Leveraging Stragglers in Coded Computing with Heterogeneous Servers

Xiaodi Fan*, Pedro Soto*, Xiaomei Zhong†, Dan Xi*, Yan Wang†, Jun Li*
*School of Computing and Information Sciences, Florida International University
†School of Software, East China Jiaotong University

***Abstract*—With the increasing sizes of models and datasets, it has become a common practice to split machine learning jobs as multiple tasks. However, stragglers are inevitable when running a job on multiple servers. Compared to replicating each task on multiple servers, running coded tasks can tolerate the same number of stragglers with much fewer servers. However, additional results of tasks running on stragglers are typically disregarded in existing schemes of coded computing, incurring a waste of the resources on such servers.**

In this paper, we leverage the results of partially finished tasks. In existing designs that utilize partially finished tasks, they have only considered servers with homogeneous performance. However, in a typical distributed infrastructure, *e.g.*, a cloud, servers with heterogeneous configurations are common. Therefore, we propose Spinner which can efficiently utilize the results of partially finished tasks even on heterogeneous servers. Spinner works with existing coding schemes for matrix multiplication, a fundamental operation in various machine learning algorithms, and can efficiently assign the workload based on the performance of the corresponding server. Furthermore, Spinner can equivalently adapt the coding scheme for heterogeneous servers, aligned with the expected workload assigned to each server, and thus save the complexity of decoding. Combining the two strategies together, we demonstrate in our experiments that Spinner can improve the time of matrix multiplication by up to 84.0% and thus improve the time of linear regression by 40.7%.

I. Introduction

Modern distributed computing systems have made it possible to train machine learning models on a large dataset. In such systems, a machine learning job can be split into multiple *tasks* that are executed on a large number of servers called *workers*. However, it is inevitable that the progress of some workers, *i.e.*, a *straggler*, may lag significantly behind others in a distributed infrastructure, such as a cloud. It has been observed that virtual machines on Amazon EC2 may be $5\times$ slower than others of the same type [1], [2]. Therefore, the performance of distributed machine learning algorithms may not necessarily be improved by simply splitting a job into more tasks, as with more workers the overall progress is more likely to be delayed by stragglers.

One of the methods to mitigate the adversarial effects of stragglers is to launch redundant tasks on additional workers [3]–[9]. Fig. 1 illustrates examples of the matrix multiplication, a pervasive operation in machine learning models. We calculate AX on four workers in Fig. 1a. The matrix A is split into two submatrices, A_1 and A_2, and the tasks of A_1X and A_2X are replicated on two workers, respectively. Therefore, any single straggler among the total four workers can be tolerated without affecting the overall performance.

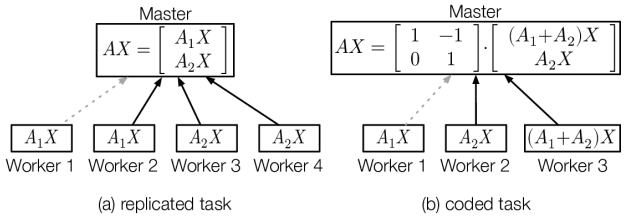

Fig. 1: Examples of distributed matrix multiplication with additional workers (running replicated or coded tasks) to tolerate one single straggler, represented with a gray dotted arrow.

However, replicating tasks on multiple workers significantly increases the resource consumption of distributed machine learning jobs. To tolerate any r stragglers, each task has to be replicated on $r+1$ workers. On the other hand, *coded tasks*, whose results can be decoded with results of other tasks, can tolerate stragglers with significantly fewer additional tasks than replication. In Fig. 1b, a third work executes a coded task $(A_1 + A_2)X$, which can be decoded to recover A_1X or A_2X if any other worker becomes 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.

In most existing designs of coded computing, however, the results of tasks running on a straggler are typically disregarded at the end of the job. In other words, a straggler is simply considered as a failed server, although performance degradation is more common on a straggler [10]. For example, in Fig. 1b, if Worker 2 and 3 have finished, the result of the task running on worker 1 will be disregarded even if it is just slightly slower than the other two servers. Moreover, even if all the three workers are not stragglers, only the results of two workers may be utilized and the other will be disregarded like a straggler since it is redundant to the other results. If there are only a small number of stragglers or the stragglers are slightly slower, a significant amount of resources will be wasted, which could have been utilized to achieve a lower job completion time.

This paper is based upon work supported by the National Science Foundation (No. CCF-1910447), the Science and Technology Project of the Department of Education, Jiangxi Province, China (No. 170384), and the National Science Foundation of Jiangxi Province, China (No. 20192BAB217006).

In order to leverage the resources on stragglers, it has been proposed that the worker can upload the result of a partially completed task instead of the whole task [1], [8], [9], [11]–[14]. However, such designs either require specific coding schemes with higher encoding and decoding overhead or need to know the stragglers in advance. Moreover, most of existing works on coded computing, including those that do not leverage stragglers, assume that all workers are homogeneous [1], [2], [15]–[25]. However, servers are more likely to be heterogeneous in distributed systems [26]. Hence, servers with lower hardware configurations will almost always be regarded as stragglers anyway, and their resources will almost always be wasted as well. Even worse, in order to decode the results of coded tasks, additional computation will be inevitable. For example, in Fig. 1b, additional matrix multiplication needs to be performed to get AX if Worker 1 is a straggler. If the size of AX is large, a significant delay will be incurred during decoding.

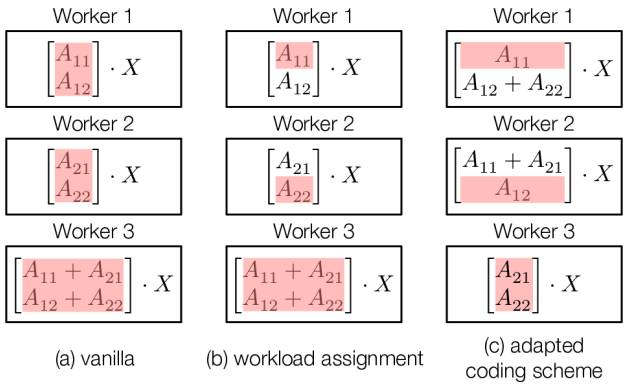

Fig. 2: Motivating examples of coded distributed matrix multiplication with one coded task on an additional worker (Worker 3).

In this paper, we propose Spinner, a coding framework for matrix multiplication that efficiently exploits the resources on stragglers among a cluster of heterogeneous servers. Instead of deploying a new coding scheme, Spinner can directly work with existing coding schemes without incurring any additional complexity. In Fig. 2, we show toy examples of multiplying a matrix A with a vector X. In Fig. 2a, the matrix is originally divided into two submatrices that are multiplied with the vector on two workers, and a third worker multiplies a coded matrix, which is the sum of the two submatrices, with the vector. Here we assume that the base performance of Worker 3, *i.e.,* the performance when it is not a straggler, is twice of Worker 1 and 2, respectively. For simplicity, we assume that the matrix on each worker has two rows. Conventionally, we need to have the results from any two workers to complete this job. As Worker 3 is faster than the other two workers, the result on Worker 1 or Worker 2 will be disregarded with a high probability, leading to a waste of resources even if it is not a straggler.

In order to fully utilize the heterogeneous performances of workers, workers in Spinner may execute their multiplications on different rows in Spinner, as highlighted in Fig. 2b where we can see that the workload on Worker 3 is twice of those on Worker 1 and 2. Since each row in the coded matrix on Worker 3 is a sum of the corresponding rows in the other two matrices, the multiplication results corresponding to the two missing rows on Worker 1 and Worker 2 can be decoded from the results on Worker 3. In this way, all workers can make contributions to the overall result of the job, and the job completion time in Fig. 2b can be 50% lower than that in Fig. 2a. When some worker becomes a straggler (relative to its base performance), Spinner will also dynamically adjust the workload in each task, still taking advantage of the resources of all workers even including the straggler.

Furthermore, unlike the conventional coding scheme for coded matrix multiplication where coded matrices are separated from uncoded ones (Fig. 2a and Fig. 2b), the coding scheme in Spinner can also be equivalently converted from an existing coding scheme, by taking the dynamic workload assignment into account, in order to make data in the original matrix be proportionally placed into all coded matrices according to the base performance of corresponding workers. We therefore, minimize the complexity of decoding the results from coded tasks and still tolerate the same number of stragglers as the original coding scheme. As shown in Fig. 2c, the matrices on the three workers all contain rows from the original matrix, whose numbers are also proportional to their performances. The rows multiplied on three workers in Fig. 2c are then all from the original matrix, avoiding the need for decoding. Moreover, the coding scheme in Fig. 2c is linearly equivalent to the coding scheme in Fig. 2a and Fig. 2b, and thus we can also recover the overall result from the other two workers if any one worker becomes a straggler.

We implement Spinner for distributed matrix multiplication with OpenMPI [27], and integrate it into a distributed linear regression job. Our experiments demonstrate that compared to existing coding schemes, Spinner can save the time of matrix multiplication by up to 84.0% and thus save the time of linear regression by up to 40.7%.

II. Related Work

Stragglers are inevitable in distributed systems, due to various reasons [28], [29] including network latency, resource contention, workload imbalance, failures of hardware or software, *etc*. Straggler detection and tolerance is an important topic in distributed computing. A straggler can significantly delay the progress of a distributed computing job waiting for all tasks running in parallel to finish. Stragglers can be detected when the job is running (*e.g.*, in Hadoop [4], [30]), and the affected tasks running on the stragglers will be relaunched. However, the relaunched tasks will probably still be completed later than other tasks, as they start later than others.

In order to fully mitigate the adversarial effects of stragglers, a state-of-the-art approach is adding redundant tasks in advance to tolerate tasks affected by potential stragglers. In this way, a task affected by a straggler can be disregarded

since the result of the same task can be obtained from other servers [3]–[7], [25].

However, replication incurs a significant resource consumption, in terms of both computation and storage, as all tasks need to be replicated. Therefore, coded tasks have been proposed to be added as redundant tasks rather than replicated tasks. A coded task takes coded data as its input and the result of the job can be decoded from a subset of all tasks so that the most straggling tasks can be disregarded. Such coded computing has been supported in some representative distributed machine learning algorithms, such as linear regression [2], gradient descent [2], [20], [24], and neural networks [21], [31], [32]. Similar paradigms have also been applied in distributed data analytics systems [15], [16], [33], [34].

In this paper, we consider the problem in coded computing that a significant amount of system resources may be wasted if the results of redundant tasks are simply disregarded. In order to leverage the resources on stragglers, a simple idea is to divide each task into multiple subtasks and encode the coded tasks at the subtask level [11], [14]. Besides matrix multiplication, this idea has also been applied in gradient descent [1], [9], [12]. However, such designs all rely on specific coding schemes which have much higher complexities than existing coding schemes for coded computing. Spinner, on the other hand, can work with most existing coding schemes, as well as the coding scheme we propose for Spinner which minimizes the decoding complexity. Although in this paper we will only discuss the case of MDS codes, it can be easily extended to support more coding schemes for coded computing, such as polynomial codes [25], [35], [36] and LDPC codes [37].

Narra *et al.* [8] and Yang *et al.* [13] have also proposed to assign workload to all non-straggling servers in order to leverage their resources. Although their methods can leverage more resources of servers if the number of stragglers is small, they cannot utilize the resources of stragglers, especially when the straggler is not a failed server. Even worse, they need to know the stragglers in advance. In Spinner the workload in each task is dynamically determined during the job so that there is no need to identify stragglers in advance.

Moreover, the works above have all focused only on homogeneous servers. If servers with lower hardware configurations are considered the same as other workers, they will be almost always considered as stragglers. Hence, we need specific designs to efficiently leverage the resources on heterogeneous servers. Reisizadeh *et al.* [20] and Sun *et al.* [38] have considered the scenario where servers are heterogeneous. However, the workloads on heterogeneous servers are statically assigned and they cannot leverage the stragglers among such heterogeneous servers. To the best of our knowledge, Spinner, for the first time, efficiently leverages the resources of all heterogeneous servers.

III. System Model

In this paper, we assume that we have a job to calculate AX where A and X are both matrices. The job will be executed on n workers that execute tasks A_iX, $i = 1, \ldots, n$. For example, if there are no redundant tasks, A_i will be one of the submatrices of A.

In order to support coded tasks, we consider (n, k) MDS (Maximum Distance Separable) codes in this paper. Given k matrices $A_1, \ldots, A_k$ divided from A,[1] an (n, k) code computes $r = n - k$ parity matrices, such that $A_{k+j} = \sum_{l=1}^{k} g_{k+j,l} A_l$, $j = 1, \ldots, r$, where the coefficient $g_{k+j,l}$ comes from a generator matrix G, *i.e.,*

$$G = \begin{bmatrix} & \mathbb{I}_k & \\ g_{k+1,1} & \cdots & g_{k+1,k} \\ \vdots & \ddots & \vdots \\ g_{k+r,1} & \cdots & g_{k+r,k} \end{bmatrix}.$$

In G, the top k rows, *i.e.*, $\mathbb{I}_k$, are a $k \times k$ identity matrix, *i.e.*, the first k tasks are actually uncoded. For simplicity, we use g_i to denote the i-th row in G, $i = 1, \ldots, n$. An (n, k) code is MDS if any k rows in G are linearly independent. Therefore, we can decode the results of coded tasks as follows,[2] as long as we have results from k tasks where $1 \leq i_1 < \ldots < i_k \leq n$:

$$\begin{bmatrix} A_1X \\ \vdots \\ A_kX \end{bmatrix} = \begin{bmatrix} g_{i_1} \\ \vdots \\ g_{i_k} \end{bmatrix}^{-1} \cdot \begin{bmatrix} A_{i_1}X \\ \vdots \\ A_{i_k}X \end{bmatrix}. \quad (1)$$

Note that in this paper we only consider the case that A is encoded with MDS codes [2]. Although code constructions have been proposed where in each coded task both A and X are encoded [25], [35], [39], the methods in Spinner can also be applied to such codes in general.

In this paper, we assume that the job in Spinner runs in a master-worker architecture. The master receives results from workers until the received results are sufficient for decoding. If a worker completes a task slower than its normal performance, we consider it as a straggler. In other words, the criteria of a straggler is based on its own hardware/software configurations. A worker will not necessarily be regarded as a straggler if it is slower than another worker, but will be if it is slower than its normal performance.

The master receives the results of the n tasks. Conventionally the results of any k tasks are expected for decoding. In practice, however, the results of more than k results will be received as the straggler may be caused by the network. Hence, the communication with all n workers should not be canceled until the results from k workers have been received by the master. In other words, results of much more than k workers may have been received (even all workers when there is no straggler) at the end of the job, which will be disregarded eventually. However, Spinner can further support receiving partial results of tasks in order to prevent collecting such unnecessary results, while still tolerating at most $n-k$ stragglers.

[1] If the size of X is larger than A, we can equivalently consider X^TA^T where X^T will be encoded.

[2] The equation works when A_i contains one row, otherwise each element $g_{i,j}$ in G should be replaced with $g_{i,j}\mathbb{I}_m$ where m is the number of rows in A_i. We assume in this paper that A_i contains one row without loss of generality.

Spinner can also convert the coding scheme accordingly so as to minimize the time of decoding.

IV. Workload Assignment in Spinner

In this section, we will present the workload assignment in Spinner, which leverages stragglers by utilizing partially completed tasks. Different from existing works [8], [13], Spinner allows determining the workload dynamically, aware of both heterogeneity and stragglers. We will first present a simple case with homogeneous workers and extend the discussion to heterogeneous workers.

A. Homogeneous Workers

In order to exploit partial results of tasks, in Spinner we further partition each of $A_1, \ldots, A_k$ into n submatrices, denoted as $A_{i,1}, \ldots, A_{i,n}$, $i = 1, \ldots, k$. The value of n can be arbitrarily chosen as long as the number of rows in A_i is divisible by n.[3] Equivalently, each coded matrix A_{k+j} can also be written as n submatrices $A_{k+j,1}, \ldots, A_{k+j,n}$, where $A_{k+j,m} = \sum_{l=1}^{k} g_{k+j,l} A_{l,m}$, $m = 1, \ldots, n$, is encoded from k submatrices in the same row as shown in Fig. 3. Note that here we do not change the coding scheme at all, as a coded matrix A_{k+j}, $1 \leq j \leq n-k$, remains the same if we combine $A_{k+j,1}, \ldots, A_{k+j,n}$ back together. Therefore, we can still recover all k matrices $A_1, \ldots, A_k$ as long as we have any k matrices among $A_1, \ldots, A_n$, and tolerate the same number of stragglers as (n, k) MDS codes described in Sec. III. Similarly, each task $A_i X$ can also be written as a series of subtasks including $A_{i,m} X$, $m = 1, \ldots, n$, and the job can still be completed as long as we have k tasks among $A_i X, i = 1, \ldots, n$, completed.

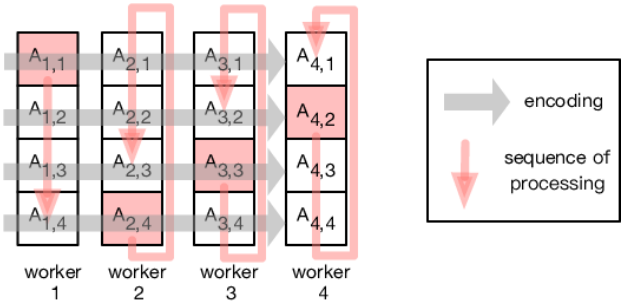

Fig. 3: An example of partitioning of $A_1, \ldots, A_4$ where $n = 4$ and $k = 3$. Horizontal arrows show that coded submatrices $A_{4,i}$ are encoded from other submatrices in the same row. Vertical arrows indicate the sequence of corresponding subtasks on a worker, where only the starting subtasks are highlighted.

However, in Spinner we do not let the master wait for the whole results from k workers since now each task contains n subtasks. Although conventionally each worker should complete all subtasks to finish its task, in Spinner we first ask each worker to start from different subtasks. Specifically, worker i will start from the subtask $A_{i,(((i-1)k) \bmod n)+1} X$. For example, when $k = n-1$, $(((i-1)k) \bmod n)+1 = ((1-i) \bmod n)+1$. Hence, Worker 1 will start from $A_{1,1}X$, Worker 2 from $A_{2,n}X$, Worker 3 from $A_{3,n-1}X$, ..., and Worker n from $A_{n,2}X$, as illustrated in Fig. 3. After finishing executing the first subtask of each worker, it will send the partial result of this subtask to the master, continue to execute the next subtask below or the first subtask if reaching the end, and eventually execute all subtasks if it is not stopped by the master. Note that the size of the result in each subtask is also $\frac{1}{n}$ of the whole task. Therefore, even though multiple messages are uploaded to the master, the overall communication overhead will not be increased. The master will keep receiving partial results from all workers until it realizes that they are sufficient to be decoded to get the result of AX. Now the overall result of AX can be recovered if all rows in Fig. 3 have at least k corresponding subtasks completed. If so, the master can stop all the incomplete tasks, and start decoding received results of their subtasks. The decoding algorithm is similar to (1), but will decode results of subtasks row by row.

In this way, we can maximize the parallelism to all workers. However, each worker will only go forward to its next subtask if needed. If there is no straggler, each worker is expected to complete k of its n subtasks and no more subtasks will be executed as they are redundant and will be disregarded anyway. Workers will continue only if there exists one or multiple stragglers so that their resources will not be wasted. In this way, no more communication is needed between workers and the master, except for uploading the result and terminating unfinished tasks. In Sec. IV-B, we will show that this mechanism can further be extended for heterogeneous workers and dynamically assign workload without additional communication overhead.

B. Heterogeneous Workers

We now assume that the performance of workers are heterogeneous. We define W_i as the base performance of Worker i,[4] *e.g.*, the number of tasks finished per unit of time, and then define w_i by normalizing W_i, *i.e.*, $w_i = \frac{W_i}{\sum_i W_i}$.

To extend the workload assignment above for heterogeneous workers, the major difference is that we consider the expected performance of each worker, by expecting a worker with a higher performance to naturally execute more subtasks. Therefore, we change the way of determining the starting subtasks of workers. We also allow the number of subtasks in all tasks, denoted by N, to be specified by the user. Hence, Worker i will start from $A_{i,j}X$, where $j = (\sum_{l=1}^{i-1} w_l k N) \bmod N + 1$. Note that N should be large enough such that for any $i = 1, \ldots, n$, $w_i k N$ is an integer. In practice, we can round $w_i k N$ to the nearest integer and we do not consider this case for simplicity.

In particular, $w_i k N$ is the expected number of subtasks executed by Worker i. In a special case where all workers are homogeneous, we will have $w_i = \frac{1}{n}$ and $N = n$ in Sec. IV-A. Each worker will then be expected to execute k subtasks, and

[3] Zero rows can be padded in A if the number of rows is not divisible by n.

[4] The base performance of a worker can be obtained by performance benchmarking or historical job completion time.

all n workers will then be expected to execute kn subtasks, which are sufficient to decode if there is no straggler. In this case, we can see that the algorithm in Sec. IV-A is simply a special case of this algorithm for heterogeneous workers. In other words, the initial subtasks to execute on each worker will be shifted by k subtasks from the previous worker.

Similarly, when we have heterogeneous workers, we expect Worker i to execute $w_i kN$ subtasks, and thus we move the starting subtasks down by $w_i kN$ when we consider the next worker. In this way, we can maximize the chance of decoding with $\sum_i w_i kN = kN$ expected subtasks, as each row in Fig. 3 will now have k subtasks executed. Otherwise, when there are no more than $r = n - k$ stragglers, other workers should continue to execute their following subtasks until the completed subtasks are sufficient to decode. We summarize the algorithms described above of the master and workers in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Master's algorithm.

1: $p\_ret = \langle\rangle$
2: **while** p_ret cannot be decoded **do**
3: receive a partial result y
4: $p\_ret = \langle p\_ret, y\rangle$.
5: **end while**
6: stop all workers
7: ret = decode(p_ret)
8: return ret

Algorithm 2 Worker's algorithm.

1: $j = (\sum_{l=1}^{i-1} w_i kN) \bmod N + 1$
2: **while** this task is not stopped by the master **do**
3: compute $y = A_{i,j}X$
4: send y to the master
5: $j = (j+1) \bmod N + 1$
6: **end while**

V. Performance Analysis

In this section, we analyze the performance of the workload assignment in Spinner. We assume that the time for a server to finish the whole job follows a runtime cumulative distribution function $F(t)$, such that the probability that the server can finish the job of calculating AX within an amount of time t is $F(t)$. We also assume that $f(t)$ is the corresponding probability density function, the expected job completion time on this server is $\mathbb{E}[t] = \int_{t=0}^{+\infty} t \cdot f(t)dt$. We further assume that the runtime distribution to complete a task can be scaled from $F(t)$, and thus the runtime distribution to finish a task of calculating $A_i X$ on this server will be $F(kt)$. When there is no redundant task, *i.e.*, $n = k$, and all other $n-1$ servers are the same as this one, the runtime distribution of the whole job will be $[F(kt)]^n$, as all n tasks need to be finished.

A. Conventional Coded Matrix Multiplication

In our analysis, we still start from homogeneous servers and then extend the analysis to heterogeneous servers. Conventional matrix multiplication does not divide tasks into subtasks, *i.e.*, $N = 1$. If all workers are homogeneous, we have $F_1(t) = \ldots = F_n(t) = F(t)$. The job can be finished when there are at least k tasks finished. Therefore, the runtime distribution of the whole job is

$$\mathbb{F}_{\text{homo}}^{N=1}(t) = \sum_{k_0=k}^{n} \binom{n}{k_0} (F(kt))^{k_0} (1 - F(kt))^{n-k_0}.$$

Note that we do not consider the time of decoding in this section, which is discussed in Sec. VI.

If workers are heterogeneous, it is now necessary to distinguish runtime distributions of different workers. We also use binary variables e_i, $i = 1, \ldots, n$ to indicate if Worker i finishes its task before the time t, *i.e.*, $e_i = 1$ if Worker i completes the task and $e_i = 0$ otherwise. Still, the job can be completed when there are at least k workers finishing their tasks, *i.e.*, $\sum e_i \geq k$. Hence, the runtime distribution of the job is

$$\mathbb{F}_{\text{het}}^{N=1}(t) = \sum_{\sum e_i \geq k} \prod_{i=1}^{n} \left(e_i F_i(kt) + (1 - e_i)(1 - F_i(kt))\right).$$

B. Spinner

Different from conventional coded matrix multiplication, in Spinner we need to consider that the matrix A_i is further split into submatrices. From Fig. 3 we can see that the job can be finished when in each row we have at least k subtasks executed.

Similar to $\mathbb{F}_{\text{homo}}^{N=1}(t)$, we can derive the runtime distribution of the j-th row. Note that for convenience, here we temporarily change the sequential index of each row so that it starts from 0 and ends with $n-1$, *i.e.*, $0 \leq j \leq n-1$.

$$\mathbb{R}_{\text{homo}}^{j}(t) = \sum_{\sum e_i \geq k} \left(e_i F\left(\frac{knt}{((j-(i-1)k) \bmod n) + 1}\right) + (1 - e_i)\left(1 - F\left(\frac{knt}{((j-(i-1)k) \bmod n) + 1}\right)\right)\right).$$

Because subtasks are sufficient for decoding when there are k subtasks completed in all $N = n$ rows, the overall runtime distribution of the job is $\mathbb{F}_{\text{homo}}^{N=n}(t) = \prod_{j=0}^{n-1} \mathbb{R}_{\text{homo}}^{j}(t)$.

When workers are heterogeneous, the major change will also be the sequence of subtasks. As each worker will shift the starting subtasks by $w_i kN$, we need to subtract such shifts when calculating the actual sequences of subtasks. Hence, we can get $\mathbb{R}_{\text{het}}^{j}(t)$, the runtime distribution of the j-th row, and the overall runtime distribution is $\mathbb{F}_{\text{het}}^{N}(t) = \prod_{j=0}^{N-1} \mathbb{R}_{\text{het}}^{j}(t)$.

$$\mathbb{R}_{\text{het}}^{j}(t) = \sum_{\sum e_i \geq k} \left(e_i F_i\left(\frac{knt}{((j - \sum_{l=1}^{i-1} w_l kN) \bmod N) + 1}\right) + (1 - e_i)\left(1 - F_i\left(\frac{knt}{((j - \sum_{l=1}^{i-1} w_l kN) \bmod N) + 1}\right)\right)\right).$$

We first prove the runtime distribution of a Spinner job in a homogeneous cluster. The job has results from sufficient subtask to decode if and only if there are k subtasks finished in all the $N = n$ rows. In each row, we use the binary variable e_i to indicate if the corresponding subtask of Worker i is finished. Hence, one row is completed if $\sum_i e_i \geq k$.

If $e_i = 1$, it means that the corresponding subtask is finished. In order to calculate the probability that it can be finished within an amount of time t, we must consider the execution sequence of subtasks of its worker. In other words, the fact that this task is finished infers that all of its previous subtasks in different rows must have also been finished. For example, if this task is the third subtasks when $n = 5$, it means that 60% of the task has been finished. We know that in a homogeneous cluster, each worker will execute k subtasks, and thus the first subtask to execute on Worker i should be $((i-1)k \bmod n)$-th subtask. Note that we have changed the sequence index from 0 to $n-1$. Now we know that the j-th subtasks finished on this worker means that there are already $((j-(i-1)k) \bmod n)+1$ subtasks finished. As there are n subtasks on a worker, the run time distribution of such subtasks finished is equivalent to that of $\frac{((j-(i-1)k) \bmod n)+1}{kn}$ of a job finished, and then we can have the scaled runtime distribution of such subtasks.

For the heterogeneous cluster, the only change is that we need to consider the number of tasks expected to execute on each worker, *i.e.*, $w_l kN$. Therefore, the first subtask executed on Worker i is $(\sum_{l=1}^{i-1} w_l kN) \bmod N$, and thus the j-th subtask completed means that $\left((j - \sum_{l=1}^{i-1} w_l kN) \bmod N\right) + 1$ tasks have finished. Therefore, we can get $\mathbb{R}_{\text{het}}^{j}(t)$.

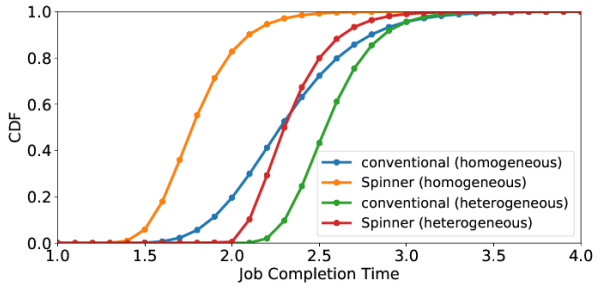

Fig. 4: A comparison of the job completion time between conventional coded matrix multiplication and Spinner.

C. Comparison

We use the analysis above to compare the performance between Spinner and conventional coded matrix multiplication with MDS codes (vanilla MDS), and matrix multiplication without coding. We assume that the runtime distribution of a worker follows a shifted-exponential distribution, *i.e.*, $F_i(x) = 1 - e^{-\mu_i(x-c_i)}, x \geq c_i$. Therefore, we have $W_i = \mathbb{E}[f_i(kt)] = k(\frac{1}{\mu_i} + c_i)$.

In our evaluation, we assume $n = 12$ and $k = 8$. We consider both two cases of homogeneous and heterogeneous workers. For the case that all workers are homogeneous, we let $c_i = 1$, $i = 1, \ldots, 20$. When workers are heterogeneous, we assume that $c_i = 3$ when $1 \leq i \leq 10$, and $c_i = 1$ when $11 \leq i \leq 20$. In this way, the baseline (non-straggling) performance of the first ten workers will be twice as that of the last ten workers.

We show the runtime cumulative distribution function of the whole job in Fig. 4. We can see that for both homogeneous workers and heterogeneous workers, Spinner can significantly improve the overall job completion time. For the medium job completion time, Spinner can save the time by 21.0% (homogeneous) and 24.5% (heterogeneous). For the tail latency, we also observe that the 90-percentile tail of the job completion time can be improved by 22.0% (homogeneous) and 21.5% (heterogeneous). In fact, when workers are heterogeneous, we can see in Fig. 4 that the top 36% of the jobs without coding may even be faster than those with conventional coded matrix multiplication. The reason is that with the same number of workers, the overhead of each task can be lower without coding, as the matrix A can now be divided into $n = 12$ matrices rather than $k = 8$ matrices. However, Spinner can always enjoy better job completion time thanks to its better parallelism.

VI. Coding Scheme in Spinner

A. Decoding Complexity

Another problem we aim to mitigate in Spinner is the complexity of decoding. When the result of the job is large, decoding results of coded tasks may significantly increase the overall job completion time. In conventional coded matrix multiplication, decoding results from coded tasks is inevitable, unless all uncoded tasks are executed by the k fastest workers. The more results from coded tasks, the more time it takes to decode them.

We first analyze the decoding complexity, in terms of the expected number of results from coded tasks, in conventional coded matrix multiplication. We assume that workers are randomly chosen regardless of their performance. Therefore, each worker has the same chance to become one of the fastest k workers. We assume that $n \leq 2k$, *i.e.*, the number of coded tasks is less than the number of uncoded tasks, which is a common practice in coded distributed computing. Then, the expected number of coded task to decode is

$$\sum_{x=0}^{n-k} \left(x \cdot \frac{\binom{n-k}{x}\binom{k}{k-x}}{\binom{n}{k}} \right) = \frac{k(n-k)}{n}.$$

We now consider the case where there is only one straggler, as in practice having one straggler is more likely than having more stragglers [20]. If only one worker becomes a straggler, which completes its task slower than its expected time, we prove that the expected number of coded tasks to decode does not change.

We only need to consider the case that the straggler should have been faster than at least $n-k$ other workers. Otherwise, the number of results from coded tasks should not change

at all. In this case, if the straggler is running a coded task (with a probability of $\frac{n-k}{n}$), then it has a chance of $\frac{k}{n-1}$ to be replaced by a worker running an uncoded task. Similarly, if the straggler is running an uncoded task (with a probability of $\frac{k}{n}$), then it has a chance of $\frac{n-k}{n-1}$ to be replaced by a worker running a coded task. Therefore, we can see that the expected number of changes is $\frac{n-k}{n} \cdot \frac{k}{n-1} - \frac{k}{n} \cdot \frac{n-k}{n-1} = 0$.

On the other hand, after applying the workload assignment algorithm in Spinner, the expected number of results from coded subtasks will be $\sum_{i=k+1}^{n} w_i kN$. The expected value of $\sum_{i=k+1}^{n} w_i$ is $\frac{n-k}{n}$ as $\sum_i w_i = 1$, and thus the expected number of results from coded subtasks is $\frac{k(n-k)N}{n}$, the same as that without Spinner.

We can also prove that the expected number of coded tasks will not change when there is one worker becoming a straggler. Considering one subtask missed by this straggler, applying a similar argument as above, we can prove that the expected number of uncoded subtasks replaced by coded subtasks equals the expected number of coded subtasks replaced with uncoded subtasks. Hence, the expected number of results from coded tasks will not change.

B. Coding scheme

By the analysis above, we can see that the workload assignment algorithm in Spinner cannot improve the decoding complexity. However, in Spinner, we can convert an existing coding scheme according to the performance heterogeneity, while the number of tolerable stragglers and the coding complexity will not change. The technique we use is known as the symbol remapping, which has been used in the construction of some existing erasure codes [33], [34], [40], [41].

For simplicity, we show a toy example of how symbol remapping works in Spinner. Assume that we originally have a $(3, 2)$ MDS code which encodes AX into three tasks: A_1X, A_2X, and $(A_1 + A_2)$X. In Spinner, we further partition each task into $N = 2$ subtasks. We show the generator matrix after partitioning and the equation of encoding in Fig. 5a .

In a generator matrix G, if a square submatrix G_0 is invertible, we can have a new code $\hat{G} = G \cdot G_0^{-1}$. From Sec. IV, we can see that the kN subtasks expected to complete in each worker are sufficient to decode, and thus their corresponding rows in the generator matrix are also an invertible submatrix. In Fig. 5a we highlight an example of such expected subtasks, assuming three heterogeneous workers. As shown in [40], the new code $\hat{G}$ will be linearly equivalent to the original code so that the decoding can be performed in the same way with the same complexity.

In Fig. 5b, we show a linearly equivalent code after applying symbol remapping from Fig. 5a. We can still tolerate any $n - k = 1$ straggler as the new code is still MDS, and recover AX with the kN subtasks if the corresponding subtasks in Fig. 5a can be decoded. Hence, the algorithms of workload assignment in Sec. IV will still work.

After symbol remapping, if none of the workers are stragglers, there will be no coded subtask to decode. Now we consider the case that one worker becomes a straggler. We assume

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_{1,1}X \\ A_{1,2}X \\ A_{2,1}X \\ A_{2,2}X \end{bmatrix} = \begin{bmatrix} A_{1,1}X \\ A_{1,2}X \\ A_{2,1}X \\ A_{2,2}X \\ (A_{1,1}+A_{2,1})X \\ (A_{1,2}+A_{2,2})X \end{bmatrix}$$

(a) before symbol remapping

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_{1,1}X \\ A_{1,2}X \\ A_{2,1}X \\ A_{2,2}X \end{bmatrix} = \begin{bmatrix} A_{1,1}X \\ (-A_{1,2}+A_{2,2})X \\ (-A_{1,1}+A_{2,1})X \\ A_{1,2}X \\ A_{2,1}X \\ A_{2,2}X \end{bmatrix}$$

(b) after symbol remapping

Fig. 5: An illustration of symbol remapping in Spinner where $n = 3$, $k = 2$, $N = 2$.

that when the job finishes, the straggler misses x subtasks, *i.e.,* the number of its completed subtasks is $w_i kN - x$. We can then infer that such missed subtasks will be replaced by the same number of coded subtasks, and thus the expected number of coded tasks to be decoded is also $\frac{x}{N} \in (0, w_i k)$. If each of the n workers has the same chance of becoming a straggler, the upper bound of the expected number of coded tasks to decode is $\frac{\sum_i w_i k}{n} = \frac{k}{n}$.

In Fig. 6, we summarize the expected number of coded tasks to be decoded of the three schemes discussed above. We can see that with symbol remapping, Spinner can improve the decoding complexity by at least $n-k$ times when there is one straggler. When the straggler is only slightly slower than its expected performance, the decoding complexity can be even lower.

scheme	no straggler	one straggler
conventional coded matrix multiplication	$\frac{k(n-k)}{n}$	$\frac{k(n-k)}{n}$
Spinner (workload assignment only)	$\frac{k(n-k)}{n}$	$\frac{k(n-k)}{n}$
Spinner (workload assignment + symbol remapping)	0	$\leq \frac{k}{n}$

Fig. 6: Comparison of the coded tasks of conventional coded matrix multiplication, Spinner with workload assignment only, and Spinner with workload assignment and symbol remapping.

VII. Evaluation

In this section, we present the empirical results of running Spinner-based distributed matrix multiplication on homogeneous and heterogeneous workers. We then integrate Spinner into linear regression which requires matrix multiplication and evaluate the performance with existing coding schemes.

	A	X
SQUARE $\times$ SQUARE	2880 $\times$ 2880	2880 $\times$ 2880
FAT $\times$ THIN	960 $\times$ 50000	50000 $\times$ 960
THIN $\times$ FAT	4800 $\times$ 100	100$\times$ 4800

Fig. 7: Configurations of jobs.

A. *Matrix multiplication*

We first implement a distributed matrix multiplication job that calculates the multiplication of AX using OpenMPI [27]. We first place corresponding matrices A_i and X on each worker. The master will then instruct each worker to multiply the two corresponding matrices in each task, and continuously polls (using MPI.Probe) to check if there is one worker which has finished one subtask. The matrix multiplication on each worker is calculated using the NumPy library. When a worker finishes one subtask, it will send the result back to the master using MPI.Send. When the master receives one result of a subtask, it will also check if all received results are sufficient to be decoded. If so, the master will stop receiving any new results and start decoding.

We run jobs of matrix multiplication on 25 virtual machines hosted in Microsoft Azure. One of such virtual machines is used as the master across all experiments, of type F4. The other 24 virtual machines are used as workers, and they have different types in different experiments below.

We measure the performance of the following schemes in our experiments: conventional matrix multiplication with MDS codes (vanilla MDS), Spinner with workload assignment only (SP-WA), and Spinner with both workload assignment and its coding scheme (SP). Specifically, in vanilla MDS we actually use Reed-Solomon codes, a popular family of MDS codes, to encode the matrix and decode results of tasks. Besides, we implement one more coding scheme (GLO) proposed by Kiani *et al.* that also supports subtasks [14]. Different from Reed-Solomon codes and Spinner, the matrix in each task or subtask is no longer encoded from k tasks or subtasks, but from all subtasks in k original tasks. Theoretically, GLO should have a better performance than Spinner as the results from any kN subtasks should be decoded. However, in our experiments, we observe that Spinner actually outperforms GLO because of the high decoding complexity of GLO and the workload assignment of Spinner.

For each scheme, we measure task completion time, which is the time that the master can get sufficient results for decoding, and decoding time, which is the time that the master spends to decode results from tasks or subtasks. The job completion time is then the sum of task completion time and decoding time. In each of the following figures, we repeat each experiment by 20 times and present the average result and the standard deviation.

We first study the job completion time and decoding time with different types of jobs. We run three jobs with different shapes of input matrices: SQUARE $\times$ SQUARE, THIN $\times$ FAT, and FAT $\times$ THIN. We show the sizes of A and X in Fig. 7. The values of n and k in all coding schemes are set to be 24 and 20. In this experiment, all 24 workers are virtual machines of type B4. We show the job completion time and task completion time of each scheme in Fig. 8, where the decoding time can be seen from the gap. We can see that in the three jobs, compared to vanilla MDS, SP-WA saves task completion time by 11.1%, 28.3%, and 24.2%, respectively. The decoding time in SP-WA is slightly higher than vanilla-MDS due to its subtasks. However, in SP its decoding time is much lower. With its coding scheme, the job completion time of SP is lower than vanilla-MDS by 53.6%, 35.2%, and 84.0% in the three jobs. Moreover, the task completion time of SP-WA and SP is rather close to that of GLO, which is expected to be optimal. However, since the decoding time of GLO is the highest among all schemes, its job completion time is even higher than SP.

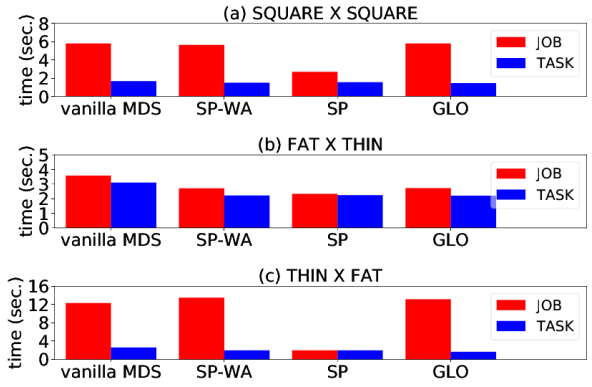

Fig. 8: Job completion time (JOB) and task completion time (TASK) in a homogeneous cluster with 24 workers.

We now run more experiments in heterogeneous clusters, which still have 24 workers. Each cluster has two types of virtual machines, of type B4 and F4. We run the same tasks on virtual machines of such two types, and observe that B4 is almost 1.5 times faster than F4. We create five clusters, where the ratio of virtual machines of type F4 increases from 0% to 25%, 50%, 75%, and eventually 100%. We run one job in all such clusters, where the sizes of two matrices are

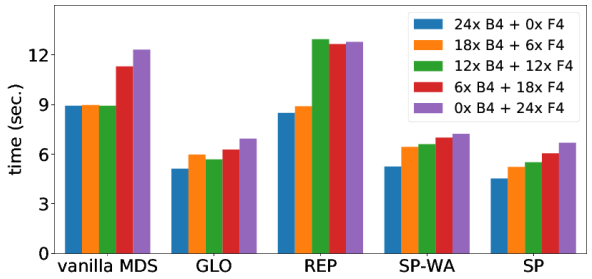

Fig. 9: Job completion time in heterogeneous clusters with 24 workers.

2880×50000 and 50000×640. The sizes of such two matrices, hence, are chosen to make the decoding time marginal in the job, and the job completion time is almost the same as the task completion time. We still let $n = 24$, but set $k = 12$. As $n = 2k$ this time, we add one more scheme of replication (REP), where each task is replicated on two workers.

In Fig. 9, we can see that with more virtual machines of type F4, the job completion time of all schemes also increases in general. However, the time of vanilla MDS and REP increases more drastically, as they cannot take advantage of partial results in slower workers, especially in such heterogeneous clusters with natively slower workers. Compared with vanilla MDS, the time saved by SP ranges between 38.2% and 49.3% in the five clusters. Similar to the results in homogeneous clusters, the job completion time of SP also outperforms GLO, thanks to its much lower decoding time.

B. Linear Regression

We now integrate the Spinner-based distributed matrix multiplication into a linear regression job. We implement a coded gradient descent job for linear regression following the approach in [2]. We consider the problem of $\min_x f(x) \triangleq \min_x \frac{1}{2}||Ax - y||^2$, where $y \in \mathbb{R}^q$ is the label vector, and $A \in \mathbb{R}^{q \times r}$ is the data matrix, and $x \in \mathbb{R}^r$ is the unknown weight vector to be trained. The gradient descent algorithm works as follows. After initializing the weight vector as $x^{(0)}$, we update it iteratively as $x^{(t+1)} = x^{(t)} - \gamma \nabla f(x^{(t)}) = x^t - \gamma A^T(Ax^{(t)} - y)$, $t \geq 0$. Therefore, east step can be completed with two matrix multiplications, *i.e.*, $z^{(t)} \triangleq Ax^{(t)}$, and $A^T(z^{(t)} - y)$.

In our implementation, we encode A and A^T with (n, k) MDS codes which will also be converted into linearly equivalent codes in Spinner. As only x is updated in each step, A and A^T need to be encoded only once for each worker. In each step, we first broadcast $x^{(t)}$ to all workers, and let each worker runs their corresponding task. The master collects the results from workers and decodes them into $z^{(t)} = Ax^{(t)}$. The master then calculates $z^{(t)} - y$ and broadcasts it to all workers so as to calculate the second matrix multiplication $A^T(z^{(t)} - y)$. After decoding, the master will get $A^T(z^{(t)} - y)$ so that we can update $x^{(t)}$ into $x^{(t+1)}$. All tasks will be immediately terminated once the master has received sufficient (partial) results for decoding and the workers will be forced to run the tasks of the next matrix multiplication.

As shown in Fig. 10, we generate the data matrix A randomly in three different sizes. We then encode A and A^T with $(12, 8)$ MDS codes. We first run the linear regression job on a cluster hosted on Microsoft Azure with the master still running on a virtual machine of type F4 and 12 other workers running on virtual machines of type B4. We run all the jobs with 500 steps. Each job is still repeated by 20 times and we present their average time. In Fig. 10, we present the completion time of the three jobs with three schemes. We can observe that in all three jobs with different shapes of A, SP can be consistently faster than vanilla MDS codes by 38.5%, 36.0%, and 32.1%, respectively. Similar to the results of matrix multiplication, GLO performs the worst due to its overwhelmingly high decoding complexity.

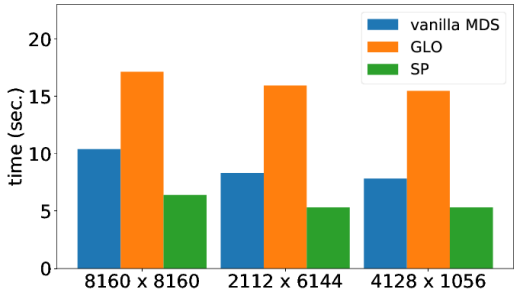

Fig. 10: Job completion time of linear regression in a homogeneous clusters with 24 workers.

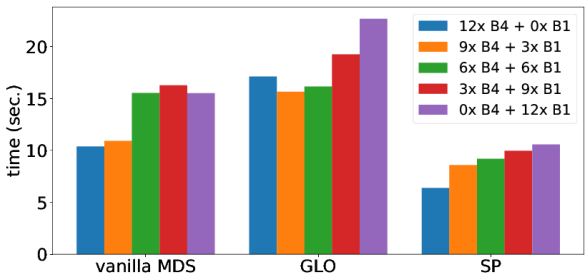

Fig. 11: Job completion time of linear regression in heterogeneous clusters with 24 workers.

We now run the job where A is a 8160×8160 matrix in five clusters with different types of virtual machines. In Fig. 11, the five clusters contain 12 virtual machines of types B4 and/or B1, where B4 servers are almost twice faster than B1 servers. We can observe similar performance patters as in Fig. 9, with the performance gain of SP being up to 40.7% better than vanilla MDS codes. While GLO is still the slowest scheme, it is interesting to observe that in two heterogeneous clusters, GLO can surprisingly perform better than itself in 12 B4 servers which are expected to be fastest among all five clusters. We find that this is because in these two heterogeneous clusters the tasks on all B4 servers are among the first k tasks that are actually uncoded tasks. As they make more contributions in the job than B1 servers, the results from coded tasks become fewer than those in homogeneous clusters where each worker makes similar contributions. Because of the high decoding complexity in GLO, such savings of decoding overhead even compensate for the losses of task completion time.

VIII. Conclusions

Existing coded matrix multiplication can lead to a significant waste of system resources, as the results on stragglers will be disregarded. In this paper, we propose Spinner, a framework for coded distributed matrix multiplication where we exploit the partial results of tasks on stragglers in a

heterogeneous cluster. We have also minimized the complexity of decoding results from coded tasks in Spinner. Combining these two methods, Spinner can significantly improve the overall performance of distributed matrix multiplication, as well as linear regression workloads with Spinner integrated.

REFERENCES

[1] 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.

[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, 2018.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] K. Narra, Z. Lin, M. Kiamari, S. Avestimehr, and M. Annavaram, "Distributed Matrix Multiplication Using Speed Adaptive Coding," 2019. [Online]. Available: https://arxiv.org/abs/1904.07098

[9] 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.

[10] 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.

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

[12] M. E. Ozfatura, D. Gunduz, and S. Ulukus, "Speeding Up Distributed Gradient Descent by Utilizing Non-persistent Stragglers," in *IEEE International Symposium on Information Theory (ISIT)*, 2019.

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

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

[15] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2016, pp. 964–971.

[16] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and S. Avestimehr, "Coded TeraSort," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 389–398.

[17] L. Chen, H. Wang, Z. Charles, and D. Papailiopoulos, "DRACO: Byzantine-resilient Distributed Training via Redundant Gradients," in *International Conference on Machine Learning (ICML)*, 2018, pp. 902–911.

[18] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded Distributed Computing: Straggling Servers and Multistage Dataflows," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2017, pp. 164–171.

[19] S. El Rouayheb and K. Ramchandran, "Fractional Repetition Codes for Repair in Distributed Storage Systems," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2010, pp. 1510–1517.

[20] A. Reisizadeh, S. Prakash, R. Pedarsani, and S. Avestimehr, "Coded Computation over Heterogeneous Clusters," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2408–2412.

[21] 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.

[22] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler Mitigation in Distributed Optimization Through Data Encoding," *Advances in Neural Information Processing Systems (NIPS)*, pp. 5434–5442, 2017.

[23] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Communication-aware Computing for Edge Processing," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2885–2889.

[24] J. Chung, K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "UberShuffle: Communication-efficient Data Shuffling for SGD via Coding Theory," *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[25] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial Codes: An Optimal Design for High-Dimensional Coded Matrix Multiplication," in *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[26] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *ACM Symposium on Cloud Computing (SoCC)*, 2012.

[27] "Open MPI: Open Source High Performance Computing," p. 2019. [Online]. Available: https://www.open-mpi.org

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

[29] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[30] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[31] H. Kim, Y. Jiang, S. Kannan, S. Oh, and P. Viswanath, "Deepcode: Feedback Codes via Deep Learning," in *35th Conference on Neural Information Processing Systems (NIPS)*, 2018.

[32] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Learning a Code: Machine Learning for Approximate Non-Linear Coded Computation," *arXiv:1806.01259*, 2018. [Online]. Available: http://arxiv.org/abs/1806.01259

[33] J. Li and B. Li, "On Data Parallelism of Erasure Coding in Distributed Storage Systems," in *International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[34] ——, "Parallelism-aware locally repairable code for distributed storage systems," in *International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[35] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2022–2026.

[36] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, "On the Optimal Recovery Threshold of Coded Matrix Multiplication," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2017.

[37] S. Wang, J. Liu, and N. Shroff, "Coded Sparse Matrix Multiplication," in *International Conference on Machine Learning (ICML)*, 2018.

[38] Y. Sun, J. Zhao, S. Zhou, and D. Gunduz, "Heterogeneous Coded Computation across Heterogeneous Workers," *arXiv:1904.07490*, 2019. [Online]. Available: https://arxiv.org/pdf/1904.07490.pdf

[39] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A Unified Coded Deep Neural Network Training Strategy Based on Generalized PolyDot Codes for Matrix Multiplication," Tech. Rep. [Online]. Available: https://arxiv.org/pdf/1811.10751.pdf

[40] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, K. Ramchandran, and K. V. Rashmi, "Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth," *USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[41] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, aug 2011.