IEEE Open Journal of the
Communications Society

Received 1 April 2023; revised 1 May 2023; accepted 17 May 2023. Date of publication 30 May 2023; date of current version 20 June 2023.

Digital Object Identifier 10.1109/OJCOMS.2023.3280174

Adaptive and Resilient Model-Distributed Inference in Edge Computing Systems

PENGZHEN LI (Student Member, IEEE), ERDEM KOYUNCU (Senior Member, IEEE), AND HULYA SEFEROGLU (Senior Member, IEEE)

Electrical and Computer Engineering Department, University of Illinois Chicago, Chicago, IL 60607, USA

CORRESPONDING AUTHOR: H. SEFEROGLU (e-mail: hulya@uic.edu)

This work was supported in part by the Army Research Laboratory (ARL) under Grant W911NF-2120272, and in part by National Science Foundation (NSF) under Grant CCF-1942878, Grant CNS-2148182, and Grant CNS-2112471.

ABSTRACT The traditional approach to distributed deep neural network (DNN) inference in edge computing systems is data-distributed inference. In this paradigm, each worker has a pre-trained DNN model. Using the DNN model, the worker processes the data that is offloaded to itself. The data-distributed inference approach (i) has high communication cost especially when the size of data is large, and (ii) is not efficient in terms of memory as the whole model should be stored and computed in each worker. Model-distributed inference is emerging as a promising solution, where a DNN model is distributed across workers. Although there is a huge amount of work on model-distributed training, the benefit of model distribution for inference is not understood well. In this paper, we analyze the potential of model-distributed inference in edge computing systems. Then, we develop an Adaptive and Resilient Model-Distributed Inference (AR-MDI) algorithm based on our optimal model allocation formulation. AR-MDI performs model allocation in a lightweight and decentralized way and it is resilient against delayed workers and failures. We implement AR-MDI in a real testbed consisting of NVIDIA Jetson TX2s and show that AR-MDI improves the inference time significantly as compared to baselines when the size of data is large, such as ImageNet.

INDEX TERMS Model-distributed inference, model parallelism, distributed deep neural networks, edge computing.

I. INTRODUCTION

Modern edge devices such as drones, autonomous robots, sensors, and self-driving cars are generating data at tremendous rates. Many applications that execute on these devices are delay-sensitive, meaning that the data generated by the applications should be processed as quickly as possible. For this purpose, transmission of the generated data to a remote cloud may be unacceptable due to transmission delays. Thus, data should be processed near its place of origin, i.e., on or near the edge. One complication in this context is that the edge devices are typically limited in terms of computation power, energy, and/or memory. Hence, the design of high-performance distributed data processing methods is crucial.

The traditional approach to distributed deep neural network (DNN) inference is *data-distributed* inference, which partitions and distributes data to workers as illustrated in Fig. 1. The workers are comprised of edge servers, end users, and/or remote cloud (if available). An end user, which would like to classify input data, offloads data to workers for classification. The end user itself could function as one of the workers by processing some of its own data. Each worker keeps a pre-trained DNN model, processes the offloaded data, and sends the output back to the end user. This approach, although very straightforward, has two disadvantages: (i) Communication cost is high especially when input data size is large (*i.e.,* high resolution data); and (ii) Each worker should store the whole model, which puts a strain especially on end user devices.

Model-distributed inference (also called *model parallelism*) is emerging as a promising solution, where a DNN model is distributed across workers, Fig. 2. The end user, which has input data, may process a few layers of a DNN model, and transmits the activation vector of its last layer

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

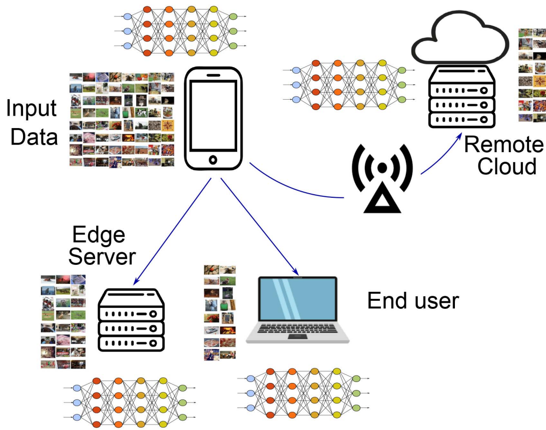


FIGURE 1. Data-distributed inference. Data is partitioned and offloaded to workers for classification.

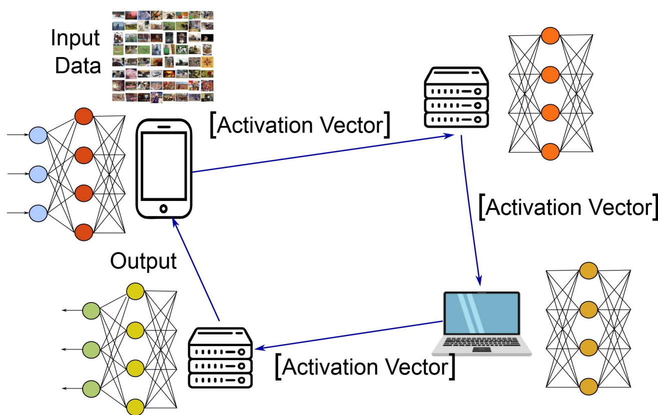


FIGURE 2. Model-distributed inference. DNN model is partitioned and distributed across workers.

to a neighboring node. The neighboring node receives an activation vector and performs the calculations of the layers that are assigned to it. Finally, the worker that calculates the last layers of the DNN model obtains and sends the output back to the end user that has the input data. We note that the workers perform parallel processing in this setup by pipelining as further explained in Section III.

Although there is huge amount of work on model-distributed training [1], [2], [3], [4], the benefit of model distribution for inference is not understood well. The potential of model distribution for training is obvious. Indeed, it is indispensable in data-distributed training to exchange the whole model among workers and a model aggregator (parameter server) for every batch of data, which introduces huge amount of communication cost. On the other hand, thanks to distributing the DNN model, model-distributed training requires to exchange only activation vectors among workers, not the whole model. Thus, model-distributed training reduces the communication cost as compared to data-distributed training.

The potential of model distributed inference in terms of reducing the communication cost is less obvious. While data-distributed inference requires the exchange of actual data (Fig. 1), model-distributed inference needs to exchange activation vectors. We observed that when the size of data is large, exchanging the actual data introduces higher communication cost, which makes model-distributed inference plausible. Building on this observation, we analyze the potential of model-distributed inference as compared to data-distributed inference in a homogeneous setup, where all workers have the same amount of computing power.

It is crucial to exploit the potential of model-distributed inference in a heterogeneous and dynamic setup, where the computing power of workers may be different and change over time. A model partitioning mechanism based on dynamic programming is proposed in [5] for this purpose. However, this approach introduces too much computing cost to determine the optimal model allocation. Also, it is not adaptive to time-varying resources. Instead, we design a lightweight, adaptive, and decentralized model allocation mechanism, which we name Adaptive and Resilient Model-Distributed Inference (AR-MDI) based on the solution to our optimal model-allocation formulation.

One of the weaknesses of model distribution as compared to data distribution is its vulnerability to failing workers. For example, if one of the workers in Fig. 2 fails, the whole system fails. Thus, we design a recovery mechanism as part of our AR-MDI algorithm. The recovery mechanism of AR-MDI is inspired by the peer management mechanism of Chord-like P2P systems as further detailed in Section V. The following are the key contributions of this work:

- We provide inference time analysis for both model- and data-distributed inference in a homogeneous setup, and show that model-distributed inference has smaller inference time if the size of input data is large.
- We formulate a model-allocation problem across workers for model-distributed inference in a heterogeneous setup. Building on the solution to the optimization problem, we design a lightweight, adaptive, and decentralized model allocation algorithm, which we name Adaptive and Resilient Model-Distributed Inference (AR-MDI) algorithm.
- We fortify our AR-MDI algorithm with a recovery mechanism against delayed and failing workers.
- We implemented AR-MDI as well as baselines; EdgePipe [5] and Data-Distributed Inference (DDI) in a heterogeneous testbed of NVIDIA Jetson TX2 cards. Our experiments including CIFAR-10 [6] and ImageNet [7] datasets, and VGG16 [8] and MobileNetV2 [9] DNN models show that AR-MDI significantly reduces the data inference time as compared to the baselines.

The structure of the rest of this paper is as follows. Section II presents the related work. Section III introduces our system model and provides preliminaries on model-distributed inference. Section IV analyzes the potential of model-distributed inference for the case of homogeneous transmission delays and worker computing powers. We formulate an optimal model allocation problem

for the heterogeneous setup, and design our Adaptive and Resilient Model-Distributed Inference (AR-MDI) algorithm based on the structure of the optimal solution in Section V. In Section VI, we provide experimental results on a real-life testbed. Section VII concludes the paper.

II. RELATED WORK

The deployment of machine learning algorithms in the edge computing systems calls for novel distributed learning and inference solutions by taking into account network and computation resources such as bandwidth, transmission power, battery limitations, computing power, etc. One of the promising approaches is distributed inference [10].

Earlier work on distributed inference splits a DNN into two parts, where the first few layers are computed in an end user device, while the rest of the layers are offloaded to an edge server [11]. In such a scenario, an end user does initial feature extraction and stops if it is confident about the result. Otherwise, the output of the first few layers are transmitted to a server to obtain a more confident result. This approach is extended in [12] to take advantage of parallel processing in the end user device and edge server by partitioning the layers between these two devices. The layer partitioning between an end user and an edge server and their corresponding computation are further considered in [13] by modeling DNN as a directed acyclic graph. This line of work is limited to two parallel processing devices and cannot take full advantage of edge computing systems with several end users and edge servers.

Model-distributed inference is emerging as a promising solution, where a DNN model could be distributed to more than two workers. Model distribution is originally considered for parallel DNN training to overcome communication and storage limitations and provide scalability for DNNs. GPipe [1] achieves close-to-linear speed up in DNN training. PipeDream [2], [4] eliminates idle workers and thus improves efficiency via a dedicated scheduling and batching mechanism. Weight prediction can be used to improve the performance of both GPipe and PipeDream [3]. A resilient model-distributed training schemes that are robust to failing or severely straggling workers is designed in [14]. Model-distributed DNN training is designed for memory-constrained edge devices in [15]. As compared to this line of work, we consider model distribution for inference rather than training.

A model partitioning problem for model-distributed inference is designed using dynamic programming [5] with the cost of high complexity. As compared to [5], our AR-MDI algorithm (i) reduces the computation cost as our algorithm makes on the fly calculations while [5] solves a dynamic programming problem, (ii) makes model distribution fully decentralized while [5] relies on a genie to solve the dynamic programming problem and make the corresponding model distribution, (iii) is adaptive to time-varying resources, and (iv) recovers if there is a failure in the network. Model distributed inference is considered for a multi-source setup in [16]. As compared to [16], our AR-MDI algorithm provides a lightweight and decentralized model allocation mechanism.

We note that there are various techniques to reduce the communication and memory cost in DNN inference such as pruning [17], gradient sparsification [18], [19], compression and quantization [20], [21], [22], [23], [24]. DNN partitioning between an end user and edge server is combined with pruning [25], [26], [27] to speed up inference time and reduce communication and memory costs. Our work is complementary to these techniques in the sense that the performance of AR-MDI can be further improved by employing one or more of these methods.

III. MODEL AND PRELIMINARIES

Setup and Topology: We consider an edge computing system comprised of end user devices, edge servers, and remote cloud (if available) for distributed DNN inference. One of the devices (source device) has an input dataset and would like to learn the output of a DNN model. The other devices behave as workers for model distributed inference.

We assume that there are N workers in the system, where $\mathcal{N} = \{\eta_1, \ldots, \eta_N\}$ is the set of workers. Workers form a Chord-like [28] circular overlay topology, where the source device becomes the first node, *i.e.,* n_1, in the topology as in Fig. 2. Such an overlay topology can be constructed by taking into account the location of each workers to reduce the number of transmissions in the lower-layers [29], [30].[1]

The workers have the following properties: *(i) Failures:* Workers may fail or "sleep/die" or leave the network before finishing their assigned computational tasks. *(ii) Stragglers:* Workers incur probabilistic delays while they are performing the computations that are assigned to them.

DataSet and DNN Model: Let K be the number of data/images in the source node that need to be processed by a DNN, where the set of data is represented by $\mathcal{K} = \{A_1, \ldots, A_K\}$. The DNN model consists of L layers. The set of layers is $\mathcal{L} = \{l_1, \ldots, l_L\}$, where l_i is the ith layer. The pre-trained model is represented with $\mathbf{w} = \{\mathbf{w}_i\}_{i=1}^{L}$, where $\mathbf{w}_i$ is the vector of weights associated with layer l_i. The number of parameters (weights) at layer i is W_i, *i.e.,* $|\mathbf{w}_i| = W_i$. The total number of parameters in the model is $W = \sum_{i=1}^{L} W_i$, where $|\mathbf{w}| = W$.

Model-Distributed Inference: Let us assume that the source node would like to process the kth data, *i.e.,* A_k. The set of layers that worker η_n computes for processing data A_k is $\Lambda_n(k)$, where $\Lambda_n(k) = \{\lambda_n^1(k), \ldots, \lambda_n^i(k), \ldots, \lambda_n^{|\Lambda_n(k)|}(k)\}$ and $\lambda_n^i(k) \in \mathcal{L}$. The last layer of $\Lambda_n(k)$ is denoted as $l_n(k) = \lambda_n^{|\Lambda_n(k)|}(k)$. We assume that $\Lambda_i(k) \cap \Lambda_j(k) = \emptyset$, $\forall i \neq j$ as two different workers do not process the same layers.

According to the model-distributed inference, the source node takes the kth data A_k, and calculates the activation vectors of all the layers in $\Lambda_1(k)$. The activation vector of

1. The topology construction can be optimized in conjunction with the model allocation yet with increased cost. Thus, we consider that topology is constructed by following [29], [30].

the first layer, *i.e.,* $a_1(k)$ is calculated as $a_1(k) = \mathbf{w}_1 A_k$. The activation vectors of the next layers are calculated as $a_i(k) = \mathbf{w}_i a_{i-1}(k)$, $\forall i \in \Lambda_1(k)$ and $i \neq 1$. The source node sends the activation vector of its last layer, *i.e.,* $a_{l_n(k)}(k)$ to its next hop neighbor, which is worker η_2.

Each worker η_n $(n \neq 1)$ calculates $a_i(k) = \mathbf{w}_i a_{i-1}(k)$, $\forall i \in \Lambda_n(k)$ and sends the output of the last layer $a_{l_n(k)}(k)$ to worker $\eta_{(n+1) \bmod N}$. Note that $\bmod N$ is needed due to the circular structure of the overlay topology. If $a_{l_n(k)}(k)$ is the output of the last layer of the DNN model, *i.e.,* $(n+1) \bmod N = 1$ is satisfied, it is transmitted to the source node.

Above process can be pipe-lined in a way that the source node can start processing $k+1$th data immediately after calculating $a_1(k)$. Thus, in a homogeneous scenario, all devices can be occupied and compute layers in parallel. The crucial questions in this context are: (i) What is the potential of model-distributed inference as compared to data-distributed inference, which can also do parallel processing?; and (ii) How to exploit the potential of model distribution in a heterogeneous setup? The next two sections address these questions.

IV. POTENTIAL OF MODEL DISTRIBUTED INFERENCE

Model-distributed inference is promising to reduce the communication cost and memory usage. Its potential regarding memory usage is obvious as it does not require storing and processing the whole model in workers, Fig. 2, while it is required in data-distributed inference, Fig. 1. However, its potential in terms of reducing the communication cost, hence improving the inference time, is not obvious. Therefore, in this section, we provide inference time analysis for both model- and data-distributed inferences in a homogeneous setup, where the computing power of workers as well as the transmission delay among workers are the same.[2]

Let d^D and d^M denote the processing time of each data sample at any given worker for DDI and MDI schemes, respectively. The two parameters can be related as $d^M = \frac{d^D}{N}$, which follows thanks to distributing the DNN model across workers in the MDI scheme. In other words, each worker in MDI processes $(1/N)$th of the layers, while in DDI it processes all the layers. Let τ^D denote the transmission time of data from the source node to a worker in DDI. Also, let τ^M denote the transmission time of an activation vector from one worker to another worker in the MDI scheme. Depending on the data size, actual network environment, and the DNN model, either one of τ^D or τ^M may be larger than the other. The following theorem characterizes the inference times for the MDI and the DDI schemes in terms of the processing times and the delay parameters of the workers.

Theorem 1: Let D^M and D^D denote that total inference time for processing K data samples in dataset $\mathcal{K}$ for the MDI and the DDI schemes, respectively. We have $D^D = K \max\{\frac{d^D}{N}, \tau^D\} + o(K)$ and $D^M = K \max\{\frac{d^D}{N}, \tau^M\} + o(K)$, as $K \to \infty$.

2. We denote model-distributed inference with MDI and data distributed inference with DDI in the rest of the paper for brevity.

Proof: We first calculate the inference time for the DDI scheme. Suppose that K is a multiple of N. According to the DDI scheme, we divide the K samples to blocks of N samples each. It takes $N\tau^D$ seconds to transmit one of the blocks of length N for the data-distributed inference to N workers. These data samples can be processed in d^D seconds in parallel. The total inference time for $\frac{K}{N}$ blocks is thus at least $\frac{K}{N} \max\{d^D, N\tau^D\} = K \max\{\frac{d^D}{N}, \tau^D\}$ seconds. For an upper bound, note that a total of τ^D seconds may be required for initial transmission of packets, after which reception of new packets and processing of previous packets may be performed concurrently. This results in the upper bound $D^D \leq (K+1) \max\{\frac{d^D}{N}, \tau^D\}$, and thus concludes the proof of theorem for the case of the DDI scheme. An MDI scheme can be thought as N single-client DDI schemes running in parallel. We thus obtain $D^M = K \max\{d^M, \tau^M\} = K \max\{\frac{d^D}{N}, \tau^M\}$. This concludes the proof of the theorem. ■

Theorem 1 shows that for large N values, the total inference times of MDI and DDI reduces to $D^M = K\tau^M$ and $D^D = K\tau^D$. This means that if $\tau^M \leq \tau^D$, MDI is always better than DDI. Noting that τ^D and τ^M correspond to transmission time of data and activation vectors, respectively, we can conclude that MDI performs better when the input data size is large. We note that our analysis in this section assumes the homogeneity of the resources, but it can still give us some guidance in a heterogeneous setup in the next section when layers are distributed across workers considering the resource limitations of workers as well as transmission delay.

V. AR-MDI: ADAPTIVE AND RESILIENT MDI

In this section, we formulate an optimization problem to determine the optimal allocation of DNN layers across workers for the model-distributed inference in a heterogeneous setup. Our goal is to get better performance from MDI by optimizing the allocation of layers across workers by taking into account their processing and transmission delays. Then, based on the structure of the optimization problem, we design AR-MDI, which has two modules; practical model allocation and recovery.

Optimization Formulation: Let us assume that the number of parameters at worker η_n for processing layers in $\Lambda_n(k)$ is $\rho_n(k) = \sum_{\forall l \in \Lambda_n(k)} W_l$. The per parameter computing delay at worker η_n while processing data A_k is $\gamma_n(k)$. The per parameter transmission delay from worker η_n to its next neighbor while transmitting the output of $l_n(k)$ is $\theta_n(k)$. Thus, the processing delay at worker η_n for data A_k becomes $d_n(k) = \gamma_n(k)\rho_n(k)$ and the transmission delay becomes $\tau_n(k) = \theta_n(k) W_{l_n(k)}$. We note that the superscript M is omitted from $d_n(k)$ and $\tau_n(k)$ to simplify the notation.

The goal of our optimization formulation is to determine the layers that should be assigned to each worker, *i.e.,* to determine the model allocation policy $\Lambda(k) = \{\Lambda_1(k), \ldots, \Lambda_N(k)\}$ that minimizes the inference time for each data A_k. The set of all possible model allocation policies is determined by $\mathcal{S}(k)$, where $\Lambda(k) \in \mathcal{S}(k)$. Thus, our

problem is formulated as

$$\min_{\Lambda(k)\in\mathcal{S}} \max_{\forall n}\{\max\{d_n(k), \tau_n(k)\}\} \quad (1)$$

The inner max term in (1), *i.e.,* $\max\{d_n(k), \tau_n(k)\}$ comes from the fact that each device is able to do computations of previously received data while receiving new data. The outer max term holds, because model is distributed and workers process layers in parallel. The worker with the largest delay determines the inference time per image.[3] Thus, our optimization problem in (1) determines the optimal layer allocation that minimizes the inference time for each data.

The solution of (1) needs to check all possible combinations of layer allocations, which introduces a computation cost. Also, such a solution requires a centralized coordinator to solve the problem, make allocations, and inform each device to determine which layers to process. Such a centralized approach is not ideal in edge computing systems, where the centralized controller may fail or become a bottleneck of the system (both in terms of computation and communication). Thus, we resort to a more practical solution building on the solution of (1). As seen from (1), the optimization problem equalizes the delay across workers by minimizing the maximum delay. We follow the same approach by splitting computing and transmission delays. Although such a split looses optimality, our implementation results in the next section shows the efficiency of our algorithm.

Practical Model Allocation: We first assume that the computing delay $d_n(k)$ is the dominant term while $\tau_n(k)$ is small in (1). Under this assumption, the optimization problem in (1) reduces to $\min_{\Lambda(k)\in\mathcal{S}} \max_{\forall n}\{d_n(k)\}$. The next theorem provides a compact solution to this problem assuming that the number of coefficients at worker devices, i.e., $\rho_1(k), \ldots, \rho_N(k)$ could be arbitrary real numbers satisfying the constraint $\sum_{n=1}^{N} \rho_n(k) = W$.

Theorem 2: For $d_n(k) = \gamma_n(k)\rho_n(k)$ and $\rho(k) = \{\rho_1(k), \ldots, \rho_N(k)\}$, the solution to the optimization problem $\min_{\rho(k)\in\mathbb{R}^N} \max_n d_n(k)$ subject to the constraint $\sum_{n=1}^{N} \rho_n(k) = W$ is given by

$$\rho_n^*(k) = W\frac{1/\gamma_n(k)}{\sum_{n=1}^{N} 1/\gamma_n(k)}. \quad (2)$$

Proof: In the following, we drop the dependence on k for brevity. We observe that the optimal solution can satisfy

$$\rho_n^\star \gamma_n = \rho_m^\star \gamma_m \quad (3)$$

for every $m, n \in \{1, \ldots, N\}$ with $m \neq n$. Otherwise, if the equality does not hold for certain indices i and j, one can appropriately tune $\rho_i^\star$ and $\rho_j^\star$ so that $\max\{\rho_i^\star\gamma_i, \rho_j^\star\gamma_j\}$ is decreased, $\rho_i^\star + \rho_j^\star$ remains the same (so that the constraint is not violated), and $\rho_i^\star\gamma_i = \rho_j^\star\gamma_j$. Repeated application of the tuning process, if necessary, thus provides an optimal solution that satisfies (3). According to (3), we can then obtain

$$\rho_m^\star = \rho_1^\star\gamma_1/\gamma_m,\ m \in \{2, \ldots, N\}. \quad (4)$$

Substituting (4) to the constraint $\sum_{n=1}^{N} \rho_n^\star = 1$, we can solve for $\rho_1^\star$. Substituting the expression for $\rho_1^\star$ to (4), we can obtain the remaining weight allocations $\rho_2^\star, \ldots, \rho_N^\star$. This concludes the proof of the theorem. ■

Now, we assume that the communication delay $\tau_n(k)$ is the dominant term in (1), which reduces the optimization problem in (1) to $\min_{\Lambda(k)\in\mathcal{S}} \max_{\forall n}\{\tau_n(k)\}$, where $\tau_n(k) = \theta_n(k)W_{l_n(k)}$. As the size of $W_{l_n(k)}$ does not differ across layers dramatically except for a few fully connected layers, we consider it as constant. Thus the solution to $\min_{\Lambda(k)\in\mathcal{S}} \max_{\forall n}\{\tau_n(k)\}$ becomes $\rho_n^*(k) = W/N$.

We design the model allocation module of AR-MDI by constantly measuring computing and communication delays. When measured computing delay is larger than the communication delay, $\rho_n(k)$ is determined according to (2). Otherwise, $\rho_n(k)$ is set to W/N. The model allocation module of AR-MDI is summarized in Algorithm 1.

Algorithm 1 has two key features: (i) Although $\rho_n(k)$ could be any real number in our solutions, this should be converted to a value of feasible model allocation noting that $\Lambda(k) \in \mathcal{S}(k)$ should be satisfied; and (ii) The implementation should be performed in a decentralized manner. This means that each worker η_n should be able to decide which layers that it should compute.[4]

Each worker η_n receives an activation vector $a_{l_{n-1}(k)}(k)$ from the previous worker as shown in line 5 of the algorithm. In addition to the activation vector, it receives the index of the layer that should be processed next, *i.e.,* $i_{n-1}(k)$ as well as partial and complete rates $\delta_{n-1}(k)$ and $\Delta(k-1)$ from the previous node. The partial rate $\delta_{n-1}(k)$ corresponds to the rate in the denominator of (2), *i.e.,* $\delta_{n-1}(k) = \sum_{z=1}^{n-1} 1/\gamma_z(k)$.[5] On the other hand, $\Delta(k-1)$ corresponds to the whole term, *i.e.,* $\Delta(k-1) = \sum_{z=1}^{N} 1/\gamma_z(k-1)$. Each worker calculates their partial rate by summing up the previous worker's partial rate and its own rate, *i.e.,* $\rho_n(k)/d_n(k)$ as shown in line 34. The partial rate at the last node η_N corresponds to the total rate, so $\delta_N(k-1)$ is set to $\Delta(k-1)$ in line 9.

The number of parameters that worker η_n can process for data A_k is determined in line 11 by mimicking (2). Indeed, each worker records the number of parameters it had for the previous data $\rho_n(k-1)$, delay $d_n(k-1)$ as well as the total rate $\Delta(k-1)$. Using these parameters, it can calculate the number of parameters for data A_k, *i.e.,* $\rho_n(k)$. The notation of $\lfloor . \rceil$ shows that the result is rounded to the nearest integer.

3. We note that AR-MDI has a recovery mechanism from very delayed (straggling) and unresponsive workers as described later in this section.

4. We assume that each worker η_n has the whole model and depending on the model allocation, some layers, *i.e.,* $\Lambda_n(k)$ set, are activated for computation. This can be relaxed in memory-constrained devices by exchanging layers among workers when needed.

5. We call these parameters "rates" as their unit is the number of parameters over processing delay, so the rate term better fits in this context.

Algorithm 1 AR-MDI Model Allocation Logic at Worker η_n for data A_k

1: Initialize: Randomly allocate $\Lambda_n(1)$, $\forall n$. $\delta_0(k) = 0$. $\Delta(0) = 0$. $i_n(1) = 0$.
2: **if** n == 1 **then**
3: $\quad n - 1 \leftarrow N$
4: **end if**
5: Receive: $a_{l_{n-1}(k)}(k)$, $\delta_{n-1}(k)$, $i_{n-1}(k)$, $\Delta(k-1)$ from $\eta_{(n-1)}$
6: **if** k != 1 **then**
7: $\quad \Lambda_n(k) = \emptyset$
8: $\quad$ **if** n==1 **then**
9: $\quad\quad$ Set $\Delta(k-1) = \delta_N(k-1)$
10: $\quad$ **end if**
11: $\quad$ Calculate $\rho_n(k) = \left\lfloor W\frac{\rho_n(k-1)/d_n^M(k-1)}{\Delta(k-1)}] \right\rceil$
12: $\quad$ **if** $\tau_n(k-1) > d_n(k-1)$ **then**
13: $\quad\quad$ Calculate $\rho_n(k) = \lfloor W/N \rceil$
14: $\quad$ **end if**
15: $\quad$ **if** n != N **then**
16: $\quad\quad$ **if** n == 1 **then**
17: $\quad\quad\quad \Lambda_1(k) = \Lambda_1(k) \cup l_1$
18: $\quad\quad\quad \rho_1(k) = \rho_1(k) - W_1$, $i_n(k) = 2$
19: $\quad\quad$ **end if**
20: $\quad\quad$ **while** $\rho_n(k) > W_{i_n(k)}$ **do**
21: $\quad\quad\quad \Lambda_n(k) = \Lambda_n(k) \cup l_{i_n(k)}$
22: $\quad\quad\quad \rho_n(k) = \rho_n(k) - W_{i_n(k)}$, $i_n(k) = i_n(k) + 1$
23: $\quad\quad$ **end while**
24: $\quad\quad$ **if** $|\rho_n(k) - W_{i_n(k)}| < W_{i_n(k)}/2$ **then**
25: $\quad\quad\quad \Lambda_n(k) = \Lambda_n(k) \cup l_{i_n(k)}$, $i_n(k) = i_n(k) + 1$
26: $\quad\quad$ **end if**
27: $\quad$ **else**
28: $\quad\quad \Lambda_N(k) = \Lambda_N(k) \cup \{l_{i_n(k)}, \ldots, l_L\}$
29: $\quad$ **end if**
30: **end if**
31: Calculate all output vectors $a_l(k)$, $\forall l \in \Lambda_n(k)$
32: Measure delay $d_n(k)$ for processing all the layers in $\Lambda_n(k)$
33: Calculate $\rho_n(k) = \sum_{l \in \Lambda_n(k)} W_l$
34: Calculate the partial rate $\delta_n(k) = \delta_{n-1}(k) + \rho_n(k)/d_n(k)$
35: Send $a_{l_n(k)}(k)$, $\delta_n(k)$, $i_n(k)$, $\Delta(k-1)$ to $\eta_{(n+1) \mod N}$ and measure the communication time $\tau_n(k)$

Lines 12-14 update the value of $\rho_n(k)$ if the communication delay is larger than the computing delay.

The value of $\rho_n(k)$ can be any integer according to the calculation in line 11. Therefore, our algorithm converts this value to a model allocation so that $\Lambda(k) \in \mathcal{S}$ is satisfied. Lines 16-19 make sure that the source node η_1 should always process the first layer. The other layers are allocated in lines 20-23. Since worker η_n knows the index of the layer that should be processed next, it starts from that layer and allocates all the layers as long as $\rho_n(k)$ is larger than the sum of all the parameters in these layers. Lines 24-26 handle the last layer allocation. If the number of parameters left after allocating all the previous layers is close to half of the parameters in the next layer, the next layer is also assigned to worker η_n. Finally, the last worker η_N processes all the layers that are not assigned to previous layers in line 28.

After the model allocation is completed in worker η_n, it calculates all of the activation vectors of the layers that it processes in line 31. It measures the delay $d_n(k)$ for processing all the layers in $\Lambda_n(k)$ in line 32. The number of parameters that node η_n processes is calculated as $\rho_n(k) = \sum_{l \in \Lambda_n(k)} W_l$ in line 33. Note that this calculation corresponds to the actual number of parameters while the terms in line 11 or 13 only give an approximate value of the number of parameters. The partial rate is calculated in line 34 using the actual number of parameters and measured delay. Finally, worker η_n transmits $a_{l_n(k)}(k)$, $\delta_n(k)$, $i_n(k)$, $\Delta(k-1)$ to the next worker and measures the communication time $\tau_n(k)$.

Computational Complexity: Let us now analyze the computational complexity and convergence of our AR-MDI algorithm as provided by Algorithm 1. The main result is summarized via the following proposition.

Proposition 1: The per-input per-worker time complexity of Algorithm 1 is at most a constant multiple of L, where L is the number of layers of the neural network that is distributed. Moreover, Algorithm 1 converges after a finite number of steps that is within a constant factor of L at every worker.

Proof: Given a fixed data to the distributed neural network (a fixed k), inspection of Algorithm 1 reveals that the only time complexity stems from the while loop in Lines 20-23. By the definition of $\rho_n(k)$ in Line 11, the number of loops is at most L. The finite convergence time also follows immediately. ∎

Note that the complexity bounds as provided by Proposition 1 are worst case complexities and are very low as compared to the solution of the original optimization problem in (1). The latter needs to look at all possible model allocation combinations. By elementary combinatorics, the complexity in this case is $O\left(\binom{L}{N}\right)$, which grows much faster than $O(NL)$ overall worst-case complexity of Algorithm 1. In particular, in the practical regime of interest of deep networks, where the number of layers L is large and the number of workers N is comparably lower/constant, the complexities of exhaustive search and Algorithm 1 are $O(L^{N-1})$ and $O(L)$, respectively.

Recovery: Our AR-MDI design relies on Algorithm 1 to equalize computing delay across workers. Yet, the transmission delay, especially in the case of unresponsive workers, is a very important factor that affects the interference time. Therefore, in this section, we design a recovery algorithm against delayed and failing workers with the goal of keeping the composite delay low across workers.

Our design idea is inspired by the peer management mechanism of Chord-like P2P systems [28]. All workers are labeled with identifiers, *e.g.,* worker η_n's identifier is n. All workers know the identifiers and IP addresses of their

neighboring nodes as well as one-hop away neighbors. For example, worker η_n keeps track of the identifiers and IP addresses of its previous two workers, *i.e.,* $\eta_{\min\{n-2,0\}+N}$ and $\eta_{\min\{n-1,0\}+N}$, $n \geq 1$ as well as its next worker, *i.e.,* $\eta_{(n+1) \mod N}$.

Each worker η_n also keeps track of the delay of its previous neighbor. Let us assume that the time that η_n receives the output vector from the previous node or data A_{k-1} and A_k at time $t_n(k-1)$ and $t_n(k)$, respectively. We define an inter-arrival delay between two activation vectors as $\mu_n(k) = \alpha\mu_n(k-1) + (1-\alpha)(t_n(k) - t_n(k-1))$, where $0 < \alpha < 1$ is a scaling factor.

After receiving an activation vector corresponding to data A_k, worker η_n waits for $\beta\mu_n(k)$ time duration, where β is a constant that can be determined by looking at the variance of inter-arrival delay. If worker η_n does not receive a new activation vector while waiting for $\beta\mu_n(k)$ time duration, then it concludes that worker $\eta_{\min\{n-1,0\}+N}$ is delayed too long or not responsive, so it contacts worker $\eta_{\min\{n-2,0\}+N}$. Workers $\eta_{\min\{n-2,0\}+N}$ and η_n are connected to each other, and $\eta_{\min\{n-2,0\}+N}$ sends its last activation vector to η_n, which processes all the layers in $\Lambda_{\min\{n-1,0\}+N}(k) \cup \Lambda_n(k)$. A new model allocation is performed for data A_{k+1} according Algorithm 1.

If worker $\eta_{\min\{n-1,0\}+N}$ would like to participate in the system, it should send small messages and convince η_n that it is responsive again. In particular, η_n calculates inter-arrival time for the messages that $\eta_{\min\{n-1,0\}+N}$ is sending. If this inter-arrival time is comparable with the inter-arrival time of its current previous hop $\eta_{\min\{n-2,0\}+N}$, then η_n disconnects from $\eta_{\min\{n-2,0\}+N}$ and connects to $\eta_{\min\{n-1,0\}+N}$.

VI. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our algorithm; AR-MDI in a real and heterogeneous testbed of NVIDIA Jetson TX2s. First, we describe the datasets, DNN models, and the testbed. We then provide the corresponding evaluation results.

A. DATASETS, DNN MODELS AND TESTBED DESCRIPTION

We utilize the standard ImageNet [7] and CIFAR-10 [6] datasets for image classification. ImageNet contains more than 1 million training images, 50,000 validation images, 100,000 test images and 1,000 object classes. The average image resolution on ImageNet is 469×387 pixels. A pre-processing step typically clips or downsamples the images to 256×256. CIFAR-10 contains 32×32 color images in 10 different classes. There are a total of 60,000 images, 10,000 of which are test images. We use the MobileNetV2 [9] or VGG16 [8] as the inference models, where MobileNetV2 is 53 layers deep, and VGG16 is 16 layers deep.

Our testbed consists of NVIDIA Jetson TX2 cards, which we shall hereby simply refer to as "Jetson"s. Each Jetson incorporates an NVIDIA Pascal Architecture GPU and dual Denver 64-bit CPUs. The Jetsons are connected to each other through Wi-Fi, where an access point in the link layer provides a connected topology. We note that we implement this topology as a proof-of-concept noting that our AR-MDI algorithm operates over a circular topology in the overlay and it works with any lower-layer topology.

We compare our algorithm AR-MDI with baselines; (i) EdgePipe [5], where model distribution is used for inference and the inference time is minimized as a solution of a dynamic programming problem, (ii) DDI, where data is distributed across workers, not the model itself, (iii) Local, where the source node has the complete DNN model and prefers to process all the data by itself, and (iv) Optimal, which is a solution to the optimization problem in (1). However, since the solution of (1) is computationally intensive, it incurs very high processing time in real implementation. Thus, we solved the optimal solution in (1) considering all possible computing and communication delay ranges for this setup, and recorded the solutions. Then, we used the recorded solutions without changing any parameters in the setup, which returned the baseline "Optimal". We note that this approach is not practical as it is not feasible to solve the optimal solution for all possible delays in every setup. The only purpose of considering the baseline Optimal in this experiment is to evaluate the performance of AR-MDI as compared to the optimal solution. We present the experimental results of our AR-MDI algorithm as compared to the baselines.

B. RESULTS

Small Data: As we discussed in Section IV, we can exploit the full potential of MDI in a scenario that data size is large. To confirm this analysis empirically, we first evaluate AR-MDI with CIFAR-10 dataset, where the size of data is small; 32×32 color images. We consider a homogeneous scenario, where all workers are Jetson's and they are not doing any other computationally intensive operations other than processing the layers that are allocated to them.

Fig. 3 shows the inference time, which we also call processing time, results on the CIFAR-10 dataset using the VGG16 model versus the number of images for five Jetsons. Due to the inherent computational complexity of the layer allocation algorithm, Edgepipe performs the worst in this scenario. Indeed, EdgePipe checks all possible model allocations, so its complexity is exponential. Since, EdgePipe performs model allocation at the start, it has high processing time even though the number of images is small. In fact, Local performs better than EdgePipe for small number of images. EdgePipe performs better than Local for larger number of images thanks to using parallel processing across workers. We can observe that DDI performs slightly better than AR-MDI in this scenario. We actually expected this result as AR-MDI performs better for large images, but CIFAR-10 images are small. As seen, AR-MDI is very close to the optimal solution, which shows the effectiveness of our design.

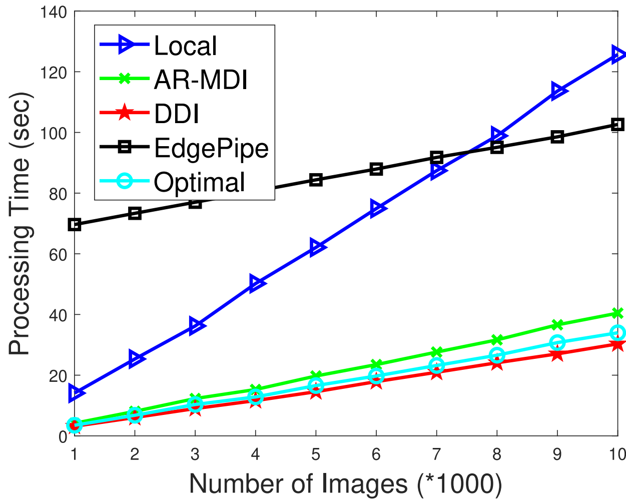
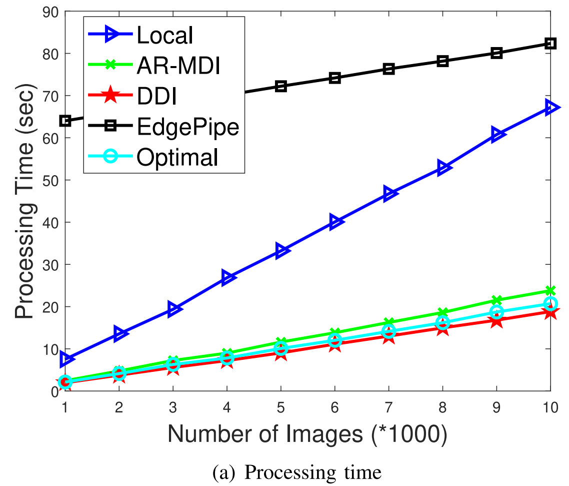
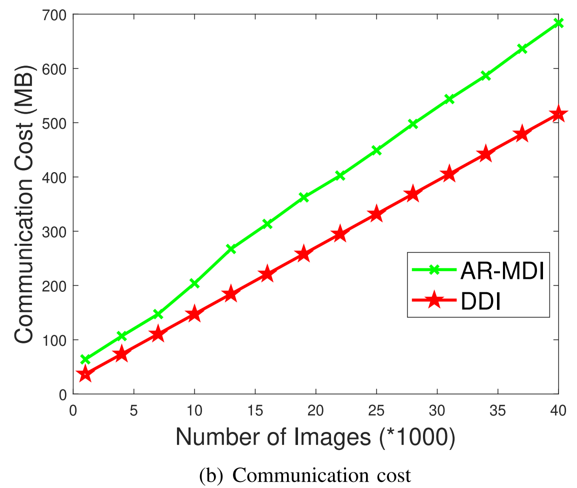


FIGURE 3. Processing time versus the number of images for CIFAR-10 using VGG16. Homogeneous setup.



(a) Processing time



(b) Communication cost

FIGURE 4. Processing time and the communication cost versus the number of images on CIFAR-10 using MobileNetV2. Homogeneous setup.

The next scenario we consider is CIFAR-10 dataset using the MobileNetV2 model. Our goal is to understand the impact of the model size on the performance of our algorithm noting that MobileNetV2 is a larger model than VGG16. Fig. 4(a) shows this scenario for five Jetsons. As seen, DDI is still slightly better than AR-MDI, which confirms our analyis in Section IV that the most important factor is the size of the data, not the model. Indeed, Fig. 4(b) shows the communication cost versus the number of images in this setup. As seen, the communication cost of DDI is lower than AR-MDI, so it performs better. Fig. 4(a) shows that EdgePipe performs worse than in Fig. 3 as the computational complexity of EdgePipe depends on the size of the DNN model. Similar to Fig. 3, AR-MDI is close to the optimal solution in Fig. 4(a).

Large Data: Now, we evaluate AR-MDI with ImageNet dataset, where the size of data is large; 256 × 256 color images. We consider a homogeneous scenario as in the "Small Data" setup.

Fig. 5 shows the inference results on the ImageNet dataset using the VGG16 model for five Jetsons. We observe that the processing times for Edgepipe, MDI, and DDI have a similar growth rate, which is indicated by the slope of the corresponding lines. However, Edgepipe has a larger starting point for the first image due to the time it takes to run the computationally-complex model allocation algorithm for the first time. Although EdgePipe takes as much computation time as in Fig. 3, it looks negligible in Fig. 5, because the overall processing time is higher in this scenario. AR-MDI reduces the processing time as compared to DDI and close to Optimal. This is expected as the size of the data is larger in this scenario. On the other hand, the improvement of AR-MDI as compared to EdgePipe is small. This is also expected as EdgePipe makes its layer allocation for the first image and uses the same allocation for all images. This works perfectly fine as this is a homogeneous scenario. Local performs the worst as processing a dataset with large input images is costly on a single device.

Fig. 6(a) shows the inference results on the ImageNet dataset using MobileNetV2 for five Jetsons. As seen, EdgePipe performs worse in this scenario as compared to Fig. 5 as MobileNetV2 is a deeper model than VGG16 and the computational complexity of EdgePipe increases with increasing DNN size. Thus, AR-MDI reduces the processing time as compared to EdgePipe thanks to its light weight model allocation, and its very close the Optimal. AR-MDI is still better than DDI as the image size in ImageNet is large. Fig. 6(b) shows the communication cost versus the number of images. As seen, the communication cost of DDI is very high as compared to AR-MDI, so AR-MDI reduces the processing time significantly. In fact, AR-MDI reduces the processing time by 80.2% and 44.5% as compared to DDI and EdgePipe, respectively in Fig. 6(a). The processing time performance of the Local is the worst, which is expected, so we do not include Local in the rest of the experiments.

Heterogeneous Setup: So far we considered a homogeneous setup where Jetsons only process the

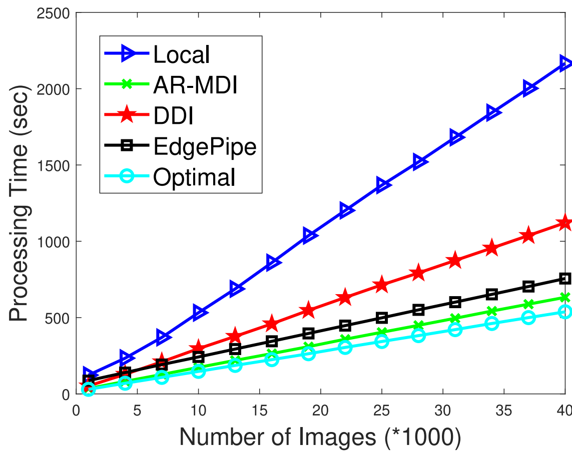
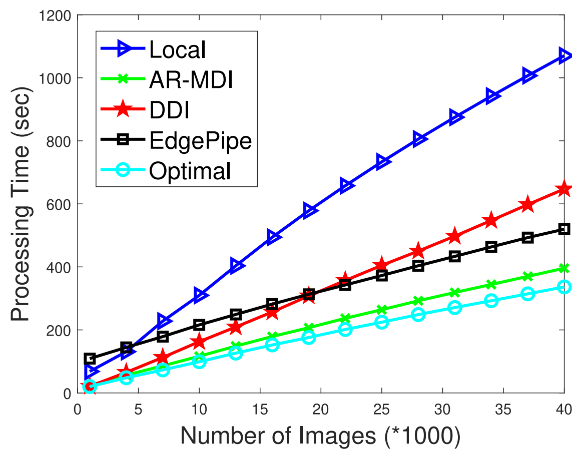
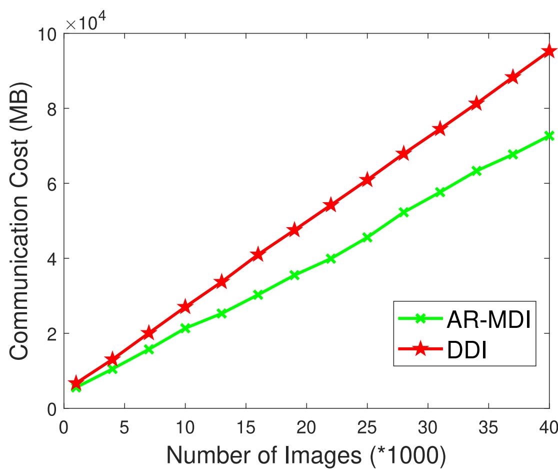


FIGURE 5. Processing time versus number of images on ImageNet using VGG16. Homogeneous setup.



(a) Processing time



(b) Communication cost

FIGURE 6. Processing time and the communication cost versus the number of images on ImageNet using MobileNetV2. Homogeneous setup.

layers that are allocated to themselves. Now, we consider a more realistic scenario, where Jetsons have other tasks to compute. In particular, two out of five Jetsons trains

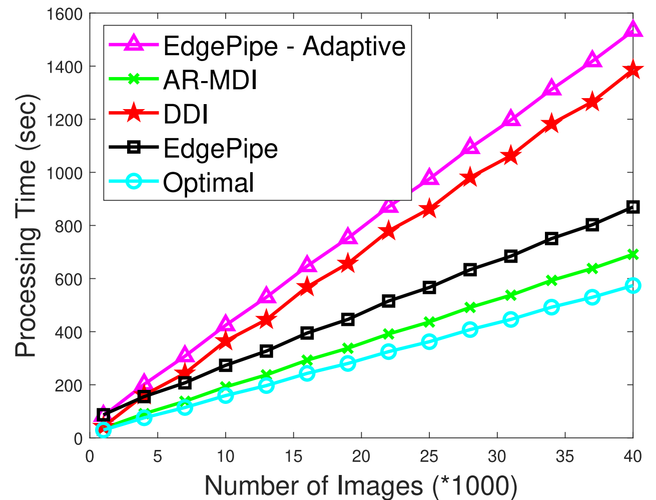


FIGURE 7. Processing time versus the number of images on ImageNet using VGG16. Heterogeneous setup.

a VGG16 model with ImageNet dataset in an ON/OFF manner. In particular, while doing inference computations for the 3000 images, VGG16 training continues, but it stops for the next 3000 images. This ON/OFF computation load continues until the end of the experiment.

In Fig. 7 shows the inference results on the ImageNet dataset using VGG16 for five Jetsons. As seen, AR-MDI improves the processing time as compared to both EdgePipe and DDI and close to Optimal thanks to its adaptive nature to varying computing resources. EdgePipe performs worse as its model allocation is not adaptive to time varying resources. The improvement of AR-MDI as compared to EdgePipe and DDI are 25.7% and 100.6%, respectively. To make the comparison fair, we also implemented EdgePipe - Adaptive, which re-allocates the model every time available resources change. This performs the worst as the model allocation algorithm of EdgePipe is computationally expensive, so introduces too much delay.

Fig. 8 shows the inference results on the ImageNet dataset using MobileNetV2 for five Jetsons. As seen, AR-MDI is very close to the optimal solution; Optimal is better than AR-MDI only by 11.7%, which shows the effectiveness of our design. Furthermore, the improvement of AR-MDI is higher as compared to EdgePipe in this setup as MobileNetV2 is a larger DNN model. In particular, the improvement of AR-MDI as compared to EdgePipe and DDI are 68% and 88%, respectively. The performance of EdgePipe-Adaptive is the worst as it computes its computationally intensive model allocation algorithm every time resources change. As seen, AR-MDI adapts to the time-varying resources very well without introducing additional computational complexity thanks to its lightweight model allocation mechanism.

Impact of Number of Workers: Now, we consider the performance of our AR-MDI algorithm as well as baselines for a varying number of workers. Fig. 9 shows processing time of 40,000 images versus the number of devices for

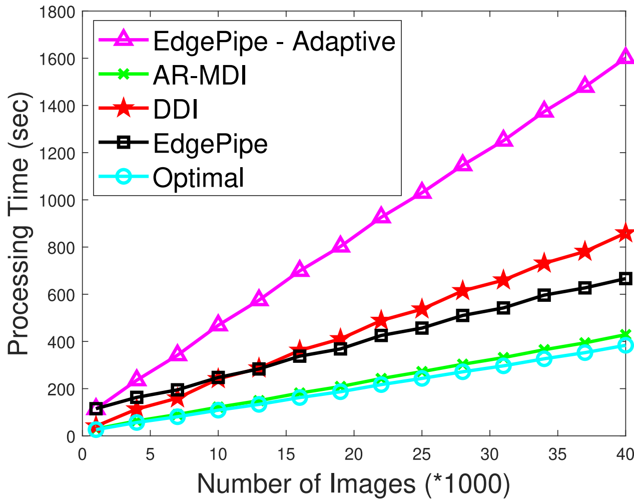


FIGURE 8. Processing time versus the number of images on ImageNet using MobileNetV2. Heterogeneous setup.

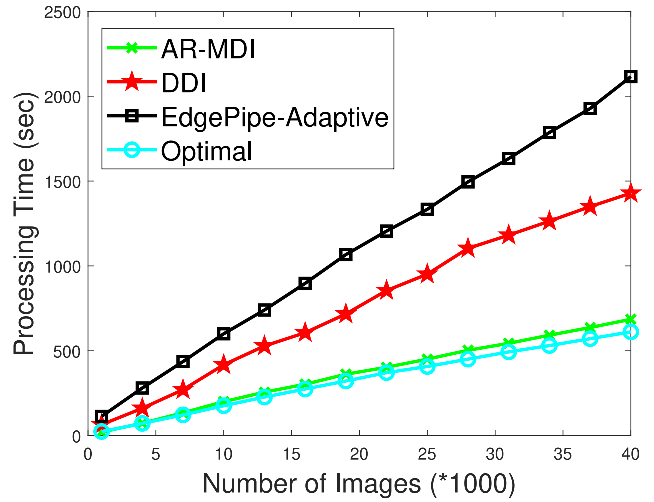


FIGURE 10. Processing time versus number of workers on ImageNet using MobileNetV2. Homogeneous setup in term of computing powers. One of the devices stops working randomly in an ON/OFF manner. AR-MDI uses its recovery module.

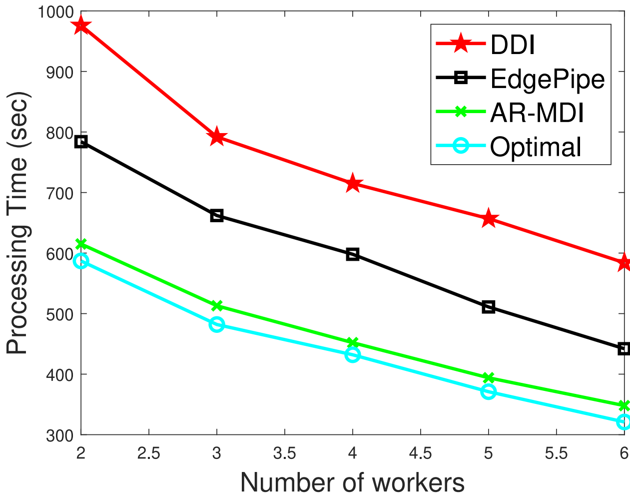


FIGURE 9. Processing time versus the number of workers for the ImageNet dataset using MobileNetV2 model. Homogeneous setup.

the ImageNet dataset and using the MobileNetV2 model. The resources are homogeneous in this scenario. We can observe that AR-MDI has the best performance and very close to Optimal, thanks to its lightweight and on the fly model allocation mechanism. Edgepipe performs better then DDI since DDI incurs a lot of communication cost due to the large dimensions of the ImageNet images.

Recovery: Our AR-MDI algorithm is resilient against delayed and failing workers thanks to its recovery module. To demonstrate the efficiency of the recovery module of AR-MDI, we implement a worker failure model. There are five Jetson's in the system in a circular overlay topology. One of the Jetson's stops working in an ON/OFF manner. The ON and OFF durations are random values selected from an exponential distribution with mean 30 sec.

Fig. 10 presents the processing time versus number of images for dataset ImageNet and model MobileNetV2. EdgePipe does not work in this scenario as it needs to stop when a worker stops working, because it is not adaptive to delayed and failing workers. Thus, we used EdgePipe-Adaptive, which reallocates the model whenever a worker starts/stops working. Since the model allocation algorithm of EdgePipe is computationally intensive, we see that EdgePipe-Adaptive performs poorly. DDI on the other hand works opportunistically in a way that it uses all the workers available when processing images. As seen, AR-MDI significantly improves the processing time as compared to both EdgePipe-Adaptive and DDI thanks to re-allocating model when a worker starts/stops working. In particular, we see 2× and 3× improvement of AR-MDI as compared to DDI and EdgePipe, which is significant. This shows the importance of light-weight model allocation and recovery mechanisms of AR-MDI. Optimal is better than AR-MDI only by 12.4%, which shows the effectiveness of our design.

VII. CONCLUSION

In this paper, we focused on distributed inference in edge computing systems by exploring model distribution across workers. First, we analyzed the potential of model distributed-inference in edge computing systems. Then, we developed an Adaptive and Resilient Model-Distributed Inference (AR-MDI) algorithm based on our optimal model allocation formulation. Our AR-MDI algorithm performs model allocation on the fly and in a decentralized way and it is resilient against delayed workers and failures. We implemented AR-MDI in a real testbed consisting of NVIDIA Jetson TX2s. Our experiment results show that AR-MDI improves the inference time significantly as compared to baselines; EdgePipe and DDI when (i) the size of data is large and (ii) workers are delayed and failures occur. Our experiment results also show that AR-MDI performs very close to the optimal solution.

REFERENCES

[1] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

[2] A. Harlap et al., "PipeDream: Fast and efficient pipeline parallel DNN training," 2018, *arXiv:1806.03377*.

[3] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, "Efficient and robust parallel DNN training through model parallelism on multi-GPU platform," 2018, *arXiv:1809.02839*.

[4] A. Harlap, "Improving ML applications in shared computing environments," Ph.D. dissertation, Dept. Electr. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2019.

[5] Y. Hu et al., "Pipeline parallelism for inference on heterogeneous edge computing," 2021, *arXiv:2110.14895*.

[6] 2009, "The CIFAR-10 dataset," University of Toronto. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.

[10] M. Chen et al., "Distributed learning in wireless networks: Recent progress and future challenges," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 12, pp. 3579–3605, Dec. 2021.

[11] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2017, pp. 328–339.

[12] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[13] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Trans. Mobile Comput.*, vol. 20, no. 2, pp. 565–576, Feb. 2021.

[14] P. Li, E. Koyuncu, and H. Seferoglu, "ResPipe: Resilient model-distributed DNN training at edge networks," in *Proc. IEEE ICASSP*, 2021, pp. 3660–3664.

[15] P. Li, H. Seferoglu, V. R. Dasari, and E. Koyuncu, "Model-distributed DNN training for memory-constrained edge computing devices," in *Proc. IEEE Int. Symp. Local Metrop. Area Netw. (LANMAN)*, 2021, pp. 1–6.

[16] P. Li, H. Seferoglu, and E. Koyuncu, "Model-distributed inference in multi-source edge networks," in *Proc. IEEE ICASSP Timely Private Mach. Learn. Netw. Workshop*, 2023.

[17] J. Liu, S. Tripathi, U. Kurup, and M. Shah, "Pruning algorithms to accelerate convolutional neural networks for edge applications: A survey," 2020, *arXiv:2005.04275*.

[18] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, 2018, pp. 1299–1309.

[19] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, 2018, pp. 5973–5983.

[20] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2285–2294.

[21] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," 2014, *arXiv:1412.6115*.

[22] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*.

[23] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1737–1746.

[24] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," 2014, *arXiv:1412.7024*.

[25] M. Jankowski, D. Gündüz, and K. Mikolajczyk, "Joint device-edge inference over wireless links with pruning," in *Proc. IEEE 21st Int. Workshop Signal Process. Adv. Wireless Commun. (SPAWC)*, 2020, pp. 1–5.

[26] W. Shi, Y. Hou, S. Zhou, Z. Niu, Y. Zhang, and L. Geng, "Improving device-edge cooperative inference of deep learning via 2-step pruning," in *Proc. IEEE INFOCOM Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, 2019, pp. 1–6.

[27] J. Shao and J. Zhang, "Communication-computation trade-off in resource-constrained edge inference," *IEEE Commun. Mag.*, vol. 58, no. 12, pp. 20–26, Dec. 2020.

[28] I. Stoica et al., "Chord: A scalable peer-to-peer lookup protocol for applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003.

[29] Q. Le-Dang, J. McManis, and G.-M. Muntean, "Location-aware chord-based overlay for wireless mesh networks," *IEEE Trans. Veh. Technol.*, vol. 63, no. 3, pp. 1378–1387, Mar. 2014.

[30] S. Burresi, C. Canali, M. E. Renda, and P. Santi, "MeshChord: A location-aware, cross-layer specialization of chord for wireless mesh networks (concise contribution)," in *Proc. 6th Annu. IEEE Int. Conf. Pervasive Comput. Commun. (PerCom)*, 2008, pp. 206–212.