Boosting Performance of Deep Learning, Machine Learning, and Dask with MVAPICH2

Tutorial at MUG ’21

by

Arpan Jain
The Ohio State University
jain.575@osu.edu
http://u.osu.edu/jain.575

Aamir Shafi
The Ohio State University
shafi.16@osu.edu
https://cse.osu.edu/people/shafi.16

Quentin Anthony
The Ohio State University
anthony.301@osu.edu
https://www.linkedin.com/in/quentin-anthony

Follow us on
https://twitter.com/mvapich
Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• Parallelization Strategies for Distributed DNN Training
• Machine Learning
• Data Science using Dask
• Conclusion
**What is Machine Learning and Deep Learning?**

- **Machine Learning (ML)**
  - “the study of computer algorithms to improve automatically through experience and use of data”

- **Deep Learning (DL) – a subset of ML**
  - Uses Deep Neural Networks (DNNs)
  - **Perhaps, the most revolutionary subset!**

- Based on learning data representation

- **DNN Examples:** Convolutional Neural Networks, Recurrent Neural Networks, Hybrid Networks

- Data Scientist or Developer Perspective for using DNNs
  1. Identify DL as solution to a problem
  2. Determine Data Set
  3. Select Deep Learning Algorithm to Use
  4. Use a large data set to train an algorithm

---

**Courtesy:**
History: Milestones in the Development of ML/DL

1800 - 1900
- Linear Regression
- PCA
- Turing Machine

1940 - 1950
- Electronic Brain
- Perceptron
- ADALINE
- K-Means

1960 - 1970
- XOR Problem
- Golden Age
- Multi-layered Perceptron (Backpropagation)

1980 - 1990
- Dark Age (“AI Winter”)
- Multi-layered Perceptron (Backpropagation)

2000 - 2010
- AlexNet
- DBN
- ResNet
- WGAN

2010 - 2020
- DBN
- ResNet
- Transformers
- AlexNet
- WGAN

1800 - 1900
- Linear Regression
- PCA
- Turing Machine

1901 - 1960
- Evolutionary Algorithms
- K-Means
- Bayesian Network

1964 - 1980
- Bayesian Network
- Decision Trees
- SVM
- XGBoost

1980 - 1990
- Multi-layered Perceptron (Backpropagation)
- Evolutionary Algorithms
- K-Means

1990 - 2000
- KNN
- SVM
- Random Forest
- XGBoost

2000 - 2020
- Deep Forest
- XGBoost
- CatBoost

Key Scientists:
- A. Legrendre - J. Gauss
- K. Pearson
- A. Turing
- S. McCulloch - W. Pitts
- F. Rosenblatt
- B. Widrow - M. Hoff
- M. Minsky - S. Papert
- J. Pearl
- D. Rumelhart - G. Hinton - R. Williams
- V. Vapnik - C. Cortes
- A. Krizhevsky
- A. Ng
- Y. LeCun
- Y. Bengio

Network Based Computing Laboratory
MUG ‘21
Deep Learning meets Super Computers

• Computation requirement is increasing exponentially
  – CPUs still dominates HPC arena and can be used for Deep Learning

Computation requirement is increasing exponentially

- CPUs still dominates HPC arena and can be used for Deep Learning

Courtesy: https://openai.com/blog/ai-and-compute/
Outline

• Introduction

• **Deep Neural Network Training and Essential Concepts**
  • Parallelization Strategies for Distributed DNN Training
  • Machine Learning
  • Data Science using Dask
  • Conclusion
So what is a Deep Neural Network?

- Example of a 3-layer Deep Neural Network (DNN) – (input layer is not counted)

![Diagram of a 3-layer Deep Neural Network](http://cs231n.github.io/neural-networks-1/)

DNN Training: Forward Pass

Input Layer

Hidden Layer

Hidden Layer

Output Layer
DNN Training: Forward Pass

Forward Pass

Input Layer

Hidden Layer

Hidden Layer

Output Layer

\[ X \rightarrow w_1 \rightarrow \text{Hidden Layer} \rightarrow w_1 \rightarrow \text{Output Layer} \]
DNN Training: Forward Pass

Forward Pass

Input Layer

Hidden Layer

Hidden Layer

Output Layer
DNN Training: Forward Pass

Input Layer

Hidden Layer

Hidden Layer

Output Layer

Forward Pass

X

\( W_1 \)

\( W_2 \)

\( W_3 \)

\( W_4 \)

\( W_5 \)

\( W_6 \)

\( W_7 \)

\( W_8 \)
DNN Training: Forward Pass

Forward Pass

\[ \text{Error} = \text{Loss}(\text{Pred}, \text{Output}) \]
DNN Training: Backward Pass

Forward Pass

Input Layer  Hidden Layer  Hidden Layer  Output Layer

Backward Pass

Error = Loss(Pred, Output)
DNN Training: Backward Pass

Forward Pass

Backward Pass

\[ \text{Error} = \text{Loss} (\text{Pred}, \text{Output}) \]
DNN Training: Backward Pass

\[
\text{Error} = \text{Loss} (\text{Pred}, \text{Output})
\]
DNN Training

Input Layer

Hidden Layer

Hidden Layer

Output Layer

X
Essential Concepts: Activation function and Back-propagation

- Back-propagation involves complicated mathematics.
  - Luckily, most DL Frameworks give you a one line implementation -- `model.backward()`

![Diagram of neural network](https://www.jeremyjordan.me/neural-networks-training/)

- What are Activation functions?
  - RELU (a Max fn.) is the most common activation fn.
  - Sigmoid, tanh, etc. are also used

Courtesy: [https://www.jeremyjordan.me/neural-networks-training/](https://www.jeremyjordan.me/neural-networks-training/)
Essential Concepts: Learning Rate ($\alpha$)

Too low

A small learning rate requires many updates before reaching the minimum point

Just right

The optimal learning rate swiftly reaches the minimum point

Too high

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Courtesy: https://www.jeremyjordan.me/nn-learning-rate/
Essential Concepts: Batch Size

- Batched Gradient Descent
  - Batch Size = $N$

- Stochastic Gradient Descent
  - Batch Size = 1

- Mini-batch Gradient Descent
  - Somewhere in the middle
  - Common:
    - Batch Size = 64, 128, 256, etc.

- Finding the optimal batch size will yield the fastest learning.

One full pass over $N$ is called an *epoch* of training.

Courtesy: https://www.jeremyjordan.me/gradient-descent/
Key Phases of Deep Learning

• Training is compute intensive
  – Many passes over data
  – Can take days to weeks
  – Model adjustment is done

• Inference
  – Single pass over the data
  – Should take seconds
  – No model adjustment

• Challenge: How to make “Training” faster?
  – Need Parallel and Distributed Training...

Courtesy: https://devblogs.nvidia.com/
Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• Parallelization Strategies for Distributed DNN Training
  • Machine Learning
  • Data Science using Dask
• Conclusion
Parallelization Strategies

• Some parallelization strategies...
  – Data Parallelism
  – Model Parallelism
  – Hybrid Parallelism

Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• Parallelization Strategies for Distributed DNN Training
  – Data Parallelism
  – Model and Hybrid Parallelism
• Machine Learning
• Data Science using Dask
• Conclusion
**Need for Data Parallelism**

Let’s revisit Mini-Batch Gradient Descent

**Drawback:** If the dataset has 1 million images, then it will take forever to run the model on such a big dataset

**Solution:** Can we use multiple machines to speedup the training of Deep learning models? *(i.e. Utilize Supercomputers to Parallelize)*
Need for Communication in Data Parallelism

Machine 1

Machine 2

Machine 3

Machine 4

Machine 5

Problem: Train a single model on whole dataset, not 5 models on different sets of dataset
Data Parallelism

Machine 1
Machine 2
Machine 3
Machine 4
Machine 5

Gradients

Reduced Gradients

MPI
AllReduce
Allreduce Collective Communication Pattern

- Element-wise Sum data from all processes and sends to all processes

```
int MPI_Allreduce (const void *sendbuf, void * recvbuf, int count, MPI_Datatype datatype,
                   MPI_Op operation, MPI_Comm comm)
```

### Input-only Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>Starting address of send buffer</td>
</tr>
<tr>
<td>recvbuf</td>
<td>Starting address of recv buffer</td>
</tr>
<tr>
<td>type</td>
<td>Data type of buffer elements</td>
</tr>
<tr>
<td>count</td>
<td>Number of elements in the buffers</td>
</tr>
<tr>
<td>operation</td>
<td>Reduction operation to be performed (e.g. sum)</td>
</tr>
<tr>
<td>comm</td>
<td>Communicator handle</td>
</tr>
</tbody>
</table>

### Input/Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recvbuf</td>
<td>Starting address of receive buffer</td>
</tr>
</tbody>
</table>

**Sendbuf (Before)**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Recvbuf (After)**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
Data Parallelism and MPI Collectives

• **Step 1:** Data Propagation
  - Distribute the data among GPUs

• **Step 2:** Forward Backward Pass
  - Perform forward pass and calculate the prediction
  - Calculate error by comparing prediction with actual output
  - Perform backward pass and calculate gradients

• **Step 3:** Gradient Aggregation
  - Call MPI_Allreduce to reduce the local gradients
  - Update parameters locally using global gradients
MVAPICH2 (MPI)-driven Infrastructure for ML/DL Training

MVAPICH2 or MVAPICH2-X for CPU Training

MVAPICH2-GDR for GPU Training

Horovod

TensorFlow
PyTorch
MXNet

Torch.distributed
DeepSpeed

PyTorch

More details available from: http://hidl.cse.ohio-state.edu
MVAPICH2 2.3.6

- **Released on 05/11/2021**
- **Major Features and Enhancements**
  - Support collective offload using Mellanox’s SHARP for Reduce and Bcast
    - Enhanced tuning framework for Reduce and Bcast using SHARP
  - Enhanced performance for UD-Hybrid code
  - Add multi-rail support for UD-Hybrid code
  - Enhanced performance for shared-memory collectives
  - Enhanced job-startup performance for flux job launcher
  - Add support in mpirun_rsh to use srun daemons to launch jobs
  - Add support in mpirun_rsh to specify processes per node using ‘-ppn’ option
  - Use PMI2 by default when SLURM is selected as process manager
  - Add support to use aligned memory allocations for multi-threaded applications
  - Architecture detection and enhanced point-to-point tuning for Oracle BM.HPC2 cloud shape
  - Enhanced collective tuning for Frontera@TACC and Expanse@SDSC
  - Add support for GCC compiler v11
  - Add support for Intel IFX compiler
  - Update hwloc v1 code to v1.11.14 & hwloc v2 code to v2.4.2
MVAPICH2-GDR 2.3.6

- Released on 08/12/2021
- Major Features and Enhancements
  - Based on MVAPICH2 2.3.6
  - Added support for 'on-the-fly' compression of point-to-point messages used for GPU-to-GPU communication
    • Applicable to NVIDIA GPUs
  - Added NCCL communication substrate for various MPI collectives
    • Support for hybrid communication protocols using NCCL-based, CUDA-based, and IB verbs-based primitives
    • MPI_Allreduce, MPI_Reduce, MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv, MPI_Scatter, MPI_Scatterv, MPI_Gather, MPI_Gatherv, and MPI_Bcast
  - Full support for NVIDIA DGX, NVIDIA DGX-2 V-100, and NVIDIA DGX-2 A-100 systems
    • Enhanced architecture detection, process placement and HCA selection
    • Enhanced intra-node and inter-node point-to-point tuning
    • Enhanced collective tuning
  - Introduced architecture detection, point-to-point tuning and collective tuning for ThetaGPU @ANL
  - Enhanced point-to-point and collective tuning for NVIDIA GPUs on Frontera @TACC, Lassen @LLNL, and Sierra @LLNL
  - Enhanced point-to-point and collective tuning for Mi50 and Mi60 AMD GPUs on Corona @LLNL
  - Added several new MPI_T PVARs
  - Added support for CUDA 11.3
  - Added support for ROCm 4.1
  - Enhanced output for runtime variable MV2_SHOW_ENV_INFO
  - Tested with Horovod and common DL Frameworks
    • TensorFlow, PyTorch, and MXNet
  - Tested with MPI4Dask 0.2
    • MPI4Dask is a custom Dask Distributed package with MPI support
  - Tested with MPI4cuML 0.1
    • MPI4cuML is a custom cuML package with MPI support
- Tested with MPI4Dask 0.2
  • MPI4Dask is a custom Dask Distributed package with MPI support
  • MPI4cuML is a custom cuML package with MPI support
Install Horovod with MVAPICH2-X and MVAPICH2-GDR

Command to install Horovod with MVAPICH2-X

$ HOROVOD_WITH_MPI=1 pip install --no-cache-dir horovod==0.22.1

Command to install Horovod with MVAPICH2-GDR

$ HOROVOD_GPU_ALLREDUCE=MPI HOROVOD_CUDA_HOME=/opt/cuda/10.1 HOROVOD_WITH_MPI=1 pip install --no-cache-dir horovod
Run PyTorch on a single GPU

+ python `pytorch_synthetic_benchmark.py` --batch-size 64 --num-ites=5

Model: resnet50
Batch size: 64
Number of GPUs: 1
Running warmup...
Running benchmark...
Iter #0: 333.9 img/sec per GPU
Iter #1: 334.2 img/sec per GPU
Iter #2: 333.9 img/sec per GPU
Iter #3: 333.8 img/sec per GPU
Iter #4: 333.9 img/sec per GPU
Img/sec per GPU: 334.0 ±0.2

Total img/sec on 1 GPU(s): 334.0 ±0.2
Run PyTorch on two nodes with 1 GPU/node (using MVAPICH2-GDR)

+ mpirun_rsh -np 2 gpu11 gpu12 MV2_USE_CUDA=1 MV2_CPU_BINDING_POLICY=hybrid MV2_HYBRID_BINDING_POLICY=spread MV2_USE_RDMA_CM=0 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so LD_PRELOAD=mv2-gdr/lib/libmpi.so python pytorch_synthetic_benchmark.py --batch-size 64 --num-iter=5

Model: resnet50
Batch size: 64
Number of GPUs: 2
Running warmup...
Running benchmark...
Iter #0: 317.0 img/sec per GPU
Iter #1: 314.9 img/sec per GPU
Iter #2: 315.4 img/sec per GPU
Iter #3: 318.0 img/sec per GPU
Iter #4: 316.7 img/sec per GPU
Img/sec per GPU: 316.4 +/- 2.2

-----------------------------------------
Total img/sec on 2 GPU(s): 632.8 +/- 4.3
-----------------------------------------

~1.89X on 2 GPUs
Scalable TensorFlow using Horovod and MVAPICH2-GDR

- ResNet-50 Training using TensorFlow benchmark on 1 DGX-2 node (16 Volta GPUs)

**Platform: Nvidia DGX-2 system, CUDA 9.2**

**Scaling Efficiency** = \( \frac{\text{Actual throughput}}{\text{Ideal throughput at scale}} \times 100\% \)

Distributed TensorFlow on ORNL Summit (1,536 GPUs)

- ResNet-50 Training using TensorFlow benchmark on SUMMIT -- 1536 Volta GPUs!
- 1,281,167 (1.2 mil.) images
- Time/epoch = 3 seconds
- Total Time (90 epochs) = 3 x 90 = 270 seconds = 4.5 minutes!

*We observed issues for NCCL2 beyond 384 GPUs

Platform: The Summit Supercomputer (#2 on Top500.org) – 6 NVIDIA Volta GPUs per node connected with NVLink, CUDA 10.1
Distributed TensorFlow on TACC Frontera (2048 CPU nodes)

- Scaled TensorFlow to 2048 nodes on Frontera using MVAPICH2 and IntelMPI

- MVAPICH2 and IntelMPI give similar performance for DNN training

- Report a peak of **260,000 images/sec** on 2048 nodes

- On 2048 nodes, ResNet-50 can be trained in **7 minutes**!

Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• **Parallelization Strategies for Distributed DNN Training**
  – Data Parallelism
  – **Model and Hybrid Parallelism**
• Machine Learning
• Data Science using Dask
• Conclusion
HyPar-Flow: Hybrid Parallelism for TensorFlow

- Why Hybrid parallelism?
  - Data Parallel training has limits! →
- We propose HyPar-Flow
  - An easy to use Hybrid parallel training framework
    - Hybrid = Data + Model
  - Supports Keras models and exploits TF 2.0 Eager Execution
  - Exploits MPI for Point-to-point and Collectives

Benchmarking large-models lead to better insights and ability to develop new approaches!

Model/Hybrid Parallelism and MPI Collectives

- HyPar-Flow is practical (easy-to-use) and high-performance (uses MPI)
  - Based on Keras models and exploits TF 2.0 Eager Execution
  - Leverages MPI Pt-to-pt. and Collectives for communication

HyPar-Flow at Scale (512 nodes on TACC Frontera)

- ResNet-1001 with variable batch size
- Approach:
  - 48 model-partitions for 56 cores
  - 512 model-replicas for 512 nodes
  - Total cores: 48 x 512 = 24,576
- Speedup
  - 253X on 256 nodes
  - 481X on 512 nodes
- Scaling Efficiency
  - 98% up to 256 nodes
  - 93.9% for 512 nodes

Why do we need Memory aware designs?

- Data and Model Parallel training has limitation!

- Maximum Batch Size depends on the memory.

- Basic Model Parallelism suffers from underutilization of memory and compute →

**GEMS: GPU Enabled Memory Aware Model Parallelism Systems**

Memory requirement increases with the increase in image size!

Exploiting Model Parallelism in AI-Driven Digital Pathology

- Pathology whole slide image (WSI)
  - Each WSI = 100,000 x 100,000 pixels
  - Can not fit in a single GPU memory
  - Tiles are extracted to make training possible

- Two main problems with tiles
  - Restricted tile size because of GPU memory limitation
  - Smaller tiles loose structural information

- Can we use Model Parallelism to train on larger tiles to get better accuracy and diagnosis?

- Reduced training time significantly
  - 7.25 hours (1 node, 4 GPUs) -> 27 mins (32 nodes, 128 GPUs)


SUPER: **SUb-Graph Parallelism for TransformERs**

Sub-Graph Parallelism

- Exploits inherent parallelism in modern DNN architectures
- Improves the Performance of multi-branch DNN architectures
- Can be used to accelerate the training of state-of-the-art Transformer models
- Provides better than Data-Parallelism for in-core models

**Simple example of a multi-branch DNN architecture**

**4-way Sub-Graph Parallelism combined with Data-Parallelism (D&SP)**

Accelerating Transformers using SUPER

- We propose sub-graph parallelism integrated with data parallelism to accelerate the training of Transformers.
- Approach
  - Data and Sub-Graph Parallelism (D&SP)
    - #-way D&SP (#: number of sub-graphs)
- Setup
  - T5-Large-Mod on WMT Dataset
  - 1024 NVIDIA V100 GPUs
- Speedup
  - Up to **3.05X** over Data Parallelism (DP)

Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• Parallelization Strategies for Distributed DNN Training

• Machine Learning
• Data Science using Dask
• Conclusion
Parallelizing K-Means Clustering

Domain Decomposition – Divide data objects amongst P processors

Recompute centroids after MPI_Allreduce

**Courtesy:** https://github.com/tmscarla/k-means-parallel  http://users.eecs.northwestern.edu/~wkliao/Kmeans/
Support for Parallel and Distributed Execution

• Scikit-learn:
  – Joblib supports multi-threaded and multi-process execution (on multiple nodes)

• XGBoost:
  – Multiple ways to run on cluster of nodes:
    • Dask (http://dask.org)
    • Ray (https://ray.io/)
    • AWS YARN
    • Apache Spark (https://spark.apache.org/) using XGBoost4J-Spark

• cuML:
  – Execution is supported on multiple nodes using Dask (http://dask.org) and NVIDIA’s NCCL
The cuML Library: Accelerating ML on GPUs

• The NVIDIA RAPIDS project aims to build end-to-end data science analytic pipelines on GPUs

• An important component is the cuML library:
  – GPU-accelerated ML library
  – GPU-counterpart of Scikit-learn
  – Supports the execution of ML workloads on Multi-Node Multi-GPUs (MNMG) systems

• Most existing ML libraries, including Scikit-learn and Apache Spark’s MLlib, only support CPU execution of ML algorithms
  – Conventional wisdom has been that only DNNs are a good match for GPUs because of high computational requirements
Main components of the cuML library

- Main components
  - Python layer
    - Provides a Scikit-learn like interface
    - Hides the complexities of the CUDA/C/C++ layer
  - Primitives and cuML algorithms built on top of CUDA
    - ML Algorithms
      - Primitives
        - Reusable building blocks for building machine learning algorithms
        - Common for different machine learning algorithms
        - Used to build different machine learning algorithms
    - Communication Support in cuML:
      - Point-to-point communication: Dask
      - Collective communication: NVIDIA Collective Communications Library (NCCL)
Parallel and Distributed Training in cuML

- **Distributed training**
  - Using data parallelism, the fit() function is executed on all workers containing a partition of the training dataset

- **Trained Model:**
  - The trained parameters are brought to a single worker using MPI_Reduce()

- **Inference Stage:**
  - The trained parameters are broadcasted to all workers with prediction partitions

Accelerating cuML with MVAPICH2-GDR

- Utilize MVAPICH2-GDR (with mpi4py) as communication backend during the training phase (the fit() function) in the Multi-node Multi-GPU (MNMG) setting over cluster of GPUs
- Communication primitives:
  - Allreduce
  - Reduce
  - Broadcast
- Exploit optimized collectives
MPI4cuML Release

- MPI4cuML 0.1 was released in Feb ‘21 adding support for MPI to cuML:
  - Can be downloaded from: http://hidl.cse.ohio-state.edu

- Features:
  - Based on cuML 0.15
  - MVAPICH2 support for C++ and Python APIs
    - Included use of cuML C++ CUDA-Aware MPI example for KMeans clustering
    - Enabled cuML handles to use MVAPICH2-GDR backend for Python cuML applications
      - KMeans, PCA, tSVD, RF, LinearModels
    - Added switch between available communication backends (MVAPICH2 and NCCL)
  - Built on top of mpi4py over the MVAPICH2-GDR library
  - Tested with
    - Mellanox InfiniBand adapters (FDR and HDR)
    - Various x86-based multi-core platforms (AMD and Intel)
    - NVIDIA V100 and P100 GPUs
MPI4cuML Installation

- MPI4cuML is available to download from: [http://hidl.cse.ohio-state.edu/](http://hidl.cse.ohio-state.edu/)
  - The userguide is available at: [http://hidl.cse.ohio-state.edu/download/hidl/mpi4cuml/mpi4cuml-userguide.pdf](http://hidl.cse.ohio-state.edu/download/hidl/mpi4cuml/mpi4cuml-userguide.pdf)

- Setup Instructions
  - Installation Pre-requisites:
    - Install the MVAPICH2-GDR Library
    - Install the mpi4py Library
  - Install MPI4cuML

- Running GPU-based cuML Applications
  - Writing the host file
  - KMeans on real and synthetic data
Setup Instructions: Pre-requisites

Install the MVAPICH2-GDR Library

Install the mpi4py Library

$ git clone https://github.com/mpi4py/mpi4py.git
$ cd mpi4py
$ edit mpi.cfg file
# MVAPICH2
# -----------------
[MVAPICH2]
mpi_dir = /path/to/MVAPICH2-GDR/install/directory
mpicc = %(mpi_dir)s/bin/mpicc
mpicxx = %(mpi_dir)s/bin/mpicxx
include_dirs = %(mpi_dir)s/include
library_dirs = %(mpi_dir)s/lib64
runtime_library_dirs = %(library_dirs)s

$ python setup.py build --mpi=MVAPICH2-GDR
$ pip install .
Setup Instructions: Install MPI4cuML

Install MPI4cuML

$ wget http://hidl.cse.ohio-state.edu/download/hidl/cuml/mpi4cuml-0.1.tar.gz
$ tar -xvf mpi4cuml-0.1.tar.gz
$ cd mpi4cuml-0.1/
$ conda env update -n mpi4cuml --file=conda/environments/cuml_dev_cuda10.2.yml
$ export LIBRARY_PATH=/path/to/miniconda3/envs/mpi4cuml/lib:$LIBRARY_PATH
$ export LD_LIBRARY_PATH=/path/to/miniconda3/envs/mpi4cuml/lib:$LIBRARY_PATH
$ ./build.sh cpp-mgtests

$ conda list
$ conda list | grep cuml

Build KMeans

$ cmake .. -DCUML_LIBRARY_DIR=/path/to/directory/with/libcuml.so
    -DCUML_INCLUDE_DIR=/path/to/directory/with/kmeans/kmeans_c.h
$ make
$ LD_PRELOAD=$MPILIB/lib64/libmpi.so:$CUML_HOME/cpp/build/libcuml++.so
Running MPI4cuML Applications

Writing the host file

$ cd mpi4cuml/cpp/examples/mg_kmeans
$ vim hosts
.. write name of the compute nodes ..

Run KMeans with synthetic data

$ MPILIB/bin/mpirun_rsh -export-all -np 4 -hostfile hosts MV2_USE_CUDA=1
   MV2_USE_GDRCOPY=1 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so
./kmeans_mg_example

Run KMeans with real dataset

.. Download data from https://www.kaggle.com/c/homesite-quote-conversion/data ..
$ unzip all.zip
$ ./prepare_input.py [train_file=train.csv] [test_file=test.csv] [output=output.txt]
$ LD_PRELOAD=$MV2_HOME/lib/libmpi.so:$CUML_HOME/cpp/build/libcuml++.so
$ MV2_HOME/bin/mpirun_rsh --export-all -np 4 -hostfile=hosts MV2_USE_CUDA=1
   MV2_USE_GDRCOPY=1 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so
./kmeans_mg_example -num_rows 260753 -num_cols 298 -input output.txt
Run K-Means at C++ Layer on one node with 1 GPU/node (using MVAPICH2-GDR)

$ sbatch -N 1 run_cuml_kmeans_single.sh

+ time /opt/tutorials/dl-tutorial-21/mv2-gdr/bin/mpirun_rsh -np 1 gpu04 MV2_USE_CUDA=1 MV2_SUPPORT_DL=1 /opt/tutorials/dl-tutorial-21/cuML/mpi4cuml-0.1/cpp/examples/mg_kmeans/kmeans_mg_example -num_rows 260753 -num_cols 298 -input data.txt -k 100 -max_iterations=3000

Running on 1 GPU(s)
Reading input with 260753 rows and 298 columns from ../data.txt.
Run KMeans with k=100, **max_iterations=3000**

<table>
<thead>
<tr>
<th>num_pts</th>
<th>inertia</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.096572e+11</td>
</tr>
<tr>
<td>1</td>
<td>1.245608e+11</td>
</tr>
<tr>
<td>2</td>
<td>1.362806e+11</td>
</tr>
<tr>
<td>3</td>
<td>1.267454e+11</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>1.048909e+11</td>
</tr>
<tr>
<td>99</td>
<td>1.310442e+11</td>
</tr>
</tbody>
</table>

Global inertia = 1.246384e+13

**real** 0m31.455s
**user** 0m0.019s
**sys** 0m0.008s
Run K-Means at C++ Layer on two nodes with 1 GPU/node (using MVAPICH2-GDR)

$ sbatch -N 1 run_cuml_kmeans_single.sh

+ time /opt/tutorials/dl-tutorial-21/mv2-gdr/bin/mpirun_rsh -np 2 gpu05 gpu04 MV2_USE_CUDA=1 MV2_SUPPORT_DL=1 /opt/tutorials/dl-tutorial-21/cuML/mpi4cuml-0.1/cpp/examples/mg_kmeans/kmeans_mg_example -num_rows 260753 -num_cols 298 -input data.txt -k 100 -max_iterations=3000

Running on 2 GPU(s)

Reading input with 260753 rows and 298 columns from ../data.txt.

Run KMeans with k=100, max_iterations=6000

<table>
<thead>
<tr>
<th>num_pts</th>
<th>inertia</th>
<th>inertia</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2466 1.096572e+11</td>
<td>2466 1.096572e+11</td>
</tr>
<tr>
<td>1</td>
<td>2638 1.245608e+11</td>
<td>2638 1.245608e+11</td>
</tr>
<tr>
<td>2</td>
<td>2739 1.362806e+11</td>
<td>2739 1.362806e+11</td>
</tr>
<tr>
<td>3</td>
<td>2659 1.267454e+11</td>
<td>2659 1.267454e+11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>98</td>
<td>2343 1.048909e+11</td>
<td>2343 1.048909e+11</td>
</tr>
<tr>
<td>99</td>
<td>2687 1.310442e+11</td>
<td>2687 1.310442e+11</td>
</tr>
</tbody>
</table>

Global inertia = 1.246384e+13

real 0m33.078s
user 0m0.035s
dsys 0m0.017s
Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• Parallelization Strategies for Distributed DNN Training
• Machine Learning
• **Data Science using Dask**
• Conclusion
Introduction to Dask

- Dask is a popular task-based distributed computing framework:
  - Scales Python applications from laptops to high-end systems
  - Builds a task-graph that is executed lazily on parallel hardware
  - Natively extends popular data processing libraries like numPy, Pandas

- Dask Distributed library supports parallel and distributed execution:
  - Built using the asyncio package that allows execution of asynchronous/non-blocking/concurrent operations called coroutines:
    - These are defined using async and invoked using await
  - Dask Distributed library currently has two communication backends:
    - TCP: Tornado-based
    - UCX: Built using a Cython wrapper called UCX-Py
    - MPI4Dask: MVAPICH2-based MPI communication device

- Other Data Science frameworks include Apache Spark and Ray
Dask Distributed Execution Model

Cluster

Client

Scheduler

Worker

Worker

Worker
MPI4Dask: MPI-based Communication Backend for Dask

- Dask currently has two communication backends:
  - TCP: Tornado-based
  - UCX: Built using a GPU-aware Cython wrapper called UCX-Py

- MPI4Dask is an MVAPICH2-based communication backend for Dask:
  - First MPI-based communication device for Dask
  - Optimizes CPU and GPU communication in the Dask framework on modern HPC clusters
Dask Architecture

- Task Graph
  - Dask-MPI
  - Dask-CUDA
  - Dask-Jobqueue
  - Scheduler
  - Worker
  - Client
  - Comm Layer
    - tcp.py
    - ucx.py
    - MPI4Dask
    - mpi4py
    - UCX
    - UCX-Py (Cython wrappers)
    - MVAPICH2-GDR
  - Laptops/Desktops
  - High Performance Computing Hardware

- Dask
  - Dask Bag
  - Dask Array
  - Dask DataFrame
  - Delayed
  - Future

- Distributed
- Network Based Computing Laboratory
MPI4Dask: Bootstrapping and Dynamic Connectivity

- Several ways to start Dask programs:
  - Manual
  - Utility classes:
    - LocalCUDACluster, SLURMCluster, SGECluster, PBCCluster, and others

- MPI4Dask uses the Dask-MPI to bootstrap execution of Dask programs

- Dynamic connectivity is established using the asyncio package in MPI4Dask:
  - Scheduler and workers listen for incoming connections by calling asyncio.start_server()
  - Workers and client connect using asyncio.open_connection()
MPI4Dask: Point-to-point Communication Coroutines

• Implements communication coroutines for point-to-point MPI functions:
  – Using mpi4py (Cython wrappers) and MVAPICH2-GDR

• mpi4py provides two flavors of point-to-point communication functions:
  – **Send()**/**Recv()** – for exchanging data in buffers (faster and used in MPI4Dask)
  – **send()**/**recv()** – for communicating Python objects (pickle/unpickle)
  – GPU buffers implement the **__cuda_array_interface__**

• Implemented **chunking** mechanism for large messages

• The send and receive communication coroutines are as follows:

```python
request = comm.Isend([buf, size], dest, tag)
status = request.Test()
while status is False:
    await asyncio.sleep(0)
    status = request.Test()
```

```python
request = comm.Irecv([buf, size], src, tag)
status = request.Test()
while status is False:
    await asyncio.sleep(0)
    status = request.Test()
```
MPI4Dask Release

- **MPI4Dask 0.2** was released in Mar ‘21 adding support for MPI to Dask:
  - Can be downloaded from: [http://hibd.cse.ohio-state.edu](http://hibd.cse.ohio-state.edu)

- **Features:**
  - Based on Dask Distributed 2021.01.0
  - Compliant with user-level Dask APIs and packages
  - Support for MPI-based communication in Dask for cluster of GPUs
  - Implements point-to-point communication co-routines
  - Efficient chunking mechanism implemented for large messages
  - *(NEW)* Built on top of mpi4py over the MVAPICH2, MVAPICH2-X, and MVAPICH2-GDR libraries
  - *(NEW)* Support for MPI-based communication for CPU-based Dask applications
  - Supports starting execution of Dask programs using Dask-MPI
  - Tested with
    - *(NEW)* CPU-based Dask applications using numPy and Pandas data frames
    - *(NEW)* GPU-based Dask applications using cuPy and cuDF
    - Mellanox InfiniBand adapters (FDR and EDR)
    - Various multi-core platforms
    - NVIDIA V100 and Quadro RTX 5000 GPUs
MPI4Dask Installation

- MPI4Dask is available to download from: http://hibd.cse.ohio-state.edu/
  - The userguide is available at: http://hibd.cse.ohio-state.edu/static/media/hibd/dask/mpi4dask-0.2-userguide.pdf

- Section 3: Setup Instructions
  - 3.1 Installation Pre-requisites:
    - 3.1.1 Install Miniconda
    - 3.1.2 Modules/Libraries for GPU-based Dask Applications:
    - 3.1.3 Modules/Libraries for CPU-based Dask Applications:
    - 3.1.4 Install the MVAPICH2 Library (MVAPICH2-X, MVAPICH2, or MVAPICH2-GDR)
    - 3.1.5 Install the mpi4py Library
    - 3.1.6 Install Dask-MPI package
  - 3.2 Install MPI4Dask

- Section 4. Running GPU-based Dask Applications
  - 4.1 Writing the host file
  - 4.2 Sum of cuPy Array and its Transpose
  - 4.3 cuDF Merge

- Section 5. Running GPU-based Dask Applications
  - 5.1 Writing the host file
  - 5.2 Sum of numPy Array and its Transpose
  - 5.3 Sum of Pandas Data Frame
  - 5.4 SVD
Installation Pre-requisites

Modules/Libraries for GPU-based Dask Applications:

$ conda install -c conda-forge -c rapidsai -c nvidia automake make libtool pkg-config libhwloc psutil python=3.8 setuptools cython cudatoolkit=10.2 cupy dask-cudf dask==2021.1.1 distributed numpy rmmod

Modules/Libraries for CPU-based Dask Applications:

$ conda install -c conda-forge -c rapidsai -c nvidia automake make libtool pkg-config libhwloc psutil python=3.8 setuptools cython dask==2021.1.1 distributed=2021.1.1 numpy

Install the MVAPICH2 Library (MVAPICH2-X, MVAPICH2, or MVAPICH2-GDR)

Install the mpi4py Library

$ git clone https://github.com/mpi4py/mpi4py.git
$ edit mpi.cfg file
[MVAPICH2]
mpi_dir = /path/to/MVAPICH2-GDR/install/directory
mpicc = %(%mpi_dir)s/bin/mpicc
mpicxx = %(%mpi_dir)s/bin/mpicxx
include_dirs = %(%mpi_dir)s/include
library_dirs = %(%mpi_dir)s/lib64
runtime_library_dirs = %(%library_dirs)s

$ python setup.py build --mpi=MVAPICH2-GDR; $ pip install .
Install Dask-MPI and MPI4Dask

Install Dask-MPI package

$ git clone https://github.com/dask/dask-mpi.git
$ cd dask-mpi
$ python setup.py build
$ pip install .

Install MPI4Dask

$ wget http://hibd.cse.ohio-state.edu/download/hibd/dask/mpi4dask-0.2.tar.gz
$ tar -xzvf mpi4dask-0.2.tar.gz
$ cd mpi4dask-0.2/distributed
$ python setup.py build
$ pip install .

$ conda list
$ conda list | grep distributed
$ conda list | grep dask
Running GPU and CPU Dask Applications

Sum of cuPy Array and its Transpose [GPU]

$ cd mpi4dask-0.2/dask-apps/gpu
$ LD_PRELOAD=$MPILIB/lib64/libmpi.so $MPILIB/bin/mpirun_rsh -export-all -np 4 -hostfile hosts MV2_USE_CUDA=1 MV2_USE_GDRCOPY=1 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so MV2_CPU_BINDING_LEVEL=SOCKET MV2_CPU_BINDING_POLICY=SCATTER python cupy_sum_mpi.py

cuDf Merge [GPU]

$ cd mpi4dask-0.2/dask-apps/gpu
$ LD_PRELOAD=$MPILIB/lib/libmpi.so $MPILIB/bin/mpirun_rsh -export-all -np 4 -hostfile hosts MV2_USE_CUDA=1 MV2_USE_GDRCOPY=1 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so MV2_CPU_BINDING_LEVEL=SOCKET MV2_CPU_BINDING_POLICY=SCATTER python cudf_merge_mpi.py --type gpu --protocol mpi --runs 20 --chunk-size 1_000_000_00

Sum of numPy Array and its Transpose [CPU]

$ cd mpi4dask-0.2/dask-apps/cpu
$ LD_PRELOAD=$MPILIB/lib/libmpi.so $MPILIB/bin/mpirun_rsh -export-all -np 4 -hostfile hosts MV2_CPU_BINDING_LEVEL=SOCKET MV2_CPU_BINDING_POLICY=SCATTER python numpy_sum_mpi.py

Sum of Pandas DataFrame [CPU]

$ cd mpi4dask-0.2/dask-apps/cpu
$ LD_PRELOAD=$MPILIB/lib/libmpi.so $MPILIB/bin/mpirun_rsh -export-all -np 4 -hostfile hosts MV2_CPU_BINDING_LEVEL=SOCKET MV2_CPU_BINDING_POLICY=SCATTER python dask-cudf_sum_mpi.py
Sum of cuPy Array and its Transpose (with TCP device)

```
$ srun -N 4 --reservation=dltutorial run_cupy_sum_tcp.sh
```

Sample Output:

```
+ /opt/tutorials/dl-tutorial-21/dask/mvapich2/install/bin/mpirun_rsh -np 4 -hostfile /tmp/shafi.16/hosts LD_PRELOAD=/opt/tutorials/dl-tutorial-21/dask/mvapich2/install/lib/libmpi.so MV2_USE_CUDA=1 MV2_CPU_BINDING_LEVEL=SOCKET MV2_CPU_BINDING_POLICY=SCATTER MV2_USE_GDRCOPY=1 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so MV2_SUPPRESS_JOB_STARTUP_PERFORMANCE_WARNING=1 /opt/tutorials/dl-tutorial-21/dask/miniconda3/envs/mpi4dask_tutorial/bin/python /opt/tutorials/dl-tutorial-21/labs/lab1/cupy_sum_tcp.py
<Client: 'tcp://10.3.1.1:46566' processes=2 threads=56, memory=269.79 GB>
Time for iteration 0 : 15.613967895507812
Time for iteration 1 : 12.97205138206482
Time for iteration 2 : 13.211132526397705
Time for iteration 3 : 12.941233396530151
Time for iteration 4 : 13.074704647064209
Median Wall-clock Time: 13.07 s
+ set +x
```
Sum of cuPy Array and its Transpose (with MPI device)

$ srun -N 4 --reservation=dltutorial run_cupy_sum_mpi.sh

Sample Output:
+ /opt/tutorials/dl-tutorial-21/dask/mvapich2/install/bin/mpirun_rsh -np 4 -hostfile /tmp/shafi.16/hosts LD_PRELOAD=/opt/tutorials/dl-tutorial-21/dask/mvapich2/install/lib/libmpi.so MV2_USE_CUDA=1 MV2_CPU_BINDING_LEVEL=SOCKET MV2_CPU_BINDING_POLICY=SCATTER MV2_USE_GDRCOPY=1 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy2.0/lib64/libgdrapi.so MV2_SUPPRESS_JOB_STARTUP_PERFORMANCE_WARNING=1 /opt/tutorials/dl-tutorial-21/dask/miniconda3/envs/mpi4dask_tutorial/bin/python /opt/tutorials/dl-tutorial-21/labs/lab1/cupy_sum_mpi.py

<Client: 'mpi://10.3.1.1:31549' processes=2 threads=56, memory=269.79 GB>
Time for iteration 0 : 9.499887466430664
Time for iteration 1 : 5.288544416427612
Time for iteration 2 : 4.123088598251343
Time for iteration 3 : 4.088623523712158
Time for iteration 4 : 4.115798234939575
Median Wall-clock Time: 4.12 s
+ set +x

MVAPICH2-GDR: 3.17x faster

Configurable options in the script (cupy_sum_mpi.py)

- **DASK_PROTOCOL** = 'mpi' # Options are ['mpi', 'tcp']
- **GPUS_PER_NODE** = 1 # number of GPUs in the system
- **RUNS** = 5 # repetitions for the benchmark
- **DASK_INTERFACE** = 'ib0' # interface to use for communication
- **THREADS_PER_NODE** = 28 # number of threads per node.
Benchmark #1: Sum of cuPy Array and its Transpose (RI2)

3.47x better on average

6.92x better on average


MPI4Dask 0.2 release (http://hibd.cse.ohio-state.edu)
Benchmark #1: Sum of cuPy Array and its Transpose (TACC Frontera GPU Subsystem)

1.71x better on average

https://arxiv.org/abs/2101.08878

MPI4Dask 0.2 release
(http://hibd.cse.ohio-state.edu)

https://arxiv.org/abs/2101.08878

MPI4Dask 0.2 release
(http://hibd.cse.ohio-state.edu)
Benchmark #2: cuDF Merge (TACC Frontera GPU Subsystem)

2.91x better on average

2.90x better on average

A. Shafi, J. Hashmi, H. Subramoni, and D. K. Panda, Efficient MPI-based Communication for GPU-Accelerated Dask Applications, CCGrid '21
https://arxiv.org/abs/2101.08878

MPI4Dask 0.2 release
(http://hibd.cse.ohio-state.edu)
Outline

• Introduction
• Deep Neural Network Training and Essential Concepts
• Parallelization Strategies for Distributed DNN Training
• Machine Learning
• Data Science using Dask

• Conclusion
Convergence of ML/DL, Data Science, and HPC

• Is Machine Learning/Deep Learning and Data Science an HPC Problem?
  – Distributed Model/DNN Training is definitely an HPC problem
  – Inference – not yet an HPC problem
  – Support for Data Science frameworks on HPC systems is improving (yet lagging)

• Why HPC can help?
  – Decades of research for communication models and performance optimizations
  – MPI, PGAS, and other communication runtimes can help for “data-parallel” training

• Some of the needs for ML/DL and Data Science frameworks are an exact match
  – Compute intensive problem

• Some needs are new for distributed/parallel communication runtimes
  – Large Message Communication
  – CUDA-Aware Communication
Exploiting GPUs and Unified Communication Layer for Data Science Frameworks

- The support for GPUs has been added to Dask as part of the NVIDIA RAPIDS project
- The RAPIDS Accelerator for Apache Spark is currently adding support for GPU processing for the Apache Spark Framework
  - SQL Plugin for executing SQL queries entirely on GPUs
  - Shuffle Plugin based on the UCX communication library (GPU-to-GPU)
  - [https://nvidia.github.io/spark-rapids/](https://nvidia.github.io/spark-rapids/)

- Opportunity to utilize GPU-to-GPU communication provided by MPI libraries like MVAPICH2-GDR

- Is it possible to have an MPI-based communication layer for Data Science frameworks?
Conclusion

• Exponential growth in Machine Learning/Deep Learning/Data Science frameworks

• Provided an overview of issues, challenges, and opportunities for designing efficient communication runtimes
  – Efficient, scalable, and hierarchical designs are crucial for ML/DL/Data Science frameworks
  – Co-design of communication runtimes and ML/DL/Data Science frameworks will be essential

• Presented use-cases to demonstrate the complex interaction between DL/ML/Dask middleware with the underlying HPC technologies and middleware

• Need collaborative efforts to achieve the full potential
Funding Acknowledgments

Funding Support by

Equipment Support by
## Acknowledgments to all the Heroes (Past/Current Students and Staffs)

### Current Students (Graduate)
- N. Alnaasan (Ph.D.)
- Q. Anthony (Ph.D.)
- C.-C. Chun (Ph.D.)
- N. Contini (Ph.D.)
- A. Jain (Ph.D.)
- K. S. Khorassani (Ph.D.)
- A. H. Tu (Ph.D.)
- P. Kousha (Ph.D.)
- B. Michalowicz (Ph.D.)
- B. Ramesh (Ph.D.)
- K. Al Attar (M.S.)
- N. Sarkauskas (Ph.D.)

### Past Students
- A. Awan (Ph.D.)
- A. Augustine (M.S.)
- P. Balaji (Ph.D.)
- M. Bayatpour (Ph.D.)
- R. Biswas (M.S.)
- S. Bhagvat (M.S.)
- A. Bhat (M.S.)
- D. Buntinas (Ph.D.)
- L. Chai (Ph.D.)
- B. Chandrasekharan (M.S.)
- S. Chakraborthy (Ph.D.)
- N. Dandapanthula (M.S.)
- V. Dhanraj (M.S.)
- C.-H. Chu (Ph.D.)
- K. Gangadharappa (M.S.)
- K. Gopalakrishnan (M.S.)
- J. Hashmi (Ph.D.)
- W. Huang (Ph.D.)
- W. Jiang (M.S.)
- J. Jose (Ph.D.)
- M. Kedia (M.S.)
- S. Kini (M.S.)
- M. Koop (Ph.D.)
- K. Kulkarni (M.S.)
- R. Kumar (M.S.)
- S. Krishnamoorthy (M.S.)
- K. Kandalla (Ph.D.)
- M. Li (Ph.D.)
- P. Lai (M.S.)
- J. Liu (Ph.D.)
- M. Luo (Ph.D.)
- A. Mamidala (Ph.D.)
- G. Marsh (M.S.)
- V. Meshram (M.S.)
- A. Moody (M.S.)
- S. Naravula (Ph.D.)
- R. Noronha (Ph.D.)
- X. Ouyang (Ph.D.)
- S. Pai (M.S.)
- S. Potluri (Ph.D.)
- K. Raj (M.S.)
- R. Rajachandrasekar (Ph.D.)

### Past Research Scientists
- K. Hamidouche
- S. Sur
- X. Lu

### Past Post-Docs
- D. Banerjee
- X. Besseron
- M. S. Ghazimeersaeed
- H.-W. Jin
- E. Mancini
- A. Ruhela
- J. Lin
- K. Manian
- J. Vienne
- M. Luo
- S. Marcarelli
- H. Wang

### Current Research Scientists
- A. Shafi
- H. Subramoni
- A. H. Tu (Ph.D.)
- S. Xu (Ph.D.)
- Q. Zhou (Ph.D.)
- K. Al Attar (M.S.)
- N. Sarkauskas (Ph.D.)
- P. Lai (M.S.)
- J. Liu (Ph.D.)
- M. Luo (Ph.D.)
- A. Mamidala (Ph.D.)
- G. Marsh (M.S.)
- V. Meshram (M.S.)
- A. Moody (M.S.)
- S. Naravula (Ph.D.)
- R. Noronha (Ph.D.)
- X. Ouyang (Ph.D.)
- S. Pai (M.S.)
- S. Potluri (Ph.D.)
- K. Raj (M.S.)
- R. Rajachandrasekar (Ph.D.)
- D. Shankar (Ph.D.)
- G. Santhanaraman (Ph.D.)
- N. Sarkauskas (B.S.)
- N. Senthil Kumar (M.S.)
- A. Singh (Ph.D.)
- J. Sridhar (M.S.)
- S. Srivastava (M.S.)
- S. Sur (Ph.D.)
- H. Subramoni (Ph.D.)
- K. Vaidyanathan (Ph.D.)
- A. Vishnu (Ph.D.)
- J. Wu (Ph.D.)
- W. Yu (Ph.D.)
- J. Zhang (Ph.D.)

### Current Students (Undergrads)
- M. Lieber
- L. Xu

### Current Software Engineers
- B. Seeds
- N. Shineman

### Past Senior Research Associate
- J. Hashmi

### Past Programmers
- A. Reifsteck
- D. Bureddy
- J. Perkins

### Past Research Specialist
- M. Arnold
- J. Smith
Thank You!

{jain.575, shafi.16, anthony.301}@osu.edu

Network-Based Computing Laboratory
http://nowlab.cse.ohio-state.edu/

The MVAPICH Project
http://mvapich.cse.ohio-state.edu/

The High-Performance Deep Learning Project
http://hidl.cse.ohio-state.edu/

Follow us on
https://twitter.com/mvapich