Asynchronous vs Synchronous Distributed Training

Overview

This document explains the differences between asynchronous and synchronous distributed training approaches, using SLURM for cluster management and NCCL for GPU communication. We provide complete code examples and configurations for both methods.

Architecture

System Components

- **SLURM**: Cluster workload manager for job scheduling and resource allocation
- **NCCL**: NVIDIA Collective Communication Library for GPU-to-GPU communication
- **PyTorch Distributed**: High-level distributed training framework
- **Multi-node Setup**: 2-5 compute nodes with GPUs connected via Ethernet

Network Configuration

Each node uses specific network interfaces:

- `master-node`: ens1f1 (192.168.20.15)
- `node01`: ens1f1 (192.168.20.16)
- `node02`: ens1f0np0
- `node03`: ens1f1
- `node04`: ens1f0

Synchronous vs Asynchronous Training

Synchronous Training

How it works:

- All workers process batches simultaneously
- Parameters are synchronized after each batch using collective operations
- Workers wait for all others before proceeding to next batch

Advantages:

- Mathematically equivalent to single-GPU training
- Deterministic and reproducible results
- Simple to implement and debug

Disadvantages:

- Limited by the slowest worker (stragglers)
- Requires global synchronization barriers
- Less fault tolerant

Asynchronous Training

How it works:

- Workers train independently without strict synchronization
- Parameters are exchanged periodically (e.g., every N steps)
- Uses techniques like Polyak averaging for parameter mixing

Advantages:

- More resilient to stragglers and failures
- Higher throughput potential
- Better resource utilization

Disadvantages:

- More complex to implement correctly
- Non-deterministic results
- Convergence properties may differ from synchronous training

Implementation Examples

SLURM Configuration

```bash
```

```
#!/bin/bash
#SBATCH --job-name=distributed_training
#SBATCH --nodes=2                  # Number of compute nodes
#SBATCH --ntasks-per-node=1        # One process per node
#SBATCH --cpus-per-task=4          # CPU cores per process
#SBATCH --gpus-per-node=1          # One GPU per node
#SBATCH --nodelist=master-node,node01
#SBATCH --time=01:00:00            # Maximum runtime
#SBATCH --output=training_%j.out   # Output file
#SBATCH --error=training_%j.err    # Error file

# Master node configuration
export MASTER_ADDR=192.168.20.15
export MASTER_PORT=29500

# Launch distributed processes
srun --mpi=none bash -c '
  export RANK=$SLURM_PROCID
  export WORLD_SIZE=$SLURM_NTASKS

  # Set network interface per node
  case $RANK in
    0) export NCCL_SOCKET_IFNAME=ens1f1 ;;
    1) export NCCL_SOCKET_IFNAME=ens1f1 ;;
    2) export NCCL_SOCKET_IFNAME=ens1f0np0 ;;
    3) export NCCL_SOCKET_IFNAME=ens1f1 ;;
    4) export NCCL_SOCKET_IFNAME=ens1f0 ;;
  esac

  # NCCL configuration
  export NCCL_IB_DISABLE=1
  export NCCL_DEBUG=INFO
  export PYTHONUNBUFFERED=1

  echo "RANK=$RANK on $(hostname) using interface $NCCL_SOCKET_IFNAME"
  python3 training_script.py
'
```

Synchronous Training Implementation

```
python
```

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

def synchronous_training():
    # Initialize process group
    dist.init_process_group(
        backend="nccl",
        rank=int(os.environ["RANK"]),
        world_size=int(os.environ["WORLD_SIZE"]),
        init_method=f"tcp://{os.environ['MASTER_ADDR']}:{os.environ['MASTER_PORT']}"
    )

    model = MyModel().cuda()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

    for epoch in range(num_epochs):
        for batch in dataloader:
            optimizer.zero_grad()
            loss = model(batch)
            loss.backward()

            # Synchronous parameter averaging
            for param in model.parameters():
                dist.all_reduce(param.grad, op=dist.ReduceOp.AVG)

            optimizer.step()
```

Asynchronous Training Implementation

python

```python
def parameters_to_vector(model):
    return torch.nn.utils.parameters_to_vector([p.data for p in model.parameters()])

def vector_to_parameters_(vec, model):
    torch.nn.utils.vector_to_parameters(vec, [p.data for p in model.parameters()])

def polyak_average_(self_vec, peer_vec, alpha=0.5):
    self_vec.mul_(1 - alpha).add_(peer_vec, alpha=alpha)

def asynchronous_training():
    # Same initialization as synchronous
    dist.init_process_group(...)

    model = MyModel().cuda()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

    rank = dist.get_rank()
    world_size = dist.get_world_size()

    for epoch in range(num_epochs):
        for step, batch in enumerate(dataloader):
            optimizer.zero_grad()
            loss = model(batch)
            loss.backward()
            optimizer.step()

            # Asynchronous parameter exchange every N steps
            if step % 5 == 0:
                param_vec = parameters_to_vector(model)

                # Method 1: All-reduce (recommended)
                dist.all_reduce(param_vec, op=dist.ReduceOp.AVG)
                vector_to_parameters_(param_vec, model)

                # Method 2: Ring exchange (more complex)
                # next_rank = (rank + 1) % world_size
                # prev_rank = (rank - 1 + world_size) % world_size
                # recv_buf = torch.empty_like(param_vec)
                #
                # if rank % 2 == 0:
                #     send_req = dist.isend(param_vec, dst=next_rank)
                #     recv_req = dist.irecv(recv_buf, src=prev_rank)
                # else:
                #     recv_req = dist.irecv(recv_buf, src=prev_rank)
                #     send_req = dist.isend(param_vec, dst=next_rank)
                #
                # send_req.wait()
                # recv_req.wait()
                #
                # polyak_average_(param_vec, recv_buf, alpha=0.5)
                # vector_to_parameters_(param_vec, model)
```

Configuration Details

NCCL Environment Variables

```bash
export NCCL_IB_DISABLE=1           # Disable InfiniBand, use Ethernet
export NCCL_SOCKET_IFNAME=ens1f1   # Specify network interface
export NCCL_DEBUG=INFO             # Enable debug logging
export NCCL_DEBUG_SUBSYS=ALL       # Debug all subsystems
```

Process Group Initialization

```python
dist.init_process_group(
    backend="nccl",                  # Use NCCL for GPU communication
    rank=rank,                       # Current process rank
    world_size=world_size,           # Total number of processes
    init_method="tcp://master:port"  # Rendezvous method
)
```

Common Issues and Solutions

1. Hanging During Initialization

Symptoms: Process hangs at `ncclCommInitRankConfig`

Causes:

- Network connectivity issues between nodes
- Firewall blocking communication ports
- Incorrect network interface configuration

Solutions:

- Test network connectivity: `ping other_nodes`
- Check firewall settings
- Verify NCCL_SOCKET_IFNAME matches actual interfaces

2. Deadlocks in Point-to-Point Communication

Symptoms: Hangs during `dist.isend()/dist.irecv()` operations

Cause: Circular dependencies in ring communication patterns

Solution: Use alternating send/receive patterns:

```python
if rank % 2 == 0:
    send_req = dist.isend(data, dst=next_rank)
    recv_req = dist.irecv(buffer, src=prev_rank)
else:
    recv_req = dist.irecv(buffer, src=prev_rank)
    send_req = dist.isend(data, dst=next_rank)
```

3. SLURM Environment Variables

Required variables:

- `SLURM_PROCID` → `RANK`
- `SLURM_NTASKS` → `WORLD_SIZE`
- `SLURM_LOCALID` → `LOCAL_RANK`

Verification:

```bash
echo "RANK=$SLURM_PROCID, WORLD_SIZE=$SLURM_NTASKS"
```

Performance Considerations

Communication Patterns

1. **All-Reduce**: Most efficient for parameter averaging
2. **Point-to-Point**: More flexible but prone to deadlocks
3. **Ring**: Good for large-scale deployments

Synchronization Frequency

- **Frequent sync (every step)**: Better convergence, lower throughput
- **Infrequent sync (every N steps)**: Higher throughput, potentially slower convergence
- **Optimal frequency**: Depends on model size and network bandwidth

Network Optimization

- Use dedicated high-bandwidth networks when possible
- Configure NCCL for your specific network topology
- Consider network topology in process placement

Best Practices

1. **Start Small**: Test with 2 nodes before scaling up
2. **Use Collective Operations**: Prefer all-reduce over point-to-point
3. **Monitor Network Usage**: Watch for bandwidth bottlenecks
4. **Error Handling**: Implement timeouts and recovery mechanisms
5. **Debug Incrementally**: Enable NCCL debugging for troubleshooting

Debugging Checklist

1. **Network Connectivity**

   ```bash
   # Test from each node
   ping master_node_ip
   telnet master_node_ip master_port
   ```

2. **SLURM Environment**

   ```bash
   echo "Job ID: $SLURM_JOB_ID"
   echo "Nodes: $SLURM_NNODES"
   echo "Tasks: $SLURM_NTASKS"
   ```

3. **NCCL Configuration**

   ```bash
   export NCCL_DEBUG=INFO
   # Look for "Init COMPLETE" messages
   ```

4. **Process Synchronization**

   ```python
   # Test basic collective operation
   test_tensor = torch.tensor(float(rank)).cuda()
   dist.all_reduce(test_tensor, op=dist.ReduceOp.SUM)
   print(f"All-reduce result: {test_tensor.item()}")
   ```

Conclusion

Both synchronous and asynchronous distributed training have their place in deep learning workflows. Synchronous training provides consistency and ease of debugging, while asynchronous training offers better fault tolerance and potential performance benefits. The choice depends on your specific requirements for accuracy, performance, and fault tolerance.

For most applications, we recommend starting with synchronous training using collective operations (all-reduce), as it's more reliable and easier to debug than point-to-point asynchronous approaches.