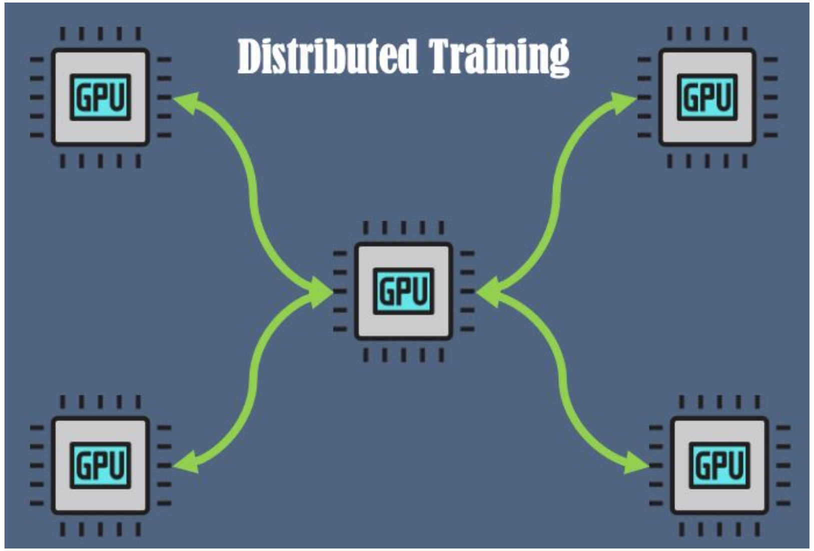
Distributed training of DNN

Шуклин Максим, 212

What is distributed training?

- **Distributed training** – is training of machine learning model using more than one machine

- **Worker** = GPU or machine



Why we need distributed training of DNN?

- Model is too big for 1 GPU
- Speed up training process
- Memory optimizations
- Accumulate computing power

Types of distributed training

- **Model parallelism**

  Model is partitioned among several workers.

- **Data parallelism**

  Each worker contains full copy of model. Data is divided into partitions and distributed by workers.

- **Tensor parallelism**

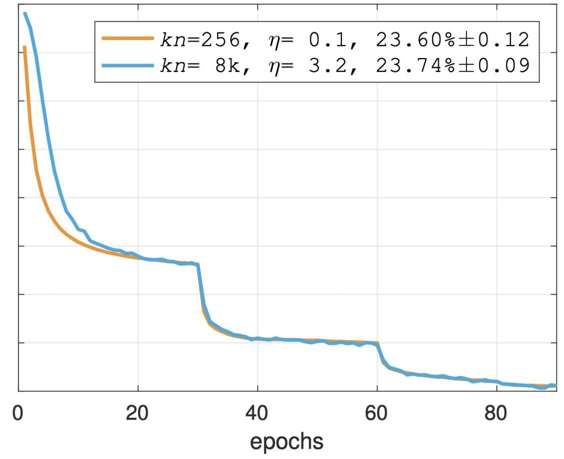
  Distribute matrix calculation by workers.

Training ImageNet in 1 Hour

Main idea: Dividing SGD mini-batches over a pool of parallel workers. **Data parallelism** takes place

In this article:

- k=256 GPUs, n=32 – mini-batch size per GPU
- Total mini-batch size is kn=8192
- ~90% scaling efficiency (8 → 256 GPU)



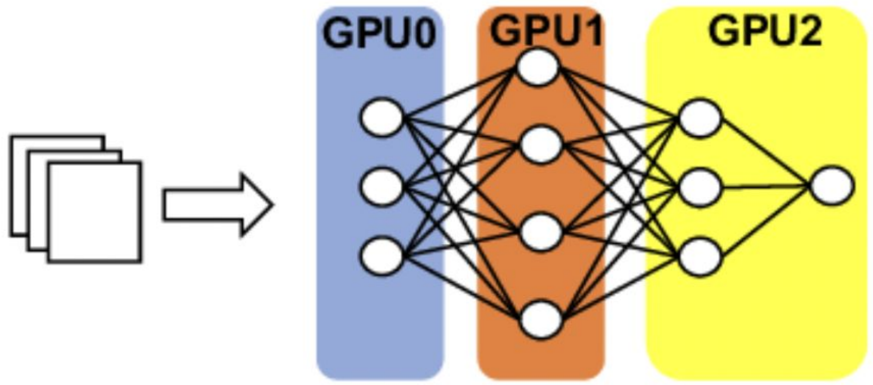
(c) gradual warmup

Based on article: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Model parallelism

Stage – sequence of consecutive layers in neural network.

Each **worker** corresponds to exactly one **stage**



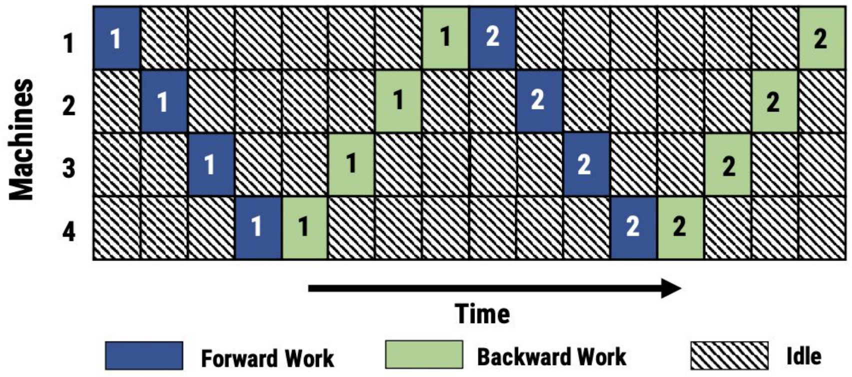
Model parallelism

Advantages:

- We can train very big models
- Memory optimization

Weak points:

- Underutilization of compute resources
- Complicated implementation

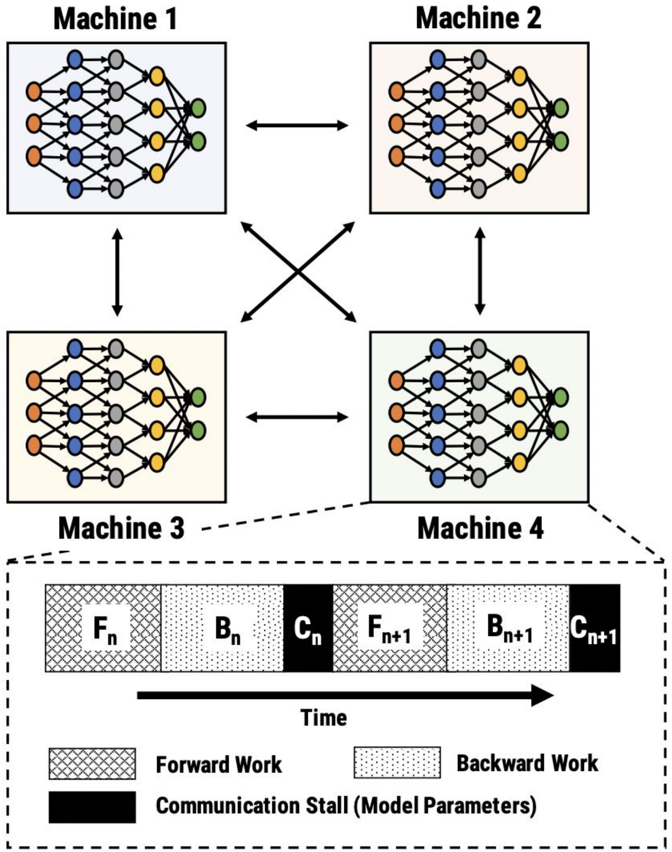


Data parallelism

In **data parallelism**, training data is distributed among all workers.

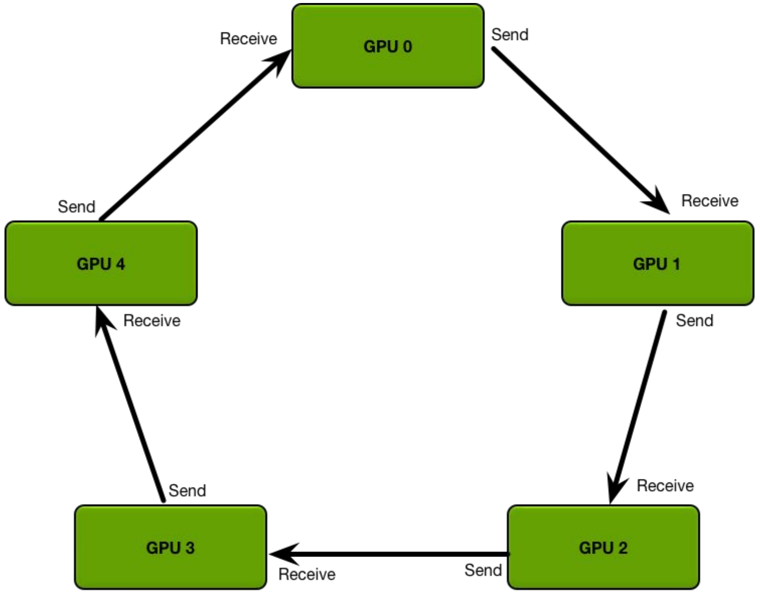
Each worker contains **full copy** of model.

Models train almost independently – with periodic **synchronization** (to update weights).

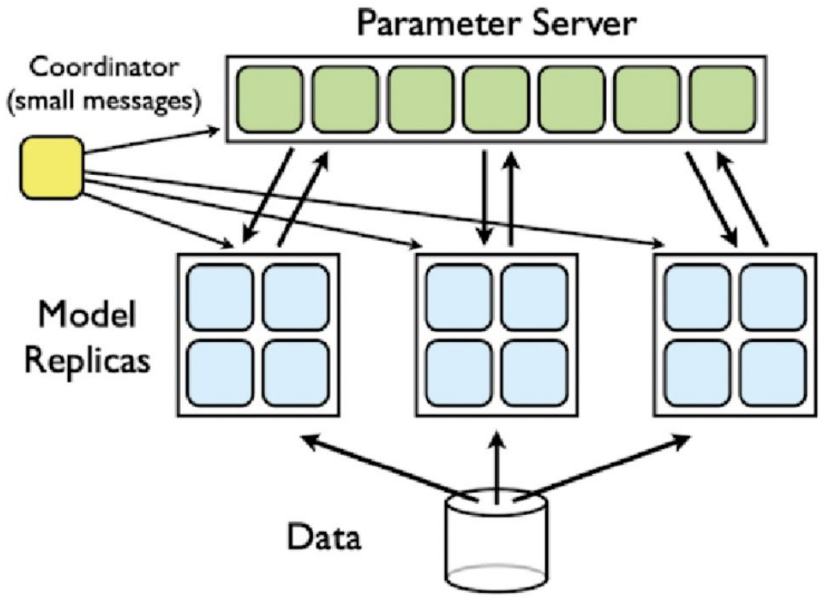


Data parallelism

Ring all-reduce



Parameter Server Architecture



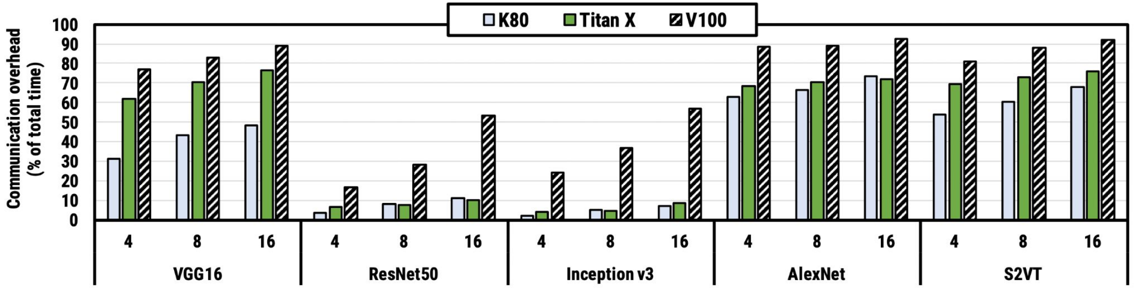
Data parallelism

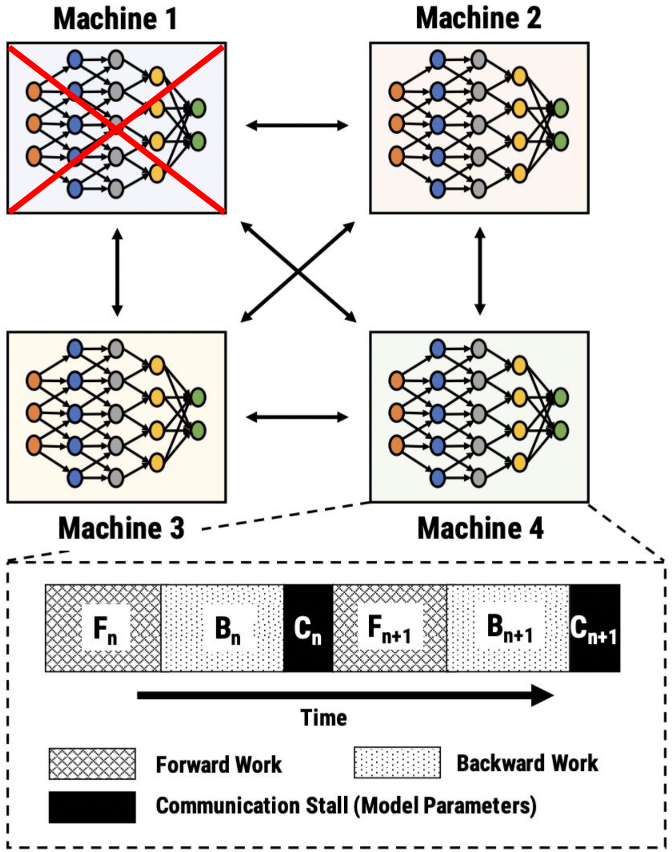
Advantages:

- Time optimization

Weak points:

- Model Synchronization
- Model Communication
- Low fault tolerance

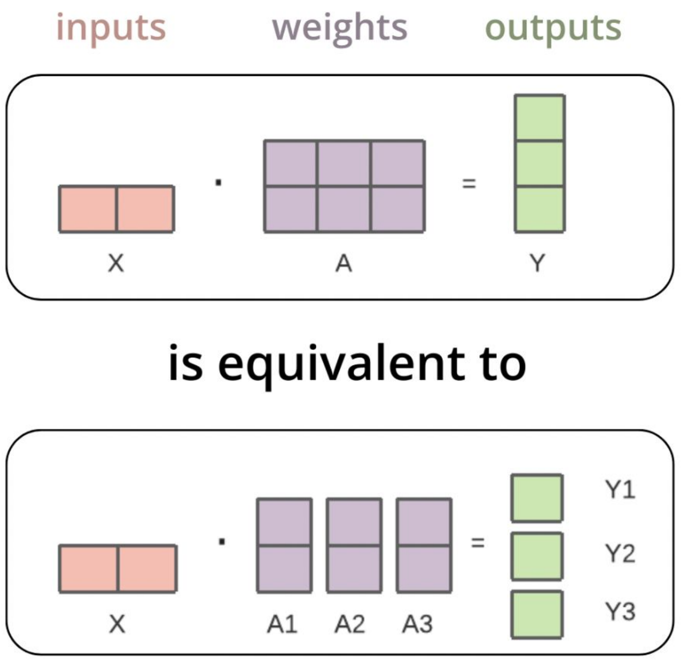




Tensor parallelism

In tensor parallelism, single weight or parameter is distributed over more than one worker.

Tensor parallelism is used for extremely large models



Modern approach (a more efficient way to train DNN)

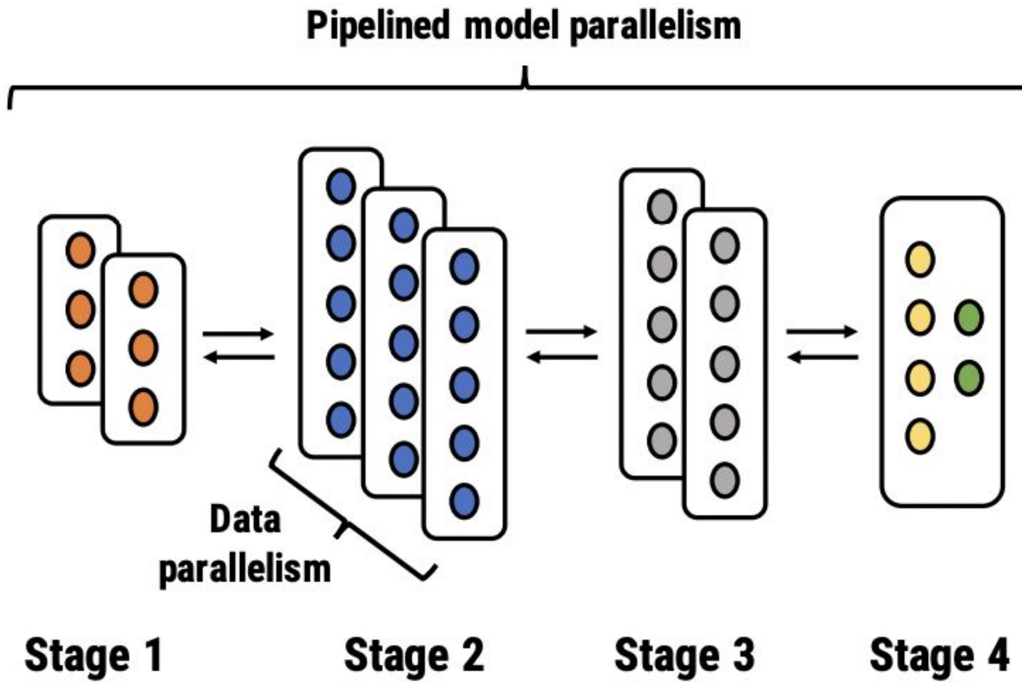
PipeDream (Microsoft, 2019)

In PipeDream: Fast and Efficient Pipeline Parallel DNN Training, published at the the 27th ACM Symposium on Operating Systems Principles (SOSP 2019), Microsoft researchers in the Systems Research Group, along with students and colleagues from Carnegie Mellon University and Stanford University, have proposed a new way to parallelize DNN training.

Long story short: **model parallelism + data parallelism**

PipeDream

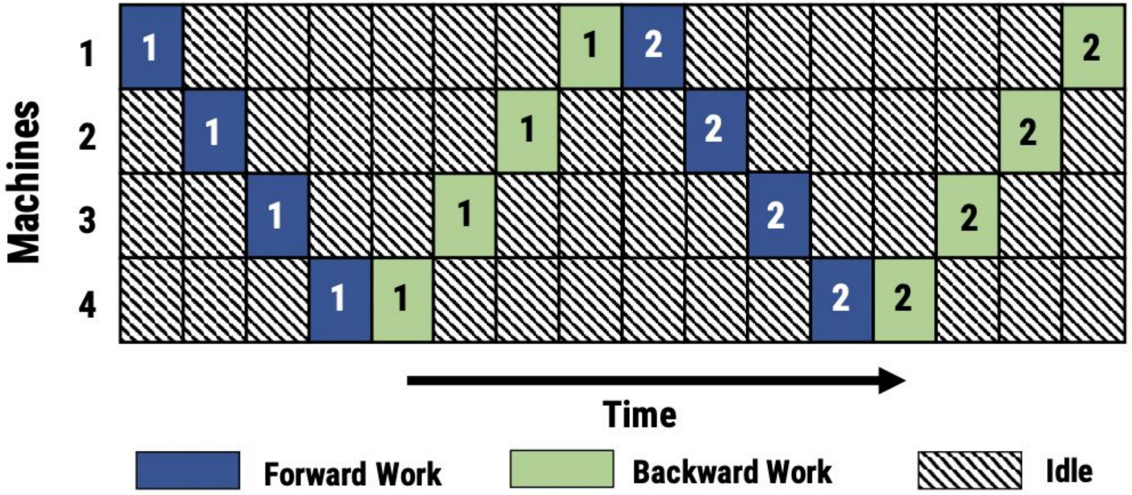
The first idea is to combine **model parallelism** and **data parallelism**



PipeDream

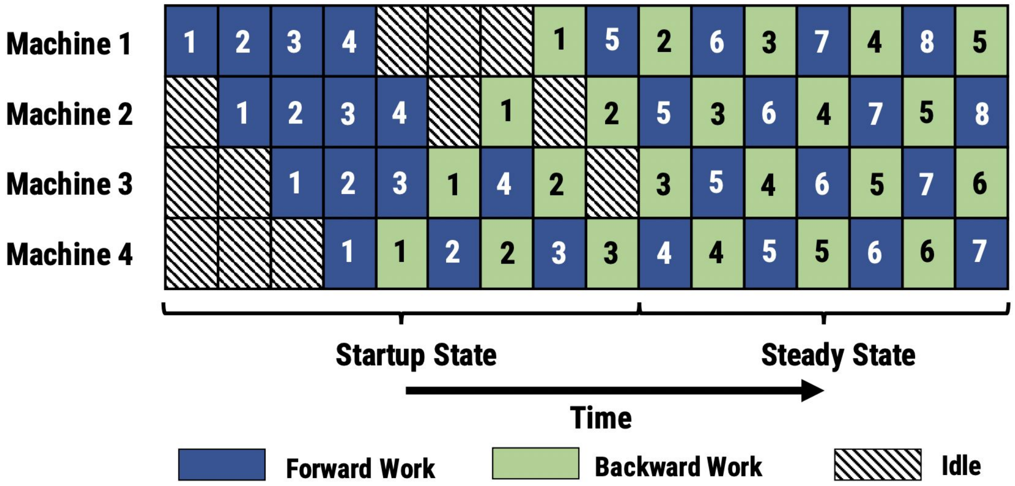
Remember problem with **model parallelism:**

Underutilization of compute resources



PipeDream

PipeDream suggests a solution for underutilization problem:



Solution: on completing the forward pass for a minibatch, each stage asynchronously sends the output activations to the next stage, while simultaneously starting to process another minibatch.

The same with backward pass.

PipeDream

Problem

The time of one iteration ≈ time of slowest stage

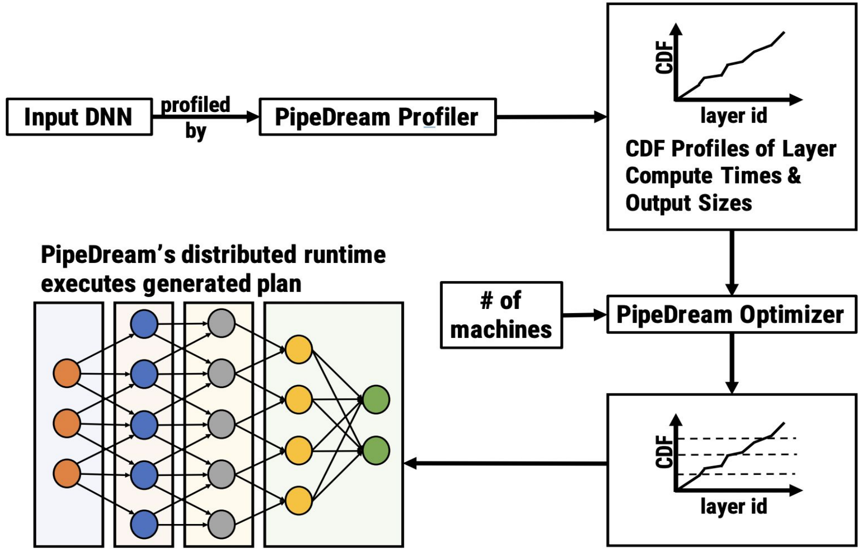
What to do?

We need to minimize the time taken by the **slowest** stage of the pipeline!

PipeDream

How to solve it?

- Profiling the DNN Model
- PipeDream's Partitioning Algorithm (PipeDream Optimizer)


Input DNN
profiled by
PipeDream Profiler
CDF
layer id
CDF Profiles of Layer Compute Times & Output Sizes
# of machines
PipeDream Optimizer
CDF
layer id
PipeDream's distributed runtime executes generated plan

PipeDream: *Profiling the DNN Model*

- Short run of the DNN model using 1000 mini-batches on one of the machines
- Measuring T_l – total computation time for layer l
- Calculate W_l^m – the time for weight synchronization for the layer when using a distributed parameter server
- The time taken by a single stage spanning layers i through j, replicated over m machines:

$$T(i \rightarrow j, m) = \frac{1}{m} \max\left(\sum_{l=i}^{j} T_l, \sum_{l=i}^{j} W_l^m \right)$$

PipeDream: *PipeDream's Partitioning Algorithm*

- Let $A(j, m)$ denote the time taken by the slowest stage in the optimal pipeline between layers 1 and j using m machines
- Goal – to find $A(N, M)$
- Calculate $A(j, m)$ using dynamic programming:

Initialization: $A(j, m) = T(1 \rightarrow j, m)$

Step:

$$A(j, m) = \min_{1 \le i < j} \min_{1 \le m' < m} \max \begin{cases} A(i, m - m') \\ 2 \cdot C_i \\ T(i + 1 \rightarrow j, m') \end{cases}$$

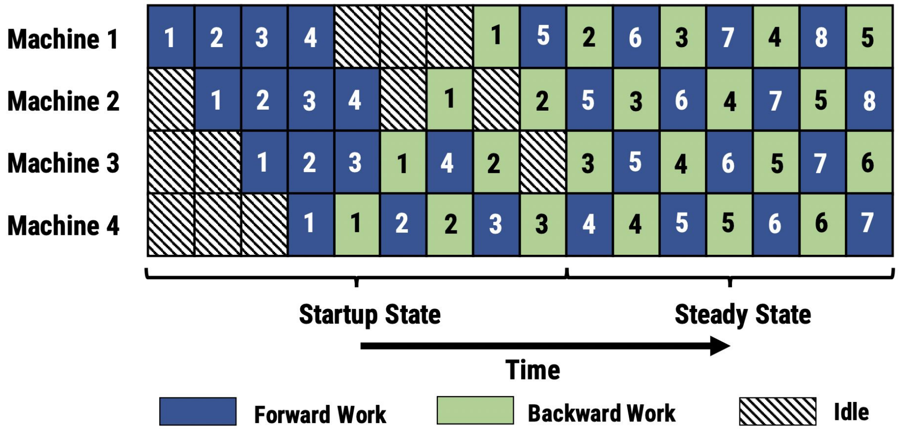
Complexity: $O(N^2 M^2)$

PipeDream: more problems!

Weight Staleness

The forward pass for each minibatch is performed using one version of parameters and the backward pass using a different version of parameters.

Furthermore, different stages in the DNN model suffer from different degrees of staleness



PipeDream: more solutions!

- **Weight Stashing**

  Store multiple versions of weights at one stage. During backward pass use the same which was used during corresponding forward pass

  Eliminates potential inconsistency **among one stage**
- **Vertical Sync**

  Eliminates potential inconsistency **among stages**

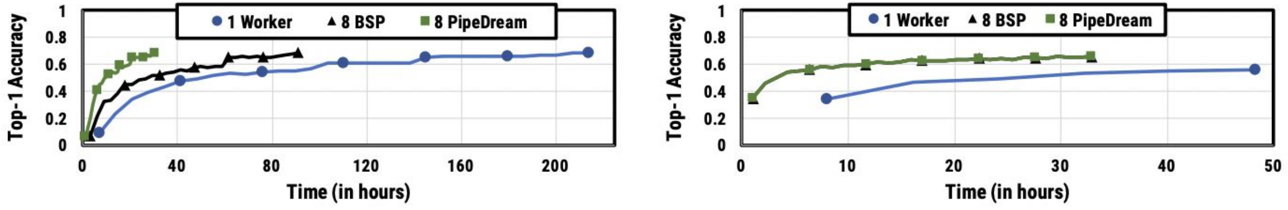
  Each mini-batch at each stage use the same version of weights

- *Weight stashing* is critical for meaningful learning.
- Impact of *vertical sync* is negligible and usually it’s not used

PipeDream

DNN Model	# Machines (Cluster)	BSP speedup over 1 machine	PipeDream Config	PipeDream speedup over 1 machine	PipeDream speedup over BSP	PipeDream communication reduction over BSP
VGG16	4 (A)	1.47×	2-1-1	3.14×	2.13×	90%
	8 (A)	2.35×	7-1	7.04×	2.99×	95%
	16 (A)	3.28×	9-5-1-1	9.86×	3.00×	91%
	8 (B)	1.36×	7-1	6.98×	5.12×	95%
Inception-v3	8 (A)	7.66×	8	7.66×	1.00×	0%
	8 (B)	4.74×	7-1	6.88×	1.45×	47%
S2VT	4 (A)	1.10×	2-1-1	3.34×	3.01×	95%

Table 1: Summary of results comparing PipeDream with data-parallel configurations (BSP) when training models to their advertised final accuracy. "PipeDream config" represents the configuration generated by our partinioning algorithm—e.g., "2-1-1" is a configuration in which the model is split into three stages with the first stage replicated across 2 machines.



(a) VGG16 **(b)** Inception-v3

Figure 10: Accuracy vs. time for VGG16 and Inception-v3 with 8 machines on Cluster-A

Conclusion

Distributed learning helps to:

- Train big models, that can’t be stored on 1 GPU
- Accelerate model training

Among **data parallelism, model parallelism and tensor parallelism the first one is the most popular and simple.**

In modern approaches base algorithms are usually combined:

- **Pipeline Parallelism = Model Parallelism + Data Parallelism**
- **PipeDream is up to 5x faster in time-to-accuracy compared to data parallel training**

References

- PipeDream: Fast and Efficient Pipeline Parallel DNN Training

- About model and tensor parallelism: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

- Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

- Modern approaches in distributed DNN training