# Asymmetric Coded Distributed Computation for Resilient Prediction Serving Systems

Lin Wang, Yuchong Hu(✉), Yuxue Liu, Renzhi Xiao, and Dan Feng

Huazhong University of Science and Technology, Wuhan, China
{wanglin2021,yuchonghu,yuxueliu,rzxiao,dfeng}@hust.edu.cn

**Abstract.** With the surge of AI services, prediction serving systems (PSSes) have been widely deployed. PSSes are often run on many workers and thus are prone to stragglers (slowdowns or failures), so it is critical to design straggler-resilient PSSes for low latency of prediction. A traditional way is replication that assigns the same prediction job to multiple workers, while incurring significant resources overheads due to its replicated redundant jobs. Recently, coded distributed computation (CDC) has become a more resource-efficient way than replication, as it encodes the prediction job into parity units for prediction reconstruction via decoding. However, we find that state-of-the-art CDC methods either trade accuracy for low latency with the encoder and decoder *both simple*, or trade latency for high accuracy with the encoder and decoder *both complicated*, leading to unbalance between accuracy and latency due to the above *symmetry* between the encoder and decoder.

Our insight is that the encoder is always used in CDC-based prediction, while the decoder is only used when stragglers occur. In this paper, we first propose a new *asymmetric* CDC framework based on the insight, called AsymCDC, composed of a simple encoder but a complicated decoder, such that the encoder's simplicity makes a low encoding time that reduces the latency largely, while the decoder's complexity can be helpful for accuracy. Further, we design the decoder's complexity in two steps: i) an exact decoding method that leverages an invertible neural network's (INN) invertibility to make the decoding have no accuracy loss, and ii) a decoder compacting method that reshapes INN outputs to utilize knowledge distillation effectively that compacts the decoder for low decoding time. We prototype AsymCDC atop Clipper and experiments show that the prediction accuracy of AsymCDC is approximately the same as state-of-the-arts with the encoder and decoder both complicated, while the latency of AsymCDC only exceeds that of state-of-the-arts with the encoder and decoder both simple by no more than 2.6%.

## 1 Introduction

*Prediction serving systems* (PSSes) are platforms that host *deployed models* for inference (i.e., the process of outputting prediction results from a deployed model in answer to prediction jobs input from users) and return prediction results to


J. Carretero et al. (Eds.): Euro-Par 2024, LNCS 14802, pp. 485–499, 2024.
https://doi.org/10.1007/978-3-031-69766-1_33

users. Recently, prediction services at a large scale commonly employ distributed PSS architectures [3], which distribute prediction jobs across multiple servers that host copies of the deployed model, and then return prediction results from the servers. Nevertheless, distributed systems are susceptible to *stragglers* (e.g., slowdowns and failures) [4], incurring high prediction latency.

To address the straggler problem, traditional wisdom utilizes replication [3], which replicates the same prediction job to multiple servers as identical inputs and chooses the fastest output as the prediction result. Replication can effectively reduce the latency caused by stragglers, but incurs significant resource overhead due to its replicated redundant jobs. Alternatively, erasure coding techniques, which are traditionally used in storage and communication systems [8,20], have been recently adopted in distributed computing, called *coded distributed computation* (CDC in short) [14]. Specifically, CDC uses an *encoder* to encode the prediction job into coded inputs that are then distributed across servers, such that a *decoder* can collect multiple coded outputs from non-straggler servers to reconstruct the prediction results. In this way, CDC incurs much less resource overhead than replication [11–13].

However, we find that the state-of-the-art CDC approaches [11–13] either design a simple encoder and a simple decoder to trade accuracy for low latency [11,12], or design a complicated encoder and a complicated decoder to trade latency for high accuracy [13] (see Sect. 3 for details). In other words, the above *symmetry* between the encoder and decoder (i.e., they are both simple or both complicated) cannot achieve both high accuracy and low latency simultaneously.

In this paper, based on our insight that the CDC-based prediction always runs the encoder but only triggers the decoder when stragglers occur, we propose AsymCDC, which is a new *asymmetric* CDC framework which couples a simple encoder with a complicated decoder, such that the simple encoder have a low encoding time that can reduce the prediction latency significantly, and the complicated decoder can be used for accuracy. We also propose an exact decoding method that can reconstruct straggler results with no accuracy loss based on invertible neural networks (INNs) and a decoder compacting method that can effectively utilize the knowledge distillation (KD) technique to get a small and fast decoder with a slight loss of accuracy.

Our contributions are as follows:

- We observe that the simple symmetry between the encoder and decoder has a lower coding time while the complicated symmetry has a higher prediction accuracy, and stragglers happen much less frequently than non-stragglers in PSSes. This motivates us to use a simple encoder but a complicated decoder to achieve both low latency and high accuracy simultaneously (Sect. 3).
- We propose an asymmetric CDC framework, called AsymCDC, which couples a simple encoder with a complicated decoder. We further design the complicated decoder via proposing an exact decoding method that can reconstruct straggler results with no accuracy loss, and a decoder compacting method that can reduce the decoder size for lower decoding time (Sect. 4).

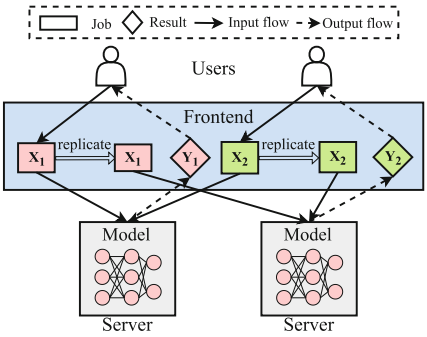



**Fig. 1.** Replication-based PSS

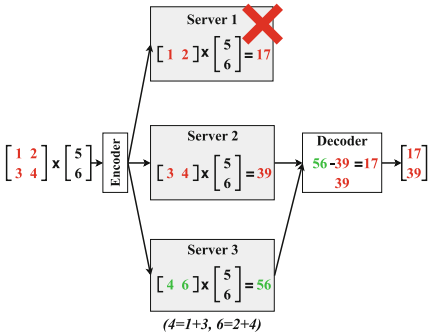



**Fig. 2.** Illustration of CDC

– We prototype AsymCDC atop Clipper [3] and conduct cloud experiments via Alibaba Cloud [1] on a variety of datasets. Experimental results show that AsymCDC has approximately the same accuracy as the state-of-the-arts with a complicated symmetry, and AsymCDC's accuracy only exceeds state-of-the-arts with a simple symmetry by no more than 2.6% (Sect. 5). The prototype is open-sourced at https://github.com/YuchongHu/AsymCDC.

## 2 Background and Related Work

### 2.1 Prediction Serving System and Its Resilience

A prediction serving system (PSS) hosts *deployed models*, receives prediction jobs as inputs from users, performs inference on hosted models, and sends prediction results as outputs to users. To serve large-scale prediction services, PSSes often employ distributed architectures to overcome the computation and memory limitations of a single server. Figure 1 shows the distributed architecture of a PSS, composed of a frontend and deployed models: the frontend receives jobs and dispatches them to deployed models for inference; the deployed models residing on different servers return prediction results after performing inference.

**Replication-Based Resilience in PSS.** Distributed systems are prone to stragglers (slowdowns or failures) that will increase the prediction latency. To deal with the stragglers, replication is commonly used to impart resilience to PSSes [3], which replicates the prediction jobs to multiple workers and waits for the fastest result. As illustrated in Fig. 1, two jobs $X_1$ and $X_2$ are replicated to two servers and two results $Y_1$ and $Y_2$ are returned from the fast servers.

However, replication incurs significant resource overhead due to the replicated redundant jobs, so in this paper, we focus on another way to maintain the PSS's resilience that has lower resource overhead (see Sect. 2.2).

### 2.2 Coded Distributed Computation

**Erasure Coding.** Traditionally, erasure coding techniques [18] are used in storage and communication systems against failures. For example, two data units $D_1$ and $D_2$ are encoded into a parity unit $D_1 \oplus D_2$, and then $D_2$ and $D_1 \oplus D_2$ can recover $D_1$ if it fails. Recently, erasure coding has been adopted in distributed computing to handles stragglers, called *coded distributed computation* (CDC).

**CDC Basics.** CDC divides an original *job* into $k$ original *tasks* and then encodes them into $m$ coded tasks, where the above $n = k + m$ tasks are then dispatched to $n$ different servers that perform computation on all tasks to produce $n$ *task results*. If one task result becomes a straggler, called *straggler result*, its corresponding task is called *straggler task*. We can use $k$ task results ($k < n$) from $k$ non-straggler servers to decode the original *job result*. As illustrated in Fig. 2, an original multiplication job $\left[\begin{smallmatrix}1 & 2\\ 3 & 4\end{smallmatrix}\right] \times \left[\begin{smallmatrix}5\\ 6\end{smallmatrix}\right]$ is encoded into three tasks $\left[1\ 2\right] \times \left[\begin{smallmatrix}5\\ 6\end{smallmatrix}\right]$, $\left[3\ 4\right] \times \left[\begin{smallmatrix}5\\ 6\end{smallmatrix}\right]$, and $\left[4\ 6\right] \times \left[\begin{smallmatrix}5\\ 6\end{smallmatrix}\right]$, where $\left[4\ 6\right] = \left[1\ 2\right] + \left[3\ 4\right]$. If server 1 is a straggler, CDC can use two task results from servers 2 and 3 to decode the original job result as $\left[\begin{smallmatrix}17\\ 39\end{smallmatrix}\right]$.

**Decoding in CDC.** There are two types of decoding in CDC: i) *exact decoding* [9], which can decode the $k$ task results to obtain exactly the same as the original job result, often happens when the original job is *linear* computation, such as matrix multiplication [9,14] in Fig. 2; ii) *approximate decoding* [21], which only decodes the final result approximating to the original job result, often happens when the original job is *non-linear* computation, such as neural network based inference [12,13]. Clearly, compared to the original job result, the exact-decoding CDC has no accuracy loss, while the approximate-decoding one lowers the accuracy.

**State-of-the-Art CDCs in PSSes.** A prediction job relies on neural network based inference that are non-linear (e.g., activation functions and max-pooling [12]), so state-of-the-art CDCs in PSSes focus on how to handle the non-linearity to improve accuracy. We specify them as follows:

- *ParM* [12] proposes a *learning-based* model to handle the non-linearity in neural network based inference. As shown in Fig. 3(a), ParM uses a simple encoder based on addition and a simple decoder based on subtraction, where the encoder divides the original job into two original tasks $X_1$ and $X_2$ and then adds them into a parity task $X_p$. Then $X_1$ and $X_2$ are dispatched to two servers hosting the deployed model (denoted by $F$); $X_p$ is dispatched to a server hosting a learned-based parity model (denoted by $F_p$), which is expected to make the coded result $F(X_p)$ be the sum of the two results $F(X_1)$ and $F(X_2)$. If straggler exists (e.g., $F(X_1)$ fails), ParM will use the subtraction decoder to obtain $\hat{F}(X_1) = F_p(X_p) - F(X_2)$, which approximates to $F(X_1)$.
- *ApproxIFER* [21] uses *approximate computing* [11] that forms a simple encoder and decoder to improve the accuracy over ParM. As shown in Fig. 3(b), ApproxIFER divides the original job into two original tasks $X_1$

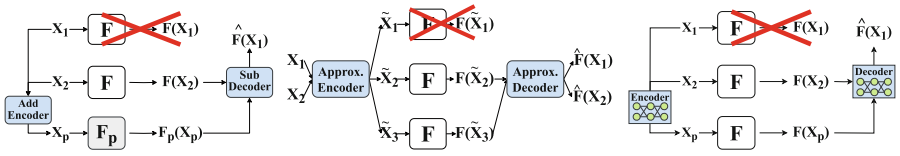



**Fig. 3.** Examples of ParM, ApproxIFER and Learned-CC.

and $X_2$, encodes them into three coded tasks $\tilde{X}_1$, $\tilde{X}_2$, and $\tilde{X}_3$, and dispatches them to three servers to perform inference to obtain three coded results $F(\tilde{X}_1)$, $F(\tilde{X}_2)$, and $F(\tilde{X}_3)$. When $F(\tilde{X}_1)$ is unavailable due to stragglers, ApproxIFER will use a decoder to use the other two available coded results $F(\tilde{X}_2)$, and $F(\tilde{X}_3)$ to decode the results $\hat{F}(X_1)$ and $\hat{F}(X_2)$, which approximate to two original task results $F(X_1)$ and $F(X_2)$.

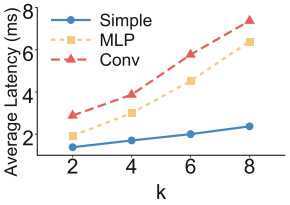
– *Learned-CC* [13] adopts a complicated encoder and a complicated decoder based on neural networks to further improve the accuracy over ParM and ApproxIFER. As shown in Fig. 3(c), Learned-CC's encoder first encodes two original tasks $X_1$ and $X_2$ into a parity task $X_p$ via a neural network, and its decoder uses the fastest two task results to approximate to the job result. However, the complexity may increase the prediction latency.

**Symmetry in State-of-the-Art CDCs.** Existing studies on CDC for PSSes either use a simple encoder and a simple decoder (e.g., ParM and ApproxIFER), or use a complicated encoder and a complicated decoder (e.g., Learned-CC). We define the above CDC framework as *symmetric* CDC (*SymCDC* in short). We then name a *SymCDC* with a simple (or complicated) encoder and decoder *simple SymCDC* (or *complicated SymCDC*). In this paper, we aim to explore another family of CDC that is not symmetric instead of SymCDC (see Sect. 3).

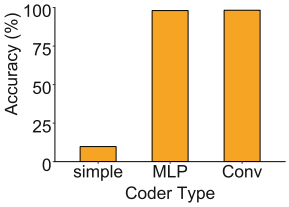
## 3 Motivation

We first show two observations: the first one is based on experiments (Fig. 4(a) and (b)) and the second one is based on existing data from literatures (Fig. 4(c)). We then introduce our main idea based on the two observations.

**Observation #1: Simple SymCDC has a lower coding time, while complicated SymCDC has a higher prediction accuracy.** We conduct our experiments on a machine equipped with an Intel Xeon Platinum 8163 CPU (8 cores), a 32GiB RAM, and a 100GiB SSD. We implement our encoders and decoders in Pytorch and evaluate them on the popular MNIST dataset and ResNet-18 model. We set $k$ from 2 to 8, and $m$ to 1. We choose addition and subtraction as the simple encoder and decoder (*SimpleCoder*), like ParM [12]. We choose two neural network architectures a multi-layer perceptron (*MLPCoder*) and a convolutional neural network (*ConvCoder*) as the complicated encoder and




(a) Coding time




(b) Prediction accuracy

| Task Type | Ratio |
|---|---|
| Straggler | 20% |
| Non-straggler | 80% |

(c) Straggler ratio

**Fig. 4.** Observation # 1-2: Coding time and prediction accuracy of SimpleCoder, MLP-Coder and ConvCoder.

decoder, like Learned-CC [13]. We measure the average coding time of encoders and decoders of each experiment over 1000 runs.

Figure 4(a) shows the average coding time of simple and complicated SymCDC. We see that the average coding time of SimpleCoder is much lower than both MLPCoder and ConvCoder. Specifically, the average coding time of SimpleCoder is lower than MLPCoder by up to 65.3% (when $k = 8$) and ConvCoder by up to 67.8% (when $k = 8$). Figure 4(b) shows the accuracy of simple and complicated SymCDC. We see that, compared with simple SymCDC, complicated SymCDC can maintain much higher accuracy. Specifically, the accuracy of simple SymCDC is lower than complicated SymCDC by up to 90%.

**Observation #2: The ratio of stragglers is much lower than that of non-stragglers.** According to the CMU Hadoop production cluster traces [19], the average ratio of stragglers is 0–40%. As shown in Fig. 4(c), many studies [15] use the average ratio of stragglers as 20%, much lower than that of non-stragglers as 80%. Since the encoder is always used whether stragglers occur while the decoder is only used when (infrequent) stragglers occur, encoding takes place much often than decoding in a CDC-based PSS. Therefore, reducing the encoding time can largely reduce the prediction latency.

**Main Idea:** The above two observations motivate us to propose an *asymmetric* CDC framework, called AsymCDC, including a simple encoder that reduces the encoding time and thus reduces the prediction latency, and a complicated decoder that has a high prediction accuracy. However, how to design AsymCDC's complicated decoder to cooperate with its simple encoder for high accuracy and low latency remains unexplored (see Sect. 4).

## 4 Design

We design AsymCDC with the following design goals:

– **Low Encoding Time.** AsymCDC achieves the low encoding time via a simple encoder based on linear computation, leaving the decoder complicated for high accuracy (Sect. 4.1).

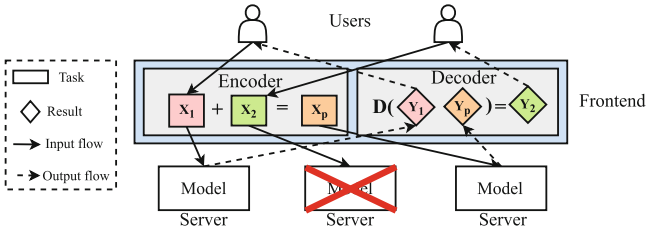



**Fig. 5.** Framework of AsymCDC with $k = 2$.

- **High Prediction Accuracy.** AsymCDC's decoder has no accuracy loss by exact decoding via leveraging the invertibility of invertible neural networks (INN) (Sect. 4.2).
- **Reduced Decoding Time.** AsymCDC's decoder reduces the decoding time by reshaping the INN outputs such that the decoder can be compacted effectively via knowledge distillation (Sect. 4.3).

### 4.1 Asymmetric Framework of AsymCDC

We show the framework of AsymCDC in Fig. 5, where AsymCDC has two typical components, frontend and deployed models as stated in Sect. 2.1. AsymCDC also has two CDC components, encoder and decoder, specified as follows.

**Simple Encoder** performs encoding operation to generate a parity task based on the original tasks. According to the main idea Sect. 3, we design our encoder in a simple way based on the simple linear computation to ensure the low encoding time. Specifically, assume that the encoder encodes each $k$ input tasks $X_1, X_2, ..., X_k$ into one parity task $X_p$ by $X_p = \sum_{i=1}^{k} \frac{1}{k} X_i$. The reason why the tasks are multiplied by $\frac{1}{k}$ before adding them together is that it can keep the value range of the parity task $X_p$ within the same range as other tasks.

**Complicated Decoder** performs decoding to reconstruct the straggler result based on the non-straggler results. Based on the main idea in Sect. 3, we design our decoder in a complicated way based on neural networks. Specifically, for the $k+1$ tasks generated by the simple encoder, i.e., $X_1, X_2, ..., X_k, X_p$, deployed models perform inference on these $k+1$ tasks and return $k+1$ prediction results, i.e., $Y_1, Y_2, ..., Y_k, Y_p$. If $Y_i$ is the straggler result, the decoder will reconstruct it by $Y_i = D(Y_1, ..., Y_{i-1}, Y_{i+1}, ..., Y_p)$, where $D$ represents the decoding function based on neural networks, which will be specified in Sect. 4.2 and Sect. 4.3.

### 4.2 Exact Decoding in AsymCDC

**Challenge.** To apply CDC in PSSes, state-of-the-art methods (Sect. 2.2) try to address the non-linearity of PSSes by approximating the straggler results

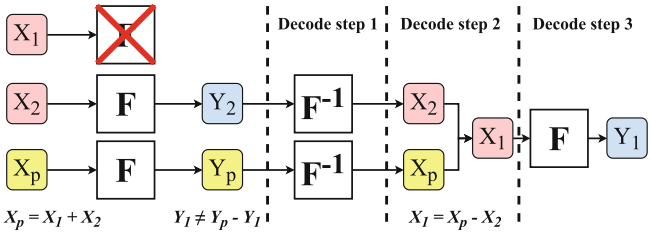



**Fig. 6.** Design idea of exact decoding in AsymCDC.

via neural networks (ParM and Learned-CC) or approximate coding (Approx-IFER). However, these approximate decoding based methods often cause accuracy loss when stragglers exist. It is natural to ask if we can design exact decoding (or an *exact decoder*) that can decode the straggler results without accuracy loss?

**Design Idea.** We state the design idea of our exact decoding method in AsymCDC by an example shown in Fig. 6. We find that the coded tasks $X_1, X_2, X_p$ always satisfy linear relationship because they are encoded by the simple encoder which is based on linear computations. Clearly, their task results $Y_1, Y_2, Y_p$ no longer satisfy linear relationship because they are computed by non-linear deployed model $F$, which makes a straggler task result (say $Y_1$) cannot be decoded exactly.

Our design idea is that if we can use the non-straggler task results $Y_2$ and $Y_p$ to obtain their corresponding tasks $X_2$ and $X_p$, the straggler task $X_1$ can be easily computed, and $Y_1$ can also be easily computed again based on the deployed model $F$, as shown in Fig. 6. This design idea motivates us to find a way to map the results to tasks. We find that invertible neural networks (INNs) [2,5,10,16], which are neural networks with invertibility which have recently drawn a lot of attention by researchers and productions. INNs can offer us this opportunity that enables the inverse mapping from results to tasks.

**Design Details.** We show our design details of the exact decoding based on INNs in the following steps.

*Step 1 (Inversing).* First, exact decoding inverses the non-straggler prediction results via INNs to get the corresponding prediction tasks.

*Step 2 (Linear Decoding).* Then, exact decoding gets the task corresponding to the straggler result based on the linear encoding of the simple encoder.

*Step 3 (Re-Computing).* Finally, the straggler result can be re-computed by inputting the straggler tasks again in the deployed model.

Specifically, we use an example shown in Fig. 6 to show how exact decoding works. Assume that the deployed model is based on an INN $F$ (i.e., for any $y = F(x)$, there exists $x = F^{-1}(y)$) and $k = 2$, $m = 1$. Two inputs $X_1$ and $X_2$ are first encoded into a parity input $X_p$ via the formula $X_p = \sum_{i=1}^{2} X_i$) in the simple encoder, and all inputs are dispatched to different servers to produce

prediction results. Assume that the results of $X_1, X_2$ and $X_p$ are $Y_1, Y_2$ and $Y_p$, respectively, and $Y_1$ is the straggler result. To reconstruct $Y_1$, the exact decoding first inverses the $Y_2$ and $Y_p$ by utilizing the invertibility of INNs to get the inputs $X_2$ and $X_p$, respectively (*Decode step 1* in Fig. 6). Then we can compute $X_1$ corresponding to $Y_1$ by simply transforming the encoding formula into $X_1 = (X_p - X_2)$ (*Decode step 2* in Fig. 6). Finally, the straggler result $Y_1$ can be exactly computed by deployed model: $Y_1 = F(X_1)$ (*Decode step 3* in Fig. 6).

Here, we prove that if the deployed model is based on INN and the encoder is based on linear computation, the exact decoding can always exactly reconstruct the straggler results. We first give two definitions to support our proof.

**Definition 1.** $F$ is invertible if: $\forall F : x \to y, \exists F^{-1} : y \to x$.

**Definition 2.** $E$ is linear if: $\forall x_1, x_2, E(x_1 + x_2) = E(x_1) + E(x_2)$ and $E(ax_1) = aE(x_1)$.

**Theorem 1.** *If $F$ is invertible, $E$ is linear, and we have $x_1, ..., x_k$, and $x_p = E(x_1, ..., x_k)$ and $y_i = F(x_i)$, $y_p = F(x_p)$ where $1 \leq i \leq k$, $\forall j \in [1, k]$, $\exists G : y^k \to y$, s.t. $G(y_1, ..., y_{j-1}, y_{j+1}, ..., y_k, y_p) = y_j$.*

*Proof.* Without loss of generality, we assume that $E(x_1, ..., x_k) = \sum_{i=1}^{k} x_i$. Then we can construct $G = F(F^{-1}(F(\sum_{i=1}^{k} x_i) - \sum_{l=1}^{k} F^{-1}(F(x_l))), l \neq j$. Thus, we can get $G = F(\sum_{i=1}^{k} x_i - \sum_{l=1}^{k} x_l) = F(x_j) = y_j, j \neq l$. This proves that if $F$ is invertible and $E$ is linear, a $G$ that can exactly reconstruct $y_j$ always exists. □

**Discussion.** Note that one may say that the exact decoding can be achieved by just re-sending the straggler tasks as inputs to deployed models to re-compute the straggler results again. However, this method needs us to cache all the tasks until all the corresponding results are returned, or to fetch them from checkpoints store in disks. This will incur huge memory overhead, or slow disk I/Os which will also prolong the prediction latency. Our exact decoding does not need to cache or store any tasks, but still has a long decoding time because of the *inversing* and *re-computing* steps. Therefore, we further propose a *decoder compacting* method to compact the decoder atop the exact decoder for low decoding time (see Sect. 4.3).

### 4.3 Compacted Decoder in AsymCDC

**Knowledge-Distillation-Based Compacting.** As mentioned in Sect. 4.1, the decoder in AsymCDC is a neural network model, so we can compact the model into a smaller one for low decoding time via using a model compression technique. Knowledge distillation (KD) [7], which targets the transfer of knowledge (i.e., a learned mapping from inputs to outputs) from one larger model to another smaller model, is a typical and popular model compression technique. Therefore, we choose to use KD to compact the exact decoder into a smaller and faster decoder, called *compacted decoder*. Specifically, based on the process of KD [7],

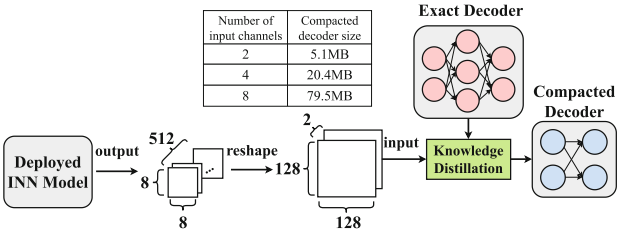



**Fig. 7.** Design idea of decoder compacting in AsymCDC.

we first view the exact decoder designed in Sect. 4.2 as KD's large model, which takes $k$ non-straggler results as the exact decoder's inputs and produces the straggler results as the exact decoder's outputs; we then choose another smaller neural network model as the compacted decoder, which has the same inputs and outputs as the exact decoder but has a smaller architecture.

**Challenge.** However, we find it is challenging to enable the above process of KD to compact the exact decoder into a smaller compacted decoder. The reason is that the inputs of the compacted decoder are from INNs' outputs which have too many *channels* (i.e., neural networks usually takes 3D data as inputs, and one of the dimensions is called channel). In this case, we observe that if the decoder has too large amount of input channels, then KD cannot effectively compact the decoder. We conduct an experiment which has the same setup as Sect. 5 to reveal the relationship between the number of input channels and the size of the compacted decoder. As shown in the table of Fig. 7, our evaluated results imply that more channels incurs a less compacted size. Therefore, it is natural to ask if we can minimize the number of channels such that KD can be most effectively used to compact the decoder?

**Design Idea.** To find out the minimal number of input channels, our design idea is to reshape the INN's outputs such that the number of channels is minimized, so as to minimize the size of the compacted decoder. On the other hand, we require the *height* and *width* (the other two dimensions of 3D data) to be the same because square inputs (i.e., height equals to width) are preferred by many popular deep neural network to simplify the input processing, as square dimensions streamline convolutional operations [6].

**Design Details.** Based on the design idea, we introduce the design details of the compacted decoder in the following steps:

*Step 1 (Getting the optimal number of input channels of the compacted decoder).* Assume that we have the shape of the INN output, including the original number of channels $c$, height $h$ and width $w$. To get the optimal number of input channels (denoted by $\alpha$) of the compacted decoder, we first initialize it as one, i.e., $\alpha = 1$. Then we loop the $\alpha$ until reaching the number of the original channels. In each loop, we check if the $\sqrt{c \times h \times w/\alpha}$ is an integer: i) if not, it means that even though the $\alpha$ is small, we cannot get an integer height and width based on this

$\alpha$, so it is impractical for reshaping, and then we just increase $\alpha$ by 1; ii) if yes, it means that we have found the smallest and practical $\alpha$, and then we can just break out of the loop and return $\alpha$, which is the optimal number of input of channels of our compacted decoder that can have the minimized size.

*Step 2 (Reshaping inputs).* We then reshape the inputs based on the optimal number of channels $\alpha$. Specifically, we reshape the original input's shape with $c \times \ h \times w$ into $\alpha \times \sqrt{\frac{C \times H \times W}{\alpha}} \times \sqrt{\frac{C \times H \times W}{\alpha}}$.

*Step 3 (Training compacted decoder via KD).* Finally, we get our compacted decoder by training it based on the exact decoder and the reshaped inputs via KD.

Specifically, let the shape of the original inputs be $512 \times 8 \times 8$. First we initialize the $\alpha$ to 1 and find that $\sqrt{c \times h \times w/\alpha} = \sqrt{32768}$ is not an integer. Then we increase $\alpha$ to 2 and find that $\sqrt{c \times h \times w/\alpha} = \sqrt{16384} = 128$, which is an integer. Then $\alpha = 2$ is the optimal (the smallest) number of input channels of our compacted decoder. Then we reshape the input data into $2 \times 128 \times 128$, and finally train the compacted decoder via KD.

## 5 Evaluation

We conduct experiments to evaluate the prediction accuracy and latency of AsymCDC, and aim to address two key questions: i) Can AsymCDC achieve high prediction accuracy via using a complicated decoder (Sect. 5.3)? ii) Can AsymCDC achieve low prediction latncy via using a simple encoder (Sect. 5.4)?

### 5.1 Implementation

We prototype AsymCDC atop Clipper [3], which is an open-sourced and low-latency online PSS. Our prototype mainly includes the following components: i) `Client` provides interfaces to users to interact with the AsymCDC; ii) `Frontend` acts as an agent to receive the requests from clients and performs encoding operations and decoding operations when straggler exists; iii) `DeployedModels` perform inference tasks and are deployed on multiple Docker containers, which are run on multiple seperate nodes; iv) `SimpleEncoder` performs simple encoding on prediction tasks from clients and is deployed in the `Frontend` component; v) `ComplicatedDecoder` performs decoding when straggler exists, and is deployed in the `Frontend` component.

### 5.2 Experimental Setup

2**Testbed.** We conduct our experiments via Alibaba Cloud [1] with 16 ecs.gn6i-c8g1.2xlarge instances (the same cluster size as ParM [12]), each of which is equipped with an Intel Xeon (Skylake) Platinum 8163 CPU (8 cores), a 32GiB RAM, a 1TiB SSD, and an NVIDIA T4 GPU. Each instance is installed with Ubuntu 16.04 LTS and all instances are connected via a 10Gbps network.

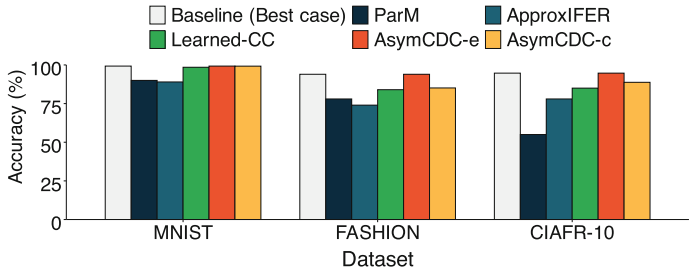
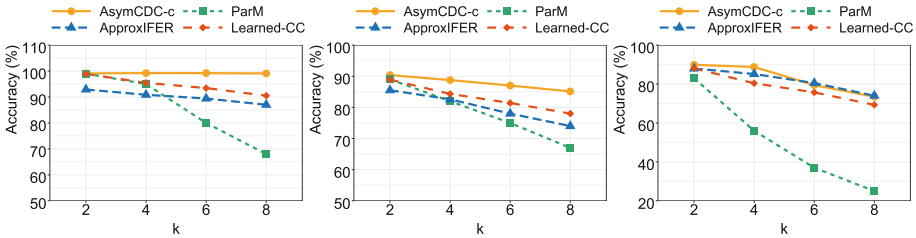



**Fig. 8.** Experiment 1: Prediction accuracy on different datasets when $k = 4$.




(a) MNIST (b) FASHION (c) CIFAR-10

**Fig. 9.** Experiment 2: Prediction accuracy on different datasets with different $k$.

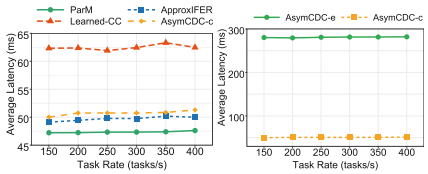
**Deployed INN Model.** We choose i-RevNet [10] to be the deployed INN model, a representative invertible neural network architecture that has been used in many studies [2] and has been shown efficiency on various datasets.

**Datasets and Parameters.** The experiments are based on popular image classification datasets (MNIST, FASHION and CIFAR-10). We set the coding parameter $k$ to 2, 4, 6 and 8. We set $m = 1$ to tolerate single-failures which account for 98% of the cases in production [17]. We vary the task rate (i.e., how many prediction tasks per second) from 150 to 400, like ParM [12]. We report the average results of each experiment over five runs.

**Baseline and Comparative Approaches.** We choose the original task results of the deployed INN model as the baseline (best case), called *Baseline*. We compare AsymCDC with three state-of-the-arts approaches, including ParM [12], ApproxIFER [21] and Learned-CC [13]. We name two proposed approaches based on exact decoding (Sect. 4.2) and compacted decoder (Sect. 4.3) as AsymCDC-e and AsymCDC-c, respectively.

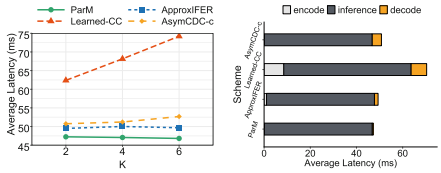
### 5.3 Accuracy Experiments

**Experiment 1 (Accuracy):** We measure the prediction accuracy of Baseline (best case), AsymCDC-e, AsymCDC-c and three state-of-the-art approaches (ParM, ApproxIFER and Learned-CC) on different datasets when $k = 4$.




(a) state-of-the-arts (b) AsymCDC-e

**Fig. 10.** Experiment 3: average latency under different task rates.




(a) Exp4: different $k$ (b) Exp5: breakdown

**Fig. 11.** Experiments 4-5: average latency under different $k$, and the breakdown of the overall latency.

Figure 8 shows that AsymCDC-e has the same prediction accuracy as the Baseline since its exact decoding can decode with no accuracy loss. Also, we can see that the prediction accuracy of AsymCDC-c always outperforms ParM and ApproxIFER. Specifically, AsymCDC-c outperforms ParM by up to 61.4% (on CIFAR-10 dataset) and is also approximately the same as Learned-CC on all datasets. Compared to AsymCDC-e, the prediction accuracy of AsymCDC-c only drops by 9% on average, and only drops by 0.04% at least on MNIST dataset.

**Experiment 2 (Impacts of $k$):** We measure the prediction accuracy of ParM, ApproxIFER, Learned-CC and AsymCDC-c under different $k$ on different datasets. Figure 9 shows that compared with ParM, AsymCDC-c always has a higher prediction accuracy with $k$. Compared with Learned-CC, AsymCDC-c also has an even higher prediction accuracy on all datasets. Compared with ApproxIFER, AsymCDC-c has a higher prediction accuracy on MNIST and FASHION datasets, and has approximately the same prediction accuracy on CIFAR-10 dataset.

### 5.4 Latency Experiments

**Experiment 3 (Latency):** We measure the average latency of ParM, ApproxIFER, Learned-CC, AsymCDC-e and AsymCDC-c under the different task rates when $k = 2$. Figure 10(a) shows the prediction latency of AsymCDC-c and three state-of-the-art approaches. We see that the average latency of AsymCDC-c is much lower than Learned-CC. The reason is that the coding time of Learned-CC is higher than that of AsymCDC-c. Specifically, AsymCDC-c can reduce the latency by up to 20% compared to Learned-CC (when the task rate is 350). Additionally, the average latency of AsymCDC-c is slightly higher than ParM and ApproxIFER, because the decoder of AsymCDC-c is more complicated than ParM and ApproxIFER, but the latency of AsymCDC-c only exceeds that of ParM by no more than 7% and exceeds that of ApproxIFER by no more than 2.6% (when the task rate is 400). Figure 10(b) shows the average latency of AsymCDC-e and AsymCDC-c. We see that AsymCDC-c's latency is much lower than AsymCDC-e. The reason is that knowledge distillation compacts the exact decoder into a much smaller decoder which has much lower decoding time than

the exact decoder. Specifically, AsymCDC-c can reduce the prediction latency by up to 82.1% (when the task rate is 150).

**Experiment 4** (**Impacts of** $k$): We measure the average latency of ParM, ApproxIFER, Learned-CC and AsymCDC-c under different $k$. Figure 11(a) shows that the average latency of AsymCDC-c increases slightly with $k$ and that of Learned-CC increases rapidly with $k$. The reason is that knowledge distillation makes the architecture of AsymCDC-c's decoder simpler than Learned-CC, so the latency of AsymCDC-c increases with $k$ more slowly than that of Learned-CC. Specifically, AsymCDC-c can reduce the average latency by up to 29% compared to Learned-CC (when $k$ is 6).

**Experiment 5 (Breakdown analysis):** We measure the breakdown of the average latency to analyze the improvement of AsymCDC-c. We decompose the overall latency into three important parts: (i) encoding time, (ii) inference time, and (iii) decoding time. Figure 11(b) shows the breakdown of the overall latency of ParM, ApproxIFER, Learned-CC and AsymCDC-c when $k$ is 2 and the task rate is 200. We see that the encoding times of ParM, ApproxIFER and AsymCDC-c are all much lower than that of Learned-CC because the encoders of ParM, ApproxIFER and AsymCDC-c are all simpler than that of Learned-CC. Also, we see that the decoding time of AsymCDC-c is also lower than Learned-CC due to the compacted decoder.

## 6 Conclusion

In this paper, we propose a new asymmetric coded distributed computation framework, called AsymCDC, which is composed of a simple encoder but a complicated decoder for both high accuracy and low latency of PSSes. We further design the complicated decoder via an exact decoding method that can reconstruct the straggler results with no accuracy loss and a decoder compacting method that can compact the decoder size for low decoding time. We prototype AsymCDC atop Clipper and the experiments show the performance gain over state-of-the-arts of AsymCDC in both prediction accuracy and latency.

**Acknowledgments..** This work was supported by the Development Program of China for Young Scholars (No. 2021YFB0301400), and Key Laboratory of Information Storage System Ministry of Education of China.

## References

1. Alibaba cloud (2024). https://www.aliyun.com/
2. Behrmann, J., Grathwohl, W., Chen, R.T., Duvenaud, D., Jacobsen, J.H.: Invertible residual networks. In: Proceedings of ICML (2019)
3. Crankshaw, D., Wang, X., Zhou, G., Franklin, M.J., Gonzalez, J.E., Stoica, I.: Clipper: a low-latency online prediction serving system. In: Proceedings of USENIX NSDI (2017)

4. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**(2), 74–80 (2013)
5. Finzi, M., Izmailov, P., Maddox, W., Kirichenko, P., Wilson, A.G.: Invertible convolutional networks. In: Proceedings of ICML (2019)
6. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition (2015)
7. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)
8. Huang, C., et al.: Erasure coding in windows azure storage. In: Proceedings of USENIX ATC (2012)
9. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **100**(6), 518–528 (1984)
10. Jacobsen, J.H., Smeulders, A.W., Oyallon, E.: i-RevNet: deep invertible networks. In: Proceedings of ICLR (2018)
11. Jahani-Nezhad, T., Maddah-Ali, M.A.: Berrut approximated coded computing: straggler resistance beyond polynomial computing. IEEE Trans. Pattern Anal. Mach. Intell. **45**(1), 111–122 (2022)
12. Kosaian, J., Rashmi, K., Venkataraman, S.: Parity models: erasure-coded resilience for prediction serving systems. In: Proceedings of ACM SOSP (2019)
13. Kosaian, J., Rashmi, K., Venkataraman, S.: Learning-based coded computation. IEEE J. Sel. Areas Inf. Theory (2020)
14. Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., Ramchandran, K.: Speeding up distributed machine learning using codes. IEEE Trans. Inf. Theory **64**(3), 1514–1529 (2017)
15. Phan, T.-D., Ibrahim, S., Zhou, A.C., Aupy, G., Antoniu, G.: Energy-driven straggler mitigation in MapReduce. In: Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.) Euro-Par 2017. LNCS, vol. 10417, pp. 385–398. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64203-1_28
16. Radev, S.T., Mertens, U.K., Voss, A., Ardizzone, L., Köthe, U.: Bayesflow: learning complex stochastic models with invertible neural networks. IEEE Trans. Neural Netw. Learn. Syst. **33**(4), 1452–1466 (2020)
17. Rashmi, K.V., Shah, N.B., Gu, D., Kuang, H., Borthakur, D., Ramchandran, K.: A solution to the network challenges of data recovery in erasure-coded distributed storage systems: a study on the Facebook warehouse cluster. In: USENIX Workshop on HotStorage (2013)
18. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. Soc. Ind. Appl. Math. **8**(2), 300–304 (1960)
19. Ren, K., Kwon, Y., Balazinska, M., Howe, B.: Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. Proc. VLDB Endow. **6**(10), 853–864 (2013)
20. Rizzo, L.: Effective erasure codes for reliable computer communication protocols. ACM SIGCOMM Comput. Commun. Rev. **27**(2), 24–36 (1997)
21. Soleymani, M., Ali, R.E., Mahdavifar, H., Avestimehr, A.S.: ApproxIFER: a model-agnostic approach to resilient and robust prediction serving systems. In: Proceedings of AAAI (2022)