RedCoast: 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 *RedCoast (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. Firstly, to automate model parallelism, 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 on a set of LLM architectures, such as GPT-J, LLaMA, T5, and OPT, up to the size of 66B. Secondly, we propose a mechanism that allows for the customization of diverse ML pipelines through the definition of merely three functions, avoiding 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. [1]

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 (Zhao et al., 2023a). Notable examples include GPT-4 (OpenAI, 2023) and LLaMA (Touvron et al., 2023). 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 (Rasley et al., 2020), Alpa (Zheng et al., 2022), and FSDP (Zhao et al., 2023b) provide diverse model parallelism solutions, but 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 users without MLSys expertise, such as algorithm developers or researchers. At times, the intricacy of coding for model parallelism proves to be even more daunting than the algorithm design itself.

In this work, we introduce *RedCoast (Redco)*[2], a lightweight and user-friendly tool designed to automate the distributed training and inference of LLMs, thereby users without MLSys expertise can also effortlessly use the tool without additional coding or intricate configurations. Furthermore, we

[1]`RedCoast (Redco)` has been released under Apache 2.0 license at `https://github.com/tanyuqian/redco`.

[2]For simplicity, we will use Redco more frequently in the rest of this paper.

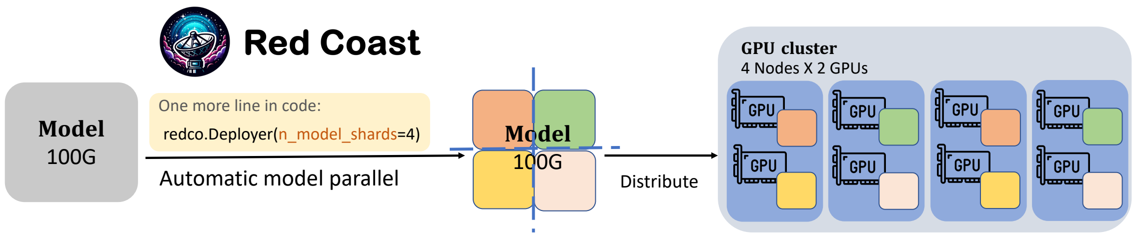


Figure 1: With a number of shards specified by user, Redco automatically conduct the model partitioning and distribution across hosts and devices.

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, checkpointing, 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 (Shoeybi et al., 2019) and DeepSpeed (Rasley et al., 2020) 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 LLMs including LLaMA-7B (Touvron et al., 2023), T5-11B (Raffel et al., 2020), and OPT-66B (Zhang et al., 2022). Moreover, pipelines driven by Redco demonstrate efficiency superior to those implemented with FSDP (Zhao et al., 2023b), and closely matching the performance of Alpa (Zheng et al., 2022), the tool with state-of-the-art model parallel efficiency.

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, checkpointing, 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 (seq2seq), 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 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 usually 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, and every device maintains a full copy of the model parameters. 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 (Shoeybi et al., 2019; Narayanan et al., 2021), DeepSpeed (Rasley et al., 2020), FSDP (Zhao et al., 2023b), and Alpa (Zheng et al., 2022), have been developed to facilitate model parallelism. These tools support the model partitioning but still require significant coding and configuration efforts based on specific model architecture and hardware settings. In this work, 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 users' MLSys expertise.

Pipeline development tools. In the development process using neural network libraries such as PyTorch (Paszke et al., 2019) and Flax (Heek et al., 2023), certain boilerplate code is consistently present. Common operations, such as backpropagation, 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 (Falcon et al., 2019) 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 (Wolf et al., 2020) 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 (Chollet et al., 2015) delivers higher-level APIs on top of TensorFlow (Abadi et al., 2015), enabling users to specify data, model, and loss functions. However, it is not well-suited for handling complex pipelines. 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 complex algorithms such as federated learning, meta-learning, and reinforcement learning.

3 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.

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 this work, we develop an automatic model parallel strategy in Redco that applies across LLMs without requiring users' MLSys expertise or extra coding efforts.

3.1 Rules to Automate Tensor Parallelism

In Redco, we automate model parallelism via tensor parallelism. A tensor sharding strategy requires a dimension specified for each tensor. Along the dimension, the tensor is sharded and distributed across multiple devices. The objective of sharding strategy design is to minimize memory and time overhead associated with inter-device communication, which is usually brought by `reduce` or `gather` operations. For example, consider a tensor t is defined either as $t = t_1 + t_2$ or $t = (t_1, t_2)$ (concatenation), with t_1 and t_2 being stored on distinct devices. In this case, the computation of t requires message passing between the two devices (GPUs or TPUs).

Consider a dense layer in a neural network

$$y = \sigma(xA)$$

where x denotes the input tensor, σ is an element-wise activation function (e.g., ReLU, SiLU), and A is to the weight matrix, which is the model parameter of the dense layer. [3] When the weight matrix A is divided along its first dimension (dimension 0), a `reduce` operation becomes necessary to compute y. Formally,

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \Longrightarrow y = \sigma(x_1 A_1 + x_2 A_2)$$

where x_1 and x_2 represent the first and second halves of the input tensor x's dimensions, respectively. The computation on two devices are indicated by the colors.

Conversely, when A is partitioned along its second dimension (dimension 1), a `gather` operation is required to obtain y:

$$A = (A_1, A_2) \Longrightarrow y = (\sigma(xA_1), \sigma(xA_2))$$

Therefore, when the weight parameter of each dense layer is partitioned across an arbitrary dimension, operations for reduction or gathering would

[3]The bias term is omitted here because its computation is non-significant.

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
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 servers. Numbers inside the brackets are the maxinum length in training. All the settings are with full precision (fp32) with AdamW optimizer.

occur within every dense layer. However, by examining a pair of consecutive dense layers

$$y = \sigma(\sigma(xA)B)$$

where A and B denote the respective weights, and partitioning A and B across dimensions 1 and 0, it becomes feasible to consolidate these operations with a single time of reduce operation:

$$A = (A_1, A_2), \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

$$\begin{aligned} \Longrightarrow y &= \sigma(\sigma(xA)B) \\ &= \sigma\left((\sigma(xA_1), \sigma(xA_2)) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}\right) \\ &= \sigma(\sigma(xA_1)B_1 + \sigma(xA_2)B_2) \end{aligned}$$

Consequently, the operations $\sigma(xA_1)B_1$ and $\sigma(xA_2)B_2$ can be performed independently on separate devices, which takes only a single reduce operation for two dense layers.

Based on the observation above, we get heuristic insights in terms of inter-device communication on the transformer architecture, specifically within feed-forward and attention layers. For feed-forward layers, given the nature of matrix multiplication, dividing two consecutive 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 1 and dimension 0.
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. Users only specify the number of shards to split the given model, without additional coding or configuration efforts.

Note that our proposed rules are similar to a part of suggested configurations of Megatron (Shoeybi et al., 2019), 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[4]. To customize any new architectures under Megatron, users still have to rewrite the model's forward function and manually implement their model parallel strategy.

3.2 Implementation inside Redco

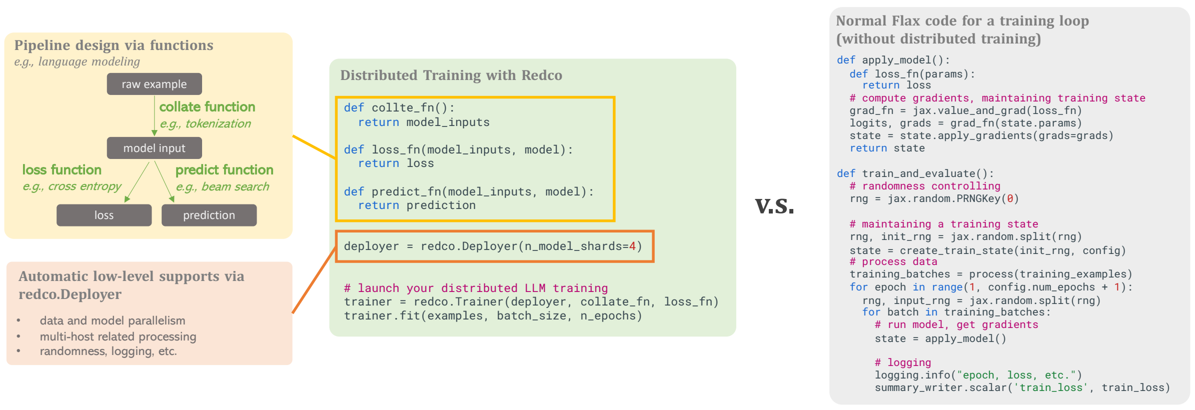
We implement tensor parallelism with the proposed strategy on top of `jax.pjit` function. This function compiles the computational graph and it merges operations on the same device to reduce unnecessary communication overhead [5].

To integrate the proposed tensor parallel strategy, Redco has a function that takes in an arbitrary transformer architecture and produces a parameter sharding strategy based on the proposed rules. Moreover, in addition to automatically generating sharding strategies, Redco also enables their customization. This allows users with more advanced strategies to execute their approaches. Practical examples of the sharding strategies produced by Redco, applied to GPT-J and LLaMA, are showcased in the Appendix.

3.3 Evaluation

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 under full precision (fp32) with the widely-used, yet memory-intensive, AdamW optimizer. We report all operable LLMs on GPU and TPU servers with varying memory capacities.

[4]https://github.com/NVIDIA/Megatron-LM/tree/main
[5]`https://jax.readthedocs.io/en/latest/notebooks/Distributed_arrays_and_automatic_parallelization.html`


Pipeline design via functions
e.g., language modeling
raw example
collate function
e.g., tokenization
model input
loss function
e.g., cross entropy
predict function
e.g., beam search
loss
prediction
Automatic low-level supports via redco.Deployer
data and model parallelism
multi-host related processing
randomness, logging, etc.
Distributed Training with Redco
v.s.
Normal Flax code for a training loop (without distributed training)

```
def collte_fn():
  return model_inputs

def loss_fn(model_inputs, model):
  return loss

def predict_fn(model_inputs, model):
  return prediction

deployer = redco.Deployer(n_model_shards=4)

# launch your distributed LLM training
trainer = redco.Trainer(deployer, collate_fn, loss_fn)
trainer.fit(examples, batch_size, n_epochs)
```

```
def apply_model():
  def loss_fn(params):
    return loss
  # compute gradients, maintaining training state
  grad_fn = jax.value_and_grad(loss_fn)
  logits, grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)
  return state

def train_and_evaluate():
  # randomness controlling
  rng = jax.random.PRNGKey(0)

  # maintaining a training state
  rng, init_rng = jax.random.split(rng)
  state = create_train_state(init_rng, config)
  # process data
  training_batches = process(training_examples)
  for epoch in range(1, config.num_epochs + 1):
    rng, input_rng = jax.random.split(rng)
    for batch in training_batches:
      # run model, get gradients
      state = apply_model()

      # logging
      logging.info("epoch, loss, etc.")
      summary_writer.scalar('train_loss', train_loss)
```

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.

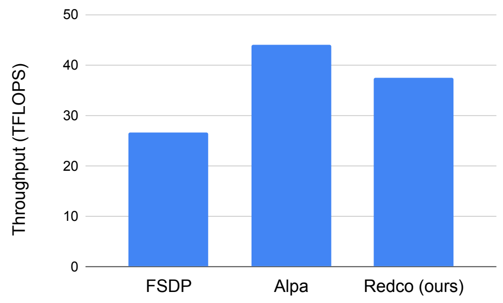


Figure 3: The comparison of throughput in the running of GPT-J-6B on a 4× A100 server. Redco's performance surpasses that of FSDP and is close to that of Alpa, the tool with state-of-the-art model parallel efficiency but intricate engineering.

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 small servers, such as those equipped 2 × 1080Ti, our strategy successfully runs large versions of BART and GPT-2 with text lengths up to 512 and 1024, respectively. On the larger servers such as the one with 16 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 4 × 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 Figure 3. The observed throughput reveals 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.

4 RedCoast: 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 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 that enables this mechanism.

4.1 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. These examples showcase that both simple and complex algorithms can be naturally defined under the proposed mechanism.

```python
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.

```python
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.

4.2 Pipeline Execution with Automatic Low-level Supports

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, checkpointing, and other convenient features such as randomness control and logging management. The final execution of the pipeline is carried out by `Trainer` and `Predictor` of Redco. Supported by `Deployer`, they execute the training and inference of the pipeline defined by users via the functions as mentioned in Section 4.1. [6]

4.2.1 Multi-host Supports

Large-scale distributed training typically involves intricate processes to accommodate multiple hosts. These processes include allocating data samples to each node and aggregating gradients or parameters across hosts, etc. Redco offers automatic support for multi-host environments and demonstrates compatibility with various platforms, including SLURM (Yoo et al., 2003), XManager[7], as well as bare nodes interconnected via IP addresses. Notably, Redco allows users to maintain their existing pipeline design and execution code without additional efforts for multi-host environments.

4.2.2 Checkpointing

In distributed training frameworks, each typically employs a distinct formatting for checkpoint saving and loading, leading to a closed-loop system. For instance, Megatron utilizes a unique approach where model parameters and optimizer states are segmented based on the configuration of model parallelism. These checkpoints are inherently tied to Megatron, necessitating considerable effort for conversion into standard PyTorch checkpoint formats. Conversely, in Redco, we adopt a standardized and well-accepted checkpointing method. Here, both model parameters and optimizer states are encapsulated comprehensively within dictionaries of tensors. This approach is independent of the specific distributed training configurations, offering the advantage of simplicity in loading and utilization, even without Redco installation.

4.2.3 Lightweight and Flexible Dependency

Distributed training frameworks, such as Megatron and Alpa, often include modifications to foundational Python packages or CUDA kernels, resulting in stringent environment installation requirements. For instance, Alpa modifies the fundamental `jaxlib`[8] library, thereby limiting its compatibility to jaxlib version 0.3.22 and CUDA version 11.3. They have been outdated compared to advanced versions of jaxlib 0.4.32 and CUDA 12.2,

[6]We include a complete example in the Appendix implementing a distributed seq2seq pipeline with Redco.

[7]`https://github.com/google-deepmind/xmanager`

[8]`https://pypi.org/project/jaxlib/`

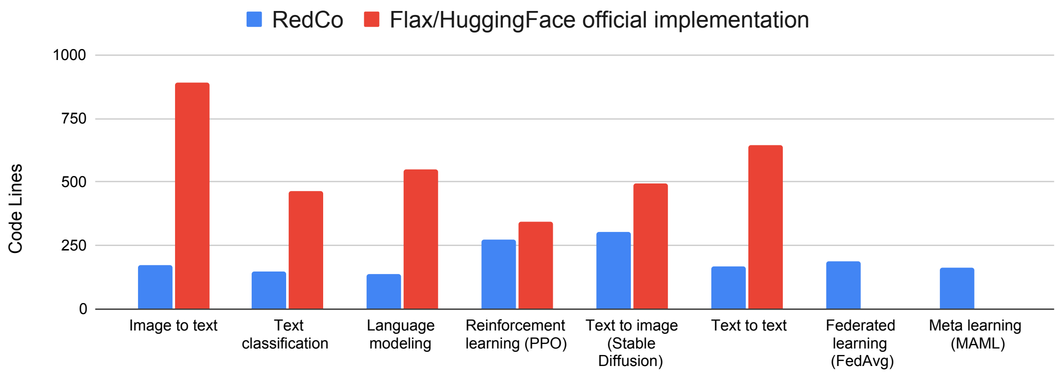


Figure 6: The comparison of code lines across a diverse set of ML algorithms. (There is no well-accepted official Flax implementations for FedAvg and MAML.)

which are prevalent in many cluster environments today. In contrast, Redco is developed on top of Jax and Flax, without any modification to existing packages. Consequently, Redco is able to support a wider range of versions of jax, flax, and CUDA, in addition to accommodating various device types, including GPUs and TPUs.

4.3 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 6 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.

5 Conclusion

In this work, we present a lightweight and user-friendly toolkit, *RedCoast (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 coding 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 has 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 needs significantly fewer lines of code compared to their official implementations.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

François Chollet et al. 2015. Keras. `https://keras.io`.

William Falcon et al. 2019. PyTorch Lightning.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. 2023. Flax: A neural network library and ecosystem for JAX.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. 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.

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

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. 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.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aur'elien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. 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. Association for Computational Linguistics.

Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023a. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023b. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. 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.

A Tensor Parallel Strategy Examples

We provide examples of the generated sharding strategies by Redco toward different architectures, where `PartitionSpec('mp', None)` indicates partioning a parameter by dimension 0 and `PartitionSpec(None, 'mp')` indicates partioning a parameter by dimension 1, and `None` means saving a copy of the parameter across every device.

```
params_sharding_rules_gptj = [
    (('fc_in', 'kernel'), PartitionSpec(None, 'mp')),  # Rule 1 in Section 3.1
    (('fc_out', 'kernel'), PartitionSpec('mp', None)),
    (('k_proj', 'kernel'), PartitionSpec(None, 'mp')), # Rule 2 in Section 3.1
    (('out_proj', 'kernel'), PartitionSpec('mp', None)),
    (('q_proj', 'kernel'), PartitionSpec(None, 'mp')),
    (('v_proj', 'kernel'), PartitionSpec(None, 'mp')),
    (('(bias|scale)',), None),                         # Parameters other than transformer blocks or bias or scale terms
    (('embedding',), PartitionSpec('mp', None)),
    (('lm_head', 'kernel'), PartitionSpec(None, 'mp'))
]
```

Figure 7: Sharding strategy for GPT-J generated by Redco.

```
params_sharding_rules_llama = [
    (('down_proj', 'kernel'), PartitionSpec('mp', None)), # Rule 1 in Section 3.1
    (('gate_proj', 'kernel'), PartitionSpec(None, 'mp')),
    (('up_proj', 'kernel'), PartitionSpec(None, 'mp')),
    (('k_proj', 'kernel'), PartitionSpec(None, 'mp')),    # Rule 2 in Section 3.1
    (('o_proj', 'kernel'), PartitionSpec('mp', None)),
    (('q_proj', 'kernel'), PartitionSpec(None, 'mp')),
    (('v_proj', 'kernel'), PartitionSpec(None, 'mp')),
    (('(bias|scale)',), None),                            # Parameters other than transformer blocks or bias or scale terms
    (('embedding',), PartitionSpec('mp', None)),
    (('lm_head', 'kernel'), PartitionSpec(None, 'mp')),
    (('norm', 'weight'), None),
    (('input_layernorm', 'weight'), None),
    (('post_attention_layernorm', 'weight'), None)
]
```

Figure 8: Sharding strategy for LLaMA generated by Redco.

B A Complete Distributed Training Example with Redco

We provide a complete example for the distributed training of a T5-XXL model, which is able to run on multi-host environments. It uses a modeling from HuggingFace, on a summarization dataset, evaluated by rouge scores, and it saves 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]
```

```python
        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, params, batch, 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(n_processes=None,
         host0_address=None,
         host0_port=None,
         process_id=None,
         n_local_devices=None,
         dataset_name='xsum',
         src_key='document',
         tgt_key='summary',
         model_name_or_path='facebook/bart-base',
         n_model_shards=1,
         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):
    deployer = Deployer(
        n_model_shards=n_model_shards,
        jax_seed=jax_seed,
        workdir=workdir,
        run_tensorboard=run_tensorboard,
        n_processes=n_processes,
        host0_address=host0_address,
        host0_port=host0_port,
        process_id=process_id,
        n_local_devices=n_local_devices)

    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}

    lr_schedule_fn = deployer.get_lr_schedule_fn(
        train_size=len(dataset['train']),
        per_device_batch_size=per_device_batch_size,
        n_epochs=n_epochs,
        learning_rate=learning_rate,
        schedule_type='linear',
        warmup_rate=warmup_rate)
    optimizer = optax.adamw(
        learning_rate=lr_schedule_fn, weight_decay=weight_decay)
    if accumulate_grad_batches > 1:
        optimizer = optax.MultiSteps(
            optimizer, every_k_schedule=accumulate_grad_batches)

    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,
        accumulate_grad_batches=accumulate_grad_batches,
        params_sharding_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_last_ckpt=True,
        save_argmax_ckpt_by_metrics=['rougeL'])


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