Redco: A Lightweight Tool to Automate Distributed Training of LLMs on Any GPU/TPUs

Bowen Tan[1,*] **Yun Zhu**[2], **Lijuan Liu**[2], **Hongyi Wang**[1], **Yonghao Zhuang**[1], **Jindong Chen**[2], **Eric Xing**[1,3,5], **Zhiting Hu**[4]
[1]Carnegie Mellon University, [2]Google Research, [3]Petuum Inc., [4]UC San Diego,
[5]Mohamed bin Zayed University of Artificial Intelligence

Abstract

The recent progress of AI can be largely attributed to large language models (LLMs). However, their escalating memory requirements introduce challenges for machine learning (ML) researchers and engineers. Addressing this requires developers to partition a large model to distribute it across multiple GPUs or TPUs. This necessitates considerable coding and intricate configuration efforts with existing model parallel tools, such as Megatron-LM, DeepSpeed, and Alpa. These tools require users' expertise in machine learning systems (MLSys), creating a bottleneck in LLM development, particularly for developers without MLSys background. In this work, we present *Red Coast (Redco)*, a lightweight and user-friendly tool crafted to automate distributed training and inference for LLMs, as well as to simplify ML pipeline development. The design of Redco emphasizes two key aspects. First, to automate model parallism, our study identifies two straightforward rules to generate tensor parallel strategies for any given LLM. Integrating these rules into Redco facilitates effortless distributed LLM training and inference, eliminating the need of additional coding or complex configurations. We demonstrate the effectiveness by applying Redco to a set of LLM architectures, such as GPT-J, LLaMA, T5, and OPT, up to the model size of 66B, and in the setting of multi-host. Second, we propose a mechanism that allows for the customization of diverse ML pipelines through the definition of merely three functions, eliminating redundant and formulaic code like multi-host related processing. This mechanism proves adaptable across a spectrum of ML algorithms, from foundational language modeling to complex algorithms like meta-learning and reinforcement learning. Consequently, Redco implementations exhibit much fewer code lines compared to their official counterparts. Redco is released under Apache License 2.0 at `https://github.com/tanyuqian/redco` [2].

1 Introduction

In recent years, the field of AI has witnessed profound advancements, predominantly attributed to the advent of LLMs with an impressive number of parameters, spanning from billions to hundreds of billions [16]. Notable examples include GPT-4 [7] and LLaMA [13]. Yet, the size of these LLMs presents distinct challenges in terms of model deployment for ML researchers and engineers. The primary challenge arises from the substantial memory requirements of LLMs, often exceed the capability of a single GPU or TPU. This necessitates the use of model parallelism, a technique that partitions the LLMs into various shards, subsequently distributing them across multiple devices or even different hosts. However, achieving this partitioning requires intricate engineering, including the formulation of a tensor-specific splitting strategy. While several specialized tools like DeepSpeed

[*] Work done during an internship at Google.
[2] The comprehensive version of this paper is available at `https://arxiv.org/pdf/2310.16355.pdf`.

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.

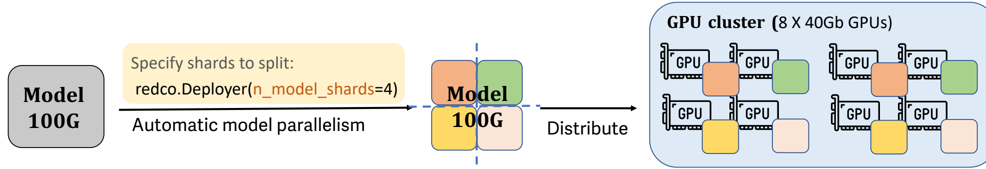

Figure 1: With a number of shards specified by user, Redco automatically conduct the model partitioning and distribution.

Server Device Memory	2 × 1080Ti 2 × 10G	4 × A100 4 × 40G	2 × TPU-v4 2 (hosts) × 4 (chips) × 32G	16 × TPU-v4 16 × 4 × 32G
2*Models	BART-Large (1024) GPT2-Large (512)	LLaMA-7B (1024) GPT-J-6B (1024)	T5-XXL-11B (512) OPT-13B (1024)	OPT-66B (512)

Table 1: Runnable model finetuning on different kinds of servers. Numbers inside the brackets are the maxinum length in training. All the settings are with full precision (fp32) with AdamW optimizer.

[10], Alpa [18], and FSDP [17] provide diverse model parallelism solutions, they demand significant additional coding and intricate configurations based on model architecture and hardware specifics, requiring in-depth understanding of MLSys. Such additional efforts make the deployment of LLMs particularly challenging, especially for those without MLSys expertise, such as algorithm developers or researchers. At times, the intricacy of coding for model parallelism proves even more daunting than the algorithm design itself. [3]

In this work, we introduce *Redco*, a lightweight and user-friendly tool designed to automate the distributed training and inference of LLMs, thereby users without prior knowledge in MLSys can effortlessly use the tool without additional coding and intricate configurations. Furthermore, we propose a novel and neat mechanism to implement ML algorithms. This method necessitates users to define merely three functions as their pipeline design, with Redco managing all the remaining details in execution, such as data parallelism, multi-host related processing, logging, etc.

Redco's design emphasizes two key aspects. The first is the automatic model parallelism. We identify two straightforward rules to generate the model parallel strategy for arbitrarily given transformer architecture, and integrate them into Redco. Unlike tools such as Megatron [12] and DeepSpeed [10] which require users to manipulate model forward function for different architecture and system specifics, Redco automates the process, where users only need to specify the desired number of shards to partition the model. We verified the effectiveness of Redco's model parallel strategy on multiple LLM architectures including LLaMA-7B [13], T5-11B [9], and OPT-66B [15]. Moreover, pipelines driven by Redco have proven to be more efficient than those implemented with FSDP [17]. They also closely match the performance of Alpa [18], the most advanced model parallelization tool.

Another pivotal feature of Redco is the neat mechanism for ML pipeline development. With Redco, users only need to write three intuitive functions to define a ML pipeline: a collate function to convert raw data examples into model inputs (e.g., text tokenization); a loss function to execute the model and compute loss (e.g., cross-entropy); and a predict function to run the model and deliver outcomes (e.g., beam search). With the defined pipeline from a user, Redco automates all the remaining of pipeline execution such as data parallelism, multi-host related processing, randomness control, log maintenance, and so forth. We demonstrate this neat mechanism is applicable to various ML paradigms, spanning from basic language modeling and sequence-to-sequence to more complex algorithms like meta-learning and reinforcement learning. Redco-based implementations consistently exhibit substantially fewer lines of code compared to their official counterparts.

2 Automatic Model Parallelism for LLMs

Model parallelism refers to distributing the computation of a large model across multiple GPUs or TPUs, in order to address the memory limitations of a single device. Two sub-paradigms within model parallelism are *pipeline parallelism* and *tensor parallelism*. Pipeline parallelism partitions the layers of the model across different devices, and tensor parallelism distributes every tensor in the model across multiple devices.

[3]We include more discussions of related work in the Appendix.

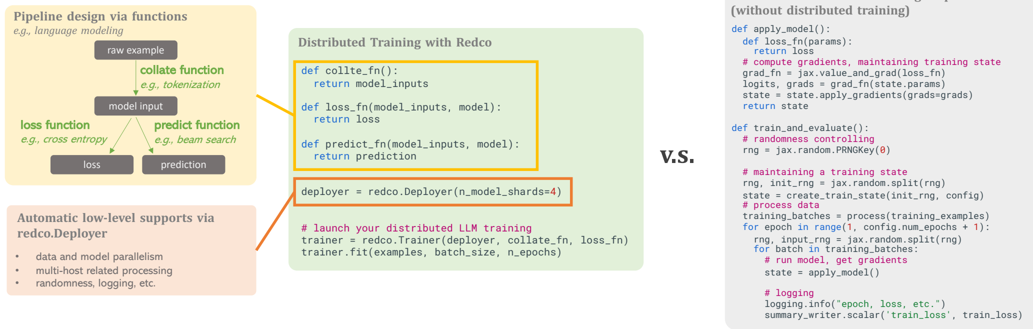

Figure 2: The template code of using redco to implement distributed training, where users only have to design a pipepine through three fucntions, without concerning data and model parallelism, multi-host related processing, randomness control, etc., which eliminates a lot of boilerplate coding.

Model parallel tools like Megatron or Alpa require a bunch of intricate configurations and extensively modifying users' code based on the model architecture and the hardware setting. For example, Megatron requires users rewriting the model forward function to customize the tensor sharding for tensor parallelism and annotate breakpoints for pipeline parallelism. This demands substantial MLSys expertise, which is not possessed by most algorithm developers or researchers.

In Redco, we automate model parallism via tensor parallelism. This requires a dimension specified for every tensor to shard and distribute it across devices. The objective of sharding strategy design is to minimize cross-device computations in order to reduce memory and time costs associated with inter-device communication.

By observing the tensor computations in transformer architectures, specifically within feed-forward and attention layers, we get some heuristic insights in terms of inter-device communication. For feed-forward layers, given the nature of matrix multiplication, dividing two matrices along different dimensions is expected to require less inter-device communication than if they are divided along a same dimension. For the attention layers, the output matrix O is multiplied with each of the Q, K, V matrices, so the matrix O should be splitted along a dimension distinct from that chosen for Q, K, V.

Based on these insights, we write two rules to determine the dimension along which to split each tensor in a model:

1. For fully-connected layers, alternate between splitting the parameter along dimension 0 and dimension 1.
2. For attention layers, split Q, K, V along dimension 0, and split the output projection matrix O along dimension 1.

Leveraging the rules above enables Redco to devise a model parallel strategy tailored for any given LLM architecture. This enables the distributed training of LLMs with almost zero user effort. The only user effort is to specify the number of shards to split the given model, without additional coding or configuration efforts nor the expertise in MLSys. [4]

Note that our proposed rules are similar to a part of suggested configurations of Megatron-LM [12], but they don't summarize their separate configurations into rules, so that only a few LLM architectures (BERT, GPT, and T5) are supported in their implmentation[5]. To customize any new architectures, users have to rewrite the model's forward function and manually implement their model parallel strategy inside it.

3 Library Design

In addition to the complexities of implementing model parallelism, ML pipelines often contain repetitive boilerplate code that demands significant effort from developers. Examples of such code include back-propagation, gradient application, batch iteration, and so forth. Furthermore, the hardware upgrades usually require patches on existing codebase. For example, a code developed

[4]We include an efficiency test of the proposed model parallel strategy in the Appendix.

[5]https://github.com/NVIDIA/Megatron-LM/tree/main

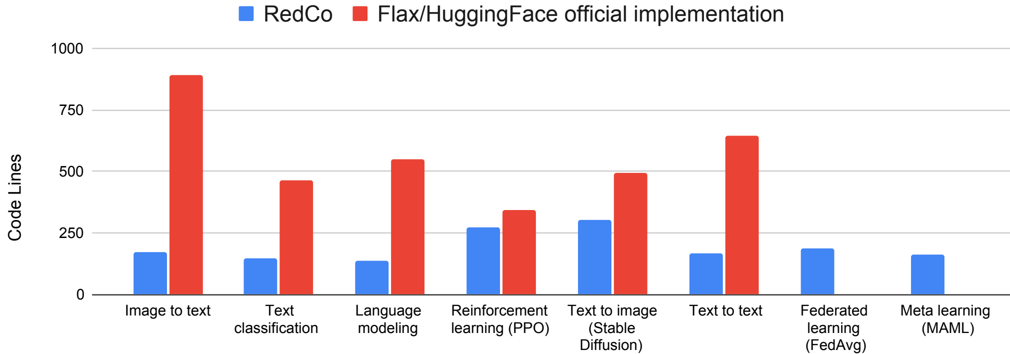

Figure 3: The comparison of code lines across a diverse set of ML algorithms. (There is no well-accepted official Flax implementations for FedAvg and MAML.)

within a single-GPU setting needs data parallelism and multi-host related processing to be added when adapted to multi-GPU machines or clusters.

In Redco, we design a neat and user-friendly mechanism to simplify ML pipeline developments. Users only have to define their pipeline through three design functions, and Redco handles all the remaining of the pipeline execution. In this section, we will introduce the software design of Redco for enabling this mechanism.

Pipeline design through three functions As shown in the yellow brick in Figure 2, in our proposed mechanism, every ML pipeline can be decoupled into three simple functions: *collate function* to convert raw examples to model inputs, e.g., converting text sentences to be a batch of token indices via tokenization; *loss function* to convert the model inputs to a scalar loss; and *predict function* to convert the model inputs to the desired model outputs, such as running beam search for the language model. We demonstrate this framework with the implementation of image captioning and a meta-learning algorithm, MAML, as shown in Figure 4 and Figure 5 in the Appendix. These examples showcase that both simple and complex algorithms can be naturally defined under the proposed mechanism.

Pipeline execution by redco For user-friendliness, there are merely three classes in Redco, i.e., `Deployer`, `Trainer`, and `Predictor`. As shown in the orange brick in Figure 2, Redco streamlines the management of low-level and boilerplate processing in pipeline development through the `Deployer` class. This includes automatic model parallelism, as discussed in the previous section, as well as automatic data parallelism, multi-host related processing, and other convenient features such as randomness control and logging management. The final execution of the pipeline is carried out by the `Trainer` and `Predictor` classes. Supported by `Deployer`, they execute the training and inference of the pipeline defined by users via the functions mentioned above. We include a complete example in the Appendix implementing a distributed sequence-to-sequence pipeline with Redco.

Evaluation We implement a variety of machine learning paradigms using Redco, ranging from fundamental supervised learning techniques such as classification and regression, to more sophisticated algorithms including reinforcement learning and meta-learning. Figure 3 illustrates a comparison between the number of code lines in our implementation and those in officially published versions. The majority of these paradigms can be efficiently implemented using Redco with only 100 to 200 lines of code. This efficiency boost of development can be attributed to Redco's ability to significantly reduce the need for writing boilerplate code.

4 Conclusion

In this work, we present a lightweight and user-friendly toolkit, *Redco*, designed to automate the distributed training of LLMs and simplify the ML pipeline development process. Redco incorporates an automatic model parallelism strategy, fundamentally based on two intuitive rules, without requiring additional efforts or MLSys expertise from the users. We evaluate its effectiveness on an array of LLMs, such as LLaMA-7B, T5-11B and OPT-66B. Furthermore, Redco provides a novel and neat pipeline development mechanism. This mechanism requires users to specify only three intuitive pipeline design functions to implement a distributed ML pipeline. Remarkably, this mechanism is general enough to accommodate various ML algorithms and requires significantly fewer lines of code compared to their official implementations.

Acknowledgements

The name of this package, *Redco*, is inspired by *Red Coast Base*, a key location in the story of Three-Body. From Red Coast Base, humanity broadcasts its first message into the vast universe. We thank Cixin Liu for such a masterpiece!

We thank Google Research for supporting Bowen Tan working as a student researcher in an internship. Eric Xing, Yonghao Zhuang, Hongyi Wang, and Bowen Tan has also been graciously supported by NGA HM04762010002, NSF IIS1955532, NSF CNS2008248, NIGMS R01GM140467, NSF IIS2123952, NSF BCS2040381, an Amazon Research Award, NSF IIS2311990, and DARPA ECOLE HR00112390063. Zhiting Hu is partically supported by DARPA ECOLE HR00112390063.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[3] F. Chollet et al. Keras. `https://keras.io`, 2015.

[4] W. Falcon et al. PyTorch Lightning, 3 2019.

[5] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee. Flax: A neural network library and ecosystem for JAX, 2023.

[6] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[7] OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[9] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[10] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[11] A. Roberts, H. W. Chung, A. Levskaya, G. Mishra, J. Bradbury, D. Andor, S. Narang, B. Lester, C. Gaffney, A. Mohiuddin, et al. Scaling up models and data with \texttt {t5x} and\texttt {seqio}. *arXiv preprint arXiv:2203.17189*, 2022.

[12] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[13] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023.

[14] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, Oct. 2020. Association for Computational Linguistics.

[15] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[16] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[17] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[18] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

A Related work

Distributed Machine Learning. Distributed machine learning refers to the utilization of multiple computing devices, typically GPUs or TPUs, for the efficient training and inference of ML models with large datasets or large models. It includes data parallelism and model parallelism. Data parallelism involves dividing a large dataset into multiple subsets, with each subset processed independently by a separate computing device. All devices maintain a copy of the model parameters, and during training, parameter gradients are aggregated across devices to update the model. However, data parallelism is limited in its ability to handle large models that exceed the memory constraints of individual devices. Model parallelism addresses this limitation by splitting and distributing the model across multiple devices, with each responsible for a portion of the model. Although it offers a solution for large models, model parallelism is more complex to implement than data parallelism due to the necessity of careful model partitioning. Tools such as Megatron-LM [12, 6], DeepSpeed [10] and FSDP [17] which are compatible with PyTorch[8], and Alpa [18] which works with Jax [2], have been developed to facilitate model parallelism. These tools support the automatic model partitioning strategies based on model and hardware specifications but still require significant coding and configuration efforts tailored to model architecture and hardware settings. Our tool, Redco, offers automatic data parallelism by default, and provides automatic model parallelism for LLMs, which is the majority of model parallelism use cases. Prioritizing user-friendliness, Redco enables users to execute distributed LLM training and inference by simply specifying the number of model shards for partitioning, without requiring any specialized knowledge of MLSys.

Pipeline development tools. In the development process using neural network libraries such as PyTorch and Flax [5], certain boilerplate code is consistently present. Common operations, such as back-propagation, gradient application, and batch iteration, recur in nearly every ML pipeline. A variety of tools aim to streamline pipeline development by eliminating repetitive code while maintaining as much development flexibility as possible. PyTorchLightning [4] offers a default training loop within PyTorch, allowing users to customize their pipelines by inheriting a Trainer class and modifying hook functions such as loss function and checkpoint saving. However, this mechanism may not be intuitive for all users. For some people, it requires a learning curve to become comfortable. Furthermore, it may be unclear how to implement these hook functions for more complex algorithms, such as federated learning. HuggingFace-Transformers [14] provides a Trainer for PyTorch models, but it heavily relies on models defined in its specific transformer classes and primarily focuses on natural language processing pipelines. Keras [3] delivers higher-level APIs on top of TensorFlow [1], enabling users to specify data, model, and loss functions. However, it is not well-suited for handling complex pipelines. T5X [11] is based on Flax, and offers pre-defined

pipelines to support the distributed training of LLMs, but its applicability is limited to platforms like Google Cloud or XManager, and users can only run pre-defined LLM pipelines by using gin files to set hyperparameters. Our proposed Redco is based on Flax, and uses a more intuitive and flexible approach for users to design their pipelines. This mechanism can be applied to a wide array of ML algorithms together with the automatic support for distributed training, including federated learning, meta-learning, reinforcement learning, and more.

B Examples of Redco's Pipeline Design

```
def collate_fn(raw_examples):
  return {
      "pixel_values": # pixel values of the images
      "token_ids": # token indicies of captions
  }

def loss_fn(batch, params):
  logits = model(params, batch['pixel_values'], batch['token_ids']) # run model and get logits
  return cross_entropy(logits, batch['token_ids'])

def pred_fn(batch, params):
  return model.beam_search(params, batch['pixel_values'])
```

Figure 4: Pipeline design functions of image captioning under Redco.

```
def collate_fn(raw_examples):
  return {
      "train_data": # a batch of few-shot training tasks
      "eval_data": # a batch of few-shot evaluation tasks
  }

def loss_fn(batch, params):
  params_inner = params - alpha * jax.grad(inner_loss)(params, batch['train_data'])
  return inner_loss(params_inner, batch['eval_data'])

def pred_fn(batch, params, model):
  params_inner = params - alpha * jax.grad(inner_loss)(params, batch['train_data'])
  return model(params_inner, batch['eval_data'])
```

Figure 5: Pipeline design functions of meta-learning (MAML) for few-shot learning under Redco. MAML's loss $L = \mathcal{L}(\mathcal{T}_{eval}, \theta')$ and $\theta' = \theta - \alpha\nabla_\theta\mathcal{L}(\mathcal{T}_{train}, \theta)$, where $\mathcal{T}_{train}$, $\mathcal{T}_{eval}$ refer to the data of training and evaluation tasks, and $\mathcal{L}(\cdot, \cdot)$ refers to the original loss function (`inner_loss`), such as the cross-entropy loss for classification.

C Evaluation of Redco's Model Parallel Strategy

Applicability test We assess the applicability of our proposed automatic model parallel approach by applying Redco on an assorted collection of LLMs across various GPU and TPU servers. We conduct distributed training for LLMs without compromising the optimizer settings or precision. More precisely, we execute the distributed training at full precision (fp32) using the widely adopted, yet memory-intensive, AdamW optimizer, and report all operable LLMs on these GPU and TPU servers with varying memory capacities. The findings, as displayed in Table 1, indicate that our straightforward, yet effective, automatic model parallel strategy is highly applcable across LLMs. For example, on smaller servers, such as those equipped with $2 \times$ 1080Ti, our strategy successfully runs

OPT-2.7B	Perplexity	Time (mins)
FSDP	10.29	27
Alpa	10.43	16
Redco (ours)	10.31	18

GPT-J-6B	Perplexity	Time (mins)
FSDP	9.11	48
Alpa	9.01	29
Redco (ours)	8.45	34

Table 2: Performance and running speed of different tools of two models on a a $4\times$ A100 server. All the settings use full precision (float 32) and AdamW optimizer.

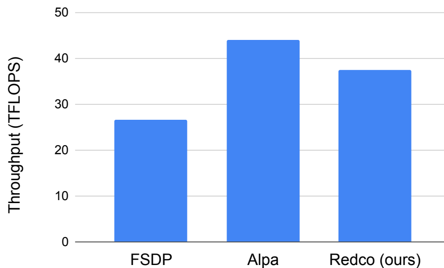

Figure 6: The comparison of throughput in the running of GPT-J-6B on a $4\times$ A100 server.

large versions of BART and GPT-2 with text lengths up to 512 and 1024, respectively. On the larger servers such as 16 $\times$ TPU-v4 hosts, Redco effectively handles the training of the giant OPT-66B.

Efficiency test We evaluate the efficiency of our proposed automatic model parallelism strategy in Redco on a server equipped with four A100 GPUs. We perform experiments by finetuning OPT-2.7B and GPT-J-6B, on the WikiText dataset, with full precision and AdamW optimizer. We compare Redco with two advanced model parallel tools: FSDP and Alpa. The experimental results are summarized in Table 2 and Figure 6. The perplexity values obtained with Redco are consistent with those achieved using FSDP and Alpa, thereby validating the correctness of our implementation for Redco's model parallelism. Furthermore, the observed running times reveal that Redco surpasses FSDP and is close to Alpa, the state-of-the-art model parallel tool. Notably, Alpa's implementation requires advanced MLSys expertise and significant coding efforts.

D A Complete Example

We provide a complete example for the distributed training of a T5-XXL model, with a Flax modeling from HuggingFace, on a summarization dataset, evaluated by rouge scores, and saving the checkpoints with best rouge-2 and rouge-L scores.

```
from functools import partial
import fire
import numpy as np
import jax
import jax.numpy as jnp
import optax
import datasets
from transformers import AutoTokenizer, FlaxAutoModelForSeq2SeqLM
import evaluate
from redco import Deployer, Trainer


def collate_fn(examples,
               tokenizer,
               decoder_start_token_id,
               max_src_len,
               max_tgt_len,
               src_key='src',
               tgt_key='tgt'):
    model_inputs = tokenizer(
        [example[src_key] for example in examples],
        max_length=max_src_len,
        padding='max_length',
        truncation=True,
        return_tensors='np')

    decoder_inputs = tokenizer(
        [example[tgt_key] for example in examples],
        max_length=max_tgt_len,
        padding='max_length',
        truncation=True,
        return_tensors='np')

    if tokenizer.bos_token_id is not None:
        labels = np.zeros_like(decoder_inputs['input_ids'])
        labels[:, :-1] = decoder_inputs['input_ids'][:, 1:]
        decoder_input_ids = decoder_inputs['input_ids']
        decoder_input_ids[:, 0] = decoder_start_token_id
    else:
        labels = decoder_inputs['input_ids']
        decoder_input_ids = np.zeros_like(decoder_inputs['input_ids'])
        decoder_input_ids[:, 1:] = decoder_inputs['input_ids'][:, :-1]
        decoder_input_ids[:, 0] = decoder_start_token_id

    model_inputs['labels'] = labels
    decoder_inputs['input_ids'] = decoder_input_ids

    for key in decoder_inputs:
        model_inputs[f'decoder_{key}'] = np.array(decoder_inputs[key])

    return model_inputs


def loss_fn(train_rng, state, params, batch, is_training):
    labels = batch.pop("labels")
    label_weights = batch['decoder_attention_mask']

    logits = state.apply_fn(
        **batch, params=params, dropout_rng=train_rng, train=is_training)[0]

    loss = optax.softmax_cross_entropy_with_integer_labels(
        logits=logits, labels=labels)
```

```
    return jnp.sum(loss * label_weights) / jnp.sum(label_weights)


def pred_fn(pred_rng, batch, params, model, gen_kwargs):
    output_ids = model.generate(
        input_ids=batch['input_ids'],
        attention_mask=batch['attention_mask'],
        params=params,
        prng_key=pred_rng,
        **gen_kwargs)
    return output_ids.sequences


def output_fn(batch_preds, tokenizer):
    return tokenizer.batch_decode(batch_preds, skip_special_tokens=True)


def eval_rouge(examples, preds, tgt_key):
    rouge_scorer = evaluate.load('rouge')

    return rouge_scorer.compute(
        predictions=preds,
        references=[example[tgt_key] for example in examples],
        rouge_types=['rouge1', 'rouge2', 'rougeL'],
        use_stemmer=True)


def main(dataset_name='xsum',
         src_key='document',
         tgt_key='summary',
         model_name_or_path='google/t5-xxl-lm-adapt',
         n_model_shards=16,
         n_epochs=2,
         per_device_batch_size=8,
         eval_per_device_batch_size=16,
         accumulate_grad_batches=2,
         max_src_len=512,
         max_tgt_len=64,
         num_beams=4,
         learning_rate=4e-5,
         warmup_rate=0.1,
         weight_decay=0.,
         jax_seed=42,
         workdir='./workdir',
         run_tensorboard=False):
    dataset = datasets.load_dataset(dataset_name)
    dataset = {key: list(dataset[key]) for key in dataset.keys()}

    with jax.default_device(jax.devices('cpu')[0]):
        model = FlaxAutoModelForSeq2SeqLM.from_pretrained(
            model_name_or_path, from_pt=True)
        model.params = model.to_fp32(model.params)

        tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)
        gen_kwargs = {'max_length': max_tgt_len, 'num_beams': num_beams}

    deployer = Deployer(
        jax_seed=jax_seed,
        n_model_shards=n_model_shards,
        workdir=workdir,
        run_tensorboard=run_tensorboard,
        verbose=True)

    optimizer, lr_schedule_fn = deployer.get_adamw_optimizer(
        train_size=len(dataset['train']),
        per_device_batch_size=per_device_batch_size,
        n_epochs=n_epochs,
        learning_rate=learning_rate,
        accumulate_grad_batches=accumulate_grad_batches,
        warmup_rate=warmup_rate,
        weight_decay=weight_decay)

    trainer = Trainer(
        deployer=deployer,
        collate_fn=partial(
            collate_fn,
            tokenizer=tokenizer,
            decoder_start_token_id=model.config.decoder_start_token_id,
            max_src_len=max_src_len,
            max_tgt_len=max_tgt_len,
            src_key=src_key,
            tgt_key=tgt_key),
        apply_fn=model,
        loss_fn=loss_fn,
        params=model.params,
        optimizer=optimizer,
        lr_schedule_fn=lr_schedule_fn,
        params_shard_rules=deployer.get_sharding_rules(params=model.params))

    predictor = trainer.get_default_predictor(
        pred_fn=partial(pred_fn, model=model, gen_kwargs=gen_kwargs),
        output_fn=partial(output_fn, tokenizer=tokenizer))
```

```
    trainer.fit(
        train_examples=dataset['train'],
        per_device_batch_size=per_device_batch_size,
        n_epochs=n_epochs,
        eval_examples=dataset['validation'],
        eval_per_device_batch_size=eval_per_device_batch_size,
        eval_loss=True,
        eval_predictor=predictor,
        eval_metric_fn=partial(eval_rouge, tgt_key=tgt_key),
        save_argmax_ckpt_by_metrics=['rouge2', 'rougeL'])


if __name__ == '__main__':
    fire.Fire(main)
```