A MODELING RECONFIGURABLE DL TRAINING

In this section, we model and predict training throughput using different combinations of strategies and resource allocations. We will first model each of the training parts in Fig. 5, and then show how to combine them into T_{iter}.

We denote each fittable parameter in our model as k plus a certain subscript to distinguish it from other model- or environment-related constants (summarized in Table 5).

A.1 Modeling Computation and Communication

Modeling T_{fwd}. The time for forward pass T_{fwd} under 3D parallelism can generally be obtained from profilers provided by DL frameworks (*e.g.*, in DeepSpeed) on a node with a given global batch size. We scale T_{fwd} linearly to the actual per-GPU batch size for data parallelism, and to per-GPU tensor shard size for tensor parallelism. Besides, we have special treatments for the following two strategies.

Pipeline Parallelism. PP balances the layers among GPUs. When profiling with g_p GPUs, the forward time provided by the framework (denoted as t_p) is usually the time for a single GPU to process a micro-batch, with l/g_p layers placed on it, where l is the total number of layers. The complete T_{fwd} for PP includes the time taken for the first micro-batch to be processed sequentially on each GPU, and that for all GPUs to serially process the other micro-batches (Narayanan et al., 2019). Besides, T_{fwd} is linear to the per-GPU number of layers. We then have $T_{fwd} = t_p \cdot g_p/p \cdot (m+p-1)$, where m is the number of micro-batches and p is the PP size.

Gradient Accumulation. GA aggregates per-GPU gradients over multiple passes. Therefore, the total forward time is $T_{fwd} \cdot a$, where a is the number of accumulation steps.

Modeling T_{bwd}. T_{bwd} is the time for computing the gradients during the backward pass. Transformer-based models are primarily comprised of matrix multiplication operations, where the time required for gradient computation can be generally considered to be proportional to T_{fwd}, *i.e.*, $T_{bwd} = k_{bwd} \cdot T_{fwd}$. A special case arises with gradient checkpointing (GC): GC recomputes activations during the backward pass. The time cost for the extra computation is typically equal to the time T_{fwd} (Chen et al., 2016). Therefore, when GC is used, modeling the T_{bwd} requires adding the time required for a forward pass.

Modeling T_{comm}. The communication time T_{comm} involves those for data, tensor, and pipeline parallelisms. For each part, T_{comm} is in general estimated as $T_{comm} = V/B$, where V is the volume of the data to transfer between each pair of GPUs and B is the corresponding bandwidth.

Table 5. Summary of performance model parameters.

Fittable		$k_{bwd}, k_{sync}, k_{opt}, k_{opt\_off}, k_{off}, k_{swap}, k_{const}$
Job	Model	s (seq), h (hidden), l (layers), P (param size)
	Resources	g (GPU), c (CPU)
	Parallelism	d, t, p (3D-parallel sizes, $d \cdot t \cdot p = g$)
	Others	b (batch size), m (micro-batch num), a (GA steps)
Environment		$B_{intra}, B_{inter}, B_{pcie}$

We discuss how to model B first. For each type of communication (DP/TP/PP), we basically use the bottleneck bandwidth of the GPUs involved in the communication, *i.e.*, the lowest bandwidth among all pairs of GPUs. For example, when all GPUs are co-located on the same node, the data can be transferred via a high-speed connection like NVLink. In this case, we use the intra-node bandwidth B_{intra} as B. However, when the GPUs are spread on multiple nodes, the communication is largely dominated by the bandwidth between nodes because the speed is much slower than NVLink. Hence we use inter-node bandwidth B_{inter} here. Note that different types of communication may use different B values. For example, TP is typically restricted inside each node while PP can be distributed across nodes (Narayanan et al., 2021). In this case, TP and PP will use B_{intra} and B_{inter}, respectively. The values of B_{intra} and B_{inter} are measured on the cluster offline.

Next, we model the communication volume V for different strategies, respectively. When the parallelism size of any parallel strategy is 1, then the corresponding V is 0.

Data Parallelism. DP typically uses the ring AllReduce algorithm to synchronize gradients, where each model replica sends and receives $2(d-1)/d$ times gradients (d being the DP size). The gradients generated during the entire backward pass are approximately as large as the parameter size. Considering that the gradients are partitioned and synchronized in parallel across TP and PP partitions, we have $V_{dp} = P \cdot 2(d-1)/(d \cdot t \cdot p)$, where P is the total parameter size, and t and p are TP and PP sizes, respectively. This rule also applies to the ZeRO series as they are based on DP.

Tensor Parallelism. The communication volume for TP depends on the size of output tensor of a Transformer layer, which is $b \cdot s \cdot h$ (Vaswani et al., 2017) when not sliced, where b, h, and s represent the batch size, hidden size, and sequence length, respectively. Each layer involves in total 4 communication operations in the forward and backward passes (Shoeybi et al., 2019). Considering the output tensor and the batch are partitioned by TP and DP, respectively, we have $V_{tp} = 4 \cdot 2 \cdot (t-1) \cdot b \cdot s \cdot h \cdot l/(d \cdot t)$ (this volume is not divided by the PP size p because the TP communications across pipeline stages are serialized).

Pipeline Parallelism. Micro-batches need to wait for the

communication from other pipeline stages after finishing the forward/backward pass for the current micro-batch. The communication volume for each micro-batch between each consecutive pair of devices is $b/m \cdot s \cdot h$. PP communication is involved in both forward and backward passes, and the tensors are partitioned by DP and TP along the batch size and operator dimensions. We thus have $V_{pp} = 2 \cdot p \cdot b \cdot s \cdot h/(d \cdot t)$[3].

Combining computation and communication. As depicted in Fig. 5, it is possible to overlap the communication with the forward/backward pass computation. We use an intermediate variable T_{cc} to denote the combination of computation and communication, which is calculated as follows.

3D parallelism. In 3D parallelism, the gradient synchronization of DP can be overlapped with the backward pass, whereas the communication for TP/PP cannot as it is on the critical path. We use a function $f^k_{overlap}(T_x, T_y)$ parameterized by k to model the overlapping of two stages, where the fittable parameter k represents the degree of the overlapping. Here we use k_{sync} for the overlapping of DP and backward pass, thus we have $T_{cc} = T_{fwd} + f^{k_{sync}}_{overlap}(T_{bwd}, T_{comm\_dp}) + T_{comm\_tp} + T_{comm\_pp}$, where the three communication times are calculated using the rule described above. To avoid distraction, we defer the detail of $f^k_{overlap}$ to Sec. A.3.

Gradient Accumulation. When GA is used in DP, per-GPU gradients are aggregated locally over $a-1$ forward-backward passes before being synchronized across all GPUs during the a^{th} pass. Therefore, the total backward propagation spans $a-1$ accumulation steps followed by the last step overlapped with the synchronization, that is, $T_{cc} = a \cdot T_{fwd} + (a-1) \cdot T_{bwd} + f^{k_{sync}}_{overlap}(T_{bwd}, T_{comm\_dp})$.

A.2 Modeling Optimizer and Offloading

Modeling T_{opt}. The optimizer time T_{opt} depends on the parameter size on each GPU, instead of the total parameter size, as the parameters are updated in parallel. We discuss each strategy as below.

3D parallelism or ZeRO-DP. 3D parallelism and ZeRO-DP partition model parameters by the TP/PP size and DP size, respectively, thus we have $T_{opt} = k_{opt} \cdot P/x$, where x represents $t \cdot p$ for 3D parallelism, and d for ZeRO-DP.

ZeRO-Offload. Beyond the partitioning, ZeRO-Offload updates the partition each GPU owns directly on the CPU. Thus, we add a new fittable parameter to represent the CPU computation efficiency. Since CPU resources are used in parallel to jointly compute a single weight update, increasing the number of CPUs c can also optimize T_{opt} under ZeRO-Offload, that is, $T_{opt} = k_{opt\_off} \cdot P/(d \cdot c)$.

Modeling T_{off}. T_{off} represents the time specifically required by ZeRO-Offload, which is taken by the communication between CPU and GPU. ZeRO-Offload offloads the partitioned gradients to the CPU memory after computation and moves the parameter partitions back to the GPU after the parameter update. The communication volume for each data parallel GPU to the CPU is P/d without mixed precision, thus we have $T_{off\_raw} = P/(d \cdot B_{pcie})$.

In ZeRO-Offload, the offloading is also overlapped with the gradient synchronization and the optimizer step. We use an intermediate variable T_{oo} to denote the combination of these parts. When using ZeRO-Offload, we have $T_{oo} = f^{k_{off}}_{overlap}(T_{comm\_dp}, T_{off}) + f^{k_{swap}}_{overlap}(T_{opt}, T_{off})$; otherwise, we simply have $T_{oo} = T_{opt}$.

A.3 Putting It All Together

Combining the discussion in previous sections, we model the end-to-end iteration time as:

$$T_{iter} = T_{cc} + T_{oo} + k_{const} \quad (1)$$

where we use another fittable parameter k_{const} to denote other constant overhead.

Modeling overlapping. We use the function $f^k_{overlap}(T_x, T_y)$ to represent the total time spent by x and y, considering the overlap between them. Taking the overlapping of T_{bwd} and T_{comm} as an example, if there is no overlap in data parallelism, they are combined as $T_{bwd} + T_{comm}$. If there is a perfect overlap, it should be $max(T_{bwd}, T_{comm})$. A realistic value is somewhere in between these two extremes. To capture the overlapping, we borrow the definition from prior work (Qiao et al., 2021) as $f^k_{overlap}(T_x, T_y) = \left(T_x^k + T_y^k\right)^{\frac{1}{k}}$. This formula has the property that the total time equals $T_x + T_y$ when $k = 1$, and it smoothly transitions towards $max(T_x, T_y)$ as $k \to \infty$.

Continuous model fitting. As mentioned in Sec. 4, To fit the 7-tuple fittable parameters (listed in Table 5), we require at least seven data points before scheduling corresponding jobs. Considering that three parameters involve ZeRO-Offload (k_{opt_off}, k_{off}, k_{swap}), the test runs should include three using this strategy. We minimize the root mean squared logarithmic error (RMSLE) between Eq. (1) and the collected data triples. The model can also be updated online to correct potential prediction errors.

[3]We model the commonly used 1F1B strategy for PP (Narayanan et al., 2019). This formula only considers the micro-batches whose results are needed immediately by the next pipeline stage. For some of the micro-batches in the warm-up phase of 1F1B, the communication can be overlapped, but the degree is hard to model. We assume that they are perfectly overlapped.

B ARTIFACT APPENDIX

B.1 Abstract

The artifact includes the source code and scripts to run the experiments. It can validate the core functionalities of *Rubick* and reproduce the main evaluation results of this paper.

B.2 Notes

Our experiments can be reproduced in two ways: artifact evaluation (referred to as *"artifact"* hereafter) and real GPU cluster experiments (referred to as *"real experiments"* hereafter). For the *artifact*, we have pre-collected performance values for all Transformer models in Table 1 under various resource amounts and execution plans in a 64-GPU cluster setup. Based on the data, we can quickly validate *Rubick*'s core functionalities without requiring GPUs.

For the *real experiments*, they require access to a 64-A800 GPU cluster, which incurs significant costs and also demands lengthy runtimes to complete. As a result, we do not recommend this approach. However, for those interested, we have provided instructions below and on GitHub.

It is worth highlighting that both methods use the same code to implement the core function of *Rubick*. We believe the *artifact* is sufficient to validate *Rubick*'s capabilities.

B.3 Artifact check-list

- **Algorithm:** A new scheduling algorithm is used for reconfigurable scheduling.
- **Program:** Seven deep learning training workloads, such as ResNet, GPT-2, are used as benchmarks.
- **Model:** For *artifacts*, we only need the model configuration information (*e.g.*, model structure), which has been included in the code. For *real experiments*, you will need to download their checkpoints from `https://huggingface.co/models`.
- **Data set:** For *real experiments* only. They need to be downloaded from `https://huggingface.co/datasets`.
- **Run-time environment:** The *artifact* is designed to run in a Docker container, making it OS-agnostic. Root access is not required, but Docker must be installed and configured. The *real experiments* need more support such as Kubernetes and training frameworks.
- **Hardware:** *Artifact*: CPUs. *Real experiments*: 64-A800 GPU cluster.
- **Execution:** For *real experiments* only. They need profiling for every new model. The overhead can be found in Sec. 7.3.
- **Metrics:** For each training job: iteration time and throughput. For cluster experiment: average job completion time and makespan.
- **Output:** Standard console output (stdout)/log files/figures/tables.
- **Experiments:** README, scripts, IPython/Jupyter notebook are used. See Github for more details.
- **How much disk space required (approximately)?:** *Artifact*: 1 GB. *Real experiments*: 800 GB.
- **How much time is needed to prepare workflow (approximately)?:** *Artifact*: 30 minutes. *Real experiments*: 1 day.
- **How much time is needed to complete experiments (approximately)?:** *Artifact*: 2 hours. *Real experiments*: 9 days.
- **Publicly available?:** `https://github.com/AlibabaPAI/reconfigurable-dl-scheduler`.
- **Code licenses (if publicly available)?:** Apache-2.0
- **Data licenses (if publicly available)?:** Apache-2.0
- **Archived (provide DOI):** 10.5281/zenodo.14991392

B.4 Description

B.4.1 How delivered

The artifact repository can be obtained from Github. To get the *Rubick* artifact, run:

```
git clone https://github.com/AlibabaPAI/
    reconfigurable-dl-scheduler.git
cd reconfigurable-dl-scheduler
git checkout mlsys25-artifact
```

B.4.2 Hardware dependencies

Artifact: CPUs.

Real experiments: A cluster comprised of 8 servers, each with 8 NVIDIA A800 GPUs (80 GB), 96 vCPUs, 1,600 GB memory, 400 GB/s NVLink bandwidth, and 100 GB/s RDMA network bandwidth.

B.4.3 Software dependencies

Artifact: Docker container. You can pull the Docker images we prepared or setup the containers by yourself using Dockerfile we provided.

Real experiments: Kubernetes, Kubeflow, PyTorch 1.12, DeepSpeed 0.9.2, and Megatron-DeepSpeed v2.4.

B.5 Installation

Here, we provide a brief introduction to the *artifact* installation. For more details, see `https://github.com/AlibabaPAI/reconfigurable-dl-scheduler`.

You can pull the container as:

```
docker pull zzxy180318/rubick-artifact:
    mlsys25ae
```

You can also setup the containers by yourself using Dockerfile, which takes only 1-2 minutes to build:

```
docker build -t rubick:mlsys25ae .
```

Finally, launch the Docker images as follows:

```
docker run -tid --name rubick-artifact
    rubick:mlsys25ae
docker exec -it rubick-artifact /bin/bash
```

B.6 Experiment workflow

Here, we provide a brief overview of the *artifact* workflow. For detailed steps on how it is implemented and executed, please visit the Github.

To validate the performance model of *Rubick*, we use seven models listed in Table 1 to evaluate the prediction errors (Table 2). To assess *Rubick*'s reconfigurabilities, we use *Rubick* to train the LLaMA-2-7B under different resource limits and evaluate the training performance (Fig. 7).

To demonstrate *Rubick*'s ability to maximize the throughput across jobs, we submit RoBERTa and T5 models to a 4-GPU cluster and compare the overall performance with a simple scheduler (Fig. 8). To ensure that *Rubick* preserves training accuracy, we profile the training loss across $3,000$ mini-batches under different execution plans and compare it with the loss with randomized seeds (Fig. 9).

Finally, to highlight *Rubick*'s ability to optimize cluster scheduling by maximizing cluster throughput while maintaining job performance, we use three different traces (each consisting of 406 jobs) and schedule them with *Rubick* onto a 64-GPU cluster (Table 4).

B.7 Evaluation and expected result

Here, we only discuss the expected results of the *artifact*. For the performance model validation and micro-benchmarks, the number of resources and jobs involved is relatively small. Therefore, the *artifact* results will closely match those reported in the paper.

However, for cluster experiments, each row in Table 4 requires a long runtime (*i.e.*, Makespan), during which unavoidable factors such as network fluctuations and restart delays may affect the experiment results. It is impossible to accurately predict or quantify their impact in the *artifact*, although we have taken these factors into consideration. As a result, cluster experiments may exhibit some variation from the results in the paper. As shown in Sec. 7.4, we consider a mean variation of up to 6.9% to be acceptable.