloor + 1} + \cdots + n_p$.

Moreover, for received coded sub-tasks, we have $\sum_{i=1}^t m_i = \epsilon = p$, and $0 \leq m_t \leq \cdots \leq m_1 \leq \epsilon = p$. Hence, $m_t t \leq p$, and then $m_t \leq \lfloor \frac{p}{t} \rfloor$ as $t > 0$. Then we have $m_t \leq n_{p-\lfloor \frac{p}{t} \rfloor + 1} + \cdots + n_p$. We can always use received coded sub-tasks on the t^{th} row of coded sub-tasks to recover the same number of incomplete uncoded sub-tasks. After that, since $\Delta_i = \{1, 2, 3, \cdots, p\}$, $i < t$, which means that we use all uncoded sub-tasks to generate coded sub-tasks in the i^{th} row, $i < t$, the remaining incomplete uncoded sub-tasks can be decoded by the rest coded sub-tasks in the i^{th} row of coded sub-tasks, $i < t$. Thus, we can guarantee that there won't be any extra coded sub-tasks unused, and we can solve the corresponding linear system.

Hence, SAC-II can achieve the sequence property. □

B. Proof of optimality

We now prove that Δ constructed by SAC-II achieves the optimal encoding complexity when $p = w - 1 = \epsilon$, $t = 2$, and $w \geq 3$. The encoding complexity is measured by $\sum_{i=1}^t |\Delta_i| = |\Delta_1| + |\Delta_2|$.

Theorem 3. *Given any construction of Δ (that satisfies the sequence property) when $t = 2$, $w \geq 3$, and $p = w - 1 = \epsilon$, $|\Delta_1| + |\Delta_2| \geq p + \lfloor \frac{p}{2} \rfloor$.*

Proof. To determine the design of Δ_1, we first need to consider one special placement of the ϵ incomplete uncoded sub-tasks and ϵ received coded sub-tasks, *i.e.*, all the ϵ incomplete uncoded sub-tasks are on one worker, and all the ϵ received coded sub-tasks are in the first row of coded sub-tasks. In this case, each row of incomplete uncoded sub-tasks has one incomplete uncoded sub-task, and all the received coded sub-tasks are in the first row of coded sub-tasks. Then, if the number of elements in Δ_1 is smaller than p, there will be some incomplete uncoded sub-tasks uncovered. Therefore, Δ_1 must be $\{1, 2, \ldots, p\}$, which means that the first row of coded sub-tasks needs to cover all the rows of uncoded sub-tasks.

To design Δ_2 with the minimum size, we need to consider $m_1 + m_2 = \epsilon = p$ and $m_1 \geq m_2$. We can then get $m_2 \leq \lfloor \frac{p}{2} \rfloor$. Hence, when each row of uncoded sub-tasks has one incomplete uncoded sub-task, we need to design Δ_2 with at least $\lfloor \frac{p}{2} \rfloor$ elements to make sure all the received coded sub-tasks in the second row of coded sub-tasks are used to recover the same number of incomplete uncoded sub-tasks. Since $\Delta_1 = \{1, 2, \ldots, p\}$, we can use the remaining received coded sub-tasks to recover the rest of the incomplete uncoded sub-tasks and achieve the sequence property. Thus, the minimum number of elements in Δ_2 is $\lfloor \frac{p}{2} \rfloor$.

The proof is then complete by combining the two cases above. □

By Theorem 3, we know that Δ constructed by SAC-II achieves the optimal encoding complexity since the equality is achieved.

VI. Evaluation

We use mpi4py to implement a distributed matrix-matrix multiplication job AX in a master-worker architecture. The master partitions and encodes the matrix A only. The master then sends encoded matrices to corresponding workers and also broadcasts X to all workers. Each worker performs the matrix multiplication in each sub-task using the NumPy library. When a worker finishes one sub-task, it sends the result back to the master. Meanwhile, the master continuously checks if the result of any completed sub-task is sent back until the received results are sufficient for decoding. If so, the master stops receiving the results of sub-tasks and starts decoding. We run the experiments on 17 virtual machines on Google Cloud, including one of type c2-standard-60 as the master and 16 of type c2d-standard-2 as workers. We measure the performance of the following schemes in our experiments: GLO, C^{3}LES, SAC-I, and SAC-II. We repeat each experiment 20 times and present the average result.

In the experiments, we measure the time elapsed in different parts of the job, including encoding time in which the master spends to generate all sub-tasks, computation time in which tasks are executed until all sub-tasks are complete or the master terminates the execution, decoding time in which the master decodes received results, and job completion time which is the time spent to complete the whole job including encoding, computation, and decoding.

In the first experiment, we run three jobs with different sizes of input matrices as shown in Fig. 5. We choose $(w, p, t, \epsilon) = (16, 15, 2, 7)$ for SAC-I. We compare it with GLO and C^{3}LES with the same value of (w, p, t).

	A	X
SQUARE $\times$ SQUARE	4800 $\times$ 4800	4800 $\times$ 4800
THIN $\times$ FAT	240 $\times$ 500000	500000$\times$ 240
FAT $\times$ THIN	9120 $\times$ 200	200 $\times$ 9120

Fig. 5: Configurations of jobs.

As shown in Fig. 6, compared to GLO, SAC-I saves job completion time by 53.4% in SQUARE $\times$ SQUARE, 35.7% in THIN $\times$ FAT, and 80.3% in FAT $\times$ THIN. The saving of job completion time is mainly from encoding and decoding time.

As for encoding time, we first observe that $\sum_{i=1}^{t} |\Delta_i|$ in SAC-I's Δ is 53.3% fewer than GLO, which encodes coded sub-tasks from all uncoded sub-tasks. Therefore, SAC-I's encoding time is 45.4%, 44.9%, and 39.1% lower than GLO in SQUARE $\times$ SQUARE, THIN $\times$ FAT, and FAT $\times$ THIN, respectively. The code construction of SAC-I costs only 0.01 seconds. In terms of decoding time, compared to

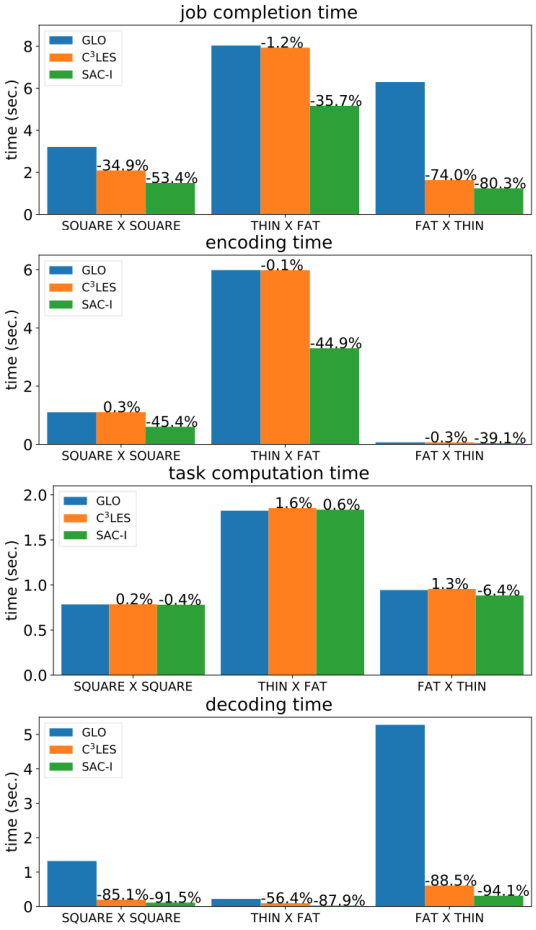

Fig. 6: Job completion time, encoding time, task computation time, and decoding time of SAC-I with $(w, p, t, \epsilon) = (16, 15, 2, 7)$, GLO, and C^{3}LES with the same (w, p, t).

GLO, SAC-I saves the decoding time by 91.5% in SQUARE $\times$ SQUARE, 87.9% in THIN $\times$ FAT, and 94.1% in FAT $\times$ THIN, respectively. The reason is that fewer sub-tasks are required in SAC-I's decoding process. The task computation time of SAC-I and GLO is almost the same because they eventually also receive similar numbers of sub-tasks, showing that SAC-I does not compromise decodability even with lower complexity. When we compare SAC-I with C^{3}LES, SAC-I saves job completion time by 28.4% in SQUARE $\times$ SQUARE, 34.8% in THIN $\times$ FAT, and 24.2% in FAT $\times$ THIN.

In the second experiment, we demonstrate that SAC-II optimizes the design of Δ generated by SAC-I and further saves job completion time. We set $(w, p, t, \epsilon) = (16, 15, 2, 15)$ which satisfies the requirement of SAC-II. We compare SAC-I and SAC-II with GLO and C^{3}LES with the same (w, p, t). The sizes of A and X are the same as SQUARE $\times$ SQUARE. Compared to GLO, SAC-I saves job completion time by 21.0%, and SAC-II further saves 11.1% in addition.

As shown in Fig. 7, SAC-II saves the encoding time by 23.7%. The reason is that SAC-II uses the least number of uncoded sub-tasks for encoding and thus enjoys the lowest

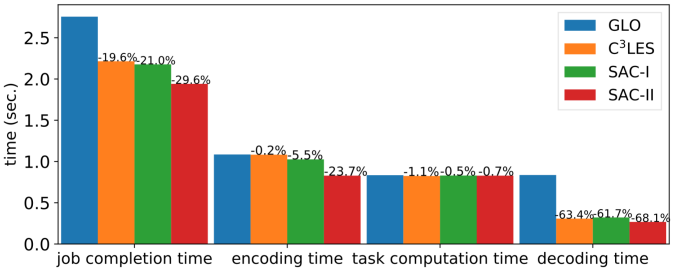

Fig. 7: Job completion time, encoding time, task computation time, and decoding time comparison of SAC-I, SAC-II, GLO, and C^{3}LES, where $(w, p, t) = (16, 15, 2)$ and $\epsilon = 15$ for SAC-I and SAC-II.

encoding complexity. The task computation time of GLO, C^{3}LES, SAC-I, and SAC-II are very similar since they eventually receive similar numbers of sub-tasks before decoding. Moreover, SAC-II achieves the lowest decoding time, which is 68.1% lower than GLO and 43.0% lower than C^{3}LES, as SAC-II performs the fewest operations of multiplication during decoding.

VII. Conclusions

Existing coding frameworks leverage partially completed tasks from stragglers by generating coded sub-tasks from all uncoded sub-tasks. In this paper, we propose a novel coding framework termed SAC with two methods SAC-I and SAC-II, constructed following the previously ignored fact that sub-tasks are executed in order. As a result, SAC can achieve significantly lower encoding and decoding complexities while still leveraging partial results from stragglers without compromise. With extensive experiments, we demonstrate that SAC can significantly reduce job completion time and also outperform other state-of-the-art coding frameworks for exploiting stragglers.

References

[1] 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.

[2] 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, 2017.

[3] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3368–3376.

[4] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[5] 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.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of The Clones," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 185–198.

[7] 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.

[8] 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.

[9] 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.

[10] K. Narra, Z. Lin, M. Kiamari, S. Avestimehr, and M. Annavaram, "Distributed Matrix Multiplication Using Speed Adaptive Coding," *arXiv preprint arXiv:1904.07098*, 2019.

[11] N. Ferdinand, B. Gharachorloo, and S. C. Draper, "Anytime Exploitation of Stragglers in Synchronous Stochastic Gradient Descent," in *IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018.

[12] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 2100–2108.

[13] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," *IEEE Transactions on Information Theory*, vol. 66, no. 3, pp. 1920–1933, 2020.

[14] X. Fan, P. Soto, X. Zhong, D. Xi, Y. Wang, and J. Li, "Leveraging Stragglers in Coded Computing with Heterogeneous Servers," in *IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 2020, pp. 1–10.

[15] S. Kiani, N. Ferdinand, and S. C. Draper, "Exploitation of Stragglers in Coded Computation," in *IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1988–1992.

[16] A. B. Das, L. Tang, and A. Ramamoorthy, "C^{3}LES: Codes for Coded Computation that Leverage Stragglers," in *IEEE Information Theory Workshop (ITW)*. IEEE, 2018, pp. 1–5.

[17] N. Ferdinand and S. C. Draper, "Hierarchical Coded Computation," in *IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1620–1624.

[18] S. Li, S. M. M. Kalan, Q. Yu, M. Soltanolkotabi, and A. S. Avestimehr, "Polynomially Coded Regression: Optimal Straggler Mitigation via Data Encoding," *arXiv preprint arXiv:1805.09934*, 2018.

[19] A. Ramamoorthy, L. Tang, and P. O. Vontobel, "Universally Decodable Matrices for Distributed Matrix-vector Multiplication," in *IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2019, pp. 1777–1781.

[20] S. Kiani, N. Ferdinand, and S. C. Draper, "Hierarchical Coded Matrix Multiplication," in *16th Canadian Workshop on Information Theory (CWIT)*. IEEE, 2019, pp. 1–6.

[21] E. Ozfatura, S. Ulukus, and D. Gündüz, "Distributed Gradient Descent with Coded Partial Gradient Computations," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 3492–3496.

[22] Y. Sun, J. Zhao, S. Zhou, and D. Gunduz, "Heterogeneous Coded Computation Across Heterogeneous Workers," in *IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.

[23] D. Kim, H. Park, and J. Choi, "Optimal Load Allocation for Coded Distributed Computation in Heterogeneous Clusters," *arXiv preprint arXiv:1904.09496*, 2019.

[24] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Coded Computation Over Heterogeneous Clusters," *IEEE Transactions on Information Theory*, vol. 65, no. 7, pp. 4227–4242, 2019.

[25] K. G. Narra, Z. Lin, M. Kiamari, S. Avestimehr, and M. Annavaram, "Slack Squeeze Coded Computing for Adaptive Straggler Mitigation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–16.

[26] E. Ozfatura, D. Gündüz, and S. Ulukus, "Speeding Up Distributed Gradient Descent by Utilizing Non-persistent Stragglers," in *IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2019, pp. 2729–2733.

[27] Y. Yang, M. Interlandi, P. Grover, S. Kar, S. Amizadeh, and M. Weimer, "Coded Elastic Computing," in *IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2019, pp. 2654–2658.

[28] K. T. Kim, C. Joe-Wong, and M. Chiang, "Coded Edge Computing," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 237–246.