A EXPERIMENT DETAILS

Our base setting is adopted from Ouyang et al. (2022), which utilizes a global batch size of 512, context length 2048, and a maximum prompt length of 1024. The global batch is divided into 8 mini-batches for PPO training.

We emphasize that the prompt and generation length may vary for different models, datasets or tasks, algorithm implementation, and even during RLHF training. To eliminate this effect and perform a fair comparison, we synthesize random data with the maximum prompt length and terminate generation only after the maximum length is reached.

We create LLaMA models of four different sizes with their detailed configurations shown in Table 1. For weak scaling experiments, we increase the model size and batch size proportionally to the number of devices. In particular, for 16, 32, 64, and 128 GPUs, the model sizes are 7B, 13B, 34B, and 70B, and batch sizes are 512, 1024, 2048, 4096, respectively. For experiments with a longer context length, we fix the number of tokens in the global batch. For instance, when the context length increases from 2048 to 8192, the global batch size decreases by a factor of 4. In experiments for strong scaling and dditional RLHF algorithms, we adopt the base setting with 70B actor/reference models and 7B critic/reward models on 16 nodes.

We show the execution plans of wall time breakdown examples (Table 6) in Tables 2 to 5.

B THE API OF REAL

Figure 18 shows an example of the API for an REAL experiment. Users define the dataflow graph of the algorithm (e.g., RLHF) using a list of `ModelFunctionCallDef` objects. These objects encapsulate the model configuration and the function call type, along with specifying input and output data dependencies. Models sharing the same `model_name` must have identical architectures (e.g., `llama7b`). They form parameter version dependencies, such that the inference and generation must wait for the training in the previous iteration. The experiment configuration is then wrapped by the `auto` decorator, which initiates the search engine to derive an efficient execution plan. This plan is transformed into a scheduling configuration for launching workers, each assigned to a specific GPU or CPU via SLURM (Yoo et al., 2003). The search engine and launcher both run under the hood. Users are free to provide distinct interface implementations to implement a diverse range of training workflows.

C SIMULATION ALGORITHM

The simulation algorithm in show in Algorithm 1.

```python
# auto is a decorator that generates worker
# scheduling configs in the cluster.
@auto(nodelist="com[01-08]", batch_size=256)
@dataclasses.dataclass
class Experiment:
    seed: int = 1
    ppo: PPOHyperparameters

    @property
    def rpcs(self) -> List[ModelFunctionCallDef]:
        return [
            ModelFunctionCallDef(
                model_name="actor",
                model_type="llama7b",
                interface_type=GENERATE,
                input_data=["prompts"],
                output_data=["seq", "logp"],
            ),
            ModelFunctionCallDef(
                model_name="reward",
                model_type="llama7b-critic",
                interface_type=INFERENCE,
                input_data=["seq"],
                output_data=["r"],
            ),
            ModelFunctionCallDef(
                model_name="actor",
                interface_type=TRAIN_STEP,
                input_data=["seq", "r", ...],
            ),
            # ref inference, critic inference,
            # and critic training
            ...,
        ]
```

Figure 18: An example of the user interface of REAL. Given the dataflow graph (represented by a list of `ModelFunctionCallDef` objects), the training batch size, and cluster specifications, REAL will automatically derive an execution plan via the `auto` decorator.

D BASELINES

In Figure 7, we show the performance comparison between REAL and 4 baseline RLHF systems: DeepSpeed-Chat (Yao et al., 2023b), OpenRLHF (Hu et al., 2024), NeMoAligner (Shen et al., 2024) and veRL (HybridFlow (Sheng et al., 2024)). The first three baselines are previous works of REAL, and veRL is concurrent to REAL. In this section, we will briefly introduce the implementation of these baseline systems. We also list the version and backend of baseline systems used in our experiments.

DeepSpeedChat is developed using modules from a popular training backend DeepSpeed (Rasley et al., 2020). It supports sequential execution of model function calls, and uses TP for the generation task, ZeRO-3 DP for the training and inference task. It also implements HybridEngine, a technique that reshards parameters between actor training and generation.

OpenRLHF exploits vLLM (Kwon et al., 2023) as their generation backend and DeepSpeed ZeRO-3 DP as their training backend. It divides GPUs into three groups, holding the actor/reference model, the critic/reward model and

Identifier	7B	13B	34B	70B
HiddenSize	4096	5120	8192	8192
IntermediateSize	14336	13824	22016	28672
NumLayers	32	40	48	80
NumAttentionHeads	32	40	64	64
NumKVHeads	8	40	8	8
VocabSize	128256	128256	128256	128256
MaxPositionEmbeddings	8192	8192	8192	8192
TotalParamCount	8030261248	14001525760	35321028608	70553706496
ParamCount w./o. Output Embedding	7504924672	13344855040	34270355456	69503033344

Table 1: The LLaMA-3 model configurations used in experiments. Because critic models have a smaller output embedding layer than the actor (i.e., the output dimension is 1 for the critic), we use the embedding-less parameter count as the identifier.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-16]	2	4	16	4	185.1
RewInf	trainer[01-16]	1	8	16	4	5.6
RefInf	trainer[01-16]	1	8	16	16	35.6
CriticInf	trainer[01-16]	1	8	16	16	5.6
CriticTrain	trainer[01-16]	8	4	4	2	20.8
ActorTrain	trainer[01-16]	2	16	4	2	108.0

Table 2: Device allocations and parallelization strategies for the 70B Actor and 7B critic searched case in Table 6.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-16]	8	4	4	8	241.8
RewInf	trainer[01-16]	8	4	4	8	12.6
RefInf	trainer[01-16]	8	4	4	8	63.5
CriticInf	trainer[01-16]	8	4	4	8	12.5
CriticTrain	trainer[01-16]	8	4	4	8	35.7
ActorTrain	trainer[01-16]	8	4	4	8	163.4

Table 3: Device allocations and parallelization strategies for the 70B Actor and 7B critic heuristic case in Table 6.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-02]	2	2	4	1	16.3
RewInf	trainer01	2	1	4	16	6.0
RefInf	trainer02	1	2	4	16	8.0
CriticInf	trainer[01-02]	1	2	8	8	4.7
CriticTrain	trainer02	4	2	1	2	28.1
ActorTrain	trainer01	2	4	1	2	26.6

Table 4: Device allocations and parallelization strategies for the 7B Actor and 7B critic searched case in Table 6.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-02]	8	1	2	4	44.2
RewInf	trainer[01-02]	8	1	2	4	7.3
RefInf	trainer[01-02]	8	1	2	4	7.6
CriticInf	trainer[01-02]	8	1	2	4	6.8
CriticTrain	trainer[01-02]	8	1	2	4	24.3
ActorTrain	trainer[01-02]	8	1	2	4	24.7

Table 5: Device allocations and parallelization strategies for the 7B Actor and 7B critic heuristic case in Table 6.

Time (s)	7B + 7B		70B + 7B	
	REAL	Heuristic	REAL	Heuristic
ActorGen (with CUDAGraph)	16.3	44.2	185.1	241.8
ActorGen (w.o. CUDAGraph)	34.5	104.6	185.1	241.8
RewInf	6.0	7.3	5.6	12.6
RefInf	8.0	7.6	35.6	63.5
CriticInf	4.7	6.8	5.6	12.5
CriticTrain	28.1	24.3	20.8	35.7
ActorTrain	26.6	24.7	108.0	163.4
End2End (with CUDAGraph)	64.0	122.6	383.1	546.8
End2End (w.o CUDAGraph)	82.2	183.0	547.4	912.3

Table 6: The RLHF wall time breakdown of two most common and representative cases. REAL reduces the end-to-end time by accelerating individual model function calls as well as concurrently executing independent computations.

the vLLM generation engine separately. It allows the concurrent execution of actor and critic training. However, the generation and training phase can not be executed concurrently due to data and parameter dependencies. This results in a significant GPU idle time.

Similarly, NeMoAligner divides GPUs into 2 disjoint GPU groups. Unlike OpenRLHF, it locates actor training and generation on the same GPU group. It splits the computations into micro batches and pipeline them to reduce the GPU idle time. It exploits TRT-LLM (Nvidia, 2024) (supports TP and resharding) as generation backend and Megatron-LM (Shoeybi et al., 2019) as training backend (supports 3D parallelization).

veRL supports colocating models on GPUs and split placement of models on different GPU groups, including the strategies adopted by three previous systems. It provides different choices for the generation (SGLang (Zheng et al., 2024) and vLLM (Kwon et al., 2023)) and training backend (Megatron-LM (Shoeybi et al., 2019) and Pytorch FSDP (Zhao et al., 2023b)) to support different parallelization strategies.

We list the version and backend of baseline systems used in our experiments in Table 7. We remark that in this experiment, REAL uses its own generation backend, model and pipeline parallelization, and adopts tensor parallelization and optimizer implementation from Megatron-LM. In a more recent version of REAL, we also support vLLM and SGLang as generation backend, which is not included in the experiments in this paper.

Algorithm 1 Calculate TimeCost($\mathcal{G}_p$)

Require: The augmented dataflow graph $\mathcal{G}_p = (\mathcal{V}_p, \mathcal{E}_p)$, device meshes $D \in \mathcal{D}$ where $\mathcal{D}$ contains all valid device meshes in the cluster.
ready_queue = PriorityQueue()*// Sorted by v.ReadyTime*
completed_set = Set() *// Contains completed nodes*
for $v \in \mathcal{V}_p$ **do**
  if v.parents=$\emptyset$ **then**
    ready_queue.push(v)
  end if
end for
while !ready_queue.empty() **do**
  Node v = ready_queue.pop()
  DeviceMesh D = v.device_mesh
  // D.last record the last completed node from all devices within D
  v.StartTime = max{v.ReadyTime, D.last.EndTime}
  v.EndTime = v.StartTime + TimeCost(v)
  completed_set.add(v)
  for $D' \in \mathcal{D}$ **do**
    if overlap(D, D') **and** D'.last.EndTime $\leq D$.last.EndTime **then**
      D'.last = v
    end if
  end for
  for $u \in v$.children **do**
    u.ReadyTime = max{u.ReadyTime, v.EndTime}
    if $w \in$ completed_set **for all** $w \in u$.parents **then**
      ready_queue.push(u)
    end if
  end for
end while
return max{v.EndTime $|v \in \mathcal{V}_p$}

System	Version	Generation Backend	Training Backend
DeepSpeedChat	commit f73a6ed	DeepSpeed v0.15.1	DeepSpeed v0.15.1
OpenRLHF	v0.4.2	vLLM v0.4.2	DeepSpeed v0.15.0
NeMoAligner	v0.4.0	TRT-LLM v0.10.0	Megatron-LM v0.8.0
veRL	0.2.0.post2	vLLM v0.6.3	Pytorch FSDP v2.4.0

Table 7: The version, generation backend and training backend used in our baseline experiments.