Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections

Marcel Wagenländer
Imperial College London

Guo Li
Imperial College London

Bo Zhao
Aalto University

Luo Mai
University of Edinburgh

Peter Pietzuch
Imperial College London

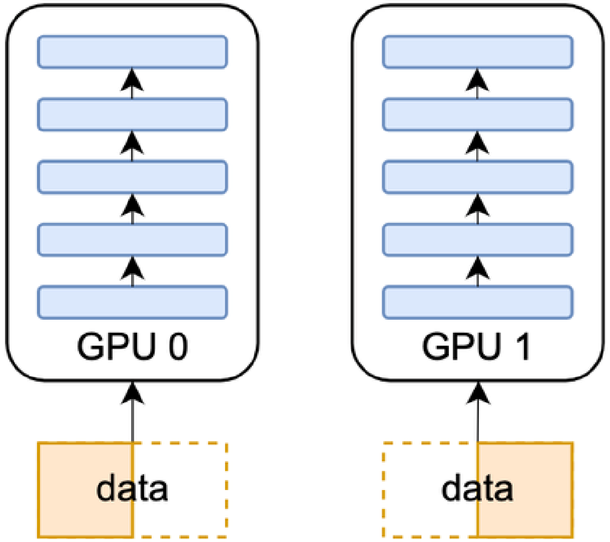
汇报人：段兆麟 张宇杭 王艺涵

天津大学智能与计算学部

College of Intelligence and Computing

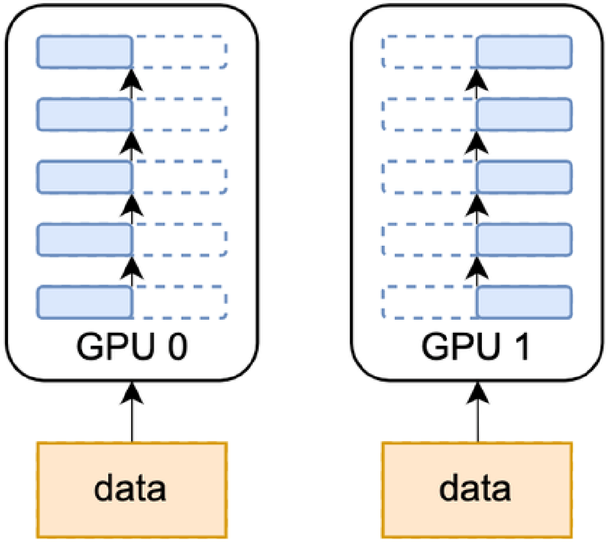
Tianjin Key Laboratory of Advanced Networking

Multi-dimensional Prallelism



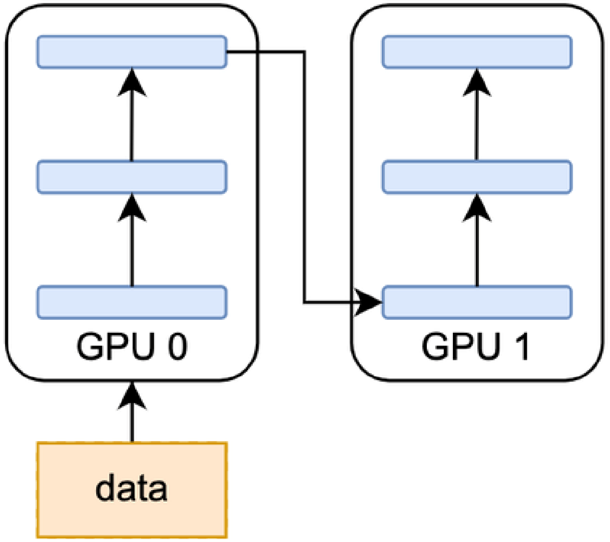
Data Parallel (DP)

Partition data across workers and replicate model



Tensor Parallel (TP)

Partition operators in the model



Pipeline Parallel (PP)

Partition model into stages

- ✗ Synchronization Overhead
- ✗ Communication Overhead
- ✗ Pipeline Bubbles

Dynamic Resource Changes

Training workloads may running days or weeks, the scheduler may change GPU allocation at runtime.

□ Elasticity:

- Dynamically scale the number of GPUs allocated to a job based on available resources or by leveraging spot instances.

□ Redeployment:

- Schedulers can reassign jobs to a new set of GPUs for operational efficiency or resource management.

□ Failure recovery:

- Long-running jobs may lose GPU resources due to failures from hardware faults, network outages, or software errors.

Dynamic Resource Changes

Current DL system do not allow DL job scheduler change GPU resource at runtime (?)

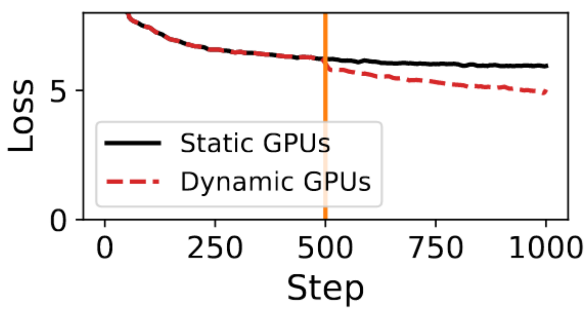
- **Lack of device-independence:** DL jobs are tightly coupled to GPUs at deployment time, preventing schedulers from changing the allocation.
- **Changing with multi-dimensional parallelism:** when GPU resources change, current parallelization strategy may no longer be optimal

changing DL job resources dynamically, with the support of multi-dimensional parallelism

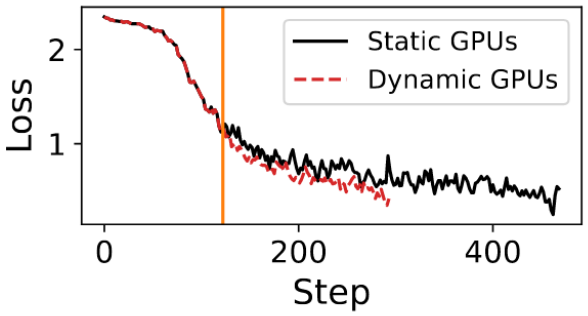
Challenge

Convergence

Training dataset
Some data may be used twice



(a) Inconsistent dataset access

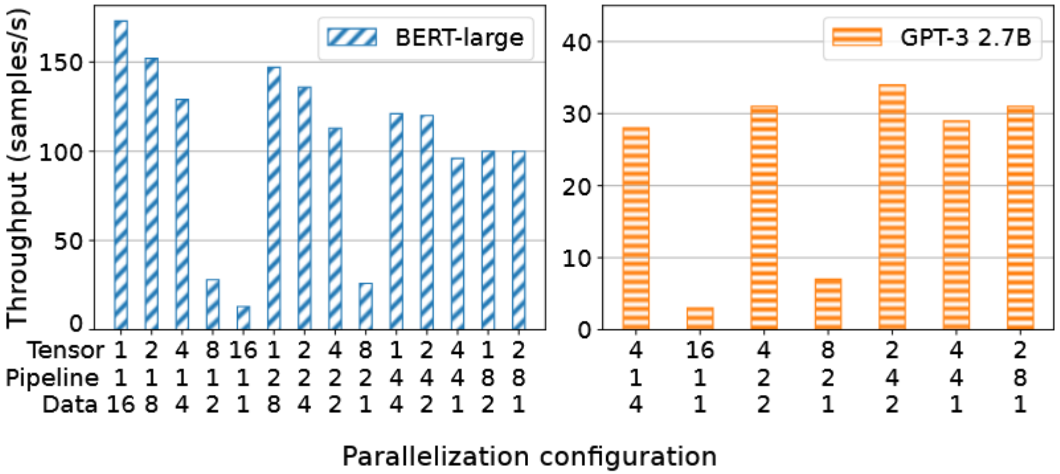


(b) Inconsistent batch size

Hyper parameters
Global batch size change

Performance

Parallelization configuration



Reconfiguration cost

Existing Work

Consistency | Multi-dimension | Dynamics | Overhead

Approach	Systems	Consistency		Parallelism						Reconfiguration overhead
		Dataset	Hyper-params	Static DP	Static PP	Static TP	Dynamic DP	Dynamic PP	Dynamic TP	
Ⓐ Model libraries	Alpa [86]	-	-	✓	✓	✓	-	-	-	-
	Megatron-LM [68]	-	-	✓	✓	✓	✓	✗	✗	full state
	Deepspeed [63]	✓	✓	✓	✓	✗	✓	✗	✗	full state
Ⓑ Elastic DL systems	Elastic Horovod [28]	✗	✗	✓	-	-	✓	-	-	full state
	Torch Distributed [57]	✓	✗	✓	✓	(✓)	✓	(✓)	(✓)	full state
	Varuna [4]	✓	✓	✓	✓	-	✓	✓	-	full state
	KungFu [43]	✓	✓	✓	-	-	✓	-	-	full state
Ⓒ Virtual devices	VirtualFlow [52]	✓	✓	✓	-	-	✓	-	-	full state
	EasyScale [40]	✓	✓	✓	-	-	✓	-	-	full state
	Singularity [69]	✓	✓	✓	✓	✓	✓	✗	✗	GPU state
State management	Tenplex	✓	✓	✓	✓	✓	✓	✓	✓	minimal state

✓ indicates support for the feature; (✓) indicates support after a job-specific implementation by the user;
✗ indicates support but without dynamic scaling; and - indicates no support.

Design Overview

Design Goal

- ensuring the consistency of the training result
- supporting arbitrary reconfiguration of jobs with multi-dimensional
- maintaining a low reconfiguration overhead parallelism

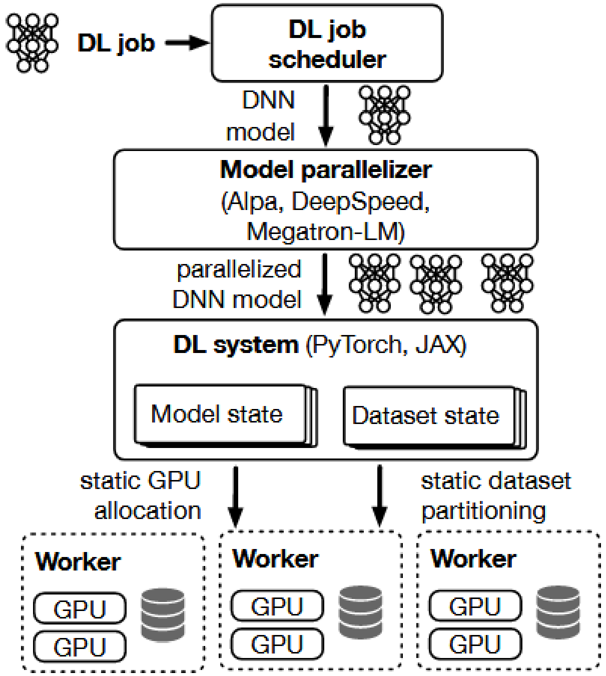
This paper propose a state management library

Externalizes and abstract state from DL job

Transform state when GPU changes

State for a DL job including model parameters and dataset

Design Overview



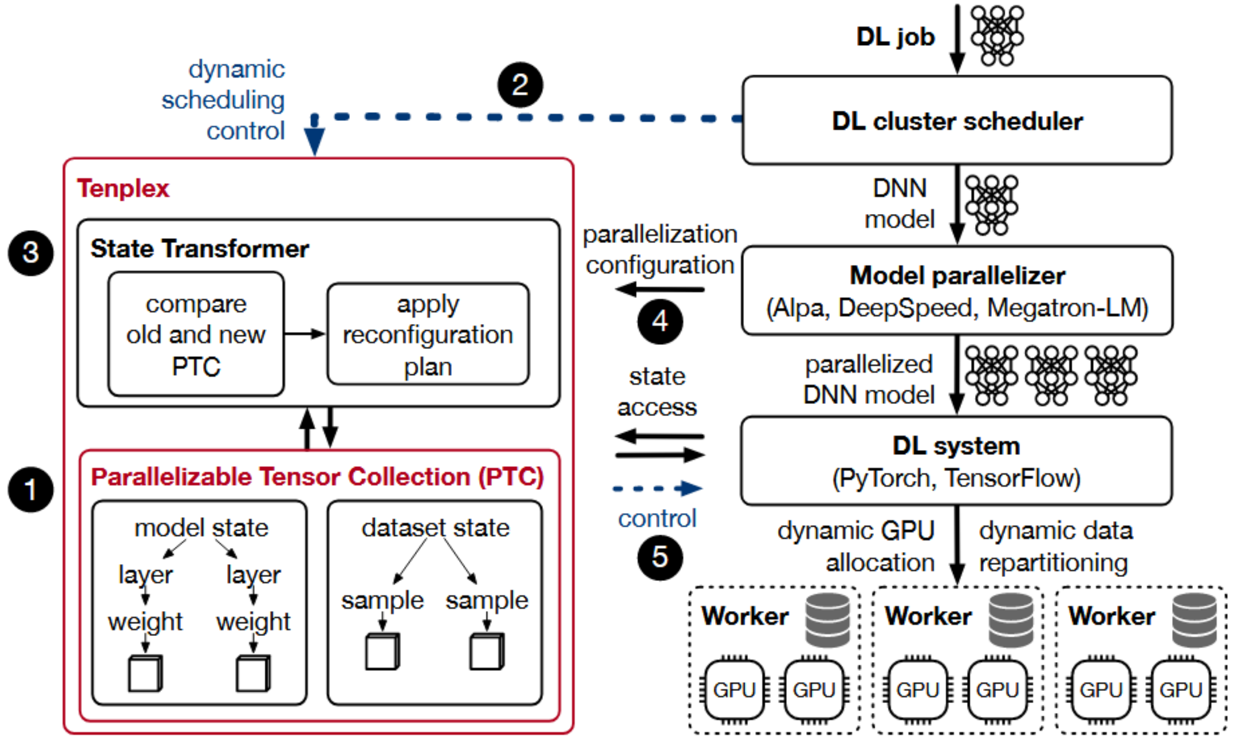


Fig. 5. Tenplex design

Tenplex manage state (model,dataset) as parallelizable tensor collection (PTC)

PTC Overview

- **Observation:** Any multi-dimensional parallelization strategy can be expressed as as a slicing of state tensors, followed by as partitioning of these tensors across GPU devices.

Define with three functions

- **Slicing (σ)**: Split tensors into sub-tensors, directed by TP.
- **Partitioning (Φ):** Group sub-tensors into collections that can be assigned to device, directed by PP and DP.
- **Allocation (α):** Map sub-tensor collections to GPU devices.

These three simple functions are sufficient to express any multi-dimensional parallelization strategies.

$$\mathrm{PTC} = (T, \sigma, \phi, \alpha)$$

T is the tensor collections (including dataset tensor and model tensor)

PTC Overview

- Slicing (σ), Partitioning (Φ), Allocation (α).

Deploy a job with DP=2, TP=2

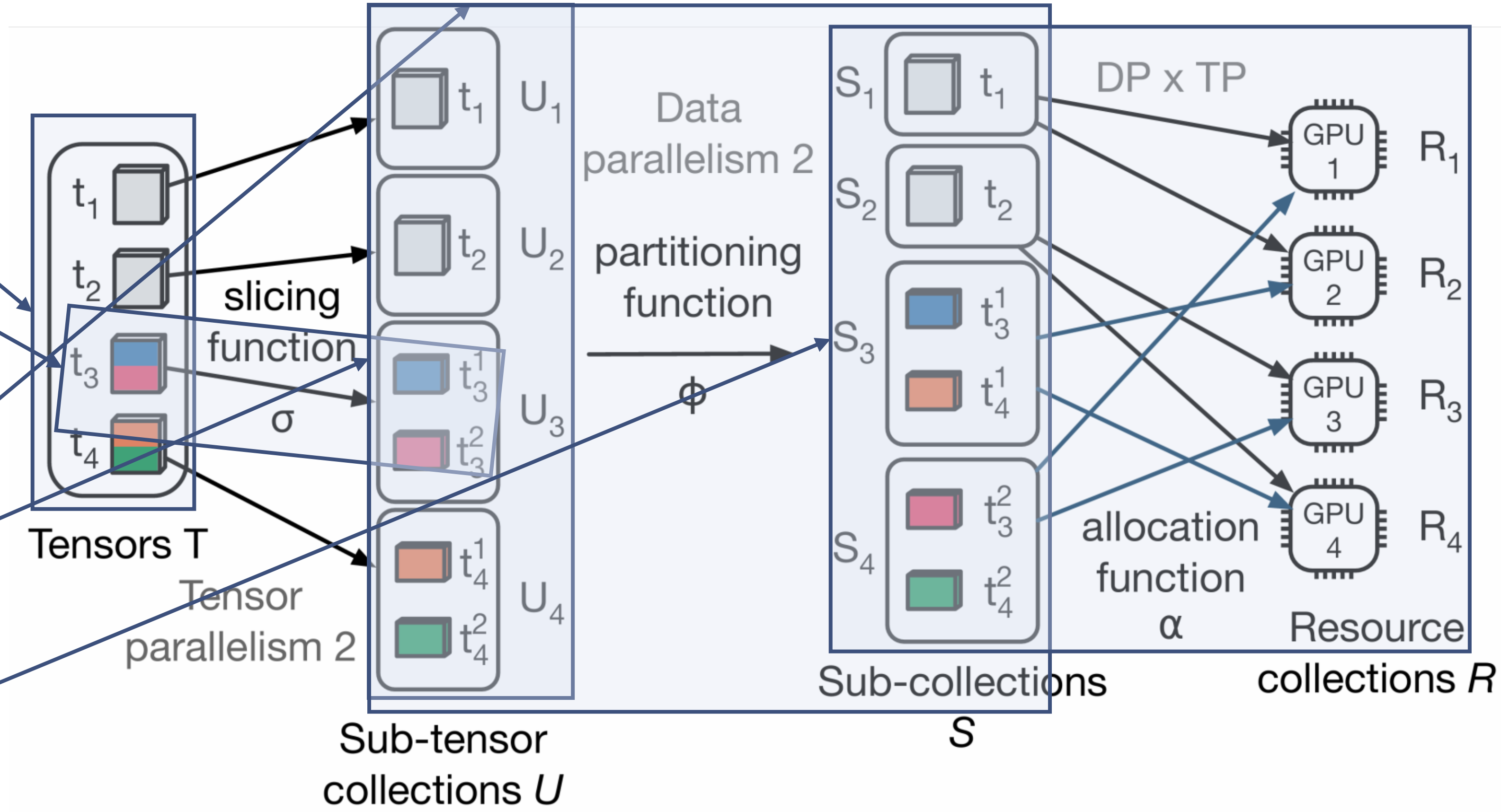
$$T = \{t_1, \ldots, t_n\}$$

$$\sigma(t) = \{t^1, .., t^m\}$$

$$U = \{\sigma(t_1) \ldots \sigma(t_n)\}$$

$$\phi(U) = \{S_1, .., S_p\}$$

$$\alpha(S_i) = \{r_1, .., r_q\}$$



Reconfiguration plan

Decide how to reconfigure by computing a delta between current PTC and new PTC'

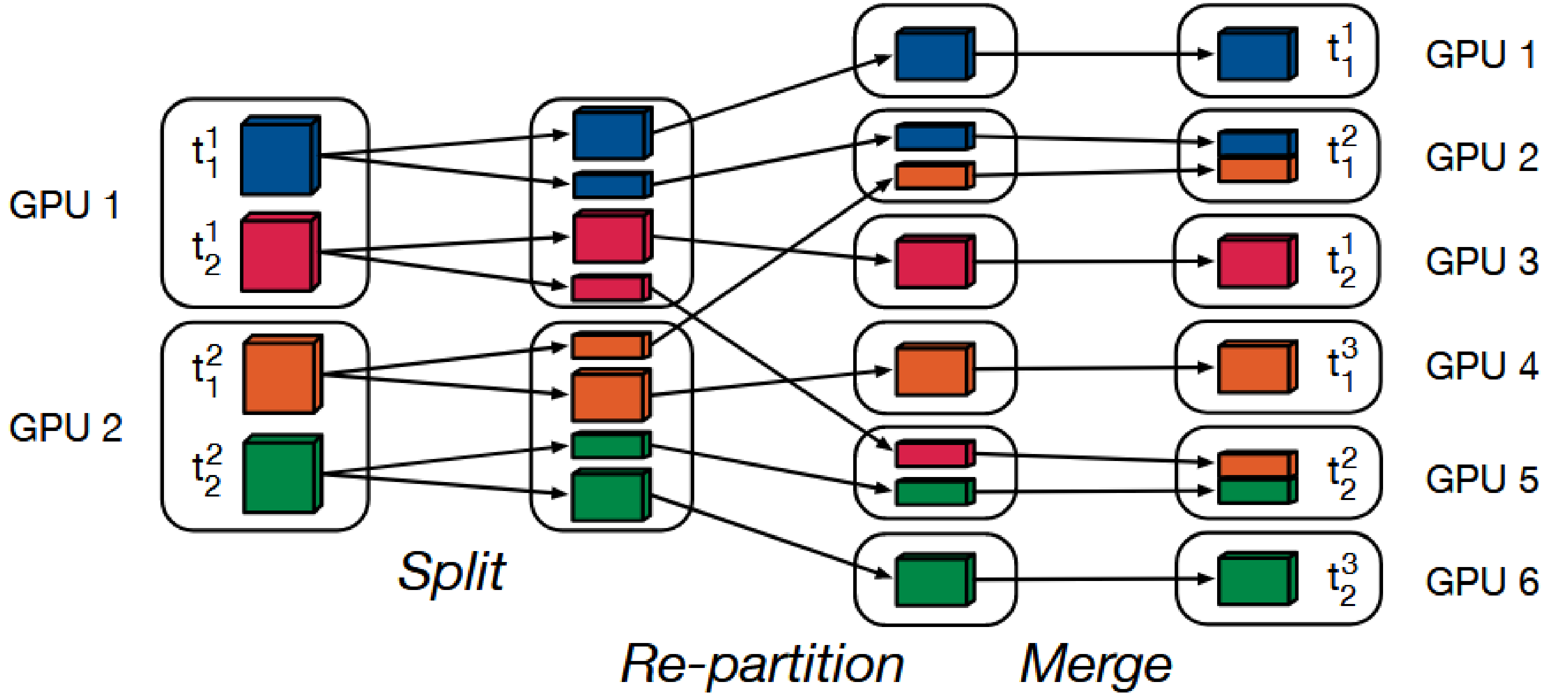
- **Reconfiguration plan:** A sequence of operations can turn state of PTC into PTC'

We can compute a reconfiguration plan which exchange minimal set of sub-tensors between GPUs

- **Split:** Slice current sub-tensors according to new slicing function σ'
- **Re-partition:** Move the split tensors from previous GPU to new GPU
- **Merge:** Combine sub-tensors were previously split but now on the same GPU.

Reconfiguration plan

- **Split:** Slice current sub-tensors according to new slicing function σ'
- **Re-partition:** Move the split tensors from previous GPU to new GPU
- **Merge:** Combine sub-tensors were previously split but now on the same GPU.



From TP=2 to TP=3,PP=2

Reconfiguration plan

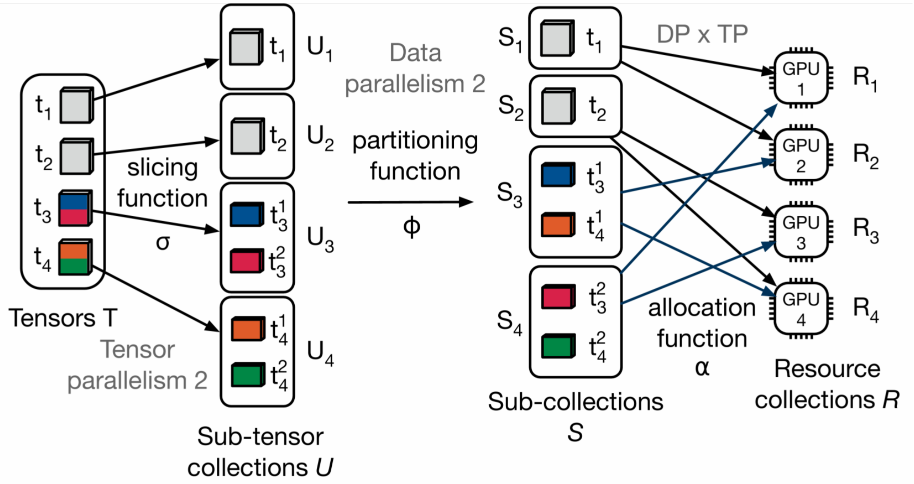
Algorithm 1: Reconfiguration plan generation

Data: $\text{PTC} = (T, \sigma, \phi, \alpha)$, $\text{PTC}' = (T, \sigma', \phi', \alpha')$
Resources R, R'
Result: Reconfiguration plan $\mathcal{P}$

$U \leftarrow \{\sigma(t) \mid t \in T\}$ *// get sub-tensor collections*
foreach $r \in R$ **do** *// start SPLIT*
  $V \leftarrow \{v \mid v \in U, \alpha(\phi(U)) = r\}$ *// get sub-tensors of r*
  foreach $v \in V$ **do**
    $\mathcal{P} \leftarrow \mathcal{P} \parallel \text{split}(v, \sigma, \sigma')$
$S' \leftarrow \phi'(\{\sigma'(t) \mid t \in T\})$ *// get sub-collections*
foreach $r' \in R'$ **do** *// start RE-PARTITION*
  $S'_r \leftarrow \{S'_i \mid S'_i \in S', \alpha(S'_i) = r\}$ *// get sub-tensors of r'*
  foreach $s' \in S'_r$ **do**
    $t \leftarrow \text{get\_base\_tensor}(\sigma', \phi', s')$
    $W \leftarrow \text{get\_split\_tensors}(t, \sigma, \sigma')$
    foreach $w \in W$ **do**
      $r_w \leftarrow \text{get\_resource}(\phi, \alpha, w)$
      $\mathcal{P} \leftarrow \mathcal{P} \parallel \text{move}(w, r_w, r')$ // add MOVE
    $\mathcal{P} \leftarrow \mathcal{P} \parallel \text{merge}(W)$ // add MERGE

1. Traverse all sub-tensors
2. generate the split function based on σ and σ'.

σ and σ' record how the original tensor was sliced, making it straightforward to create the corresponding split function.



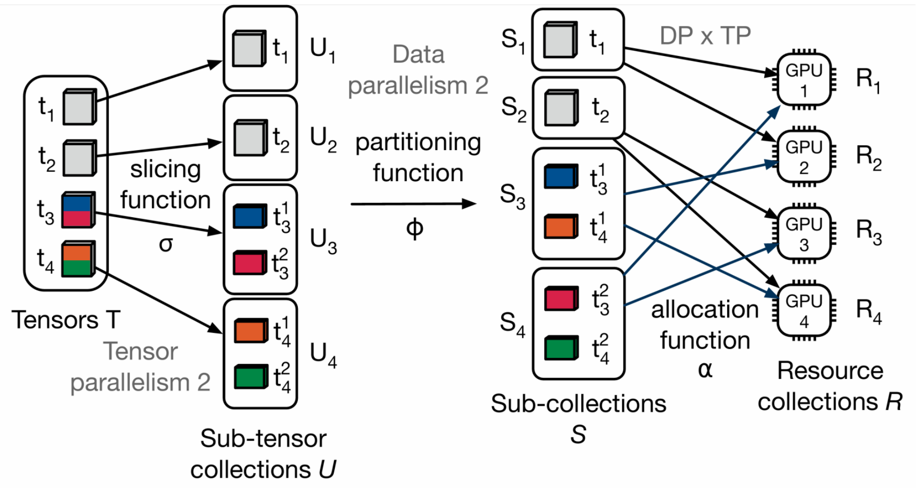
Reconfiguration plan

Algorithm 1: Reconfiguration plan generation

Data: $\mathrm{PTC} = (T, \sigma, \phi, \alpha)$, $\mathrm{PTC}' = (T, \sigma', \phi', \alpha')$
Resources R, R'
Result: Reconfiguration plan $\mathcal{P}$

$U \leftarrow \{\sigma(t) \mid t \in T\}$ *// get sub-tensor collections*
foreach $r \in R$ **do** *// start SPLIT*
  $V \leftarrow \{v \mid v \in U, \alpha(\phi(U)) = r\}$ *// get sub-tensors of r*
  foreach $v \in V$ **do**
    $\mathcal{P} \leftarrow \mathcal{P} \,\|\, \mathrm{split}(v, \sigma, \sigma')$
$S' \leftarrow \phi'(\{\sigma'(t) \mid t \in T\})$ *// get sub-collections*
foreach $r' \in R'$ **do** *// start RE-PARTITION*
  $S'_r \leftarrow \{S'_i \mid S'_i \in S', \alpha(S'_i) = r\}$ *// get sub-tensors of r′*
  foreach $s' \in S'_r$ **do**
    $t \leftarrow \mathrm{get\_base\_tensor}(\sigma', \phi', s')$
    $W \leftarrow \mathrm{get\_split\_tensors}(t, \sigma, \sigma')$
    foreach $w \in W$ **do**
      $r_w \leftarrow \mathrm{get\_resource}(\phi, \alpha, w)$
      $\mathcal{P} \leftarrow \mathcal{P} \,\|\, \mathrm{move}(w, r_w, r')$ // add MOVE
    $\mathcal{P} \leftarrow \mathcal{P} \,\|\, \mathrm{merge}(W)$ // add MERGE

1. Traverse all sub-collections in PTC'.
2. Traverse all sub-tensor in a sub-collections
3. Retrieve its original tensor T and how this tensor was sliced with SPLIT
4. For each slicing, add move by compare its r and r'
5. Merge splited tensor.



Reconfiguration plan

Algorithm 1: Reconfiguration plan generation

Data: PTC $= (T, \sigma, \phi, \alpha)$, PTC′ $= (T, \sigma', \phi', \alpha')$
Resources R, R'
Result: Reconfiguration plan $\mathcal{P}$

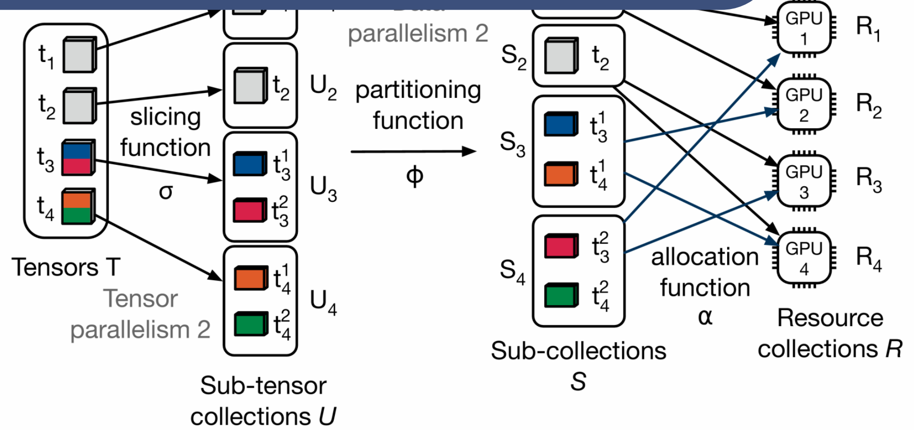
```
U ← {σ(t) | t ∈ T}                          // get sub-tensor collections
foreach r ∈ R do                             // start SPLIT
    V
    fo

S′ ← φ
foreac
    S′_r
    foreach s′ ∈ S′_r do
        t ← get_base_tensor(σ′, φ′, s′)
        W ← get_split_tensors(t, σ, σ′)
        foreach w ∈ W do
            r_w ← get_resource(φ, α, w)
            P ← P || move(w, r_w, r′)        // add MOVE
        P ← P || merge(W)                    // add MERGE
```

1. Traverse all sub-collections in PTC’.
2. Traverse all sub-tensor in a sub-collections
3. Retrieve its original tensor T and how this tensor was sliced with

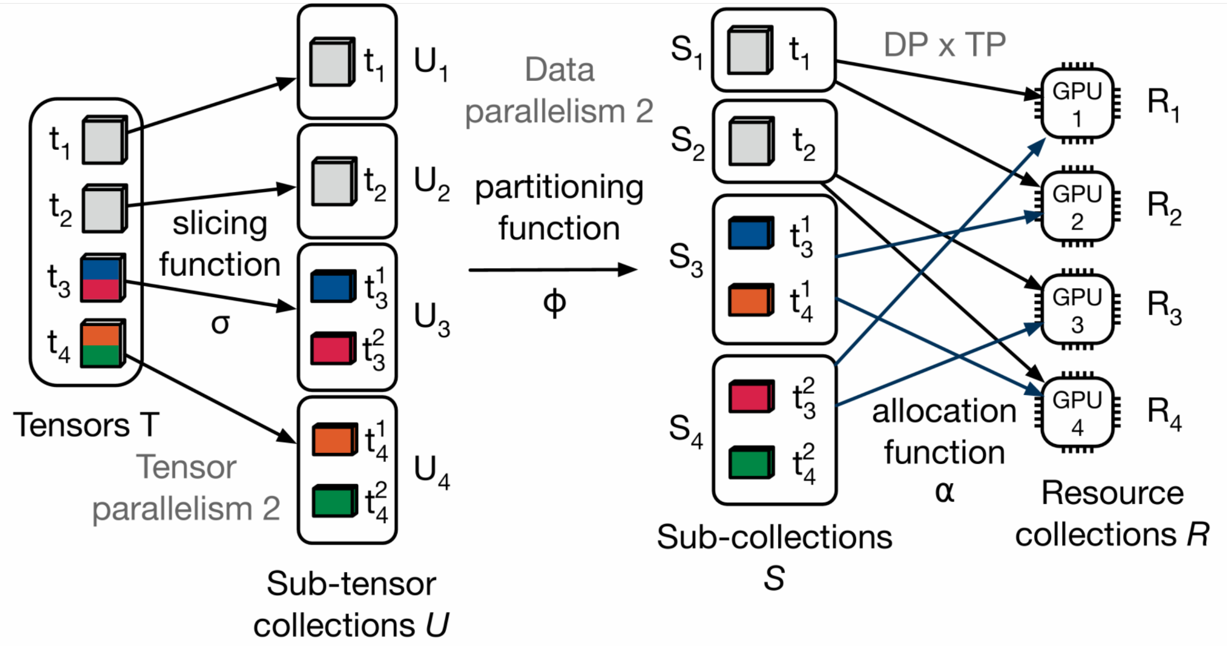
y

A very straightforward algorithm that simply leverages its abstraction.



Expanding to new parallelism strategies

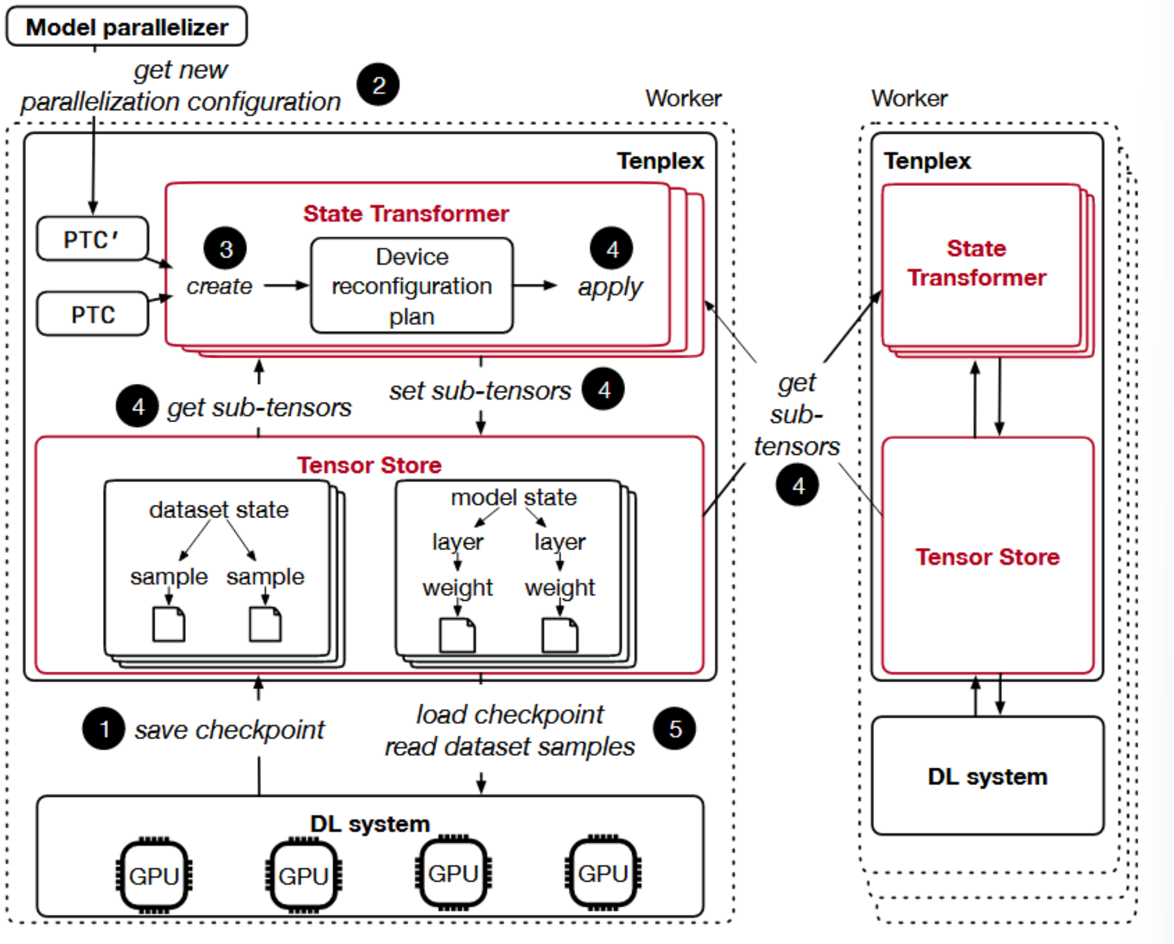
- **Expert parallelism (EP):** Modify the partition function Φ and allocation function α, without changing the slicing σ function, as EP does not split tensors.
- **Sequence parallelism(SP):** Use the slicing function σ to partition the dataset along the sequence dimension.



Tenplex Architecture

State Transformer: Apply transformation from PTC to PTC’ , according to new configuration

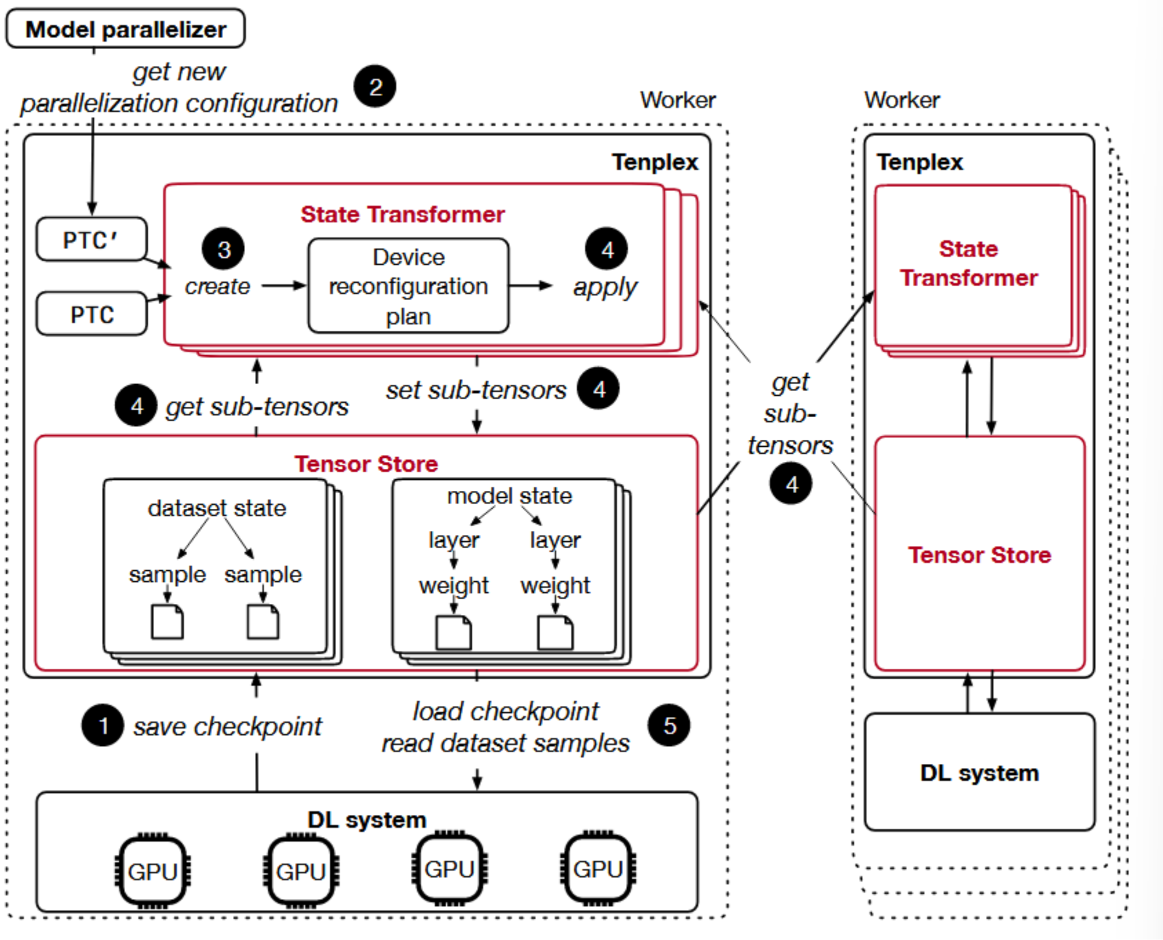
Tensor Store: Maintain model and dataset describe by PTC in a in-memory file system.



Tenplex Architecture

State Transformer: Apply transformation from PTC to PTC' , according to new configuration

1. Save checkpoint to Tensor Store
2. Get new Parallelization configuration as PTC'
3. Create reconfiguration plan using Alg.1
4. Apply split, re-partition, and merge with the help with local or remote Tensor Store
5. DL system restore job from the Tensor Store



Tenplex Architecture

Tensor Store: Maintain model and dataset describe by PTC in a in-memory file system.

Model State:

- Expose python slice-like API for State Transformer to modify sub-tensors: range=[:,2:4]
- Expose load/store API for DL system to move model in and out DL system.

Dataset State:

- Expose data sample access API to State Transformer
- Expose data access API for DL system
- Overlap training and dataset fetching, because dataset is immutable and consumed sequentially

Tenplex Architecture

◻Integration with existing training jobs

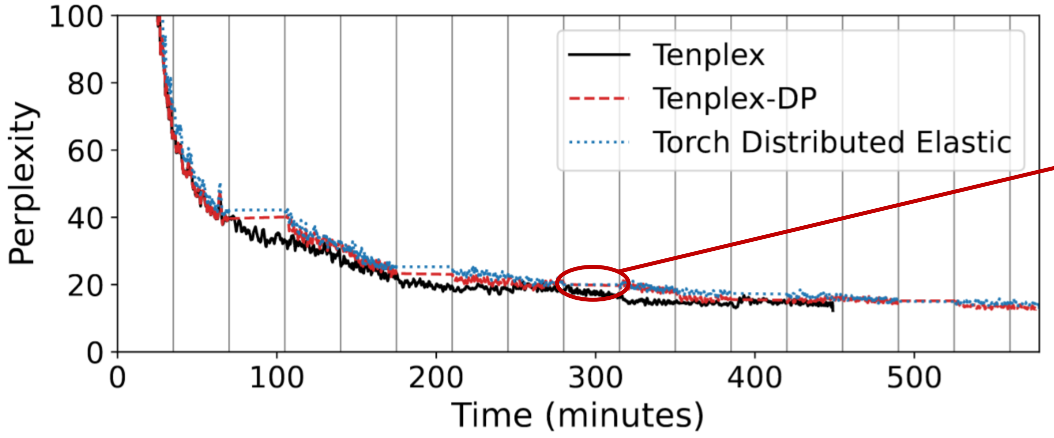
- **Job schedulers:** Notice tenplex when GPU resource changed.
  - E.g., K8s, Pollux, Ray, Sia
- **Model paralleizers:** Decide parallelization configuration according to available resource.
  - E.g.， Alpa, Megatron-LM
- **DL systems:** Allow Tenplex to externalize DL job state through APIs for load/store model
  - E.g., Pytorch, JAX
- **Training programs:** Use Tenplex’s API to access dataset and replace saving/loading checkpoint with Tenplex’s API

Evaluation

- Two Clusters:
  1. (4×NVIDIA RTX A6000) × 4
  2. (4×NVIDIA V100) × 8
- Baselines:
  - Torch Distributed Elastic v2.0
  - DeepSpeed v0.6 with Magatron-LM v23.06
  - Tenplex-DP
  - Tenplex-Central
- Models:
  - BERT-Large(340M),
  - GPT-3 (1.3B, 2.7B, 6.7B)
  - ResNet-50 (25M)
- Datasets:
  - OpenWebText
  - Wikipedia
  - ImageNet

Evaluation

- Elastic multi-dimensional parallelism:

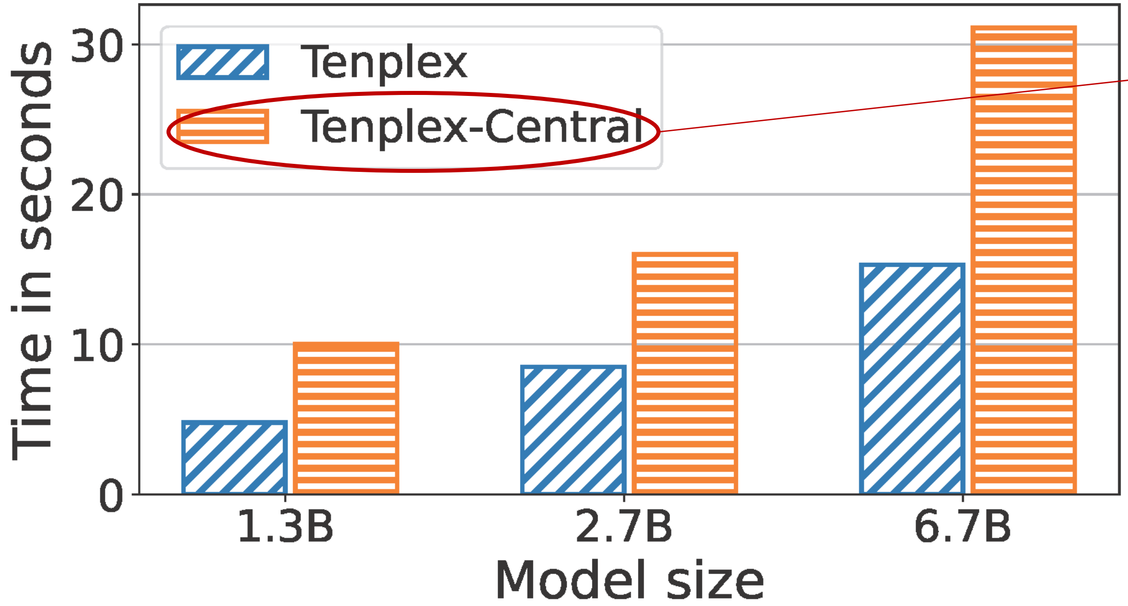


Pausing when elastic only with DP

(TP,PP,DP)	16GPU	8GPU	14GPU
Tenplex	(2,4,2)	(2,4,1)	(2,2,1)
Others	(2,4,2)	(2,4,1)	Pausing

Evaluation

- Job redeployment:
  - Redeploy a DL job from one set of 8 GPUs to another 8 GPUs.



Performs all state repartitioning at a central node

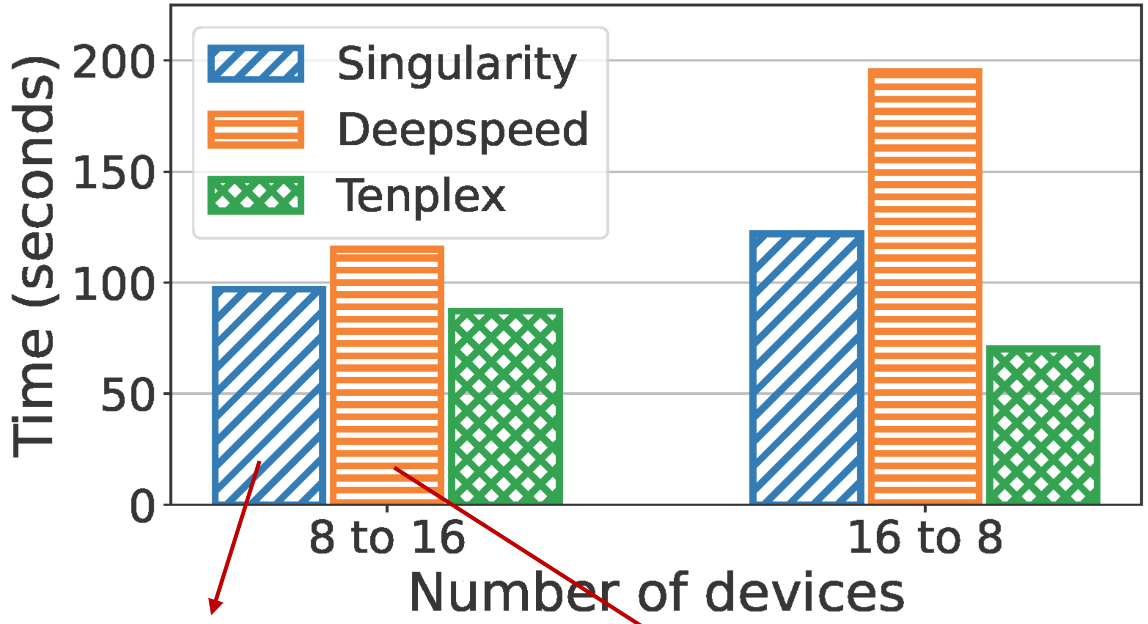
- Tenplex can **migrate state directly** between workers.
- Prevent network BW of **single worker** from becoming a bottleneck

Evaluation

□ Reconfiguration overhead:

1-> Scales up from 8 to 16 GPUs.2 -> Scales down from 16 to 8 GPUs.

Data of Singularity directly from original paper



Full GPU state transform

Relies on failure mechanism to notifying reconfiguration

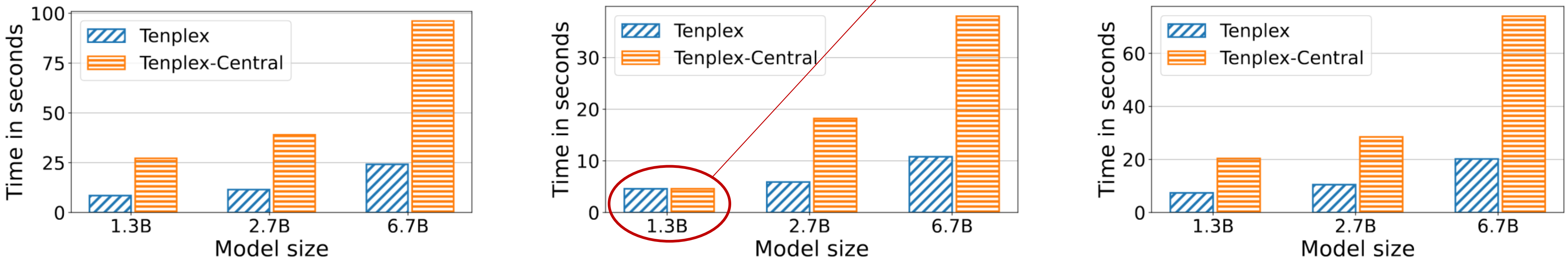
- Scales up: **24%** less time than DeepSpeed and **10%** less time than Singularity.
- Scales down: **64%** less time than DeepSpeed and **43%** less than Singularity.

Performe better when 16 to 8 because it can benefit from minimal set data movement

Evaluation

- Impact of parallelization type:
  - For DP: change from $(T, P, D) = (4, 2, 1)$ to $(4, 2, 2)$
  - For PP: change from $(T, P, D) = (4, 2, 1)$ to $(4, 4, 1)$
  - For TP: change from $(T, P, D) = (4, 2, 1)$ to $(8, 2, 1)$

- PP does not involve splitting and merging sub-tensors.
- Network is not a bottleneck here

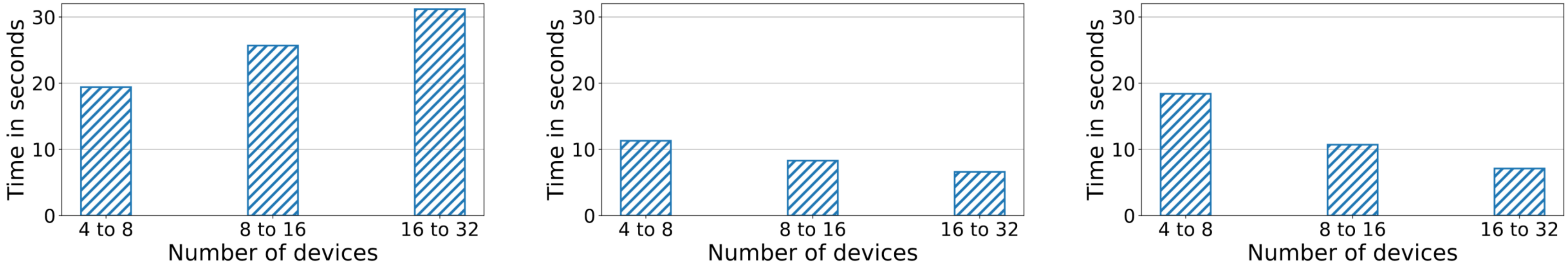


(a) Data parallelism **(b)** Pipeline parallelism **(c)** Tensor parallelism

- Tenplex takes shorter time because of a **distributed peer-to-peer** state reconfiguration

Evaluation

- Impact of cluster size for reconfiguration:
  - Keep model size fixed but change GPU resources in the cluster.
  - GPT-3-1.3B in 32-GPU testbed (V100)



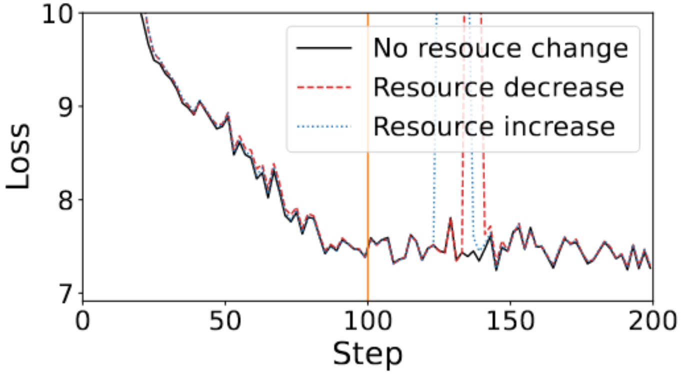
(a) Data parallelism (b) Pipeline parallelism (c) Tensor parallelism

- DP: time **increases linearly** because the number of replicas increases.
- PP: model size is constant, network BW increases with GPU count
- TP: similar to PP, but **must split and merge** sub-tensors

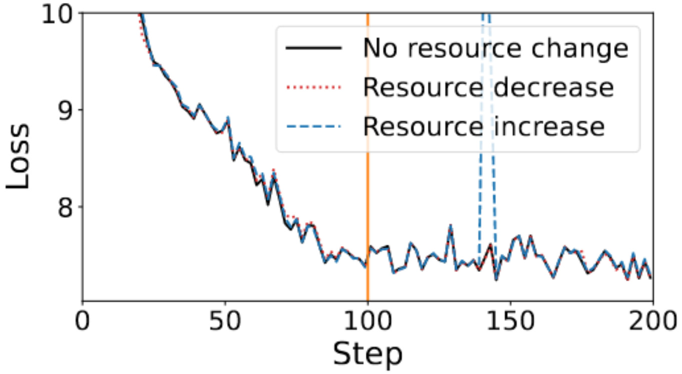
* We compare Tenplex with the baseline Tenplex-Central, as it is the only baseline that supports full multi-dimensional parallelism???

Evaluation

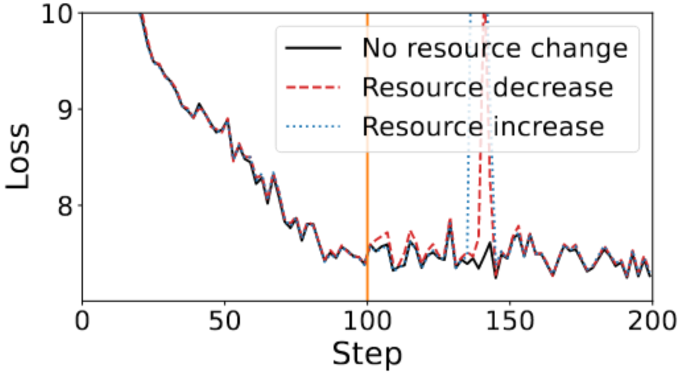
- Impact of convergency:
  - Convergency is unaffected
  - But way? No design?



(a) Data parallelism



(b) Pipeline parallelism



(c) Tensor parallelism

Concusion

- Tenplex abstracts the training state in multi-dimensional parallelization with PTC, enabling multi-dimensional transformations.
- Pros
  - A simple abstraction to describe 1. Describing state distribution in multi-dimensional parallelism 2. Managing state transitions
  - Decouples state management from DL system, allowing it to run as an external library
- Cons
  - Does not account for the time needed to find optimal multi-dimensional parallelism
  - Propose minor challenges without relevant design solutions
  - Evaluation is not convincing
  - Is it applicable to model inference?