Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Data Parallelism: Deep Learning con GPU Multiple in modo efficiente e sostenibile.

Francesco Gargiulo, Antonio Francesco Gentile, Emilio Greco

RT- ICAR-NA-24-04 **Novembre 2024**

Premessa

In questo rapporto tecnico, illustreremo i principi fondamentali del DDP. Sebbene, mostreremo a titolo esemplificativo l'addestramento distribuito di una rete di classificazione su più GPU i concetti trattati sono applicabili a qualsiasi architettura di modello e a varie applicazioni. A tal fine, sarà descritto l'uso del pacchetto **DistributedDataParallel (DDP)** di PyTorch, una libreria di deep learning progettata per velocizzare le operazioni di addestramento.

Questo rapporto tecnico è il primo di una serie di documenti tecnici volti a fornire utili informazioni per l'uso efficiente e sostenibile di infrastrutture di calcolo per l'IA basate su architetture multi-GPU. In particolare, questo lavoro è stato reso possibile grazie alle attività realizzate nell'ambito del progetto "Humanities and Cultural Heritage Italian Open Science Cloud – H2IOSC," finanziato dall'Unione europea - NextGenerationEU nell'ambito del PNRR Missione 4, "Istruzione e Ricerca" - Componente 2, "Dalla ricerca all'impresa" - Linea di investimento 3.1, "Fondo per la realizzazione di un sistema integrato di infrastrutture di ricerca e innovazione", decreto di concessione del finanziamento prot. MUR n. 112 del 20-06-2022 (CUP B63C22000730005).

I corsi di formazione forniti da NVIDIA Academy, le competenze interne all'ICAR e la documentazione acquisita attraverso vari canali, hanno permesso di approfondire e applicare tecniche all'avanguardia nel calcolo distribuito consentendo di esplorare e formalizzare modalità operative e casi d'uso delle risorse di calcolo recentemente acquistate dall'ICAR CNR, per il supporto alle ricerche avanzate condotte nell'ambito di infrastrutture per l'AI basate sul paradigma multi-GPU.

1. Introduzione al Distributed Data Parallel (DDP)

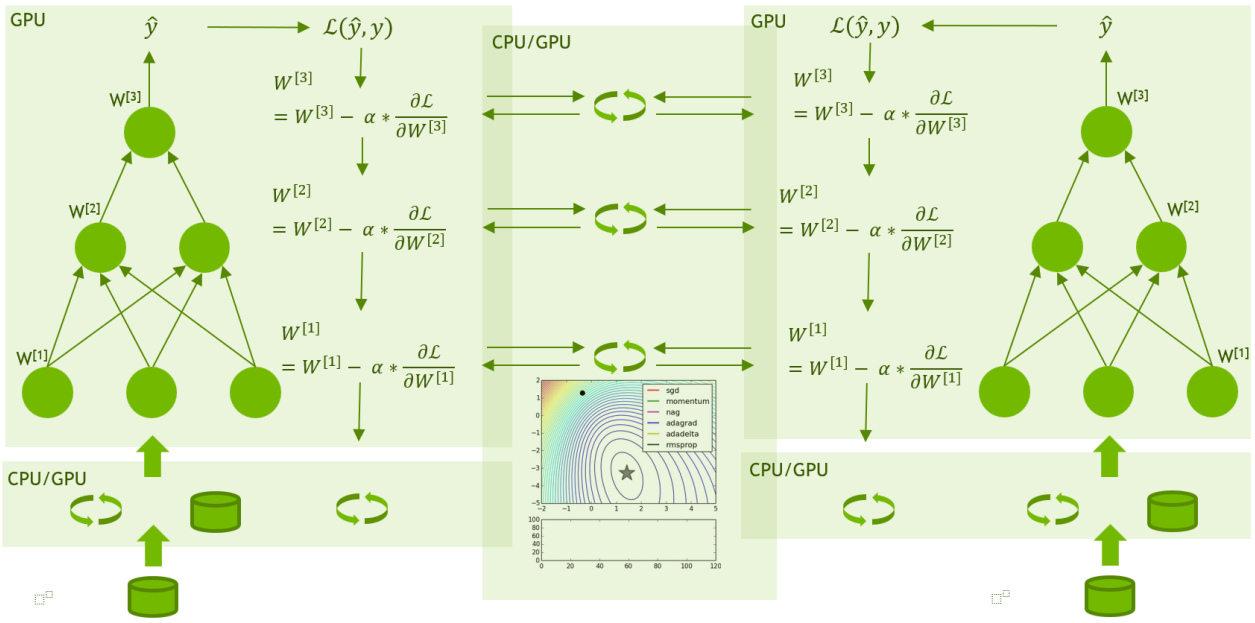
Le più comuni strategie per l'addestramento di reti neurali in modo distribuito sono essenzialmente due: il **parallelismo sul modello** e il **parallelismo sui dati**. Il parallelismo dei dati è una tecnica che sfrutta la duplicazione del modello tra più GPU. Questo approccio risulta particolarmente utile quando la dimensione del batch del modello è troppo elevata per essere gestita da una singola macchina oppure quando si cerca di accelerare il processo di addestramento. Con questa tecnica, ogni replica del modello viene addestrata simultaneamente su una porzione del dataset.
Il parallelismo del modello è una tecnica utilizzata quando i parametri del modello sono troppo grandi per essere gestiti nei limiti della memoria disponibile. Questo metodo prevede di suddividere i processi di addestramento su più GPU, eseguendoli in parallelo o in sequenza. Ogni suddivisione del modello lavora sullo stesso set di dati, rendendo necessaria la sincronizzazione delle informazioni tra le diverse parti del modello.
Nel seguito di questo documento useremo un codice di esempio e ci concentreremo sulla strategia di addestramento distribuito dei dati. La scelta del pacchetto software da utilizzare per raggiungere il nostro obiettivo dipende spesso dalla piattaforma di programmazione (come Keras, PyTorch o MXNet). Nel nostro caso pratico utilizzeremo PyTorch poichè sta diventando sempre più popolare tra i ricercatori. La libreria DDP nativa di PyTorch si sta affermando come una delle scelte

principali per l'addestramento distribuito dei dati. In questo contesto, DDP viene utilizzata come base per apprendere i principi del parallelismo basato sui dati finalizzati ad accelerare l'addestramento delle reti neurali.

Di seguito introdurremo alcuni termini chiave che sono fondamentali per comprendere il resto del documento. Chiameremo **Processo o Worker** un'istanza del programma Python. L'ipotesi di lavoro è che ogni processo controlla una singola e specifica GPU, pertanto il numero di GPU disponibili per l'addestramento distribuito determina quanti processi potranno essere avviati da DDP.
Il termine **Nodo o Host** verrà usato per riferirci ad un computer fisico con tutti i suoi componenti hardware, come CPU, GPU, RAM, ecc. Quindi ogni nodo della nostra architettura avrà un ID univoco assegnato. Ad un addestramento distribuito partecipano più processi in parallelo, per identificare il numero totale di processi che utilizzeremo DDP usa il termine **World size**.
Il termine **Global rank** è usato per descrivere l'ID univoco assegnato a ciascun processo indipendentemente dal nodo in cui viene eseguito e viene calcolato in funzione dell'architettura usata ad eccezione del **processo principale** che avrà sempre **global rank 0**.
Con il termine **Local rank** verrà identificato l'ID univoco assegnato a ciascun processo all'interno di un singolo nodo. Poiché ogni nodo può contenere più di una GPU al local rank generalmente viene assimilato l'indice della GPU su cui viene eseguito il processo.
Se prendiamo ad esempio un cluster di calcolo costituito da due nodi, ciascuno dei quali dotato di due GPU, se vogliamo utilizzare tutte le GPU disponibili per addestrare il nostro modello in parallelo, il **world size sarà 4**, ciò vuol dire che ci saranno **quattro processi** indipendenti (uno per ciascuna GPU). A ogni processo verrà assegnato un **global rank che varia da 0 a 3**, e ogni nodo/host avrà un ID univoco (0 o 1). I processi all'interno di ciascun nodo avranno un **local rank pari a 0 o 1**.

Per realizzare la tecnica di addestramento con parallelismo dei dati, l'architettura del modello dovrà essere replicata su tutte le GPU disponibili in modo che ogni GPU elabori una porzione distinta dei dati. L'addestramento viene eseguito in **modo sincronizzato** ossia, in modo che tutte le copie del modello sulle diverse GPU abbiano gli stessi pesi. A tal fine, **i pesi vengono inizializzati solo dal processo principale** (quello con rank 0) e poi trasmessi a tutti gli altri processi. Questo assicura che tutte le GPU abbiano una copia identica dei parametri del modello prima che l'addestramento inizi, la libreria DDP di pytorch gestisce internamente questo meccanismo.
Dopo l'inizializzazione del modello, ciascuna GPU riceve una porzione distinta dei dati, che viene elaborata in modo indipendente. Durante la **forward propagation**, ogni GPU elabora i suoi dati attraverso il modello per calcolare una previsione e, coseguentemente, una "loss[1]". Successivamente, vengono calcolati i gradienti a partire dalla loss, questi vengono quindi condivisi tra tutte le GPU tramite un'operazione chiamata **all-reduce**, che calcola la media dei gradienti per ciascun parametro del modello. Infine, ogni GPU utilizza i gradienti mediati per aggiornare i propri parametri durante la fase di **backward propagation**. Questo processo garantisce che tutte le GPU abbiano lo stesso modello sincronizzato, con parametri identici alla fine di ogni ciclo di addestramento.

[1] Indicatore numerico di quanto il modello è lontano dall'obiettivo preposto.



Attraverso il DDP il codice che viene scritto per addestrare un modello su una singola GPU può essere agevolmente eseguito in parallelo su più processi e su più GPU simultaneamente. Questi processi lavorano su porzioni diverse dello stesso dataset di addestramento, garantendo che ogni GPU si occupi solo di una parte distinta di esso. La libreria DDP gestisce la sincronizzazione dei modelli e dei gradienti per mantenere il corretto aggiornamento dei parametri su tutte le GPU.

La struttura del documento è la seguente: inizialmente viene presentato un riepilogo delle tecniche principali per l'addestramento distribuito, seguito da una descrizione dettagliata dell'implementazione pratica di uno script di addestramento distribuito con DDP.

2.Implementazione script di addestramento con DDP

Di seguito vedremo passo dopo passo come utilizzare la libreria DDP modificando un classico algoritmo di addestramento su singola GPU.

Per motivi esemplificativi, partiremo dalla versione di un codice mono-gpu implementato per realizzare un classificatore di immagini di abbigliamento partendo dal dataset **Fashion-MNIST**:

```
import argparse
import torch
import torch.nn as nn
import numpy as np
import os
import time
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader
```

```
# Parse input arguments
parser = argparse.ArgumentParser(description='Fashion MNIST Example',
                    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--batch-size', type=int, default=32,
            help='input batch size for training')
parser.add_argument('--epochs', type=int, default=40,
            help='number of epochs to train')
parser.add_argument('--base-lr', type=float, default=0.01,
            help='learning rate for a single GPU')
parser.add_argument('--target-accuracy', type=float, default=.85,
            help='Target accuracy to stop training')
parser.add_argument('--patience', type=int, default=2,
            help='Number of epochs that meet target before stopping')
args = parser.parse_args()


# Standard convolution block followed by batch normalization
class cbrblock(nn.Module):
   def __init__(self, input_channels, output_channels):
      super(cbrblock, self).__init__()
      self.cbr = nn.Sequential(nn.Conv2d(input_channels, output_channels, kernel_size=3, stride=(1,1),
                  padding='same', bias=False),
                  nn.BatchNorm2d(output_channels),
                  nn.ReLU()
      )
   def forward(self, x):
      out = self.cbr(x)
      return out

# Basic residual block
class conv_block(nn.Module):
   def __init__(self, input_channels, output_channels, scale_input):
      super(conv_block, self).__init__()
      self.scale_input = scale_input
      if self.scale_input:
         self.scale = nn.Conv2d(input_channels, output_channels, kernel_size=1, stride=(1,1),
                  padding='same')
      self.layer1 = cbrblock(input_channels, output_channels)
      self.dropout = nn.Dropout(p=0.01)
      self.layer2 = cbrblock(output_channels, output_channels)

   def forward(self, x):
      residual = x
      out = self.layer1(x)
      out = self.dropout(out)
      out = self.layer2(out)
      if self.scale_input:
         residual = self.scale(residual)
```

```
        out = out + residual
        return out

# Overall network
class WideResNet(nn.Module):
    def __init__(self, num_classes):
        super(WideResNet, self).__init__()
        nChannels = [1, 16, 160, 320, 640]
        self.input_block = cbrblock(nChannels[0], nChannels[1])
        # Module with alternating components employing input scaling
        self.block1 = conv_block(nChannels[1], nChannels[2], 1)
        self.block2 = conv_block(nChannels[2], nChannels[2], 0)
        self.pool1 = nn.MaxPool2d(2)
        self.block3 = conv_block(nChannels[2], nChannels[3], 1)
        self.block4 = conv_block(nChannels[3], nChannels[3], 0)
        self.pool2 = nn.MaxPool2d(2)
        self.block5 = conv_block(nChannels[3], nChannels[4], 1)
        self.block6 = conv_block(nChannels[4], nChannels[4], 0)
        # Global average pooling
        self.pool = nn.AvgPool2d(7)
        # Feature flattening followed by linear layer
        self.flat = nn.Flatten()
        self.fc = nn.Linear(nChannels[4], num_classes)

    def forward(self, x):
        out = self.input_block(x)
        out = self.block1(out)
        out = self.block2(out)
        out = self.pool1(out)
        out = self.block3(out)
        out = self.block4(out)
        out = self.pool2(out)
        out = self.block5(out)
        out = self.block6(out)
        out = self.pool(out)
        out = self.flat(out)
        out = self.fc(out)
        return out

def train(model, optimizer, train_loader, loss_fn, device):
    model.train()
    for images, labels in train_loader:
        # Transfering images and labels to GPU if available
        labels = labels.to(device)
        images = images.to(device)
        # Forward pass
        outputs = model(images)
        loss = loss_fn(outputs, labels)
        # Setting all parameter gradients to zero to avoid gradient accumulation
```

```
        optimizer.zero_grad()
        # Backward pass
        loss.backward()
        # Updating model parameters
        optimizer.step()

def test(model, test_loader, loss_fn, device):
    total_labels = 0
    correct_labels = 0
    loss_total = 0
    model.eval()
    with torch.no_grad():
        for images, labels in test_loader:
            # Transfering images and labels to GPU if available
            labels = labels.to(device)
            images = images.to(device)
            # Forward pass
            outputs = model(images)
            loss = loss_fn(outputs, labels)
            # Extracting predicted label, and computing validation loss and validation accuracy
            predictions = torch.max(outputs, 1)[1]
            total_labels += len(labels)
            correct_labels += (predictions == labels).sum()
            loss_total += loss
    v_accuracy = correct_labels / total_labels
    v_loss = loss_total / len(test_loader)
    return v_accuracy, v_loss


if __name__ == '__main__':
    train_set = torchvision.datasets.FashionMNIST("./data", download=True, transform=
                                    transforms.Compose([transforms.ToTensor()]))
    test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False, transform=
                                    transforms.Compose([transforms.ToTensor()]))
    # Training data loader
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=args.batch_size, drop_last=True)
    # Validation data loader
    test_loader = torch.utils.data.DataLoader(test_set, batch_size=args.batch_size, drop_last=True)
    # Create the model and move to GPU device if available
    num_classes = 10
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = WideResNet(num_classes).to(device)
    # Define loss function
    loss_fn = nn.CrossEntropyLoss()
    # Define the SGD optimizer
    optimizer = torch.optim.SGD(model.parameters(), lr=args.base_lr)
    val_accuracy = []
    total_time = 0
```

```
    for epoch in range(args.epochs):
        t0 = time.time()
        train(model, optimizer, train_loader, loss_fn, device)
        epoch_time = time.time() - t0
        total_time += epoch_time
        images_per_sec = len(train_loader)*args.batch_size/epoch_time
        v_accuracy, v_loss = test(model, test_loader, loss_fn, device)
        val_accuracy.append(v_accuracy)
            print("Epoch = {:2d}: Cumulative Time = {:5.3f}, Epoch Time = {:5.3f}, Images/sec = {}, Validation Loss =
{:5.3f}, Validation
                                Accuracy = {:5.3f}".format(epoch+1, total_time, epoch_time, images_per_sec, v_loss,
val_accuracy[-1]))
                    if len(val_accuracy) >= args.patience and all(acc >= args.target_accuracy for acc in
val_accuracy[-args.patience:]):
            print('Early stopping after epoch {}'.format(epoch + 1))
            break
```

Ad esempio, per addestrare il modello in 5 epoche con un batch size di 512, lanceremo il seguente comando con due attributi:

```
python myscript.py --epochs 5 --batch-size 512
```

Ottenendo il seguente output:

```
Epoch =  1: Cumulative Time = 91.131, Epoch Time = 91.131, Images/sec = 657.3424213456788, Validation Loss = 0.686, Validation Accuracy = 0.741
Epoch =  2: Cumulative Time = 182.929, Epoch Time = 91.798, Images/sec = 652.560768298225, Validation Loss = 0.612, Validation Accuracy = 0.781
Epoch =  3: Cumulative Time = 274.775, Epoch Time = 91.846, Images/sec = 652.2236577699774, Validation Loss = 0.486, Validation Accuracy = 0.826
Epoch =  4: Cumulative Time = 366.682, Epoch Time = 91.907, Images/sec = 651.7869326969766, Validation Loss = 0.415, Validation Accuracy = 0.852
Epoch =  5: Cumulative Time = 458.566, Epoch Time = 91.884, Images/sec = 651.9506140525963, Validation Loss = 0.378, Validation Accuracy = 0.864
Early stopping after epoch 5
```

Prima di apportare le modifiche necessarie per scalare il modello su più GPU, ci assicuriamo di poter addestrare correttamente la versione del modello su una singola GPU. Le informazioni prodotte da questo test ci saranno utili, in questo esempio, anche per confrontare i risultati ottenuti con l'esecuzione in parallelo.

Per comprendere lo **speed-up** ottenuto dalla parallelizzazione del processo di addestramento è fondamentale acquisire ed osservare alcuni parametri di utilizzo delle GPU come ad esempio il **consumo di energia o l'uso della memoria**. In molti casi risulta molto importante eseguire il monitoraggio continuo del carico di lavoro durante tutto il processo di addestramento, o almeno nelle fasi iniziali per capire se gli script sono stati lanciati correttamente su ogni nodo.

Dopo aver apportato le modifiche allo script descritte precedentemente, andremo ad eseguire i seguenti comandi:

1. Esecuzione del comando `watch -n 2 nvidia-smi`[2] per monitorare l'attività della GPU in un secondo terminale.
2. Sul primo terminale avvio dello script di addestramento del modello::

```
python myscript.py --epochs 1 --batch-size 512
```

```
Epoch =  1: Cumulative Time = 91.203, Epoch Time = 91.203, Images/sec = 656.8205623612605, Validation Loss = 0.574, Validation Accuracy = 0.788
```

Le seguenti line di codice dello script riportate di seguito si occupano di definire ed assegnare la GPU o CPU su cui eseguire il modello:

```
# Imposta il dispositivo per il calcolo (cuda:0 corrisponde alla prima GPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# Inizializza il modello e trasferiscilo al dispositivo
model = WideResNet(num_classes).to(device)
```

Dalle informazioni di output del comando `nvidia-smi`, che riportiamo nell'immagine seguente, ci andremo ad annotare il consumo di risorse necessarie per l'addestramento del modello sulla singola GPU selezionata del nostro nodo.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:1B.0 Off |                    0 |
| N/A   27C    P0    68W /  70W |   7365MiB / 15360MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            On   | 00000000:00:1C.0 Off |                    0 |
| N/A   22C    P8    14W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla T4            On   | 00000000:00:1D.0 Off |                    0 |
| N/A   22C    P8    14W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla T4            On   | 00000000:00:1E.0 Off |                    0 |
| N/A   23C    P8    14W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      8232      C                                   7363MiB |
+-----------------------------------------------------------------------------+
```

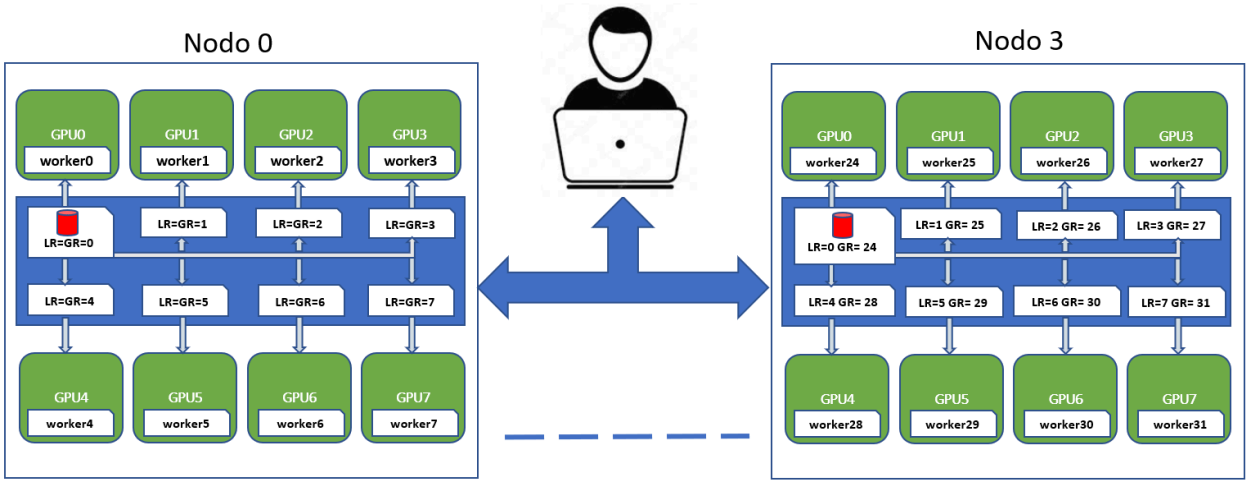
[2] Il comando `nvidia-smi` restituisce informazioni sull'utilizzo delle GPU installate su un nodo ed in ultima analisi ci consente di acquisire ulteriori informazioni come ad esempio il consumo, l'utilizzo della memoria, etc. Questo comando può essere eseguito su qualsiasi sistema dotato di GPU NVIDIA.

2.1 Definizione e Configurazione Architettura

Al fine di rendere lo script utilizzabile su più architetture è necessario aggiungere alcuni parametri di input, come ad esempio il numero di nodi e i relativi ID, nonché il numero di GPU per ciascun nodo.
Supponiamo di poter disporre, per eseguire l'addestramento, di quattro nodi di cui ogni nodo dispone di 8 GPU, questo si può tradurre nel lanciare 32 processi di addestramento dislocati su quattro terminali separati. Su ogni uno dei quattro nodi dovremmo poter lanciare il seguente comando:

```
python myscript.py --epochs 5 --batch-size 512 --node-id 0 --num-gpus 8 --num-nodes 4
python myscript.py --epochs 5 --batch-size 512 --node-id 1 --num-gpus 8 --num-nodes 4
python myscript.py --epochs 5 --batch-size 512 --node-id 2 --num-gpus 8 --num-nodes 4
python myscript.py --epochs 5 --batch-size 512 --node-id 3 --num-gpus 8 --num-nodes 4
```

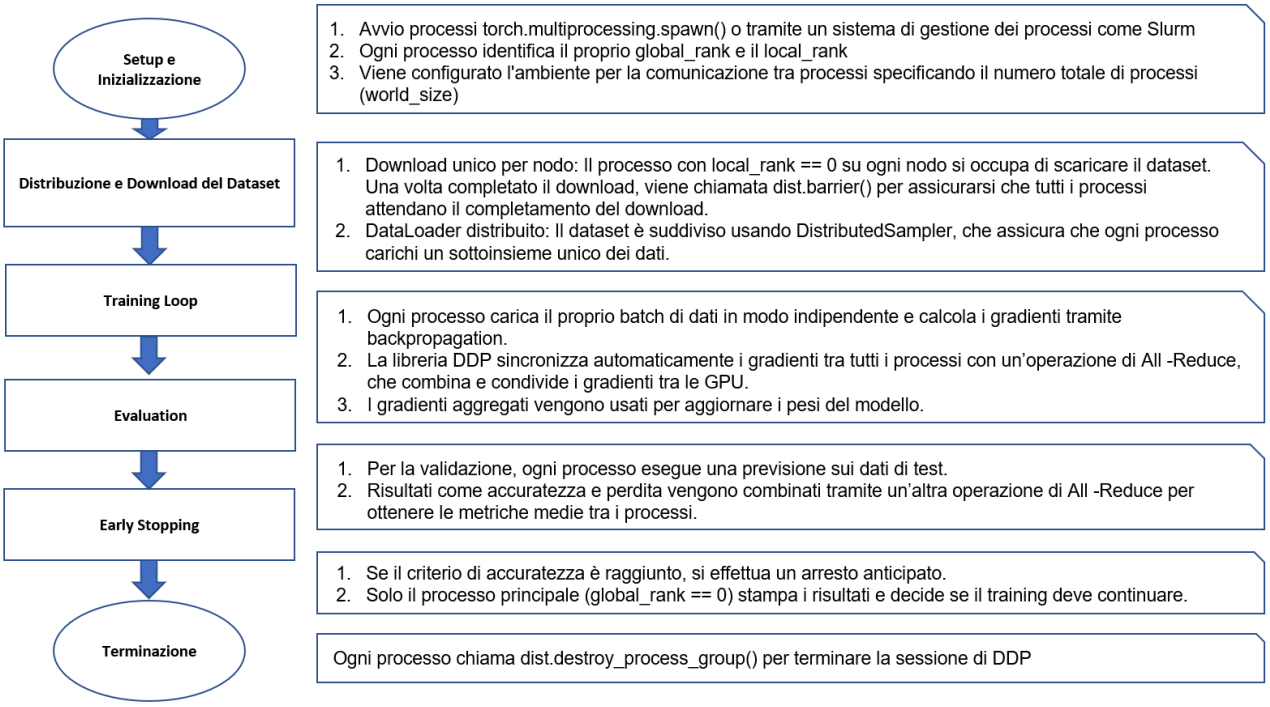
Nella Figura seguente viene mostrata la disposizione dei worker nei 4 nodi di calcolo. Si noti che solo il processo con local_rank = 0 esegue il download del dataset in ogni nodo (worker0; worker8; worker16; worker24). Infine, che ogni worker, riceve la stessa rete con i pesi inizializzati dal worker0 ed una partizione distinta del DB.



Questi semplici comandi lanciati su ogni nodo dovranno impostare l'addestramento su 32 GPU distribuite, sincronizzando i processi tra loro prima dell'addestramento ed eseguendo l'aggiornamento dei gradienti ad ogni epoca. In questo scenario, ogni nodo deve lanciare lo script separatamente e in modo indipendente, specificando l'ID del nodo e il numero totale di nodi e di GPU per nodo come parametri.

PyTorch, tramite la libreria **torch.distributed**, si occuperà poi di coordinare e sincronizzare automaticamente i processi tra i diversi nodi. Una volta avviato su ciascun nodo, lo script utilizzerà la libreria torch.distributed, che gestisce la sincronizzazione tra i processi. Questo avviene tramite

l'impostazione degli indirizzi IP e delle porte per il nodo principale, solitamente il nodo con node_id=0, quindi ogni nodo si collega all'indirizzo e alla porta del nodo principale, ed attraverso la funzione **torch.distributed.init_process_group()** gestisce il coordinamento tra i nodi e le GPU.



Allo stesso tempo la funzione DistributedDataParallel (DDP) si occuperà di sincronizzare automaticamente i gradienti e di aggiornare i parametri del modello, utilizzando operazioni come **all-reduce** per assicurarsi che tutti i pesi siano allineati tra i diversi nodi e GPU

Per modificare lo script di partenza, il primo step da compiere è ovviamente l'importazione delle librerie necessarie. Le librerie che utilizzeremo sono già incluse nell'installazione standard di PyTorch, pertanto non sarà necessario installare nuovi pacchetti. Utilizzeremo in particolare la libreria **multiprocessing** che ci fornirà le funzionalità per lanciare più processi, ognuno assegnato a una GPU e la libreria **DistributedDataParallel** (DDP) per integrare funzionalità come ad esempio torch.distributed per distribuire automaticamente il lavoro tra le GPU, sincronizzare i gradienti e aggiornare i parametri in modo trasparente per l'utente.

```
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP
```

Dovremmo quindi ampliare la sezione dello script che riguarda il parsing degli argomenti di input per includere il **Numero di Nodi, GPU e ID del Nodo**.

Naturalmente, per poter configurare opportunamente gli script per la fase di addestramento ci occorre sapere il numero di gpu presenti su ogni nodo attraverso il comando `nvidia-smi`.

Di seguito lo script con le integrazioni finora descritte:

```
import argparse
import torch
import torch.nn as nn
import numpy as np
import os
import time
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP

# Parse input arguments
parser = argparse.ArgumentParser(description='Fashion MNIST Example',
                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--batch-size', type=int, default=32,
                    help='input batch size for training')
parser.add_argument('--epochs', type=int, default=40,
                    help='number of epochs to train')
parser.add_argument('--base-lr', type=float, default=0.01,
                    help='learning rate for a single GPU')
parser.add_argument('--target-accuracy', type=float, default=.85,
                    help='Target accuracy to stop training')
parser.add_argument('--patience', type=int, default=2,
                    help='Number of epochs that meet target before stopping')
parser.add_argument('--num-nodes', type=int, default=1,
                    help='number of nodes for distributed training')
parser.add_argument('--node-id', type=int, default=0,
                    help='ID of the current node (0 is the main node)')
parser.add_argument('--num-gpus', type=int, default=1,
                    help='number of GPUs per node')
parser.add_argument('--momentum', type=float, default=0.9,
                help='SGD momentum')

args = parser.parse_args()
```

2.2 World size e processo principale

Lo script per eseguire un addestramento su più nodi contenenti più di una GPU va predisposto, pertanto dovremmo calcolare, come primo step, la dimensione della world_size in funzione dei parametri passati come input. Inoltre, dovremo configurare i parametri del nodo master. A tal fine andremo ad inseriamo nel main le seguenti righe di codice.

```
WORLD_SIZE = args.num_gpus * args.num_nodes
os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '29500'
```

Per configurare l'ambiente di esecuzione, occorre identificare l'indirizzo IP del nodo associato al processo principale (global rank 0) e la relativa porta di comunicazione che verrà successivamente utilizzata dagli altri processi per comunicare col nodo master e sincronizzare le attività. Poiché in questo esempio consideriamo l'addestramento su un singolo host, possiamo impostare questo indirizzo IP come localhost. Inoltre, occorre assegnare un numero di porta libero compreso tra 1024 e 49151 utilizzata per la comunicazione dei processi.

2.3 Configurazione dei Worker

Per definire le funzionalità dei singoli worker occorre anzitutto spostare il contenuto del blocco main dello script, visto all'inizio del documento, all'interno in una nuova funzione che per semplicità chiameremo **worker**. Questa funzione verrà invocata dal metodo main, dello stesso script, da un particolare metodo, come vedremo anche in seguito: **mp.spawn(worker, nprocs=args.num_gpus, args=(args,))** che si occupa della creazione dei processi e la loro esecuzione delle varie GPU.

La funzione worker dovrà accettare due argomenti:

- **local_rank**: Il rank locale del processo (ricordiamo che questo parametro indica su quale GPU specifica eseguire il processo all'interno del nodo).
- **args**: Gli argomenti di input passati dal parser (quante epoche, dimensione del batch, numero di nodi, ecc.).

```
def worker(local_rank, args):
    # Inserire qui il codice di addestramento trasferito dal blocco __main__
```

I processi, una volta istanziati, invocheranno la funzione worker, pertanto è necessario predisporre una fase di inizializzazione che porti ed una sincronizzazione tra tutti i processi in esecuzione. Questo obbiettivo può essere raggiunto attraverso l'uso del metodo **dist.init_process_group**. Il metodo **dist.init_process_group** sincronizza i processi prima di avviare l'addestramento distribuito,

assicurandosi che tutte le GPU abbiano lo stesso stato iniziale del modello e che quindi possano comunicare tra loro. Questa funzione accetta tre parametri:

- **Backend**: identifica quale piattaforma di multiprocessing utilizzare. Nel nostro caso utilizzeremo NVIDIA Collective Communications Library (NCCL). Questo backend è ottimizzato per comunicazioni tra GPU NVIDIA.
- **World size**: rappresenta il numero totale di processi.
- **Global rank**: Identifica univocamente ciascun processo nell'intero gruppo distribuito, permettendo a ogni processo di sapere qual è il suo identificativo all'interno dell'ambiente di esecuzione.

Il global rank di ciascun processo può essere determinato a partire dalle informazioni sull'ID del nodo, il numero di GPU per nodo e il rank locale. Per calcolare in modo semplice questo identificativo possiamo avvalerci della seguente formula:

```
def worker(local_rank, args):

    # Calcola il global rank per il processo
    global_rank = node_id * num_gpus + local_rank

    # Inizializza e sincronizza i processi
    dist.init_process_group(backend='nccl', world_size=WORLD_SIZE, rank=global_rank)
```

I parametri per il calcolo del `global_rank` sono passati in input alla chiamata dello script come nell'esempio seguente:

```
python myscript.py --node-id 0 --num-gpus 4 --num-nodes 1  --epochs 3 --batch-size 512
```

In questo caso essendoci un solo nodo, il global rank corrisponde al local rank. La funzione `mp.spawn` contenuta nel main provvederà a trasferire i parametri alla chiamata alle diverse istanze della funzione worker. Nel nostro esempio avremo pertanto 4 istanze di worker:

worker(--local_rank 0,* --epochs 3, --batch-size 512*)
worker(--local_rank 1,* --epochs 3, --batch-size 512*)
worker(--local_rank 2,* --epochs 3, --batch-size 512*)
worker(--local_rank 3,* --epochs 3, --batch-size 512*)

2.4 Gestione delle operazioni di I/O

I prossimi step di modifica dello script sono finalizzati alla corretta gestione delle operazioni di I/O in ambiente distribuito. Se ad esempio lasciassimo invariate le istruzioni di download del dataset per l'addestramento, così come mostrato di seguito:

```
def worker(local_rank, args):

    # Calcola il global rank per il processo
    global_rank = node_id * num_gpus + local_rank

    # Inizializza e sincronizza i processi
    dist.init_process_group(backend='nccl', world_size=WORLD_SIZE, rank=global_rank)

    train_set = torchvision.datasets.FashionMNIST("./data", download=True, transform=
                        transforms.Compose([transforms.ToTensor()]))
    test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False,
    transform=
                        transforms.Compose([transforms.ToTensor()]))
```

il dataset verrebbe scaricato un numero di volte pari al numero di processi worker instanziati. Per ottimizzare l'uso delle risorse, dovremmo impostare all'interno del nostro codice, che **solo un processo per nodo** scarichi il dataset. Cosa che affronteremo nel prossimo paragrafo parlando della gestione delle operazioni di Input/Output.

Quando il processo di addestramento coinvolge **più nodi** è necessario assicurarsi che ciascun nodo deve avere una copia locale del dataset anche in virtù del fatto che i nodi potrebbero non poter condividere lo stesso filesystem. Allo stesso tempo, all'interno dello stesso nodo, dobbiamo assicurarci che solo e soltanto un processo effettui le operazioni di IO altrimenti otterremo un degrado notevole delle prestazioni ed un uso eccessivo delle risorse.

Per sincronizzare il download dei dati tra i processi in esecuzione all'interno di ciascun nodo, si utilizza il metodo **dist.barrier()**. Il metodo permette di bloccare i processi locali, in un determinato punto della loro esecuzione, in attesa che il processo principale completi la fase di download del dataset. La funzione dist.barrier() è utilizzata in contesti di addestramento distribuito con PyTorch per sincronizzare i processi tra le diverse GPU. Quando viene chiamata, dist.barrier() blocca l'esecuzione di ciascun processo fino a quando **tutti i processi** all'interno del gruppo di comunicazione non raggiungono questo punto. Solo quando tutti i processi sono giunti alla barriera, possono continuare con l'esecuzione del codice successivo.

Una volta completato il download su un nodo, dist.barrier() notifica agli altri processi dello stesso nodo che il download è terminato ed i dati sono disponibili, evitando che gli stessi iniziano la fase di training ancor prima che i dati siano presenti.

```
def worker(local_rank, args):

    # Calcola il global rank per il processo
    global_rank = node_id * num_gpus + local_rank
    # Inizializza e sincronizza i processi
    dist.init_process_group(backend='nccl', world_size=WORLD_SIZE, rank=global_rank)
    # Il processo principale (local_rank == 0) scarica il dataset, gli altri attendono
    if local_rank == 0:
        train_set = torchvision.datasets.FashionMNIST("./data", download=True,
                                transform=transforms.Compose([transforms.ToTensor()]))
        test_set = torchvision.datasets.FashionMNIST("./data", download=True, train=False,
                                transform=transforms.Compose([transforms.ToTensor()]))
    dist.barrier()  # Tutti i processi attendono il completamento del download

    # Tutti i processi caricano il dataset
    train_set = torchvision.datasets.FashionMNIST("./data", download=False,
                            transform=transforms.Compose([transforms.ToTensor()]))
    test_set = torchvision.datasets.FashionMNIST("./data", download=False, train=False,
                            transform=transforms.Compose([transforms.ToTensor()]))
```

Allo stesso tempo, se abbiamo la necessità di eseguire operazioni di scrittura su file o stampe sul prompt dei comandi, dovremmo far riferimento ad un solo nodo worker per eseguire tali operazioni. Nel nostro caso utilizzeremo per tali operazioni il processo principale:

```
……

if global_rank == 0:

        print("Epoch = {:2d}: Validation Loss = {:5.3f}, Validation Accuracy = {:5.3f}".format(epoch+1,
v_loss,  val_accuracy[-1]))
```

2.5 Data Partitioning distribuita

L'obiettivo della metodologia DDP consiste nel suddividere il dataset e distribuirlo tra più GPU per addestrare la stessa replica di rete neurale. Di conseguenza il DataLoader non riceverà più tutto il dataset ma soltanto una partizione di esso. Per partizionare il dataset faremo uso della utility: **torch.utils.data.distributed.DistributedSampler.** Questa funzione facilita le operazioni di segmentazione del dataset in modo che ciascuna GPU disponga di una partizione proporzionale alla world **size** calcolata.

```
# Create Distributed Samplers
                train_sampler = torch.utils.data.distributed.DistributedSampler(train_set,
num_replicas=WORLD_SIZE, rank=global_rank , shuffle=True)
```

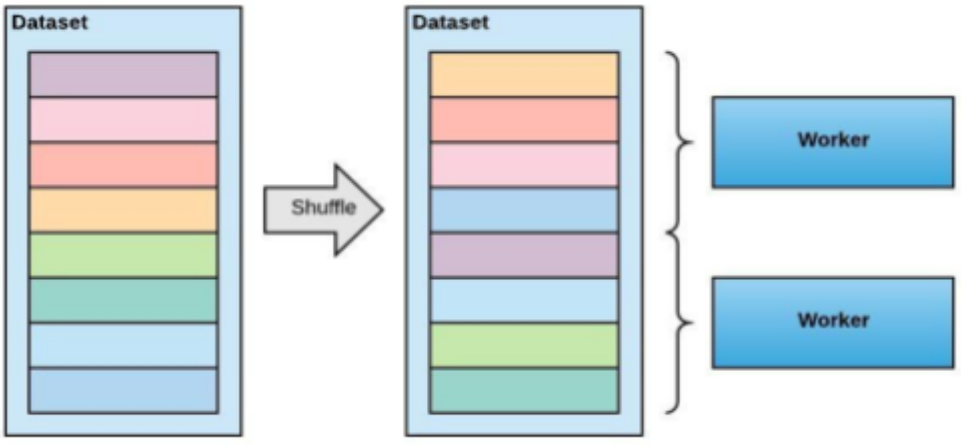
```
    test_sampler = torch.utils.data.distributed.DistributedSampler(test_set, num_replicas=WORLD_SIZE,
rank=global_rank,  shuffle=True)

    # Data loaders
    train_loader = DataLoader(dataset=train_set, batch_size=args.batch_size, sampler=train_sampler)
    test_loader = DataLoader(dataset=test_set, batch_size=args.batch_size, sampler=test_sampler)
```

La classe torch.utils.data.distributed.DistributedSampler è progettata per suddividere i dati tra diversi processi distribuiti, facilitando l'addestramento su più GPU. I parametri che accetta la funzione sono i seguenti:

- **dataset (train_set o test_set)**: indica il dataset che si vuole suddividere tra i processi. In questo caso, si utilizza train_set per il campionatore di addestramento e test_set per il campionatore di validazione.
- **num_replicas (WORLD_SIZE)**: indica il numero totale di processi (o repliche) che stanno eseguendo l'addestramento distribuito. Solitamente, WORLD_SIZE è il numero totale di GPU disponibili. Questo parametro consente a ogni campionatore di sapere quante parti deve suddividere il dataset.
- **rank (global_rank)**: rank rappresenta l'identificatore univoco di ciascun processo nel contesto distribuito. global_rank è l'ID specifico del processo corrente (va da 0 a WORLD_SIZE - 1).
- **shuffle (True)**: durante l'addestramento, spesso si utilizza shuffle=True per garantire che i dati siano distribuiti in modo casuale in ogni epoca. Per il set di test, shuffle=True è opzionale, ma si può usare per randomizzare i dati di validazione se richiesto.

In un ambiente distribuito lo shuffle del dataset è obbligato in quando occorre garantire che:

1. I modelli presenti su ogni GPU non si adattino a un ordine specifico dei dati.
2. Ogni GPU elabori una parte distinta del dataset, evitando ridondanze e garantendo efficienza computazionale.
3. L'addestramento sia sempre basato sugli stessi gruppi di dati, migliorando la generalizzazione del modello. Aggiungendo il rimescolamento dei dati tra le epoche.



Quando si utilizza **DistributedSampler**, questo suddivide il dataset in segmenti da assegnare a ciascuna GPU. Per impostazione predefinita, il parametro **shuffle=True**, il che significa che i dati vengono rimescolati prima di essere divisi in segmenti per ogni GPU. Il rimescolamento iniziale del dataset contribuisce a migliorare la generalizzazione del modello, poiché assicura che i dati vengano elaborati in ordine casuale e non in un ordine predefinito o ripetitivo. **Questo riduce il rischio di overfitting**, migliorando le capacità del modello di generalizzare su nuovi dati.
Ogni GPU riceve un **segmento fisso** del dataset che processerà per tutta la durata dell'addestramento. Questo significa che, all'inizio, ogni GPU è responsabile di una porzione specifica del dataset, definita dal sampler. Inoltre è necessario **cambiare l'ordine dei batch** all'interno di quel segmento, a ogni epoca. Questo è fatto utilizzando la funzione **train_sampler.set_epoch(epoch)**, che assicura che il dataset sia rimescolato in modo diverso all'inizio di ogni epoca.

2.6 Distribuzione del modello sulle GPU

Nello script precedente abbiamo visto come assegnare il modello della nostra rete alla prima GPU disponibile sul nodo attraverso le seguenti istruzioni:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = WideResNet(num_classes).to(device)
```

In caso di più GPU presenti sullo stesso nodo, le istruzioni verranno modificate in modo che una copia del modello venga assegnata ad ogni GPU presente avvalendoci del parametro Local_rank:

```
# Imposta il dispositivo in base al local_rank per ogni processo
device = torch.device(f"cuda:{local_rank}" if torch.cuda.is_available() else "cpu")
# Trasferisci il modello sul dispositivo assegnato
model = WideResNet(num_classes).to(device)
```

Oltre alle precedenti istruzioni occorre eseguire l'incapsulamento del modello con **DistributedDataParallel** (DDP). Questo wrapper permette al modello di distribuire in modo efficiente e trasparente l'addestramento su più dispositivi (GPU), gestendo automaticamente molti aspetti complessi della sincronizzazione e della comunicazione tra GPU.

```
model = nn.parallel.DistributedDataParallel(model, device_ids=[local_rank])
```

Quando si usa nn.parallel.DistributedDataParallel, ogni GPU avrà una copia del modello e lavorerà su un sottoinsieme dei dati, calcolando i gradienti sui suoi campioni. Questo wrapper in sintesi si occupa di:

- Distribuire i calcoli su tutte le GPU, riducendo la duplicazione dei calcoli per ciascuna GPU.

- **Sincronizzare automaticamente i gradienti** tra tutte le GPU alla fine di ogni batch, usando un'operazione di riduzione chiamata **All-Reduce**. In questo modo, ogni GPU riceve i gradienti aggiornati da tutte le altre GPU.
- **Ottimizzare la comunicazione**: DDP sfrutta la NVIDIA Collective Communications Library (NCCL), particolarmente efficiente nel ridurre i tempi di comunicazione tra GPU.

2.7 Sincronizzazione delle statistiche per la Batch Normalization (opzionale)

Quando usiamo DistributedDataParallel (DDP), **All-Reduce** è usato automaticamente da DDP per sincronizzare i gradienti dei parametri del modello tra tutti i processi, permettendo un aggiornamento coordinato dei pesi. Questo meccanismo assicura che ogni GPU usi gli stessi pesi e che i gradienti siano combinati e aggiornati in modo identico. In breve All-Reduce si occupa di sincronizzare i gradienti a livello di modello in modo che ogni GPU riceva lo stesso aggiornamento di peso. Questo è fondamentale per garantire la consistenza dei pesi tra le GPU, mantenendo ogni copia del modello sincronizzata durante l'addestramento distribuito.
In un ambiente distribuito, la Batch Normalization può presentare un problema quando il batch size per GPU è ridotto. Un batch size ridotto porta a statistiche di normalizzazione (media e deviazione standard) calcolate su ogni singola GPU che non sono rappresentative del batch complessivo del dataset. In questo caso, possiamo usare il meccanismo **All-Reduce per sincronizzare le statistiche della batch normalization**.
A tal fine è stata sviluppata la funzione **torch.nn.SyncBatchNorm** che è una versione sincronizzata della Batch Normalization nella quale, anziché calcolare le statistiche separatamente su ogni GPU, le sincronizza tra tutte le GPU utilizzando la tecnica di All-Reduce. In questo modo, tutte le GPU utilizzano le stesse stime di media e deviazione standard per la normalizzazione, rendendo le stime più rappresentative dell'intero dataset.
Usare torch.nn.SyncBatchNorm è semplice e può essere implementato con convert_sync_batchnorm per convertire tutti i layer di Batch Normalization nel modello, aumentando l'accuratezza delle stime delle statistiche in caso di batch size ridotto.

```
# Creazione del modello
model = WideResNet(num_classes)

# Step opzionale: Sincronizzazione delle statistiche di batchnorm
model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model)

# Imposta il dispositivo e trasferisci il modello
device = torch.device(f"cuda:{local_rank}")
model = model.to(device)

# Incapsula il modello con DistributedDataParallel
model = nn.parallel.DistributedDataParallel(model, device_ids=[local_rank])
```

Quando usarlo: È consigliabile usarlo solo quando il batch size per GPU è molto piccolo e influisce negativamente sulle prestazioni di addestramento. Se il batch size per GPU è sufficientemente grande, come per il nostro esempio, ogni GPU sarà in grado di calcolare buone stime delle statistiche locali, rendendo questo passaggio non necessario. L'utilizzo di un secondo step di sincronizzazione rallenterebbe il processo di addestramento.

Riepilogo degli utilizzi di All-Reduce:

- **Sincronizzazione dei gradienti** (Obbligatorio in DDP): All-Reduce si occupa di sincronizzare i gradienti dei pesi tra le GPU per garantire che ogni copia del modello abbia gli stessi parametri.
- **Sincronizzazione delle statistiche** (Facoltativo in DDP): All-Reduce sincronizza le stime di media e deviazione standard del batch tra tutte le GPU, utilizzare solo quando il batch size è piccolo.

Entrambe le operazioni utilizzano All-Reduce ma in contesti diversi: uno per i gradienti e l'altro per statistiche di normalizzazione, con obiettivi specifici di sincronizzazione dei parametri e delle metriche di normalizzazione rispettivamente.
Esiste un terzo utilizzo di All-reduce che fa uso della funzione **dist.all_reduce.** Questa funzione può essere usata a facoltà dello sviluppatore all'interno del codice per eseguire una sincronizzazione forzata dei processi. Un esempio di utilizzo, come vedremo di seguito, è per ottenere delle statistiche globali, come ad esempio l'accuratezza media del processo parallelo. Come già anticipato l'uso di all_reduce è sconsigliato in ambiente di produzione perché rallenta le operazioni di calcolo ma viene spesso utilizzato in ambiente di test per rilevare delle statistiche generali.

2.8 Calcolo del Throughput

Nel contesto dell'addestramento distribuito su più GPU, calcolare il throughput (ossia quante immagini sono elaborate per secondo dall'intero swarm) è utile per valutare le prestazioni ottenute nella parallelizzazione del processo.
Per il calcolo del throughput multiprocesso, possiamo seguire questi passaggi:

- **Calcolo del throughput locale**: ogni GPU calcola quante immagini è riuscita a processare per secondo. Ad esempio, se su una GPU una epoca richiede 2 secondi per processare 1000 immagini, il throughput sarà di 500 immagini al secondo.
- **Conversione in tensore**: per predisporre il calcolo della media su GPU inseriamo questo valore in un tensore.
- **Calcolo della media con all-reduce**: una volta che il throughput è in formato tensore, possiamo applicare una riduzione distribuita (come **all_reduce**), che somma i throughput calcolati da ciascuna GPU per ottenere un totale complessivo.

I tensori permettono un'elaborazione efficiente su GPU, rendendo semplice spostare e manipolare i dati direttamente nel luogo di calcolo e sfruttare la velocità della GPU. Questo permette un calcolo veloce su GPU usando le operazioni di sincronizzazione, come all_reduce.
Vediamo di seguito praticamente come utilizzare questa funzionalità. Possiamo modificare lo script come segue:

1) Aggiungiamo una chiamata a dist.barrier() alla fine di ogni epoca per assicurarci che tutti i processi abbiano completato l'epoca prima di procedere con il calcolo del throughput.
2) Convertiamo images_per_sec in un tensore per calcolare il throughput distribuito.
3) Utilizziamo torch.distributed.reduce per sommare li throughput tra tutte le GPU e salvare il risultato nel processo principale (global rank 0).
4) Stampiamo i risultati solo nel processo principale.

```
# ciclo di addestramento
for epoch in range(args.epochs):
    t0 = time.time()
    train_sampler.set_epoch(epoch)
    train(model, optimizer, train_loader, loss_fn, device)
    # Alla fine di ogni epoca
    dist.barrier()  # Assicurati che tutti i processi abbiano completato l'epoca
    epoch_time = time.time() - t0
    total_time += epoch_time
    # Calcola il numero di immagini processate al secondo
    # Convertilo in un tensore su GPU
    images_per_sec = torch.tensor(len(train_loader) * args.batch_size / epoch_time).to(device)
    # Riduci i valori di images_per_sec tra tutte le GPU e somma al global rank 0
    torch.distributed.reduce(images_per_sec, dst=0, op=torch.distributed.ReduceOp.SUM)

    # Stampa solo nel processo principale
    if global_rank == 0:
      print(f"Epoch = {epoch + 1}, Cumulative Time = {total_time:.3f}s, "
      f"Epoch Time = {epoch_time:.3f}s, Images/sec = {images_per_sec.item():.3f}")
```

Nello spezzone di script vediamo che ogni processo calcola images_per_sec (immagini processate per secondo) in base al proprio tempo di epoca. Questa metrica è locale a ciascun processo e rappresenta il throughput calcolato specificamente per i dati elaborati su quella GPU. Successivamente ogni processo invoca la funzione reduce, che:

- Somma i valori images_per_sec di tutti i processi.
- Salva il risultato nel tensore images_per_sec **del processo specificato con dst=0** (quindi il processo con global_rank == 0).

Dopo l'operazione reduce, solo il processo con global_rank == 0 avrà accesso alla somma complessiva di images_per_sec proveniente da tutti i processi, **poiché è lui che riceve il risultato aggregato**. Questo processo stampa quindi il valore complessivo del throughput calcolato per tutte le GPU.

2.9 Calcolo dell'Accuracy e della Loss

Poiché ora ogni GPU esegue la validazione solo su una porzione del dataset, ciascun processo produrrà risultati di validazione diversi. Per migliorare la qualità delle metriche di validazione e ridurre la varianza, calcoleremo la media dei risultati di validazione tra tutti i worker utilizzando la funzione **all-reduce** utilizzando la metodologia descritta in precedenza.

```
# ciclo di addestramento
for epoch in range(args.epochs):
    t0 = time.time()
    train_sampler.set_epoch(epoch)
    train(model, optimizer, train_loader, loss_fn, device)
    # Alla fine di ogni epoca
    dist.barrier()  # Assicurati che tutti i processi abbiano completato l'epoca
    epoch_time = time.time() - t0
    total_time += epoch_time
    # Calcola il numero di immagini processate al secondo
    # Convertilo in un tensore su GPU
    images_per_sec = torch.tensor(len(train_loader) * args.batch_size / epoch_time).to(device)
    # Riduci i valori di images_per_sec tra tutte le GPU e somma al global rank 0
    torch.distributed.reduce(images_per_sec, dst=0, op=torch.distributed.ReduceOp.SUM)

    # Validation
    v_accuracy, v_loss = test(model, test_loader, loss_fn, device)

    # Average validation accuracy and loss across GPUs
    dist.all_reduce(v_accuracy, op=dist.ReduceOp.AVG)
    dist.all_reduce(v_loss, op=dist.ReduceOp.AVG)

    val_accuracy.append(v_accuracy)

    # Print metrics only on the main process
    if global_rank == 0:
       print(f"Epoch {epoch+1}: Time = {epoch_time:.3f}s, Images/sec = {images_per_sec.item():.3f},
       Validation Accuracy = {v_accuracy.item():.3f}, Validation Loss = {v_loss.item():.3f}")
```

Nota: salviamo I valori medi complessivi dell'accurancy in un vettore poiché ci servirà come vedremo nel prossimo paragrafo per verificare i criteri di Early Stopping , ovvero impostare un parametro di patience.

2.10 Condizione di Early Stopping

Le seguenti istruzioni chiudono la definizione della funzione worker.

```
# Early stopping if accuracy condition is met
if len(val_accuracy) >= args.patience and all(acc >= args.target_accuracy for acc in val_accuracy[-args.patience:]):
    if global_rank == 0:
        print(f"Early stopping after epoch {epoch+1}")
    break

dist.destroy_process_group()
```

La prima istruzione descrive la regola d'uscita del processo di addestramento nella quale il modello ha raggiunto una precisione sufficiente per un dato numero di epoche consecutive (**patience**), quindi l'addestramento può essere interrotto.
La seconda istruzione chiude il gruppo di processi distribuiti e libera le risorse. E' importante che questa funzione venga chiamata al termine di ogni processo distribuito per una corretta gestione della memoria e della comunicazione tra le GPU.

2.11 Esecuzione dei processi

In un ambiente di calcolo distribuito, in genere si richiede che ogni nodo lanci il proprio processo di addestramento separatamente. I parametri `--node-id`, `--num-nodes`, e `--num-gpus` della chiamata: **python *myscript*.py --node-id 0 --num-gpus 4 --num-nodes 1 --epochs 3 --batch-size 512** sono necessari per configurare ciascun nodo nel contesto dell'intero cluster, permettendo la sincronizzazione tra i nodi.

Il codice finale prevede un solo metodo all'iterno della funzione main:

```
mp.spawn(worker, nprocs=args.num_gpus, args=(args,)).
```

Quindi ogni nodo chiama la funzione mp.spawn() per avviare un processo separato su ciascuna GPU disponibile.
Essa si occupa di:

1. **Lancio Multiplo di Processi**: Quando mp.spawn(worker, nprocs=args.num_gpus, args=(args,)) viene eseguito, crea un numero di processi pari a nprocs (nel nostro caso, uguale al numero di GPU specificato da args.num_gpus).

2. **Coordinamento della Comunicazione con DDP**: Ogni processo lanciato da mp.spawn() si connette al process group distribuito configurato con torch.distributed.init_process_group, che sincronizza i processi su tutte le GPU, anche tra nodi diversi.

3. **Sincronizzazione e Riduzione**: Con tutti i processi in esecuzione, DistributedDataParallel utilizza all-reduce per sincronizzare automaticamente i gradienti alla fine di ogni batch, come descritto in precedenza. Questo assicura che i parametri siano aggiornati correttamente su tutte le GPU.

Se eseguiamo un test lanciando l'esecuzione su un solo nodo:

```
python myscript.py --node-id 0 --num-gpus 4 --num-nodes 1  --epochs 3 --batch-size 512
```

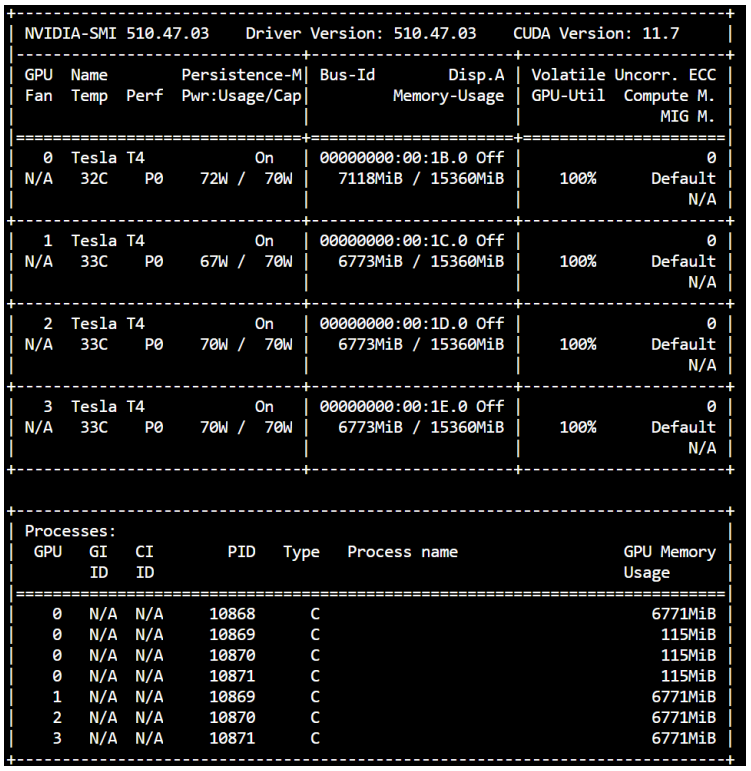
Otterremo il seguente output:

```
Epoch =  1: Cumulative Time = 24.508, Epoch Time = 24.508, Images/sec = 2423.3818359375, Validation Loss = 2.483, Validation Accuracy = 0.169
Epoch =  2: Cumulative Time = 48.048, Epoch Time = 23.540, Images/sec = 2501.712646484375, Validation Loss = 0.745, Validation Accuracy = 0.751
Epoch =  3: Cumulative Time = 71.689, Epoch Time = 23.641, Images/sec = 2512.25830078125, Validation Loss = 0.644, Validation Accuracy = 0.763
```

se andiamo a confrontare questo output con la Baseline che abbiamo ottenuto in precedenza attraverso l'esecuzione dell'addestramento su una sola GPU:

```
Epoch =  1: Cumulative Time = 91.131, Epoch Time = 91.131, Images/sec = 657.3424213456788, Validation Loss = 0.686, Validation Accuracy = 0.741
Epoch =  2: Cumulative Time = 182.929, Epoch Time = 91.798, Images/sec = 652.560768298225, Validation Loss = 0.612, Validation Accuracy = 0.781
Epoch =  3: Cumulative Time = 274.775, Epoch Time = 91.846, Images/sec = 652.2236577699774, Validation Loss = 0.486, Validation Accuracy = 0.826
Epoch =  4: Cumulative Time = 366.682, Epoch Time = 91.907, Images/sec = 651.7869326969766, Validation Loss = 0.415, Validation Accuracy = 0.852
Epoch =  5: Cumulative Time = 458.566, Epoch Time = 91.884, Images/sec = 651.9506140525963, Validation Loss = 0.378, Validation Accuracy = 0.864
Early stopping after epoch 5
```

possiamo osservare, che il tempo per processare ogni epoca è stato scalato in proporzione al numero di GPU, così come il numero di immagini processate per secondo. Le metriche ottenute alla prima epoca rispecchiano una situazione in cui ogni GPU converge a minimi locali per cui il valore mediato risulta pessimo, mentre già alla seconda epoca in cui interviene la fase di all-reduce, i valori migliorano notevolmente.
Rispetto ai consumi avremo nel primo caso una sola GPU per una potenza di 68W per un tempo di 274.755s. Pertanto nella fase di addestramento avremo un consumo di **0,247W/s**, mentre nel secondo caso, col calcolo parallelo avremo 4x70W = 280W per un tempo di 71,7s. In questo secondo scenario quindi il consumo è pari ad **3,9W/s**.



Conclusioni

Questo rapporto tecnico ha approfondito l'uso del **Distributed Data Parallel (DDP)** di PyTorch per ottimizzare l'addestramento di modelli di deep learning su larga scala, evidenziando le modalità per distribuire il carico su più GPU e migliorare il throughput del processo di training. Gli esperimenti hanno dimostrato come l'incremento del numero di GPU possa aumentare il throughput in modo quasi lineare.

Inoltre, il rapporto ha descritto i passaggi necessari per implementare un processo di addestramento distribuito, inclusi la configurazione di parametri come il **world size**, la gestione delle operazioni di **I/O**, il **partizionamento dei dati** e la **sincronizzazione tra i processi**. Questi elementi sono fondamentali per garantire un'**ottimizzazione efficiente del training** su infrastrutture multi-GPU.

I risultati di questo lavoro forniscono linee guida pratiche per l'uso delle risorse multi-GPU, favorendo un utilizzo efficiente e sostenibile delle infrastrutture di calcolo ad alte prestazioni in applicazioni avanzate di deep learning.

Negli sviluppi futuri verranno analizzate le metodologie di ottimizzazione avanzata, inclusi gli approcci di calibrazione del throughput e del batch size, per massimizzare le prestazioni, offrendo spunti ed esperimenti pratici che illustrano l'impatto delle tecniche implementate sulla velocità di addestramento e sull'accuratezza del modello.
Successivamente verranno analizzate e testate le tecniche si Parallelismo del modello.

Bibliografia

Kurth, T., Treichler, S., Romero, J., Mudigonda, M., Luehr, N., Phillips, E., ... & Houston, M. (2018, November). Exascale deep learning for climate analytics. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (p. 51). IEEE Press. arXiv:1810.01993

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. arXiv:1811.03600

You, Y., Zhang, Z., Hsieh, C., Demmel, J., & Keutzer, K. (2017). ImageNet training in minutes. arXiv: 1709.05011

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

Keskar, N. S., et al. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv:1609.04836

Li, H., Xu, Z., Taylor, G., & Goldstein, T. (2017). Visualizing the Loss Landscape of Neural Nets. arXiv:1712.09913

Hestness, J., et al. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv: 1712.00409

Zoph, Barret, et al. (2017). "Learning transferable architectures for scalable image recognition." arXiv: 1707.07012

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., & Uszkoreit, J. (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., & Keutzer, K. (2014). Densenet: Implementing efficient convnet descriptor pyramids. arXiv preprint arXiv:1404.1869.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*

Li, H., Xu, Z., Taylor, G., & Goldstein, T. (2017). Visualizing the Loss Landscape of Neural Nets. *arXiv:1712.09913*.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. arXiv:1404.5997

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677

Ioffe and Szegedy (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

Smith, S. L., Kindermans, P. J., & Le, Q. V. (2017). Don't Decay the Learning Rate, Increase the Batch Size. arXiv:1711.00489

You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. arXiv:1708.03888