Chapter 2

Parameter Server and All-Reduce

SK telecom

개요

1. 이장의 주요 목표

- Parameter server와 All-Reduce라는 두 가지 주요 데이터 병렬 학습 패러다임을 비교 분석
- 두가지 학습 패러다임에 대한 데이터 병렬(data parallel) 학습 파이프라인 설계

2. 주요 논의 내용

- Parameter Server 패러다임의 시스템 아키텍처를 설명
- PyTorch를 사용하여 Parameter Server 아키텍처를 구현하는 방법
- Parameter Server 의 단점과 All-Reduce로 대체하는 이유
- All-Reduce 아키텍처 와 Collective Communication 설명

3. 참고

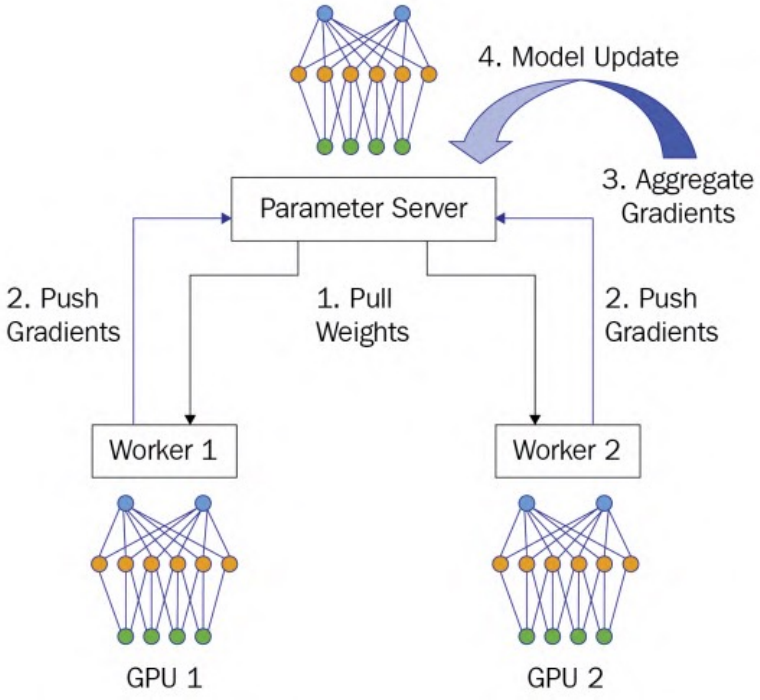
- 분산 학습 정리 자료
  - https://confluence.tde.sktelecom.com/display/GLMMODEL/5.+Distributed+Training

Parameter Server

1. Parameter Server Architecture

- Parameter Server서버와 workers 두 가지 역할로 구성
  - 전통적인 Master / Worker 아키텍처에서의 마스터 노드역활
  - worker는 모델 학습을 담당하는 컴퓨터 노드나 GPU 노드들

2. 동작 단계



1. Pull Weights
   - 모든 worker는 중앙의 Parameter Server 서버에서 모델 Parameter/weights를 가져옴

2. Push Gradients
   - 각 worker는 로컬에 할당된 학습 데이터 파티션으로 local 모델을 학습하고 local gradients를 구하고, 이후 모든 workers는 local gradients를 parameter server로 전송

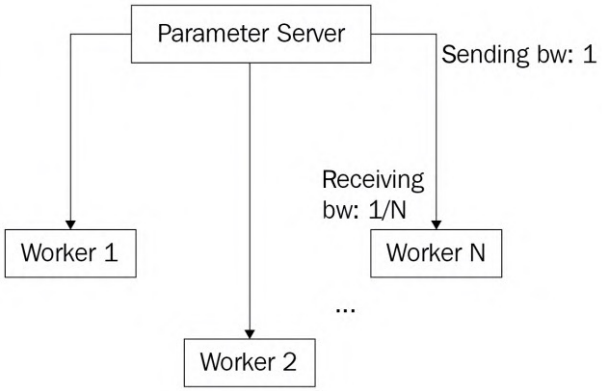
3. Aggregate Gradients
   - worker 에서 보내진 모든 gradients를 수집한 후, Parameter Server는 모든 gradients를 합산

4. Model Update
   - 집계된 gradients가 계산되면, parameter 서버의 모델 parameter를 업데이트

Communication bottleneck in the parameter server architecture

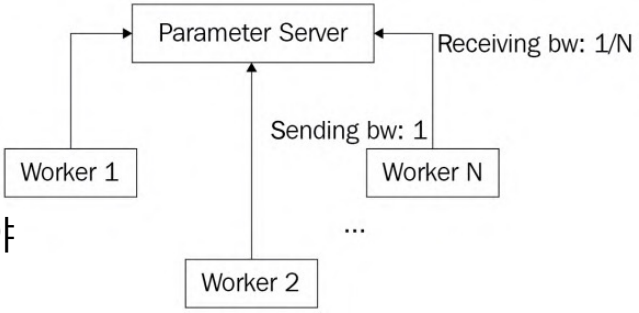
1. Pull Weights

- Fan-out 통신 패턴
- Parameter 서버가 model weights를 동시에 모든 worker 노드로 전송해야 하는 일대다 통신
- Communication bottleneck
  - Parameter server는 N개의 worker에게 모델을 동시에 전송해야 하기 때문에 각 worker에 대한 전송 대역폭(BW)은 1/N 반면에 각 worker의 수신 대역폭은 1
  - weights를 가져오는 단계에서는 Parameter Server 측에서 통신 병목 현상이 발생



2. Push Gradients

- Fan-in 통신 패턴
- 모든 worker 노드가 Parameter 서버로 전송해야 하는 다대일 통신
- Communication bottleneck
  - N개의 worker들은 각각 1의 전송 대역폭을 갖음
  - Parameter Server들은 모든 worker들로 부터 gradients를 받아야 하므로 각 worker에 대한 수신 대역폭은 1/N
  - Push Gradients 에서도 여전히 Prameter server가 병목

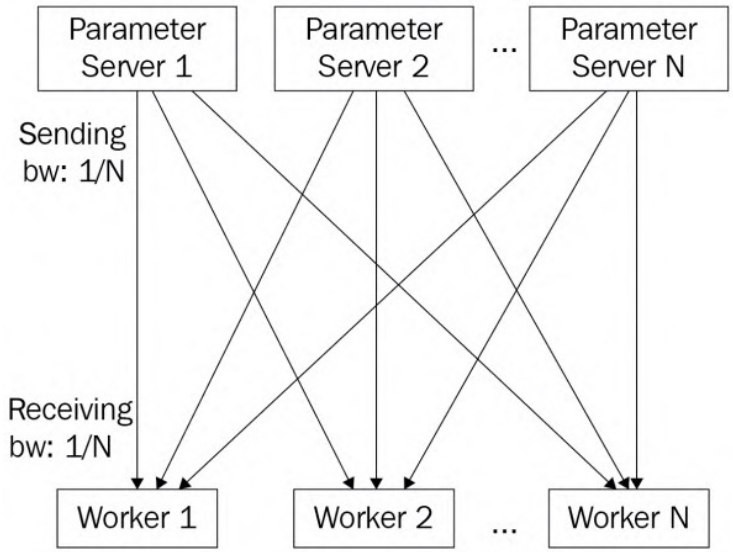


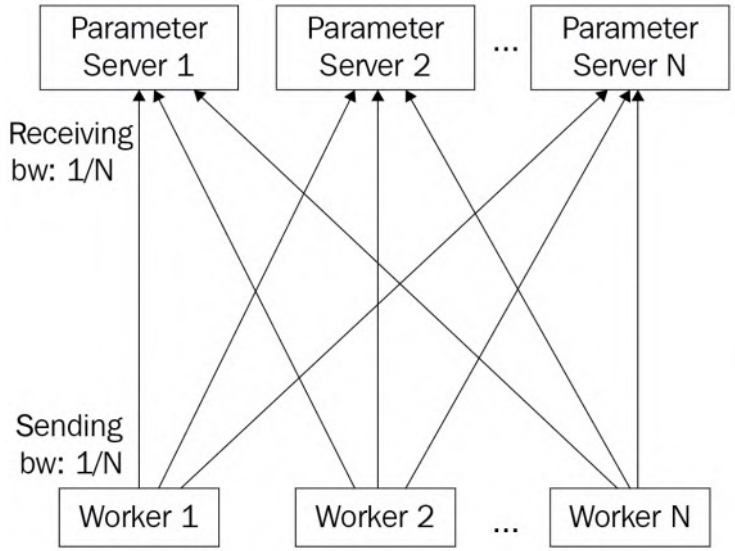
Sharding the model among parameter servers

1. **Pull Weights**
   - N개의 Parameter 서버로로 분할
   - Communication bottleneck
     - 각 worker는 대해 1/N의 대역폭으로 parameter 서버로부터 데이터 수신
     - 결국 Prameter 서버의 수신 데이터 총량은 1/N * N

2. **Push Gradients**
   - N개의 Parameter 서버로로 분할
   - Communication bottleneck
     - 각 worker는 대해 1/N의 대역폭으로 N개의 parameter 서버로 Gradients 전송

3. **Parameter Server Sharding**
   - Parameter server와 worker수의 동일할 필요는 없음
   - Parameter 서버의 분할은 적절히 조정되어야 함





Parameter Server 구현

1. Parameter Server

- get_weights
  - 모델의 weight를 반환하는 메서드
  - model.state_dict()를 호출하여 모델의 parameter를 저장하는 state dictionary을 반환
- update_model
  - parameter로 graidents를 받아와 모델 레이어에 각 parameter에 해당 graidents를 할당한 후 optimizer를 사용하여 모델을 업데이트
  - optimizer.step()을 호출하여 weights를 업데이트 한 후 optimizer.zero_grad()를 호출해 gradients 초기화

```
class ParameterServer(nn.Module):
        def __init__(self):
                super().__init__()
                self.model = MyNet()

                if torch.cuda.is_available():
                        self.input_device = torch.device("cuda:0")
                else:
                        self.input_device = torch.device("cpu")

                self.optimizer = optim.SGD(self.model.parameters(), lr = 0.05)

        def get_weights(self):
                return self.model.state_dict()

        def update_model(self, grads):
                for para, grad in zip(self.model.parameters(), grads):
                        para.grad = grad
                self.optimizer.step()
                self.optimizer.zero_grad()
```

Parameter Server 구현

2. Worker

- pull_weights
  - 모델의 weights를 받아와서 로드하는 메서드
  - load_state_dict() 메서드를 사용하여 Worker의 모델에 weights를 로드
- push_gradients
  - gradients를 계산하고 반환하는 메서드
  - 학습 데이터를 통해 graidnet 계산으
  - 모델에 학습 데이터를 입력하여 출력을 얻은 후 loss 계산,
  - Loss에 대한 back prop을 수행해서 gradient 계산하고 각 레이어의 gradients를 grads 리스트에 추가
  - 현재 배치 번호와 loss을 출력하고, gradient list를 반환

```
class Worker(nn.Module):
        def __init__(self):
                super().__init__()
                self.model = MyNet()
                if torch.cuda.is_available():
                        self.input_device = torch.device("cuda:0")
                else:
                        self.input_device = torch.device("cpu")

        def pull_weights(self, model_params):
                self.model.load_state_dict(model_params)

        def push_gradients(self, batch_idx, data, target):
                data, target = data.to(self.input_device), target.to(self.input_device)
                output = self.model(data)
                data.requires_grad = True
                loss = F.nll_loss(output, target)
                loss.backward()
                grads = []
                for layer in self.parameters():
                        grad = layer.grad
                        grads.append(grad)
                print(f"batch {batch_idx} training :: loss {loss.item()}")
                return grads
```

Parameter Server 구현

3. main

- main
  - Parameter server와 worker간의 연결을 정의하는 함수
- 학습단계
  - 먼저 ps, worker를 모두 초기화
  - Ps 에서 최선 weight을 가져옴
  - Worker가 weights를 가져오고 로컬 모델 parameter를 업데이트
  - Worker에서 gradient 계산 후 ps 로 push
  - Ps가 gradients를 수신후 해당 gradients로 모델의 weight 업데이트

```
def main():
        ps = ParameterServer()
        worker = Worker()

        for batch_idx, (data, target) in enumerate(train_loader):
                params = ps.get_weights()
                worker.pull_weights(params)
                grads = worker.push_gradients(batch_idx, data, target)
                ps.update_model(grads)
        print("Done Training")
```

Issues with the parameter server

1. PS Architeture의 문제

- Parameter Server
  - 학습을 수행하지 않으며, 학습 대역폭이 0 (학습에 참여 안함)
  - Prarameter Server가 더 많을수록 통신 대역폭이 높아지고, 모델 동기화 지연 시간이 감소
- Worker
  - Worker가 더 많을수록 학습 대역폭이 높아짐 (학습 시간이 빨라짐)
  - Worker 더 많을수록 데이터 전송이 많아지고, 모델 동기화 오버헤드가 증가
- 학습 처리량과 통신 대기시간의 균형을 맞춰야 함

- $Model\_sync\_latency = Amount\_of\_data / Total\_communication\_bw$

2. Case1 더 많은 parameter server

- PS가 더 많다는 것은 통신 지연 시간이 더 낮다는 의미
- Workder가 적을수록 학습 처리량이 낮아짐

3. Case2 더 많은 worker

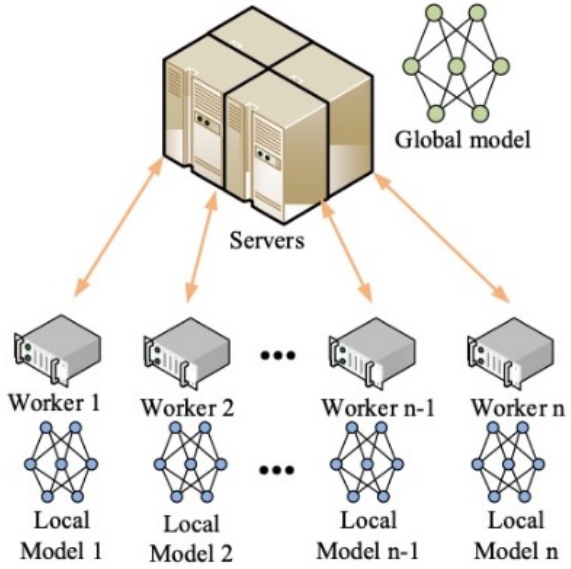
- 동기화해야 할 데이터가 더 많아지기 때문에 통신할 데이터가 많아짐.
- 더 많은 worker는 통신 지연 시간이 더 높다는 것을 의미
- 더 많은 worker는 더 높은 학습 처리량

PS vs All-Reduce

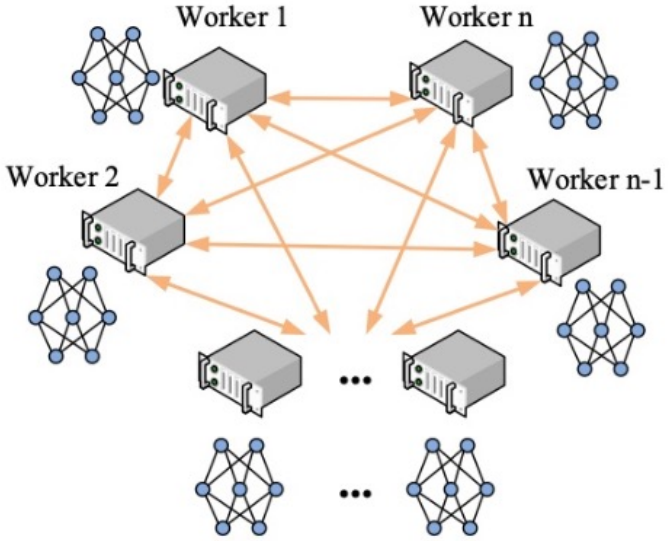
1. PS Architecture를 개선하기 위한 All-Reduce Architecture

- Collective communication
  - 집합 통신
  - Parameter 공유를 위해 모든 노드들이 참여 하는 방법

PS Architecture



All-Reduce Architecture



Collective Communication

1. Collective Communication 이란?

- 여러 개의 컴퓨팅 장치 또는 노드 간에 데이터를 교환하고 동기화 하는 방법

	Point to Point	Collective Communication
구분	• 두개 통신 노드간에 데이터를 직접 교환	• 여러대의 통신 노드간에 데이터를 교환하고 동기화
통신 유형	• 두 노드간의 통신은 일대일(point to point)통신으로 이루어지며 하나의 송신자와 하나의 수신자가 직접 통신을 수행	• 다수의 노드 간에 통신이 이루어짐, 모든 노드가 참여하여 데이터를 교환하고 동기화하는 방식
통신 패턴	• 주로 송신자가 데이터를 보내고 수신자가 데이터를 받는 단방향 통신 패턴, 데이터는 한 번에 하나의 노드로 전송	• 다양한 통신 패턴 • Reduce, All-reduce, Broadcast, Scatter, Gather, All-gather 등 다중 노드 간의 데이터 교환 및 동기화를 위한 패턴이 사용됨
데이터전송	• 데이터를 송신하고 수신하기 위해 송신자와 수신자 간에 명시적인 통신을 설정	• 라이브러리 또는 프레임워크에서 제공하는 인터페이스를 통해 연산을 호출하여 데이터 교환 및 동기화가 이루어짐 (MPI, Gloo, NCCL)
확장성	• 작은 규모의 시스템에서 효율적이고 간단한 통신을 제공 • 그러나 시스템의 크기가 커지면 통신 노드 간의 관리 및 통신 패턴의 복잡성이 증가	• 다수의 노드 간에 데이터 교환 및 동기화를 처리하는 데 효과적 • 시스템의 규모가 커져도 통신 패턴은 대부분 노드 간의 관계에 따라 구성되므로 확장성이 좋음

Multi processing with Pytorch

Collective Communication 동작 방식을 알기 위해서는

1) 먼저 multi Process를 생성한는 방법과
2) 생성된 Process들이 통신할 수 있도록 초기화 방법
3) 그리고 Process간에 Collective Communication을 지원하는 libarary 사용법

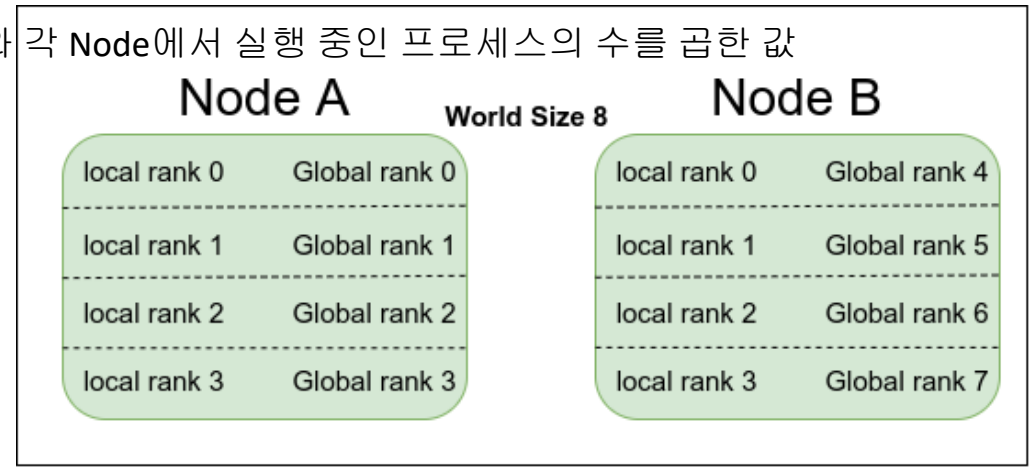
익혀야 한다.

Multi processing with Pytorch

1. 용어 정의

- Node
  - 분산 학습을 위해 사용되는 물리적 또는 가상의 컴퓨터를 나타냄

    각 Node는 여러 개의 GPU 또는 CPU를 가질 수 있으며, 독립적으로 작동할 수 있는 컴퓨팅 자원을 제공
- Global Rank
  - 각 프로세스의 전역적인 식별자
  - 각 프로세스는 고유한 Global Rank를 가지며, 일반적으로 0부터 랭크
- Local Rank
  - 각 Node 내에서의 프로세스의 로컬 식별자
  - 한 Node에는 Local Rank는 Node 내에서의 상대적인 위치를 나타내며, 동일한 Node 내의 프로세스 간에 통신 및 동기화에 사용
- World Size
  - World Size는 분산 학습 시스템에서 전체 프로세스 수
  - 이는 Node의 수와 각 Node에서 실행 중인 프로세스의 수를 곱한 값



Multi process를 생성 하는 방법

1. **사용자의 코드가 메인 프로세스가 되어 특정 함수를 subprocess로 분기**
   - spawn을 사용한 멀티 프로세스 실행
   - spawn은 Python의 multiprocessing 모듈을 사용하여 멀티 프로세스 실행하는 방법
   - PyTorch에서는 torch.multiprocessing을 통해 spawn을 지원
   - spawn을 사용하면 각 프로세스는 독립적으로 시작되며, 독립적인 메모리 공간을 갖음
   - 데이터 및 모델 파라미터를 공유하기 위해 프로세스 간 통신 기능은 torch.distributed 패키지를 사용

2. **torch.distributed.launch를 사용한 멀티 프로세스 실행**
   - PyTorch의 분산 학습을 위한 확장 라이브러리인 torch.distributed.launch에서 지원하는 기능으로 , 멀티 노드 및 멀티 프로세스 작업을 관리하는 데 사용
   - Pytorch의 launcher가 메인 프로세스가 되어 사용자 코드를 서브 프로세스로 분기
   - 대규모 분산 학습 작업 또는 여러 노드에서 실행되는 작업에 적합

multiprocessing.spawn

1. **사용자의 코드가 메인 프로세스가 되어 특정 함수를 subprocess로 분기**
   - torch.multiprocessing.spawn(fn, args=(), nprocs=1, join=True, daemon=False, start_method='spawn')
     - fn: 실행할 함수
     - args: fn에 전달할 추가적인 인자들을 튜플 형태로 지정
     - nprocs: 생성할 프로세스의 수를 지정
     - join: 프로세스 실행이 완료될 때까지 메인 프로세스가 대기할지 여부를 결정
     - daemon: 생성된 프로세스를 데몬 프로세스로 설정할지 여부
     - start_method: 프로세스 생성 방식을 지정 ('spawn', 'fork')

```python
import torch.multiprocessing as mp

# subprocess에서 실행될 함수 정의
def worker(rank, world_size):
    print(f" {world_size} rank: {rank}")

# main process 정의
if __name__ == '__main__':
    world_size = 4 # 프로세스 수
    mp.spawn(worker, args=(world_size,), nprocs=world_size)
```

```
$ python test.py
 4 rank: 3
 4 rank: 2
 4 rank: 1
 4 rank: 0
```

distriubtd.launch

2. torch.distributed.launch를 사용한 멀티 프로세스 실행

- Single node
  - python -m torch.distributed.launch --nproc-per-node=NUM_GPUS_YOU_HAVE YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3 and all other arguments of your training script)
- Multi nodes
  - node1
  - python -m torch.distributed.launch --nproc-per-node=NUM_GPUS_YOU_HAVE --nnodes=2 --node-rank=0 --master-addr="192.168.1.1" --master-port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3 and all other arguments of your training script)
  - Node2
  - python -m torch.distributed.launch --nproc-per-node=NUM_GPUS_YOU_HAVE --nnodes=2 --node-rank=1 --master-addr="192.168.1.1" --master-port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3 and all other arguments of your training script)

```python
import os

print(f" Worldsize : {os.environ['WORLD_SIZE']} rank: {os.environ['RANK']}")
```

```
$ python -m torch.distributed.launch --nproc_per_node=4 multiprocess_launch.py

 Worldsize : 4 rank: 0
 Worldsize : 4 rank: 1
 Worldsize : 4 rank: 2
 Worldsize : 4 rank: 3
```

Process들이 통신할 수 있도록 초기화

1. Initailize Process Group

- Multi process를 생성한 후 process간 통신 방법에 대한 정의가 필요
- torch.distributed.init_process_group
  - torch.distributed.init_process_group(backend=None, init_method=None, timeout=datetime.timedelta(seconds=1800), world_size=- 1, rank=- 1, store=None, group_name='', pg_options=None)
    - Backend : 프로세스 간 통신에 사용할 백엔드를 지정 (nccl, gloo, mpi)
    - init_method: 프로세스 간 통신 초기화를 위해 필요한 정보를 미리 설정
      - MASTER_ADDR, MASTER_PORT 등
      - env://은 환경 변수를 통해, file://'은 파일을 통해 초기화 방법을 설정)

2. Process Group 초기화 과정

1. process들은 backend 인자에 지정된 백엔드를 사용하여 통신을 설정한다. 예를 들어 'nccl' 백엔드를 사용하는 경우, NVIDIA의 NCCL 라이브러리를 사용하여 GPU 간의 통신이 이루어짐.
2. init_method 인자에 지정된 초기화 방법에 따라 process들이 통신을 위한 주소와 포트 번호, 랭크 등의 정보를 교환한다.
3. 각 process는 자신의 Rank와 world size를 할당 받음 Rank는 각 process의 고유 식별자로, 보통 0부터 시작하여 1씩 증가하는 순차적인 고유값이고, world size수는 분산 학습에 참여하는 총 process의 수
4. Init_method에 지정된 Master 노드에 0번 프로세스가 Master process로 지정되어 다른 worker들의 응답을 대기하기 된다.
5. 각 worker들로 부터 응답이 오면 process group이 구성되고 이를 통해 각 process들은 분산 처리를 수행할 수 있게 된다.

Ininitailze Process Group

1. Process 그룹 초기화

- Multi process를 생성한 후 process간 통신 방법에 대한 정의가 필요
- torch.distributed.init_process_group

```python
import os
import torch.distributed as dist
import torch.multiprocessing as mp

# 각 subprocess에서 실행될 내용
def worker(rank, world_size):
      dist.init_process_group(backend='nccl', rank=rank, world_siz
      e=world_size)
      group = dist.distributed_c10d._get_default_group()
      # 원하는 랭크만 새로운 그룹으로 묶기
      # group = dist.new_group([_ for _ in range(world_size)])
      # group = dist.new_group([0, 1])
      print(f"{group} - rank: {rank}")

# Mainprocess에서 실행될 내용
if __name__ == '__main__':
      world_size = 4
      os.environ["MASTER_ADDR"] = "127.0.0.1"
      os.environ["MASTER_PORT"] = "32900"
      os.environ["WORLD_SIZE"] = str(world_size)

      mp.spawn(
         worker,
         args=(world_size,),
         nprocs=world_size,
         start_method="spawn",
      )
```

```
$ python multiprocess_spawn.py

<torch.distributed.distributed_c10d.ProcessGroup object at 0x7f998ed0bb70> - rank: 0
<torch.distributed.distributed_c10d.ProcessGroup object at 0x7efbd16f5ab0> - rank: 3
<torch.distributed.distributed_c10d.ProcessGroup object at 0x7f47e0e648f0> - rank: 2
<torch.distributed.distributed_c10d.ProcessGroup object at 0x7f3ce8ee19f0> - rank: 1
```

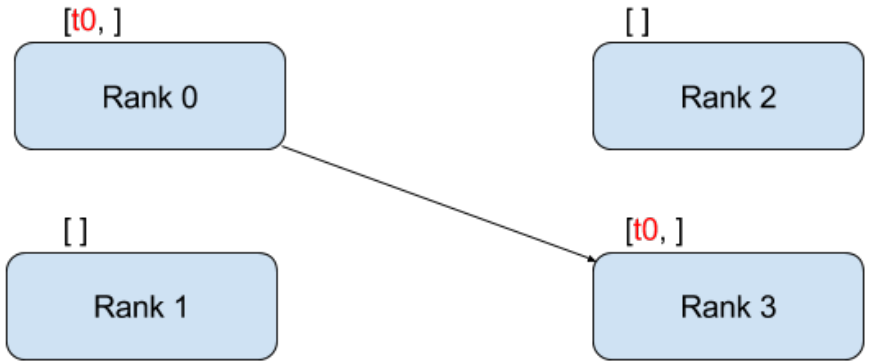
```python
import torch.distributed as dist

# Process group 초기화
dist.init_process_group(backend="nccl")

group = dist.distributed_c10d._get_default_group()
# 새로운 process 그룹 생성
#group = dist.new_group([_ for _ in range(dist.get_world_size())])

print(f"{group} - rank: {dist.get_rank()}\n")
```

```
$ python -m torch.distributed.launch --nproc_per_node=4 multiproce
ss_launch.py

<torch.distributed.distributed_c10d.ProcessGroup object at 0x7f46a
59217f0> - rank: 1
<torch.distributed.distributed_c10d.ProcessGroup object at 0x7f36b
3a2e0b0> - rank: 3
<torch.distributed.distributed_c10d.ProcessGroup object at 0x7fca5
0738bf0> - rank: 0
<torch.distributed.distributed_c10d.ProcessGroup object at 0x7fe3d
bca0130> - rank: 2
```

Distirubted Communication Package (Torch.distributed)

1. Point to point

- 하나의 프로세스에서 다른 프로세스로 데이터를 전송




```python
import torch
import torch.distributed as dist

dist.init_process_group("gloo")
# 현재 nccl은 send, recv를 지원하지 않음

if dist.get_rank() == 0:
tensor = torch.randn(2, 2)
request = dist.isend(tensor, dst=1)
elif dist.get_rank() == 1:
tensor = torch.zeros(2, 2)
request = dist.irecv(tensor, src=0)
else:
raise RuntimeError("wrong rank")

request.wait()

print(f"rank {dist.get_rank()}: {tensor}")
```

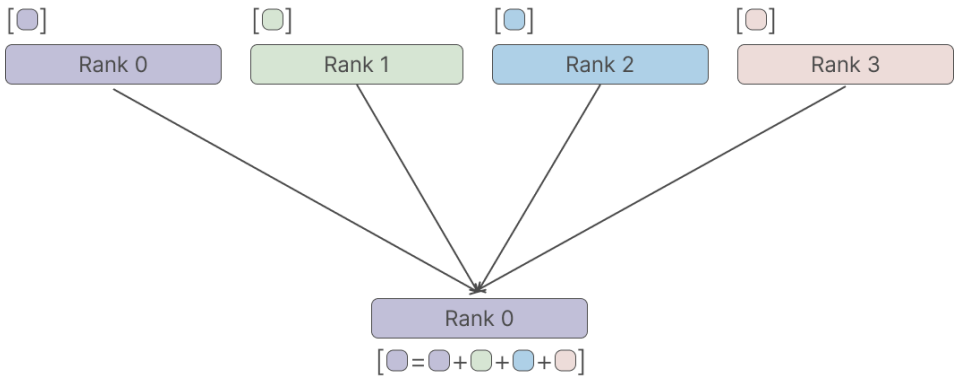
```
$ python -m torch.distributed.launch --nproc_per_node=2 reduce.py

rank 0: tensor([[ 1.2471, -0.4073],
        [ 1.6540,  0.9288]])
rank 1: tensor([[ 1.2471, -0.4073],
        [ 1.6540,  0.9288]])
```

https://tutorials.pytorch.kr/intermediate/dist_tuto.html

Distirubted Communication Package (Torch.distributed)

1. Collective Communication

- Reduce
  - 모든 프로세스에 있는 데이터를 한곳으로 축소
  - 축소 작업은 sum, product, max 등




```python
import torch
import torch.distributed as dist

dist.init_process_group("nccl")
rank = dist.get_rank()
torch.cuda.set_device(rank)

tensor = torch.ones(1).to(torch.cuda.current_device())
# rank 0번에 각 rank의 tensor 를 보내 합친 결과를 rank 0에 저장
dist.reduce(tensor, op=torch.distributed.ReduceOp.SUM, dst=0)

print(f"[{rank}] data = {tensor[0]}"
```

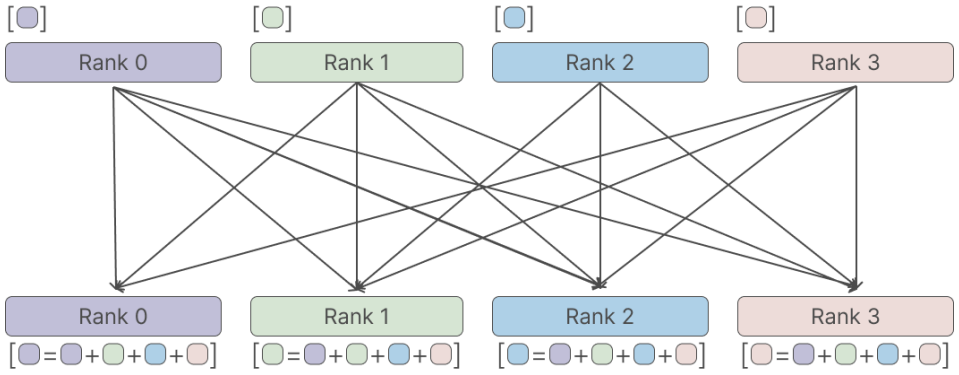
```
$ python -m torch.distributed.launch --nproc_per_node=4 reduce.py

[1] data = 1.0
[2] data = 1.0
[0] data = 4.0
[3] data = 1.0
```

https://blog.roboflow.com/collective-communication-distributed-systems-pytorch/

Distirubted Communication Package (Torch.distributed)

1. Collective Communication

- All Reduce
  - 모든 프로세스에 있는 데이터를 한곳으로 축소하고 모든 프로세스에 반환
  - 모든 프로세스에 Tensor의 평균 도는 합계와 같은 값을 쉽게 계산 할 수 있음




```python
import torch
import torch.distributed as dist

dist.init_process_group("nccl")
rank = dist.get_rank()
torch.cuda.set_device(rank)

tensor = torch.ones(1).to(torch.cuda.current_device())
# 각 rank의 tensor를 모두 합친 결과를 다시 모든 rank에 전송
dist.all_reduce(tensor, op=dist.ReduceOp.SUM )

print(f"[{rank}] data = {tensor[0]}")
```

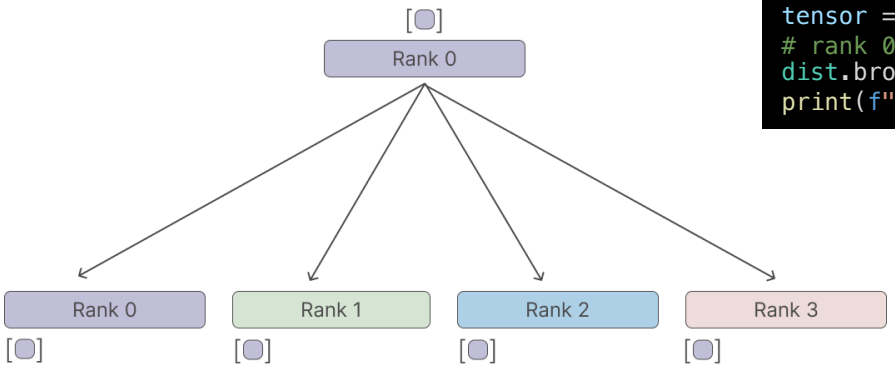
```
$ python -m torch.distributed.launch --nproc_per_node=4 all_reduce.
py

[2] data = 4.0
[1] data = 4.0
[0] data = 4.0
[3] data = 4.0
```

https://blog.roboflow.com/collective-communication-distributed-systems-pytorch/

Distirubted Communication Package (Torch.distributed)

1. Collective Communication

- Broadcast
  - 하나의 프로세스에 있는 데이터를 그룹내 모든 프로세스에 복사 하는 연산
  - 전송 프로세스는 broadcast 함수를 호출할 때 인수로 지정




```python
import torch
import torch.distributed as dist

dist.init_process_group("nccl")
rank = dist.get_rank()
torch.cuda.set_device(rank)

if rank == 0:
tensor = torch.tensor([rank], dtype=torch.float32).to(torch.cuda.current_device())
else:
tensor = torch.empty(1).to(torch.cuda.current_device())
# rank 0번의 tensor ([0.])을 다른 모든 rank에 전송
dist.broadcast(tensor, src=0 )
print(f"[{rank}] data = {tensor}")
```

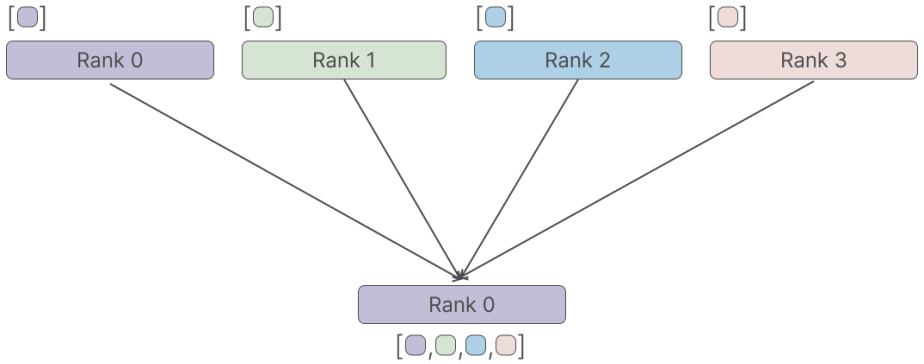
```
$ python -m torch.distributed.launch --nproc_per_node=4 broadcast.py

[0] data = tensor([0.], device='cuda:0')
[2] data = tensor([0.], device='cuda:2')
[1] data = tensor([0.], device='cuda:1')
[3] data = tensor([0.], device='cuda:3')
```

https://blog.roboflow.com/collective-communication-distributed-systems-pytorch/

Distirubted Communication Package (Torch.distributed)

1. Collective Communication

- Gather
  - 모든 프로세스에서 데이터를 수집하고 이를 하나로 연결




```python
import torch
import torch.distributed as dist

dist.init_process_group("nccl")
rank = dist.get_rank()
world_size = 4
torch.cuda.set_device(rank)

tensor = torch.tensor([rank], dtype=torch.float32).to(torch.cuda.current_
device())
# 모든 랭크의 tensor를 랭크 0번으로 보내고 하나의 리스트에 저장
if rank == 0:
tensor_list = [torch.empty(1).to(torch.cuda.current_device()) for i in ra
nge(world_size)]
dist.gather(tensor, gather_list=tensor_list, dst=0)
else:
dist.gather(tensor, gather_list=[], dst=0)
# [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])]
if rank == 0:
print(f"[{rank}] data = {tensor_list}")
```

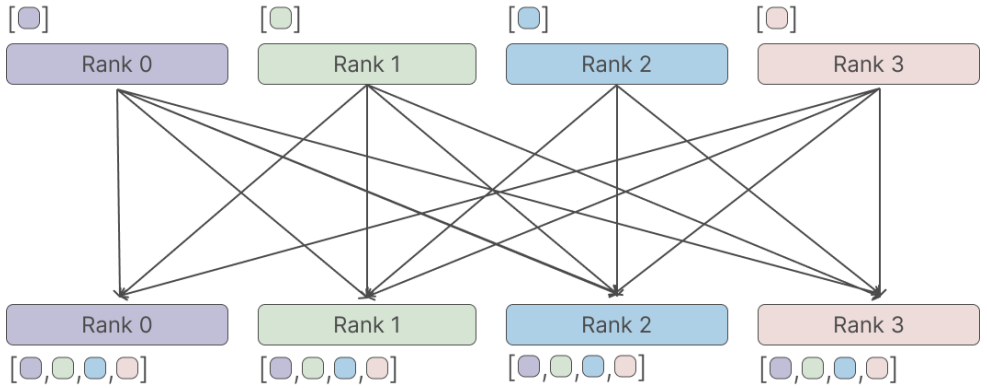
```
$ python -m torch.distributed.launch --nproc_per_node=4 gather.py

[0] data = [tensor([0.], device='cuda:0'), tensor([1.],
device='cuda:0'), tensor([2.], device='cuda:0'), tensor([3.],
device='cuda:0')]
```

https://blog.roboflow.com/collective-communication-distributed-systems-pytorch/

Distirubted Communication Package (Torch.distributed)

1. Collective Communication

- All Gather
  - 모든 프로세스에서 각각에서 다른 프로세스들의 데이터를 모아 하나로 연결




```python
import torch
import torch.distributed as dist

dist.init_process_group("nccl")
rank = dist.get_rank()
world_size = 4
torch.cuda.set_device(rank)

tensor = torch.tensor([rank], dtype=torch.float32).to(torch.cuda.current_device())
# 모든 랭크의 tensor를 다른 모든 랭크에 보내고 각각의 랭크는 수신된 tensor를 하나의 리스트로 저장
tensor_list = [torch.empty(1).to(torch.cuda.current_device()) for i in range(world_size)]
dist.all_gather(tensor_list, tensor=tensor )
# 모든 랭크들은 [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])] 을 가지게 됨
print(f"[{rank}] data = {tensor_list}")
```

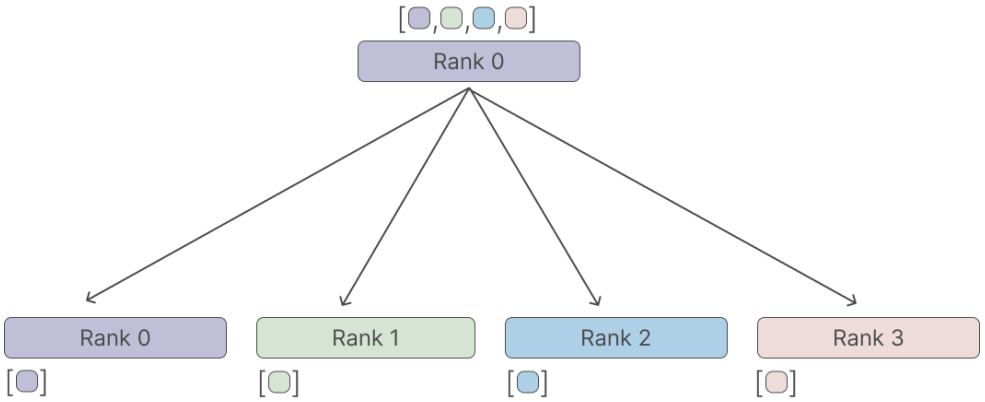
```
$ python -m torch.distributed.launch --nproc_per_node=4 all_gather.py

[2] data = [tensor([0.], device='cuda:2'), tensor([1.], device='cuda:2'), tensor([2.], device='cuda:2'), tensor([3.], device='cuda:2')]
[1] data = [tensor([0.], device='cuda:1'), tensor([1.], device='cuda:1'), tensor([2.], device='cuda:1'), tensor([3.], device='cuda:1')]
[0] data = [tensor([0.], device='cuda:0'), tensor([1.], device='cuda:0'), tensor([2.], device='cuda:0'), tensor([3.], device='cuda:0')]
[3] data = [tensor([0.], device='cuda:3'), tensor([1.], device='cuda:3'), tensor([2.], device='cuda:3'), tensor([3.], device='cuda:3')]
```

https://blog.roboflow.com/collective-communication-distributed-systems-pytorch/

Distirubted Communication Package (Torch.distributed)

1. Collective Communication

- scatter
  - 하나의 프로세스내의 데이터를 그룹내의 프로세스 개수 만큼 나눠서 각각 전송하는 것




```python
import torch
import torch.distributed as dist

#NCCL을 scatter operation을 지원하지 않음
dist.init_process_group("gloo")
rank = dist.get_rank()
world_size = 4
torch.cuda.set_device(rank)

tensor = torch.empty(1).to(torch.cuda.current_device())
# 랭크 0번의 tensor를 world size 만큼 나눠서 각각의 rank에 전송
if rank == 0:
tensor_list = [torch.tensor([i + 1], dtype=torch.float32).to(torch.
cuda.current_device()) for i in range(world_size)]
# tensor_list = [tensor(1), tensor(2), tensor(3), tensor(4)]
dist.scatter(tensor, scatter_list=tensor_list, src=0 )
else:
dist.scatter(tensor, scatter_list=[], src=0 )
print(f"[{rank}] data = {tensor[0]}")
```

```
$ python -m torch.distributed.launch --nproc_per_node=4 scatter.py

[2] data = 3.0
[3] data = 4.0
[0] data = 1.0
[1] data = 2.0
```

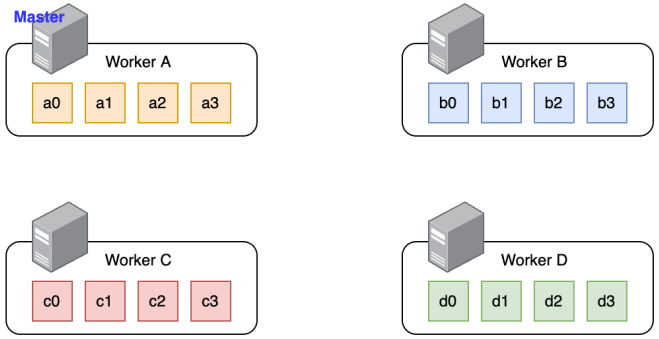
https://blog.roboflow.com/collective-communication-distributed-systems-pytorch/

All-Reduce Architecure 상세

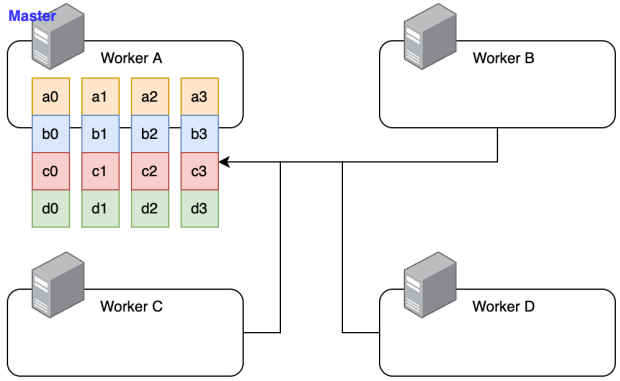
1. Master – Worker 방식

- 모든 데이터 통신은 master로 집중되므로 참여하는 worker가 증가 할 수록 Master가 bottlneck이 되며 확장에 한계가 있음.

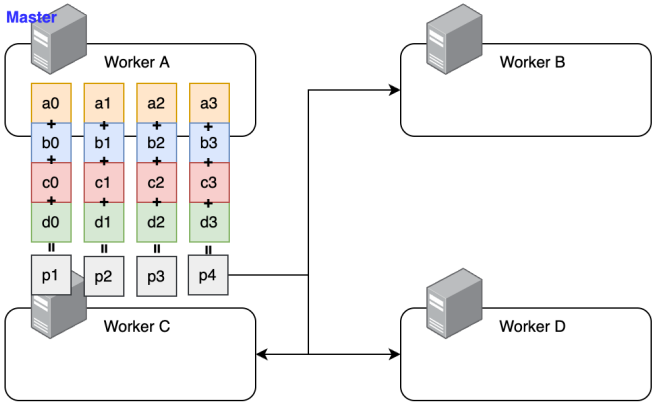
1. 하나의 worker process를 master로 지정한다.



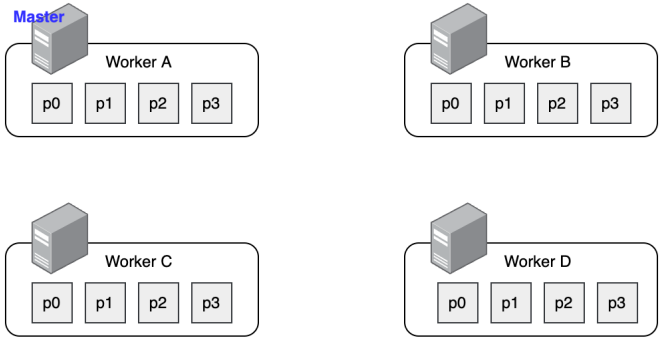
2. worker process에서 master process로 worker process들의 chunk 배열을 단일 배열로 줄인다 (reduce)



3. master process에서 계산된 단일 chunk 배열을 다시 worker process 에 반환한다 (broadcast)



4. 모든 process는 동일한 parameter chunk를 갖는다.

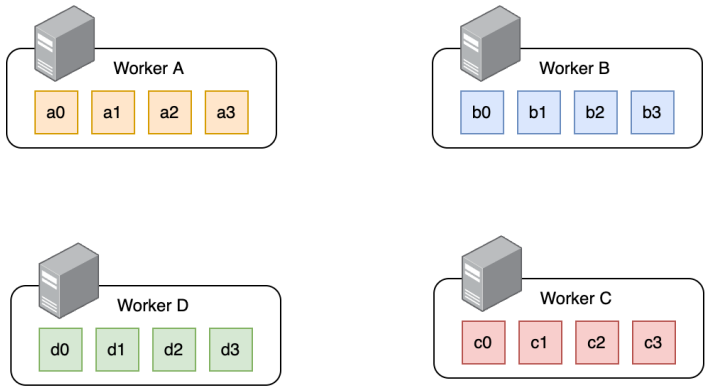


All-Reduce Architecure 상세

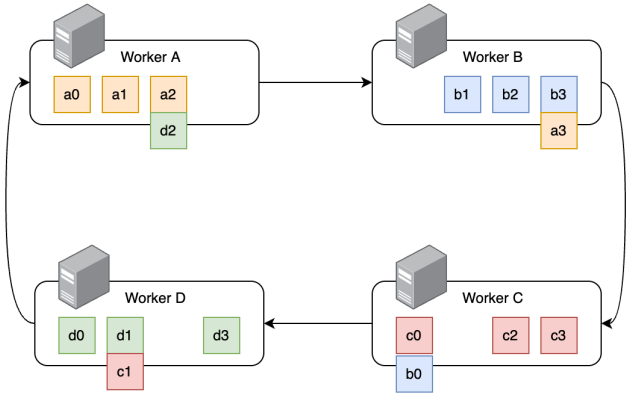
2. Ring-All-Reduce 방식

- 모든 process들이 참여하는 Ring 구조의 통신 방식을 채택하여 Master-worker 방식의 성능 및 확장성의 한계를 개선하기 위한 방법으로 제안

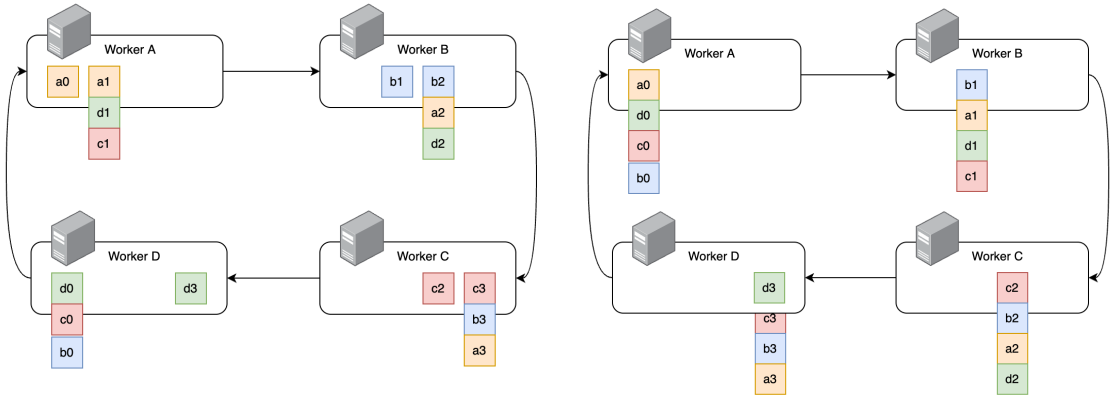
1. 각 worker process에서 parameter를 worker의 수만큼 chunk로 분리하고, chunk에 index를 표시한다. 아래는 4개의 worker process가 있어 데이터를 4개의 chunk로 분리하고 각각 0번 부터 3번까지 index를 부여한다



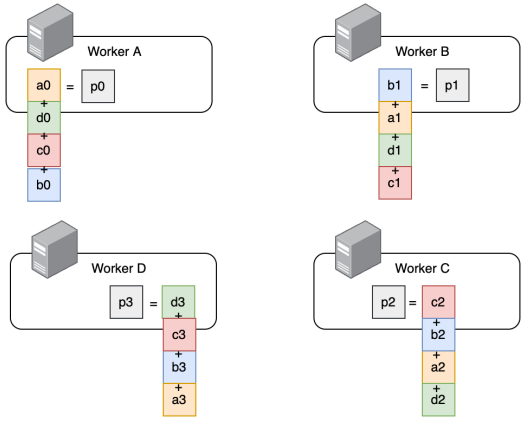
2. 각 worker process 에서 index가 중복되지 않도록 순차적으로 chunk를 다음 worker 로 전달 한다. 아래 예시를 보면 worker A의 4번째 chunk인 a3를 Woker B로 worker B의 첫번째 b0를 다시 worker C로 전달 하는 식이다.



3. 모든 worker process에 chunk가 전달 되도록 이 과정을 반복한다.



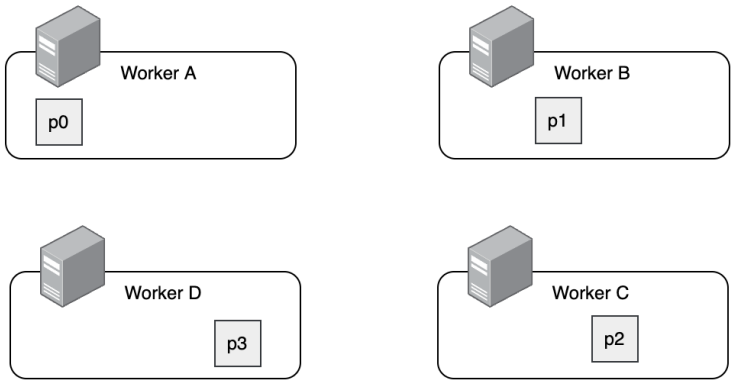
4. 결과적으로 각 worker process는 모든 worker들의 개별 chunk들을 갖게 되고 이 chunk들에 reduce를 하여 하나의 chunk를 만들어 낸다.



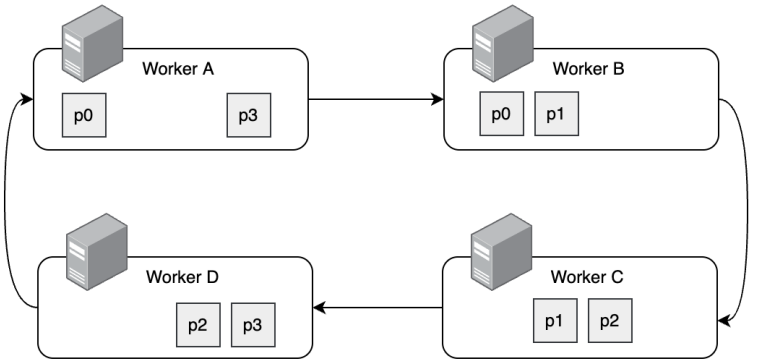
All-Reduce Architecure 상세

2. Ring-All-Reduce 방식

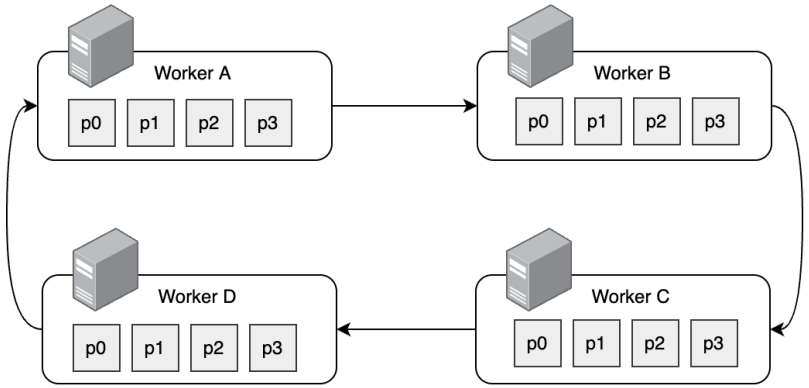
5. 각 worker들은 reduce된 개별 reduced chunk 들을 소유하게 된다.

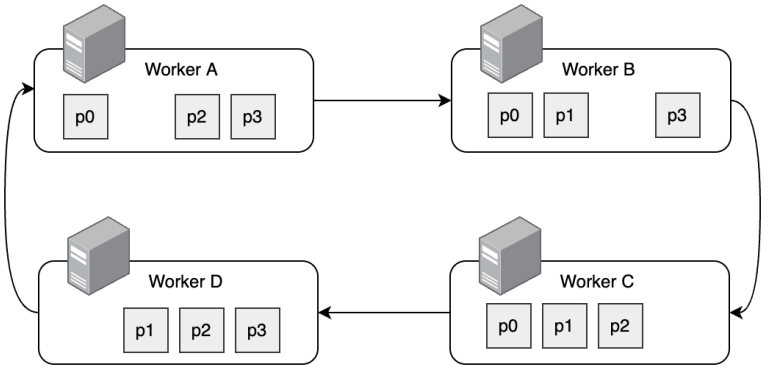


6. 각 worker의 reduced chunk를 다시 다음 worker 전달 한다.



7. n-1 번(n은 worker 수)의 회전이 돌면 모든 worker들이 reduced된 모든 chunk를 가지게 된다..





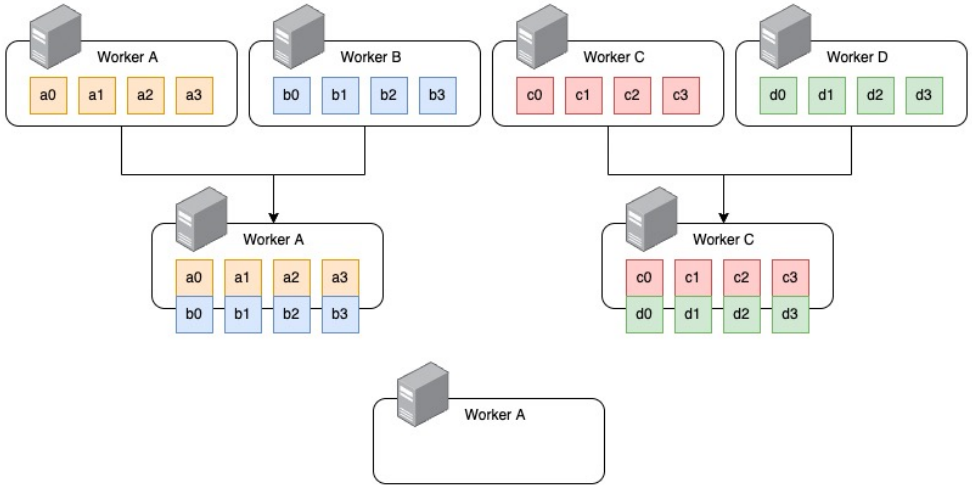
All-Reduce Architecure 상세

3. Tree-All-Reduce

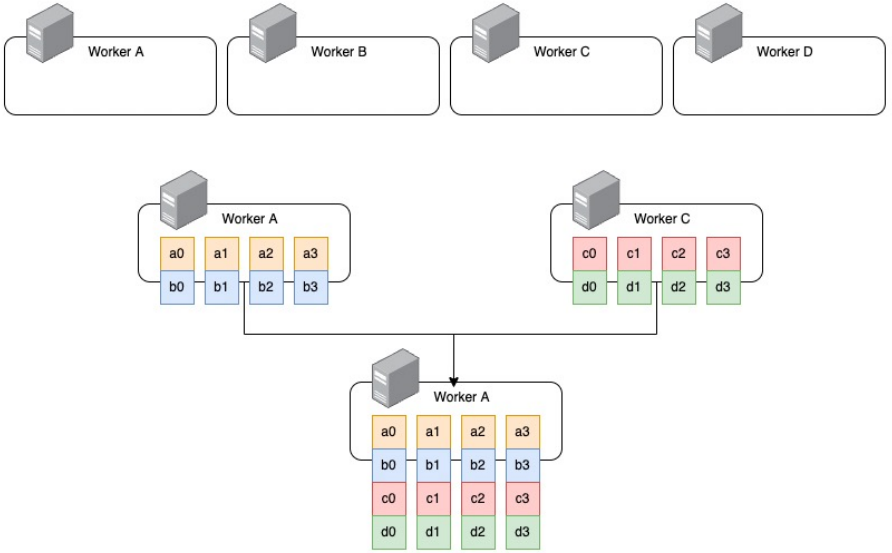
- Ring 방식이 process간 통신에 있어 Master-worker 방식보다 효율적인것은 사실이나 통신 경로가 모든 process를 거쳐가야 한다는 근본적인 문제점이 존재
- Binary Tree를 통해 거쳐가야 하는 Process의 개수는 $log_2 P$ 만큼으로 줄어들게 되어 Process 개수가 늘어 날 수록 Latency 측면에서 효과적

1. 먼저 각 worker process의 연결을 tree 구조로 만든다 아래 그림을 예를 들면 꼭대기 노드에 A가 있다면 A의 하위 가지가 C, B로 매핑하고 다시 C의 하위 가지로는 D와 C를 매핑하는 형식이다.

2. Tree의 가장 하위 worker 2개서 바로 2nd 레벨의 worker 1개로 chunk를 전송한다.



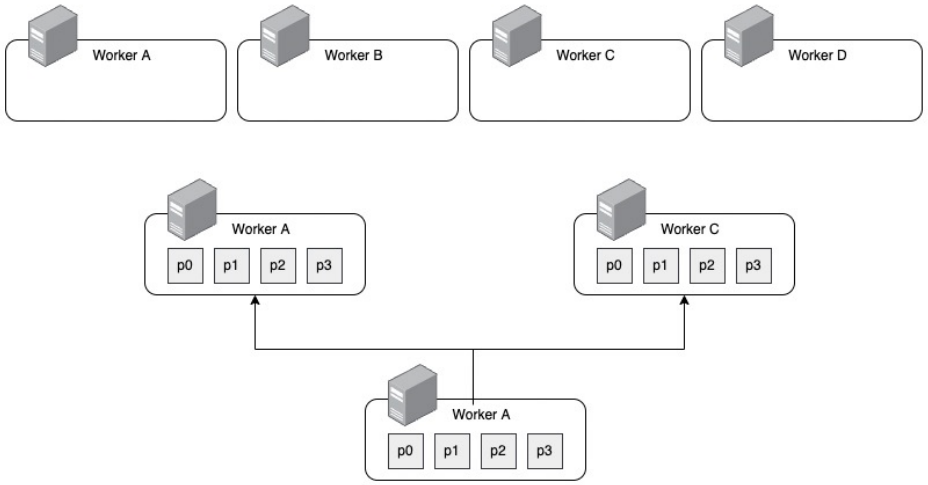
3. 2nd 레벨 worker 2개에서 바로 상위 worker 1래로 모아진 chunk를 전송한다.



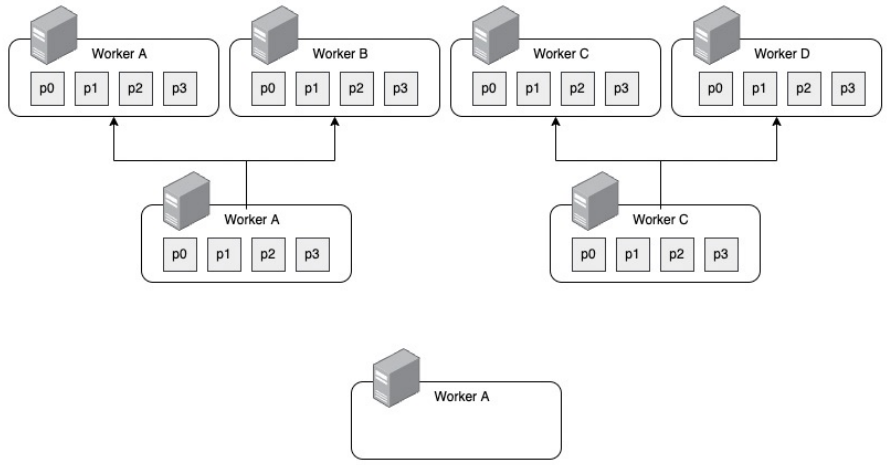
All-Reduce Architecure 상세

3. Tree-All-Reduce

4. 3rd worker에서 모아진 chunk를 reduce 하고 결과를 다시 하위 worker 2개에 반환한다.



5. 2nd worker에서 최하위 worker들에게 reduced 된 chunk를 반환하면 모든 worker들이 동일한 reduced chunk를 갖게 된다.



All-Reduce Architecure 상세

4. All-Reduce 방식에 따른 통신량 비교

- Master-worker 방식
  - $T = (P - 1) * N$
- Ring-All-Reduce 방식
  - $T = 2 * \left(\frac{N}{P}\right) * (P - 1)$
- Tree-All-Reduce 방식
  - $T = log_2 P * N$

- $P: Processes$
- $N: Message\ Size$
- $T: Total\ network\ traffic$

End of Documents