Distributed Machine Learning Frameworks

Amir H. Payberah
payberah@kth.se
2020-12-07

The Course Web Page

https://fid3024.github.io

Review of the Current Frameworks

TensorFlow (1/4)

- TensorFlow supports data parallelism and model partitioning (as of v0.8).

TensorFlow (1/4)

- TensorFlow supports data parallelism and model partitioning (as of v0.8).
- As of v2.2, the Multi Worker Mirrored Strategy (allreduce) is integrated into TensorFlow for data parallelism.

TensorFlow (1/4)

- TensorFlow supports data parallelism and model partitioning (as of v0.8).
- As of v2.2, the Multi Worker Mirrored Strategy (allreduce) is integrated into TensorFlow for data parallelism.
  - Its update rule is synchronous and it has communication and computation overlapped.

TensorFlow (1/4)

- TensorFlow supports data parallelism and model partitioning (as of v0.8).
- As of v2.2, the Multi Worker Mirrored Strategy (allreduce) is integrated into TensorFlow for data parallelism.
  - Its update rule is synchronous and it has communication and computation overlapped.
- TensorFlow also has extensions to support different parallelization approaches.

TensorFlow (2/4)

- Mesh-TensorFlow is a language for distributed deep learning in TensorFlow.

TensorFlow (2/4)

- Mesh-TensorFlow is a language for distributed deep learning in TensorFlow.
- It is capable of specifying a broad class of distributed tensor computations.

TensorFlow (2/4)

- Mesh-TensorFlow is a language for distributed deep learning in TensorFlow.
- It is capable of specifying a broad class of distributed tensor computations.
- Mainly used for model parallelism in TensorFlow.

TensorFlow (2/4)

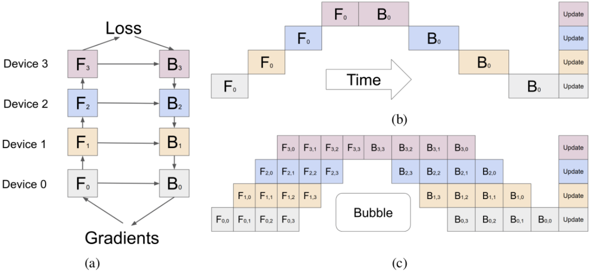
- Mesh-TensorFlow is a language for distributed deep learning in TensorFlow.
- It is capable of specifying a broad class of distributed tensor computations.
- Mainly used for model parallelism in TensorFlow.
- A mesh is an n-dimensional array of processors, connected by a network.

TensorFlow (2/4)

- Mesh-TensorFlow is a language for distributed deep learning in TensorFlow.
- It is capable of specifying a broad class of distributed tensor computations.
- Mainly used for model parallelism in TensorFlow.
- A mesh is an n-dimensional array of processors, connected by a network.
- Each tensor is distributed across all processors in a mesh.

TensorFlow (3/4)

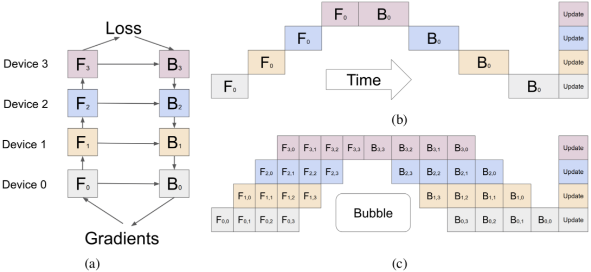
- GPipe is a pipeline parallelism library implemented under Lingvo (a TensorFlow framework focusing on seq-to-seq models).



[Huang et al., GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, 2019]

TensorFlow (3/4)

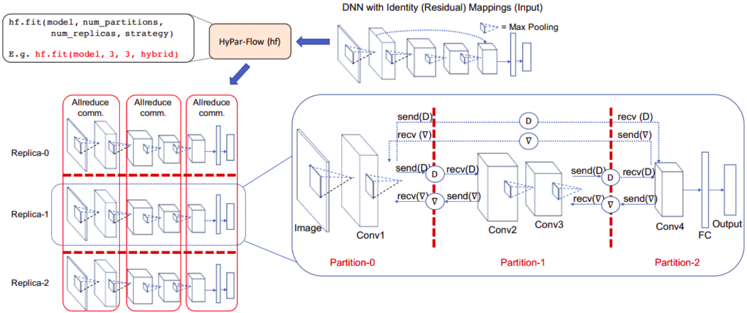
- GPipe is a pipeline parallelism library implemented under Lingvo (a TensorFlow framework focusing on seq-to-seq models).
- Partitions operation in the forward and backward pass and allows data transfer between neighboring partitions.



[Huang et al., GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, 2019]

TensorFlow (4/4)

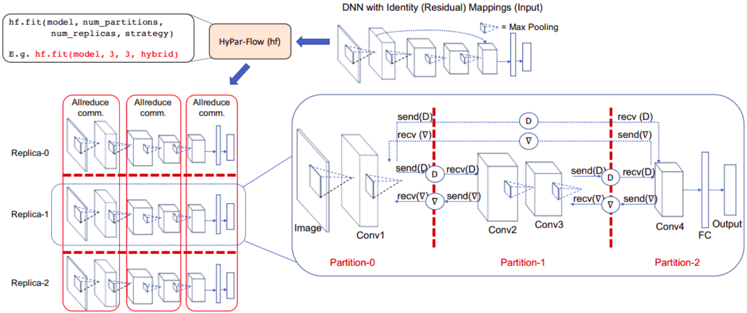
- HyPar-Flow is an implementation of data, model, and hybrid parallelization on Eager TensorFlow.



[Awan et al., HyPar-Flow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training with TensorFlow, 2020]

TensorFlow (4/4)

- HyPar-Flow is an implementation of data, model, and hybrid parallelization on Eager TensorFlow.
- It only requires the strategy, the number of model partitions, and the number of model replicas from the user to utilize them with every possible intra-iteration parallelization.



[Awan et al., HyPar-Flow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training with TensorFlow, 2020]

PyTorch (1/4)

- Caffe is a DL framework that does not support distributed training out-of-the-box.

PyTorch (1/4)

- Caffe is a DL framework that does not support distributed training out-of-the-box.
- Many extensions of Caffe to support distributed training centralized or decentralized.

PyTorch (1/4)

- Caffe is a DL framework that does not support distributed training out-of-the-box.
- Many extensions of Caffe to support distributed training centralized or decentralized.
- FireCaffe and MPI-Caffe support data and model parallelism on multi-GPU clusters, respectively.

PyTorch (1/4)

- Caffe is a DL framework that does not support distributed training out-of-the-box.
- Many extensions of Caffe to support distributed training centralized or decentralized.
- FireCaffe and MPI-Caffe support data and model parallelism on multi-GPU clusters, respectively.
- Intel-Caffe and NUMA-Caffe support data parallelism training on CPU-based clusters.

PyTorch (1/4)

- Caffe is a DL framework that does not support distributed training out-of-the-box.
- Many extensions of Caffe to support distributed training centralized or decentralized.
- FireCaffe and MPI-Caffe support data and model parallelism on multi-GPU clusters, respectively.
- Intel-Caffe and NUMA-Caffe support data parallelism training on CPU-based clusters.
- S-Caffe is a CUDA-Aware MPI runtime and Caffe for data parallelism on GPU clusters.

PyTorch (2/4)

- Chainer is a Define-by-Run (imperative) DL framework.

PyTorch (2/4)

- Chainer is a Define-by-Run (imperative) DL framework.
- It only supports data parallelism.

PyTorch (2/4)

- Chainer is a Define-by-Run (imperative) DL framework.
- It only supports data parallelism.
- It has a synchronous decentralized design for allreduce communication.

PyTorch (3/4)

- PyTorch is a successor of Caffe2, which is inspired by Chainer.

PyTorch (3/4)

- PyTorch is a successor of Caffe2, which is inspired by Chainer.
- It is an imperative DL framework using dynamic computation graphs and automatic differentiation.

PyTorch (3/4)

- PyTorch is a successor of Caffe2, which is inspired by Chainer.
- It is an imperative DL framework using dynamic computation graphs and automatic differentiation.
- PyTorch mainly focuses on ease of use, and enables users with options in training their models.

PyTorch (3/4)

- PyTorch is a successor of Caffe2, which is inspired by Chainer.
- It is an imperative DL framework using dynamic computation graphs and automatic differentiation.
- PyTorch mainly focuses on ease of use, and enables users with options in training their models.
- PyTorch RPC is developed to support model parallelism.

PyTorch (4/4)

- PyTorch Distributed Data Parallel (DPP) is an extra feature to PyTorch (available as of v1.5).

PyTorch (4/4)

- PyTorch Distributed Data Parallel (DPP) is an extra feature to PyTorch (available as of v1.5).
- PyTorch DDP utilizes some techniques to increase performance, such as

PyTorch (4/4)

- PyTorch Distributed Data Parallel (DPP) is an extra feature to PyTorch (available as of v1.5).
- PyTorch DDP utilizes some techniques to increase performance, such as
  - Gradient bucketing (small tensors bucket into one allreduce operation)

PyTorch (4/4)

- PyTorch Distributed Data Parallel (DPP) is an extra feature to PyTorch (available as of v1.5).
- PyTorch DDP utilizes some techniques to increase performance, such as
  - Gradient bucketing (small tensors bucket into one allreduce operation)
  - Overlapping communication with computation

PyTorch (4/4)

- PyTorch Distributed Data Parallel (DPP) is an extra feature to PyTorch (available as of v1.5).
- PyTorch DDP utilizes some techniques to increase performance, such as
  - Gradient bucketing (small tensors bucket into one allreduce operation)
  - Overlapping communication with computation
  - Skipping synchronization

MXNet (1/2)

- MXNet is a multi-language ML library.

MXNet (1/2)

- MXNet is a multi-language ML library.
- It blends declarative symbolic expression with imperative tensor computation.

MXNet (1/2)

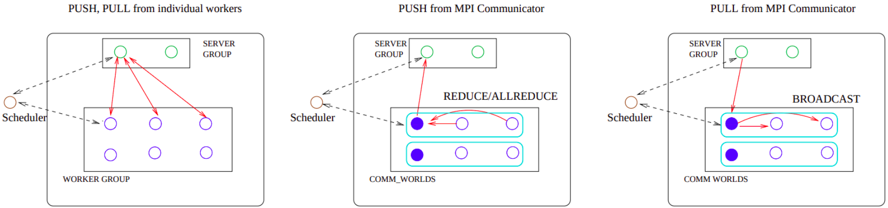
- MXNet is a multi-language ML library.
- It blends declarative symbolic expression with imperative tensor computation.
- It uses a distributed key-value store for data synchronization over multiple devices.

MXNet (2/2)

- MXNet-MPI is the extension of MXNet that replaces each worker in a parameter server architecture with a group of workers.
- Workers of each group are synced together using an MPI collective operation.

MXNet (2/2)

- MXNet-MPI is the extension of MXNet that replaces each worker in a parameter server architecture with a group of workers.
- Workers of each group are synced together using an MPI collective operation.



[Mamidala et al., MXNet-MPI: Embedding MPI parallelism in Parameter Server Task Model for Scaling Deep Learning, 2018]

Horovod

- Horovod is a stand-alone Python library for data parallelism using an optimized ring_allreduce collective and a tensor fusion algorithm.

Horovod

- Horovod is a stand-alone Python library for data parallelism using an optimized ring_allreduce collective and a tensor fusion algorithm.
- It works on top of another DL framework (e.g., TensorFlow, PyTorch, and MXNET).

Horovod

- Horovod is a stand-alone Python library for data parallelism using an optimized ring_allreduce collective and a tensor fusion algorithm.
- It works on top of another DL framework (e.g., TensorFlow, PyTorch, and MXNET).
- It has one of the most optimized asynchronous collectives.

Horovod

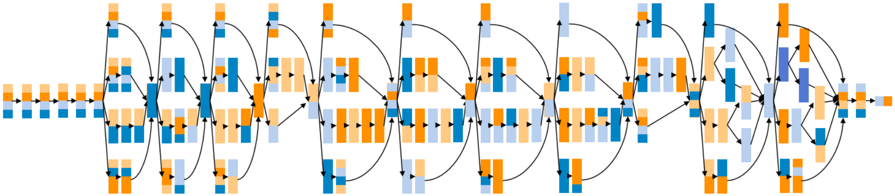
- Horovod is a stand-alone Python library for data parallelism using an optimized ring_allreduce collective and a tensor fusion algorithm.
- It works on top of another DL framework (e.g., TensorFlow, PyTorch, and MXNET).
- It has one of the most optimized asynchronous collectives.
- However, the communication overhead significantly grows with the number of nodes.

FlexFlow

- FlexFlow can parallelize a DNN in the Sample, Operation, Attribute, and Parameter (SOAP) dimensions.

FlexFlow

- FlexFlow can parallelize a DNN in the Sample, Operation, Attribute, and Parameter (SOAP) dimensions.
- It uses guided randomized search of the SOAP space to find a fast parallelization strategy for a specific parallel machine.



[Jia et al., Beyond Data and Model Parallelism for Deep Neural Networks, 2019]

BigDL

- BigDL is a distributed DL framework for data parallelism on top of Spark.

BigDL

- BigDL is a distributed DL framework for data parallelism on top of Spark.
- It does not support model parallelism.

BigDL

- BigDL is a distributed DL framework for data parallelism on top of Spark.
- It does not support model parallelism.
- It favors coarse-grained operations where data transformations are immutable.

BigDL

- BigDL is a distributed DL framework for data parallelism on top of Spark.
- It does not support model parallelism.
- It favors coarse-grained operations where data transformations are immutable.
- It runs a series of Spark jobs, which are scheduled by Spark.

BigDL

- BigDL is a distributed DL framework for data parallelism on top of Spark.
- It does not support model parallelism.
- It favors coarse-grained operations where data transformations are immutable.
- It runs a series of Spark jobs, which are scheduled by Spark.
- Due to using Spark, it is equipped with fault tolerance and a fair load balancing mechanism.

ZeRO and DeepSpeed

- ZeRO focuses on solving the memory limitation problem while attempting to minimize the overhead.

ZeRO and DeepSpeed

- ZeRO focuses on solving the memory limitation problem while attempting to minimize the overhead.
- It partitions activations, optimizer states, gradients, and parameters and distributes them equally overall available nodes.

ZeRO and DeepSpeed

- ZeRO focuses on solving the memory limitation problem while attempting to minimize the overhead.
- It partitions activations, optimizer states, gradients, and parameters and distributes them equally overall available nodes.
- It then employs overlapping collective operations to reconstruct the tensors as needed.

ZeRO and DeepSpeed

- ZeRO focuses on solving the memory limitation problem while attempting to minimize the overhead.
- It partitions activations, optimizer states, gradients, and parameters and distributes them equally overall available nodes.
- It then employs overlapping collective operations to reconstruct the tensors as needed.
- DeepSpeed brings ZeRO techniques through lightweight APIs compatible with PyTorch.

BigDL: A Distributed Deep Learning Framework for Big Data

Big Data vs. Deep Learning Frameworks

- Big data and deep learning systems have different distributed execution model.

Big Data vs. Deep Learning Frameworks

- Big data and deep learning systems have different distributed execution model.
- Big data tasks are embarrassingly parallel and independent of each other.

Big Data vs. Deep Learning Frameworks

- Big data and deep learning systems have different distributed execution model.
- Big data tasks are embarrassingly parallel and independent of each other.
- Deep learning tasks need to coordinate with and depend on others.

Big Data vs. Deep Learning Frameworks

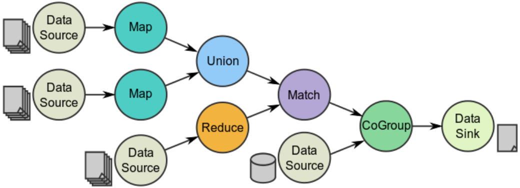
- Big data and deep learning systems have different distributed execution model.
- Big data tasks are embarrassingly parallel and independent of each other.
- Deep learning tasks need to coordinate with and depend on others.
- Several connectors, e.g., TFX, CaffeOnSpark, TensorFlowOnSpark, SageMaker.

Big Data vs. Deep Learning Frameworks

- Big data and deep learning systems have different distributed execution model.
- Big data tasks are embarrassingly parallel and independent of each other.
- Deep learning tasks need to coordinate with and depend on others.
- Several connectors, e.g., TFX, CaffeOnSpark, TensorFlowOnSpark, SageMaker.
- However, the adaptation between different frameworks can impose very large overheads in practice.

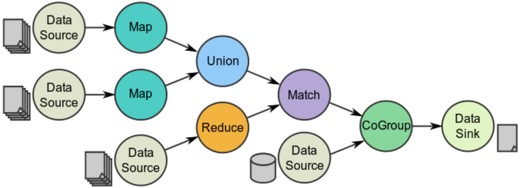
Spark Dataflow Model

- Job is described based on directed acyclic graphs (DAG) data flow.



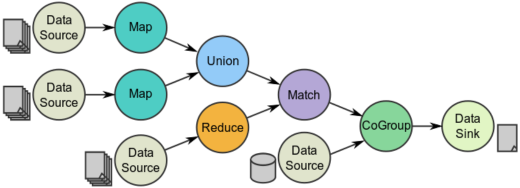
Spark Dataflow Model

- Job is described based on directed acyclic graphs (DAG) data flow.
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.



Spark Dataflow Model

- Job is described based on directed acyclic graphs (DAG) data flow.
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Parallelizable operators



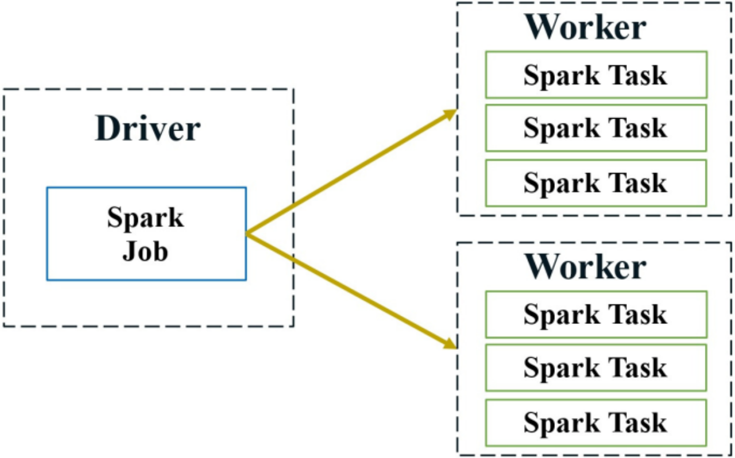
Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
  - Like a LinkedList <MyObjects>

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

Spark Execution Model



[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

BigDL

- Directly implements the distributed deep learning support in Spark.

BigDL

- Directly implements the distributed deep learning support in Spark.
- An data-analytics integrated deep learning pipeline can be executed as a standard Spark jobs.

BigDL

- Directly implements the distributed deep learning support in Spark.
- An data-analytics integrated deep learning pipeline can be executed as a standard Spark jobs.

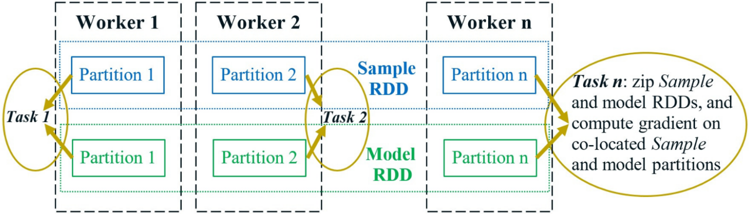
```
#distributed data processing
spark = SparkContext(appName="text_classifier", ...)
input_rdd = spark.textFile("hdfs://...")
train_rdd = input_rdd.map(lambda x: read_text_and_label(x))
                     .map(lambda data: decode_to_ndarrays(data))
                     .map(lambda arrays: to_sample(arrays))

#distributed training
model = Sequential().add(Recurrent().add(LSTM(...)))
                    .add(Linear(...)).add(LogSoftMax())
optimizer = Optimizer(model=model, training_rdd=train_rdd,
                      criterion=ClassNLLCriterion(),
                      optim_method=Adagrad(), ...)
trained_model = optimizer.optimize()

#distributed inference
test_rdd = ...
prediction_rdd = trained_model.predict(test_rdd)
```

Data-Parallel Training in BigDL (1/3)

- BigDL provides synchronous data-parallel training to train an NN model.

Data-Parallel Training in BigDL (1/3)

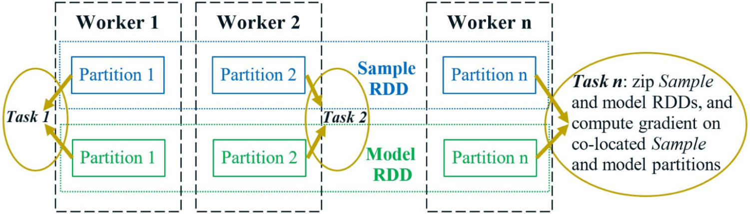
- BigDL provides synchronous data-parallel training to train an NN model.
- RDD of Samples, which are automatically partitioned across the Spark cluster.



[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

Data-Parallel Training in BigDL (1/3)

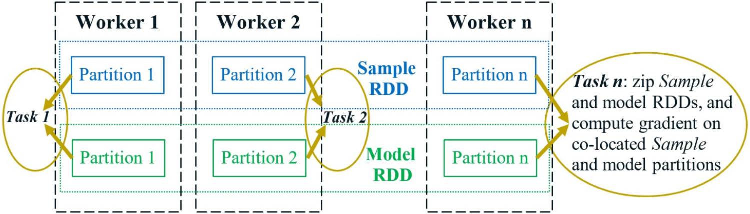
- BigDL provides synchronous data-parallel training to train an NN model.
- RDD of Samples, which are automatically partitioned across the Spark cluster.
- RDD of models, each of which is a replica of the original NN model.



[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

Data-Parallel Training in BigDL (2/3)

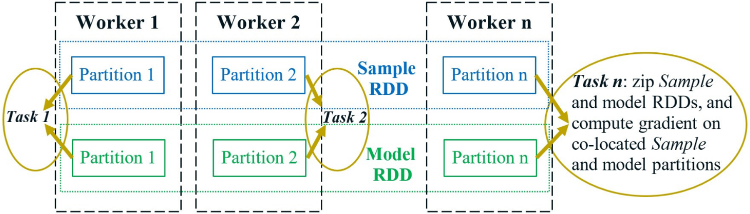
- In each iteration, a single model forward-backward Spark job.



[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

Data-Parallel Training in BigDL (2/3)

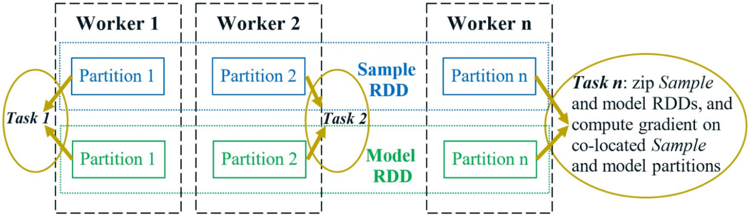
- In each iteration, a single model forward-backward Spark job.
- Applies the functional zip operator to the co-located partitions of the two RDDs.



[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

Data-Parallel Training in BigDL (2/3)

- In each iteration, a single model forward-backward Spark job.
- Applies the functional zip operator to the co-located partitions of the two RDDs.
- Then, computes the local gradients in parallel for each model replica.



[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

Data-Parallel Training in BigDL (3/3)

Algorithm 1 Data-parallel training in BigDL

```
for i = 1 to M do
    //"model forward-backward" job
    for each task in the Spark job do
        read the latest weights;
        get a random batch of data from local Sample partition;
        compute local gradients (forward-backward on local model
         replica);
    end for
    //"parameter synchronization" job
    aggregate (sum) all the gradients;
    update the weights per specified optimization method;
end for
```

[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

Parameter Synchronization in BigDL (1/2)

- Parameter synchronization based using parameter server or AllReduce requires fine-grained data access.

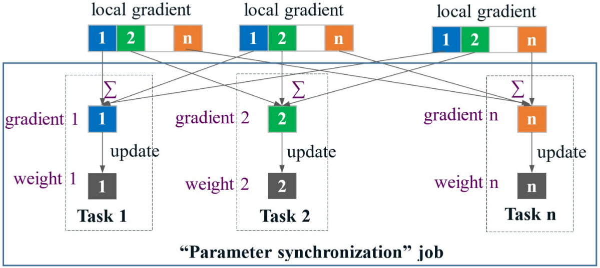
Parameter Synchronization in BigDL (1/2)

- Parameter synchronization based using parameter server or AllReduce requires fine-grained data access.
- Fine-grained operations are not supported by Spark.

Parameter Synchronization in BigDL (1/2)

- Parameter synchronization based using parameter server or AllReduce requires fine-grained data access.
- Fine-grained operations are not supported by Spark.
- BigDL directly implements an efficient AllReduce-like operation using existing primitives in Spark.

Parameter Synchronization in BigDL (2/2)



Algorithm 2 "Parameter synchronization" job

1: **for** each task n in the "parameter synchronization" job **do**
2: **shuffle** the n^{th} partition of all gradients to this task;
3: aggregate (sum) these gradients;
4: updates the n^{th} partition of the weights;
5: **broadcast** the n^{th} partition of the updated weights;
6: **end for**

[Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019]

PyTorch Distributed: Experiences on Accelerating Data Parallel Training

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.
  - In this calss, applications provide their model at construction time.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.
  - In this calss, applications provide their model at construction time.
  - Its behavior during the forward pass is specified by its `forward` member function.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.
  - In this calss, applications provide their model at construction time.
  - Its behavior during the forward pass is specified by its `forward` member function.
- A `Module` can contain `Tensors` as parameters.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.
  - In this calss, applications provide their model at construction time.
  - Its behavior during the forward pass is specified by its `forward` member function.
- A `Module` can contain `Tensors` as parameters.
  - A `LinearModule` contains a `weight` and a `bias` parameter.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.
  - In this calss, applications provide their model at construction time.
  - Its behavior during the forward pass is specified by its `forward` member function.
- A `Module` can contain `Tensors` as parameters.
  - A `LinearModule` contains a `weight` and a `bias` parameter.
  - Whose `forward` function generates the output by multiplying the input with the `weight` and adding the `bias`.

PyTorch (1/2)

- PyTorch organizes values into `Tensors`, generic n-dimensional arrays.
- A `Module` defines a transform from input values to output values.
  - In this calss, applications provide their model at construction time.
  - Its behavior during the forward pass is specified by its `forward` member function.
- A `Module` can contain `Tensors` as parameters.
  - A `LinearModule` contains a `weight` and a `bias` parameter.
  - Whose `forward` function generates the output by multiplying the input with the `weight` and adding the `bias`.
- An application composes its own `Module` by stitching together `Modules` (e.g., linear, convolution) and `Functions` (e.g., relu, pool) in a `forward` function.

PyTorch (2/2)

```
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

Data Parallelism in PyTorch (1/4)

- PyTorch provides distributed data parallel as an `nn.Module` class.

Data Parallelism in PyTorch (1/4)

- PyTorch provides distributed data parallel as an `nn.Module` class.
- All replicas start from the same initial values for model parameters.

Data Parallelism in PyTorch (1/4)

- PyTorch provides distributed data parallel as an `nn.Module` class.
- All replicas start from the same initial values for model parameters.
- They synchronize gradients to keep parameters consistent across training iterations.

Data Parallelism in PyTorch (2/4)

- PyTorch offers several tools to facilitate distributed training.

Data Parallelism in PyTorch (2/4)

- PyTorch offers several tools to facilitate distributed training.
- `DataParallel` for data parallel training on the same machine.

Data Parallelism in PyTorch (2/4)

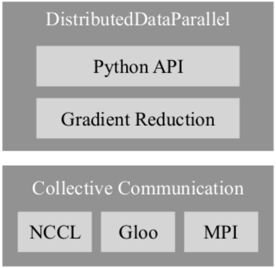
- PyTorch offers several tools to facilitate distributed training.
- `DataParallel` for data parallel training on the same machine.
- `DistributedDataParallel` (DDP) for data parallel training across GPUs and machines.

Data Parallelism in PyTorch (2/4)

- PyTorch offers several tools to facilitate distributed training.
- `DataParallel` for data parallel training on the same machine.
- `DistributedDataParallel` (DDP) for data parallel training across GPUs and machines.
- `RPC` for general distributed model parallel training.

Data Parallelism in PyTorch (3/4)

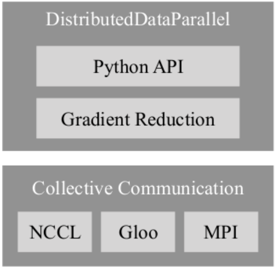
- DDP module enables data parallel training across multiple processes and machines.



[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Data Parallelism in PyTorch (3/4)

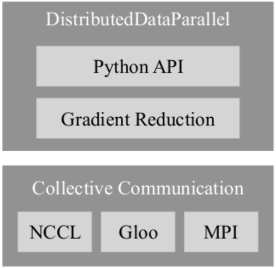
- `DDP` module enables data parallel training across multiple processes and machines.
- `AllReduce` is the primitive communication API used by `DDP`.



[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Data Parallelism in PyTorch (3/4)

- DDP module enables data parallel training across multiple processes and machines.
- AllReduce is the primitive communication API used by DDP.
- It is supported by multiple communication libraries, including NCCL, Gloo, and MPI.



[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Data Parallelism in PyTorch (4/4)

```python
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

Data Parallelism in PyTorch (4/4)

```python
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

```python
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
net = par.DistributedDataParallel(net)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

Gradient Reduction - Naive Solution (1/3)

- DDP guarantees correctness by letting all training processes:

Gradient Reduction - Naive Solution (1/3)

- DDP guarantees correctness by letting all training processes:
    1. Start from the same model state.

Gradient Reduction - Naive Solution (1/3)

- DDP guarantees correctness by letting all training processes:
  1. Start from the same model state.
  2. Consume the same gradients in every iteration.

Gradient Reduction - Naive Solution (1/3)

- DDP guarantees correctness by letting all training processes:
  1. Start from the same model state.
  2. Consume the same gradients in every iteration.

- Step 1 can be achieved by broadcasting model states from one process to all others.

Gradient Reduction - Naive Solution (1/3)

- DDP guarantees correctness by letting all training processes:
  1. Start from the same model state.
  2. Consume the same gradients in every iteration.

- Step 1 can be achieved by broadcasting model states from one process to all others.

- Step 2 can be achieved by inserting a gradient synchronization phase after the local backward pass and before updating local parameters.

Gradient Reduction - Naive Solution (2/3)

- To implement the step 2, the PyTorch accepts custom backward hooks.

Gradient Reduction - Naive Solution (2/3)

- To implement the step 2, the PyTorch accepts custom backward hooks.
- `DDP` can register autograd hooks to trigger computation after every backward pass.

Gradient Reduction - Naive Solution (2/3)

- To implement the step 2, the PyTorch accepts custom backward hooks.
- `DDP` can register autograd hooks to trigger computation after every backward pass.
- When fired, each hook scans through all local model parameters, and retrieves the gradient tensor from each parameter.

Gradient Reduction - Naive Solution (2/3)

- To implement the step 2, the PyTorch accepts custom backward hooks.
- DDP can register autograd hooks to trigger computation after every backward pass.
- When fired, each hook scans through all local model parameters, and retrieves the gradient tensor from each parameter.
- Then, it uses the `AllReduce` collective communication call to calculate the average gradients on each parameter across all processes, and writes the result back to the gradient tensor.

Gradient Reduction - Naive Solution (3/3)

- Two performance concerns:

Gradient Reduction - Naive Solution (3/3)

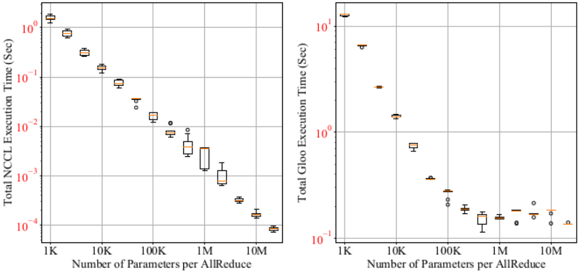
- Two performance concerns:
- 1. Collective communication performs poorly on small tensors, which will be especially prominent on large models with massive numbers of small parameters.

Gradient Reduction - Naive Solution (3/3)

- Two performance concerns:
- 1. Collective communication performs poorly on small tensors, which will be especially prominent on large models with massive numbers of small parameters.
- 2. Separating gradient computation and synchronization forfeits the opportunity to overlap computation with communication due to the hard boundary in between.

Gradient Reduction - Gradient Bucketing (1/2)

- Collective communications are more efficient on large tensors.



(a) NCCL (b) GLOO

[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Gradient Reduction - Gradient Bucketing (2/2)

- Not to launch `AllReduce` immediately after each gradient tensor becomes available.

Gradient Reduction - Gradient Bucketing (2/2)

- Not to launch `AllReduce` immediately after each gradient tensor becomes available.
- Instead, waits for a short period and buckets multiple gradients into one `AllReduce` operation.

Gradient Reduction - Gradient Bucketing (2/2)

- Not to launch `AllReduce` immediately after each gradient tensor becomes available.
- Instead, waits for a short period and buckets multiple gradients into one `AllReduce` operation.
- But not to communicate all gradients in one single `AllReduce`, otherwise, no communication can start before the computation is over.

Gradient Reduction - Gradient Bucketing (2/2)

- Not to launch `AllReduce` immediately after each gradient tensor becomes available.
- Instead, waits for a short period and buckets multiple gradients into one `AllReduce` operation.
- But not to communicate all gradients in one single `AllReduce`, otherwise, no communication can start before the computation is over.
- With relatively small bucket sizes, DDP can launch `AllReduce` operations concurrently with the backward pass to overlap communication with computation.

Overlap Computation with Communication (1/2)

- `AllReduce` on gradients can start before the local backward pass finishes.

Overlap Computation with Communication (1/2)

- `AllReduce` on gradients can start before the local backward pass finishes.
- With bucketing, `DDP` needs to wait for all contents in the same bucket before launching communications.

Overlap Computation with Communication (1/2)

- `AllReduce` on gradients can start before the local backward pass finishes.
- With bucketing, DDP needs to wait for all contents in the same bucket before launching communications.
- DDP registers one autograd hook for each gradient accumulator.

Overlap Computation with Communication (1/2)

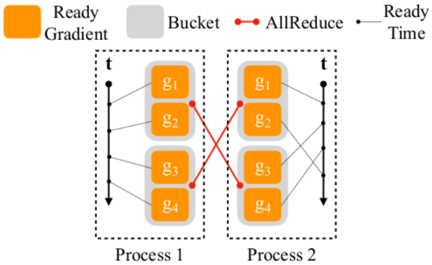
- `AllReduce` on gradients can start before the local backward pass finishes.
- With bucketing, DDP needs to wait for all contents in the same bucket before launching communications.
- DDP registers one autograd hook for each gradient accumulator.
- The hook fires after its corresponding accumulator updating the gradients.

Overlap Computation with Communication (1/2)

- `AllReduce` on gradients can start before the local backward pass finishes.
- With bucketing, DDP needs to wait for all contents in the same bucket before launching communications.
- DDP registers one autograd hook for each gradient accumulator.
- The hook fires after its corresponding accumulator updating the gradients.
- If hooks of all gradients in the same buckets have fired, then `AllReduce` on that bucket will be triggered.

Overlap Computation with Communication (2/2)

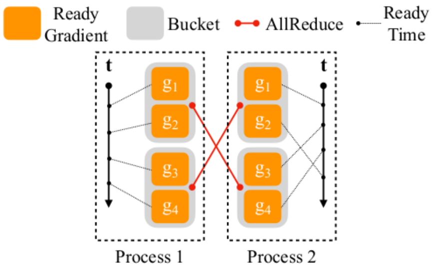
- The reducing order must be the same across all processes, otherwise, `AllReduce` contents might mismatch.



[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Overlap Computation with Communication (2/2)

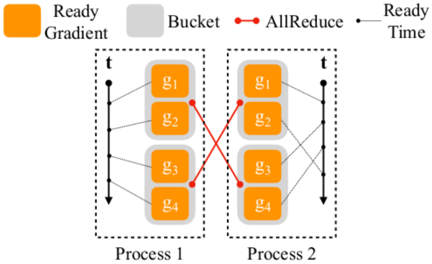
- The reducing order must be the same across all processes, otherwise, `AllReduce` contents might mismatch.
- All processes must use the same bucketing order



[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Overlap Computation with Communication (2/2)

- The reducing order must be the same across all processes, otherwise, `AllReduce` contents might mismatch.
- All processes must use the same bucketing order
- No process can launch `AllReduce` on bucket $\mathtt{i}+1$ before embarking bucket `i`.



[Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020]

Gradient Accumulation

- Reduce gradient synchronization frequencies to speed up distributed data parallel training.

Gradient Accumulation

- Reduce gradient synchronization frequencies to speed up distributed data parallel training.
- Instead of launching `AllReduce` in every iteration, it can conduct `n` local training iterations before synchronizing gradients globally.

Gradient Accumulation

- Reduce gradient synchronization frequencies to speed up distributed data parallel training.
- Instead of launching `AllReduce` in every iteration, it can conduct `n` local training iterations before synchronizing gradients globally.
- Helpful if the input batch is too large to fit into a device.

Gradient Accumulation

- Reduce gradient synchronization frequencies to speed up distributed data parallel training.
- Instead of launching `AllReduce` in every iteration, it can conduct `n` local training iterations before synchronizing gradients globally.
- Helpful if the input batch is too large to fit into a device.
    - It can split one input batch into multiple micro-batches.

Gradient Accumulation

- Reduce gradient synchronization frequencies to speed up distributed data parallel training.
- Instead of launching `AllReduce` in every iteration, it can conduct `n` local training iterations before synchronizing gradients globally.
- Helpful if the input batch is too large to fit into a device.
  - It can split one input batch into multiple micro-batches.
  - Run local forward and backward passes on every micro-batch.

Gradient Accumulation

- Reduce gradient synchronization frequencies to speed up distributed data parallel training.
- Instead of launching `AllReduce` in every iteration, it can conduct `n` local training iterations before synchronizing gradients globally.
- Helpful if the input batch is too large to fit into a device.
  - It can split one input batch into multiple micro-batches.
  - Run local forward and backward passes on every micro-batch.
  - Only launch gradient synchronization at the boundaries of large batches.

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

ZeRO (1/2)

- Data and model parallelisms exhibit fundamental limitations to fit massive models into limited device memory, while obtaining computation, communication and development efficiency.

ZeRO (1/2)

- Data and model parallelisms exhibit fundamental limitations to fit massive models into limited device memory, while obtaining computation, communication and development efficiency.
- Zero Redundancy Optimizer (ZeRO) eliminates memory redundancies in data-parallel and model-parallel training.

ZeRO (1/2)

- Data and model parallelisms exhibit fundamental limitations to fit massive models into limited device memory, while obtaining computation, communication and development efficiency.
- Zero Redundancy Optimizer (ZeRO) eliminates memory redundancies in data-parallel and model-parallel training.
- It retains low communication volume and high computational granularity.

ZeRO (1/2)

- Data and model parallelisms exhibit fundamental limitations to fit massive models into limited device memory, while obtaining computation, communication and development efficiency.
- Zero Redundancy Optimizer (ZeRO) eliminates memory redundancies in data-parallel and model-parallel training.
- It retains low communication volume and high computational granularity.
- Therefore, it allows to scale the model size proportional to the number of devices.

Where Did All the Memory Go?

- Model states
  - Optimizer states
  - Gradients
  - Parameters
- Residual memory consumption
  - Activations
  - Temporary buffers
  - Memory fragmentation

ZeRO (2/2)

- ZeRO has two sets of optimizations:

ZeRO (2/2)

- ZeRO has two sets of optimizations:
- ZeRO-DP (ZeRO Data Parallelism): aimes at reducing the memory footprint of the model states.

ZeRO (2/2)

- ZeRO has two sets of optimizations:
- ZeRO-DP (ZeRO Data Parallelism): aimes at reducing the memory footprint of the model states.
- ZeRO-R (ZeRO Residual): targetes towards reducing the residual memory consumption.

ZeRO-DP

Optimizing Model State Memory

- Model states often consume the largest amount of memory during training.

Optimizing Model State Memory

- Model states often consume the largest amount of memory during training.
  - Data-Parallel and Model-Parallel approaches do not offer satisfying solution.

Optimizing Model State Memory

- Model states often consume the largest amount of memory during training.
  - Data-Parallel and Model-Parallel approaches do not offer satisfying solution.
- Data-Parallel (DP) has good compute/communication efficiency, but poor memory efficiency.

Optimizing Model State Memory

- Model states often consume the largest amount of memory during training.
  - Data-Parallel and Model-Parallel approaches do not offer satisfying solution.
- Data-Parallel (DP) has good compute/communication efficiency, but poor memory efficiency.
- Model-Parallel (MP) can have poor compute/communication efficiency, but good memory efficiency.

Optimizing Model State Memory

- Model states often consume the largest amount of memory during training.
  - Data-Parallel and Model-Parallel approaches do not offer satisfying solution.
- Data-Parallel (DP) has good compute/communication efficiency, but poor memory efficiency.
- Model-Parallel (MP) can have poor compute/communication efficiency, but good memory efficiency.
- Both approaches maintain all the model states required over the entire training process statically, even though not all model states are required all the time during the training.

Optimization Phases of ZeRO-DP

- Optimizer state partitioning P_{os}
- Gradient partitioning P_g
- Parameter partitioning P_p

Optimizer State Partitioning P_{os}

- N_d: number of data parallel processes

Optimizer State Partitioning P_{os}

- N_d: number of data parallel processes
- Group the optimizer states into N_d equal partitions ($\frac{1}{N_d}$) on each data parallel process.

Optimizer State Partitioning P_{os}

- N_d: number of data parallel processes
- Group the optimizer states into N_d equal partitions ($\frac{1}{N_d}$) on each data parallel process.
- Each data parallel process only updates the its corresponding optimizer states.

Optimizer State Partitioning P_{os}

- N_d: number of data parallel processes
- Group the optimizer states into N_d equal partitions ($\frac{1}{N_d}$) on each data parallel process.
- Each data parallel process only updates the its corresponding optimizer states.
- Performs an all-gather across the data parallel process at the end of each training step to get the fully updated parameters across all data parallel process.

Gradient Partitioning P_g

- Each data parallel process only needs the reduced gradients for the corresponding parameters.

Gradient Partitioning P_g

- Each data parallel process only needs the reduced gradients for the corresponding parameters.
- As each gradient of each layer becomes available during the backward propagation, only the data parallel process responsible for updating the corresponding parameters will reduce them.

Gradient Partitioning P_g

- Each data parallel process only needs the reduced gradients for the corresponding parameters.
- As each gradient of each layer becomes available during the backward propagation, only the data parallel process responsible for updating the corresponding parameters will reduce them.
- This is a Reduce-Scatter operation, where gradients corresponding to different parameters are reduced to different process.

Gradient Partitioning P_g

- Each data parallel process only needs the reduced gradients for the corresponding parameters.
- As each gradient of each layer becomes available during the backward propagation, only the data parallel process responsible for updating the corresponding parameters will reduce them.
- This is a Reduce-Scatter operation, where gradients corresponding to different parameters are reduced to different process.
- After the reduction, the gradients are no longer needed and their memory can be released.

Parameter Partitioning P_p

- Each process only stores the parameters corresponding to its partition.

Parameter Partitioning $\mathrm{P_p}$

- Each process only stores the parameters corresponding to its partition.
- When the parameters outside of its partition are required for forward and backward propagation, they are received from the appropriate data parallel process through broadcast.

Parameter Partitioning P_p

- Each process only stores the parameters corresponding to its partition.
- When the parameters outside of its partition are required for forward and backward propagation, they are received from the appropriate data parallel process through broadcast.
- This approach increases the total communication volume of a baseline DP system to 1.5x, while enabling memory reduction proportional to N_d.

ZeRO-R

Optimizing Residual State Memory

- ZeRO-DP boosts memory efficiency for model states.

Optimizing Residual State Memory

- ZeRO-DP boosts memory efficiency for model states.
- The rest of the memory consumed by activations, temporary buffers, and unusable memory fragments.

Optimizing Residual State Memory

- ZeRO-DP boosts memory efficiency for model states.
- The rest of the memory consumed by activations, temporary buffers, and unusable memory fragments.
- ZeRO-R optimizes the residual memory consumed by the following three factors:

Optimizing Residual State Memory

- ZeRO-DP boosts memory efficiency for model states.
- The rest of the memory consumed by activations, temporary buffers, and unusable memory fragments.
- ZeRO-R optimizes the residual memory consumed by the following three factors:
    1. Optimizes activation memory (stored from forward pass in order to perform backward pass) by activation partitioning. It also offloads activations to CPU when appropriate.

Optimizing Residual State Memory

- ZeRO-DP boosts memory efficiency for model states.
- The rest of the memory consumed by activations, temporary buffers, and unusable memory fragments.
- ZeRO-R optimizes the residual memory consumed by the following three factors:
  1. Optimizes activation memory (stored from forward pass in order to perform backward pass) by activation partitioning. It also offloads activations to CPU when appropriate.
  2. Defines appropriate size for temporary buffers to strike for a balance of memory and computation efficiency.

Optimizing Residual State Memory

- ZeRO-DP boosts memory efficiency for model states.
- The rest of the memory consumed by activations, temporary buffers, and unusable memory fragments.
- ZeRO-R optimizes the residual memory consumed by the following three factors:
  1. Optimizes activation memory (stored from forward pass in order to perform backward pass) by activation partitioning. It also offloads activations to CPU when appropriate.
  2. Defines appropriate size for temporary buffers to strike for a balance of memory and computation efficiency.
  3. Proactively manages memory based on the different lifetime of tensors, preventing memory fragmentation.

Optimization Phases of ZeRO-R

- Partitioned activation checkpointing $\mathtt{P_a}$
- Constant size buffers $\mathtt{C_B}$
- Memory defragmentation $\mathtt{M_D}$

Partitioned Activation Checkpointing P_a

- ZeRO partitions the activations.

Partitioned Activation Checkpointing P_a

- ZeRO partitions the activations.
- Once the forward propagation for a layer of a model is computed, the activations are partitioned across all the model parallel process, until it is needed again during the backprogation.

Partitioned Activation Checkpointing P_a

- ZeRO partitions the activations.
- Once the forward propagation for a layer of a model is computed, the activations are partitioned across all the model parallel process, until it is needed again during the backprogation.
- At this point, ZeRO uses an all-gather operation to re-materialize a replicated copy of the activations.

Partitioned Activation Checkpointing P_a

- ZeRO partitions the activations.
- Once the forward propagation for a layer of a model is computed, the activations are partitioned across all the model parallel process, until it is needed again during the backprogation.
- At this point, ZeRO uses an all-gather operation to re-materialize a replicated copy of the activations.
- It works in conjunction with activation checkpointing, storing partitioned activation checkpoints only instead of replicated copies.

Constant Size Buffers C_B

- ZeRO selects the sizes of the temporal-data buffers to balance memory and compute efficiency.

Constant Size Buffers C_B

- ZeRO selects the sizes of the temporal-data buffers to balance memory and compute efficiency.
- During training, the computational efficiency of some operations can be highly dependent on the input size, with larger inputs achieving higher efficiency.

Constant Size Buffers C_B

- ZeRO selects the sizes of the temporal-data buffers to balance memory and compute efficiency.
- During training, the computational efficiency of some operations can be highly dependent on the input size, with larger inputs achieving higher efficiency.
- To get better efficiency, it fuses all the parameters into a single buffer before applying these operations.
- The memory overhead of the fused buffers is proportional to the model size, and can become inhibiting.

Constant Size Buffers C_B

- ZeRO selects the sizes of the temporal-data buffers to balance memory and compute efficiency.
- During training, the computational efficiency of some operations can be highly dependent on the input size, with larger inputs achieving higher efficiency.
- To get better efficiency, it fuses all the parameters into a single buffer before applying these operations.
- The memory overhead of the fused buffers is proportional to the model size, and can become inhibiting.
- To address this issue, ZeRO-R uses a constant-size fused buffer when the model becomes too large.

Memory Defragmentation $\mathtt{M_D}$ (1/2)

- Memory fragmentation in model training occurs as a result of activation checkpointing and gradient computation.

Memory Defragmentation $\mathtt{M_D}$ (1/2)

- Memory fragmentation in model training occurs as a result of activation checkpointing and gradient computation.
- During the forward propagation with activation checkpointing, only selected activations are stored for back propagation.

Memory Defragmentation $\mathtt{M_D}$ (1/2)

- Memory fragmentation in model training occurs as a result of activation checkpointing and gradient computation.
- During the forward propagation with activation checkpointing, only selected activations are stored for back propagation.
  - Most activations are discarded as they can be recomputed again during the back propagation.

Memory Defragmentation $\mathtt{M_D}$ (1/2)

- Memory fragmentation in model training occurs as a result of activation checkpointing and gradient computation.
- During the forward propagation with activation checkpointing, only selected activations are stored for back propagation.
  - Most activations are discarded as they can be recomputed again during the back propagation.
  - Short lived memory (discarded activations) and long lived memory (checkpointed activation).

Memory Defragmentation $\mathtt{M_D}$ (2/2)

- During the backward propagation, the parameter gradients are long lived, while activation gradients and any other buffers required to compute the parameter gradients are short lived.

Memory Defragmentation $\mathtt{M_D}$ (2/2)

- During the backward propagation, the parameter gradients are long lived, while activation gradients and any other buffers required to compute the parameter gradients are short lived.
- This interleaving of short term and long term memory causes memory fragmentation.

Memory Defragmentation $\mathtt{M_D}$ (2/2)

- During the backward propagation, the parameter gradients are long lived, while activation gradients and any other buffers required to compute the parameter gradients are short lived.
- This interleaving of short term and long term memory causes memory fragmentation.
- ZeRO does memory defragmentation on-the-fly by pre-allocating contiguous memory chunks for activation checkpoints and gradients. produced.

Summary

Summary

- BigDL
- PyTorch Distributed
- ZeRO

Reference

- Hasheminezhad et al., Towards a Scalable and Distributed Infrastructure for Deep Learning Applications, 2020
- Dai et al., BigDL: A Distributed Deep Learning Framework for Big Data, 2019
- Li et al., PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020
- Rajbhandari et al., ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, 2020

Questions?