Adjoint MPI for Non-Additive Separable Loss Functions

Kai Krajsek (k.krajsek@fz-juelich.de)
SDL Neuroscience, Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich GmbH
52425 Jülich, Germany

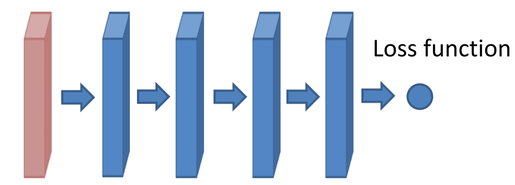
Abstract

- Most Deep Learning parallelization frameworks [1,2] follow a strict design pattern that favours a specific parallelization paradigm
- Existing communication primitives do not cover the MPI standard and in most cases do not provide automatic differentiation (E.g., PyTorch's RPC-based distributed prototype [3] offers only automatic differentiation for P2P communication but not for collective communication)
- This poster:
  - Presents adjoint MPI concepts [4] in the context of Deep Learning (DL) for distributed training
  - Introduces an approach for preserving MPI operation orders in the backward pass and for integrating MPI operations into the computational graph
  - Demonstrates the usability of the adjoint MPI concept to non-additive separable loss functions as used in contrastive learning

1. Automatic Differentiation

- Adjoint mode automatic differentiation (AD) combines partial derivatives starting from the neural network output
- Adjoint variables are computed for each forward variable representing the partial derivative of the output variable w.r.t. the forward variable
- Requires keeping track of forward variables, e.g., by Directed Acyclic Graphs (DAG)

$$y = f_5(f_4(f_3(f_2(f_1(x_0)))))$$



Backpropagation: Gradients are calculated efficiently by the chain rule, starting with the last computation

$$\frac{\partial L}{\partial w_1} = J_1^T J_2^T J_3^T J_4^T J_5^T \frac{\partial L}{\partial y}$$

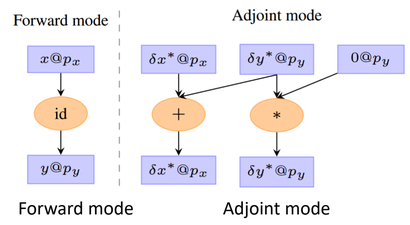
$$J_{f_m} := J_m := \left(\frac{\partial f_i}{\partial x_j}\right)_{i=1,\dots,m;j=1,\dots,n}$$

2. Adjoint MPI

Adjoint MPI considers a subset of MPI calls (e.g., one sided communication is excluded) as differentiable transformation and identifies their adjoint counterpart

MPI Functions	Adjoint MPI Functions
MPI_Bcast	MPI_Reduce
MPI_Reduce	MPI_Bcast
MPI_Gather	MPI_Scatter
MPI_Allreduce	MPI_Allreduce
MPI_Iallreduce + MPI_Wait	MPI_Wait + MPI_Iallreduce
MPI_Recv	MPI_Send
MPI_Send	MPI_Recv
MPI_Isend + MPI_Wait	MPI_Wait + MPI_Irecv
MPI_Irecv + MPI_Wait	MPI_Wait + MPI_Isend

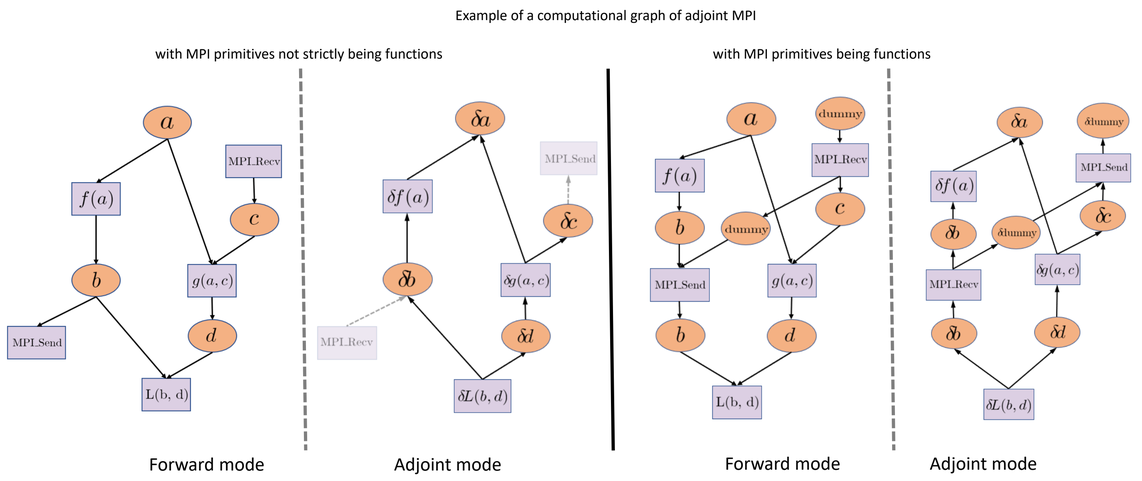
MPI functions at their adjoint counterparts



Graph representation of MPI_Send and its adjoint counterpart

Adjoint MPI operations need to be attached to the computational graph and strictly applied in reverse order of their forward counterparts to avoid deadlocks:

- Consecutive MPI functions must be made dependent on each other to assure order preserving in the backward pass
- All MPI primitives must be functions in a mathematical sense



Acknowledgements:
The author wants to thank Dr. Phillipp Knechtges (German Aerospace Center (DLR), Institute for Software Technology, High-Performance Computing) for discussing the adjoint MPI concept and Dr. Hanno Scharr (IAS-8, Forschungszentrum Jülich GmbH) about exchange on application of adjoint MPI in Deep Learning.

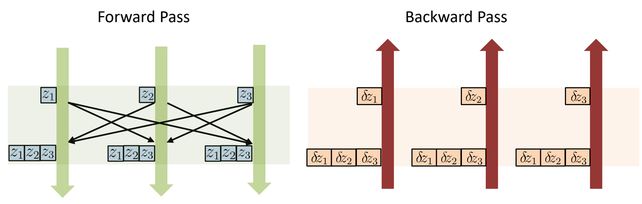
4. Distributed Contrastive Learning

Contrastive learning approaches, e.g., SimCRL [5] or CLIP [6] involve non-additive separable loss functions, i.e., they are not linear in data points, e.g.:

SimCLR Loss: $$\ell_{i,j} = -\log \frac{\exp(\mathrm{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]} \exp(\mathrm{sim}(z_i, z_k)/\tau)}$$

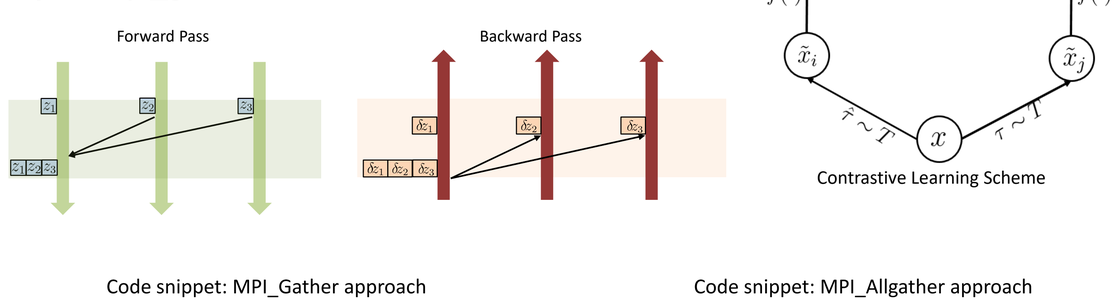
➡ Full loss function needs to be computed before backpropagation

MPI_Allgather Approach (no_grad=True):
Apply MPI_Allgather aggregating all network outputs on all nodes, compute the loss function on them and run backprop on each node independently



Contrastive learning teaches a model to compare and identify differences or similarities between data samples to learn meaningful representations of the data.

MPI_Gather Approach:
If aggregated data is too large for computation devices it can be gathered on a device with larger memory via adjoint mpi_gather



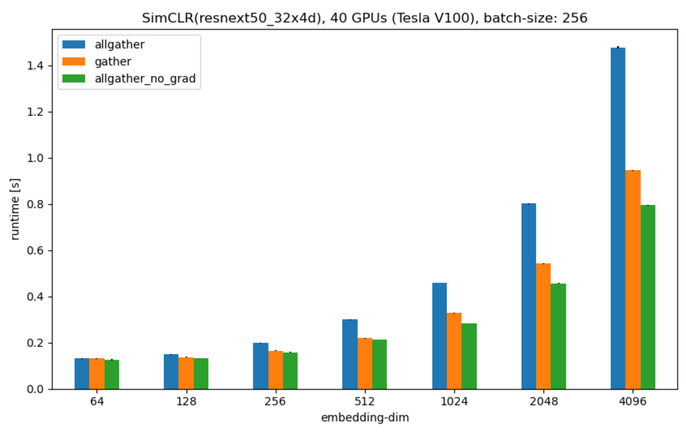
Code snippet: MPI_Gather approach

```
for xiter in range(len(train_loader)):
    train_iter = iter(train_loader)
    if comm.rank < comm.size-1:
        image, text = next(train_iter)
        image, text = image.to(device), text.to(device)

        image_features = model.encode_image(image)
        text_features = model.encode_text(text)

        x1, x2 = model(image_features, text_features)
        x = torch.cat((x1, x2), 0)

    if comm.rank == comm.size-1: # rank comm.size-1 computes the loss function
        x = torch.randn(2*args.batch_size, args.feature_dim, requires_grad=True).to(device)

    output = mpi.MPIGather(x, root=comm.size-1, comm=comm)
    output1, output2 = output.chunk(2)

    if comm.rank == comm.size-1:
        # apply single node CLIP loss function
        loss = cliploss(output1, output2, args.temperature)

        loss.backward()

    if comm.rank != comm.size-1:
        # envoce backward pass on mean of dummy variables
        (output1 + output2).mean().backward()

        optimizer.step()
```

Code snippet: MPI_Allgather approach

```
for xiter in range(len(train_loader)):
    train_iter = iter(train_loader)

    # restrict computation to ddp
    if comm.rank < comm.size-1:
        image, text = next(train_iter)
        image, text = image.to(device), text.to(device)

        image_features = model.encode_image(image)
        text_features = model.encode_text(text)

        #apply model
        x1, x2 = model(image_features, text_features)
        x = torch.cat((x1, x2), 0)

        # gather all model outputs on all ddp ranks
        output = mpi.MPIAllgather(x, comm=comm_ddp, no_grad=True)
        output1, output2 = output.chunk(2)

        # apply CLIP loss on all ddp ranks
        loss = cliploss(output1, output2, args.temperature)

        # envoce backward pass on all ddp ranks
        loss.backward()

        optimizer.step()
```

Experiment: The MPI_Allgather and MPI_Gather approaches have been tested with the SimCLR model [5] parallelized with PyTorch's Distributed Data Parallel (DDP) module on a machine learning HPC system at the Jülich Supercomputing Centre.
(4 NVIDIA Tesla V100 SXM2 GPUs with 32GB VRAM per card, 15 nodes in total, The GPUs communicate node-internally via an NVLink interconnect. The system is optimized for GPUDirect communication across node boundaries, supported by 2x Mellanox 100 Gbit EDR InfiniBand links)
For the experiment 40 GPUs were used, whereas one GPU was held back for the loss function computation.



Experimental setup: The runtime was measured by applying the Resnext50_32x4d model to two views of images sized 128x128 with a batch size of 256 and varying embedding dimensions. The resulting output was fed into the SimCLR loss function. Different strategies were compared, including MPI_Allgather with backpropagation, MPI_Allgather without backpropagation, and MPI_Gather with backpropagation. The runtime has been averaged over 100 iterations.

References:
[1] Sergeev, A. and Balso, M.D., (2018). Horovod: Fast and easy distributed deep learning in TensorFlow. In *arXiv preprint arXiv:1802.05799*
[2] Y. Huang et al., (2018). Pipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *arXiv preprint arXiv: 1811.06965*
[3] Paszke, A. et al., (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (pp. 8024-8035)
[4] Utke, J., et al. (2009). Toward adjoinable MPI. In *IEEE International Symposium on Parallel & Distributed Processing*, pp. 1-8
[5] Chen, T., et al. (2020). A Simple Framework for Contrastive Learning of Visual Representations, *Proceedings of the 37th International Conference on Machine Learning*, PMLR, (pp. 1597-1607)
[6] Radford, A. et al. (2021). Learning Transferable Visual Models From Natural Language Supervision, Proceedings of the 38th International Conference on Machine Learning, ICML, (pp. 8748-8763)