How to Boost the Performance of HPC/AI Applications Using MVAPICH2 Library?

Lab L9121 at GTC’19

by

Dhabaleswar K. (DK) Panda, Hari Subramoni, Ammar Ahmad Awan, and Ching-Hsiang Chu

The Ohio State University

E-mail: {panda,subramon@cse.ohio-state.edu}{awan.10,chu.368}@osu.edu

http://www.cse.ohio-state.edu/~panda
Overview

• Introduction
  • Setting up the Lab
  • Understanding CUDA-Aware MPI
  • Lab 1 - Distributed Training with TensorFlow
  • Lab 2 - High-Performance Particle Simulation with HOOMD-blue
  • Lab 3 - OSU Micro-Benchmarks using MVAPICH2
  • Lab 4 - Distributed Training using OSU-Caffe
• Homeworks
• Conclusion
Current and Next Generation HPC/AI Applications

• Growth of High Performance Computing
  – Growth in processor performance
    • Chip density doubles every 18 months
  – Growth in commodity networking
    • Increase in speed/features + reducing cost

• Growth of AI
  – High Performance Deep Learning
    • Computer Vision
    • Speech Recognition
    • Self Driving Cars
Parallel Programming Models Overview

- Programming models provide abstract machine models
- Models can be mapped on different types of systems
  - e.g. Distributed Shared Memory (DSM), MPI within a node, etc.
- PGAS models and Hybrid MPI+PGAS models are gradually receiving importance
Deep Learning, HPC, and GPUs

NVIDIA GPUs - main driving force for faster training of Deep Neural Networks (DNNs)

- The ImageNet Challenge - (ILSVRC)
  - DNNs like AlexNet, ResNet, and VGG
  - 90% of the ImageNet teams used GPUs in 2014*
  - GPUs: A natural fit for DL – throughput-oriented (dense-compute)

- In the High Performance Computing (HPC) arena
  - GPUs are growing in the HPC arena as well! – Top500 (Nov ‘18)
  - CUDA-Aware Message Passing Interface (MPI)
  - DGX-1, DGX1-V (Volta), and DGX-2

*https://blogs.nvidia.com/blog/2014/09/07/imagenet/
https://www.top500.org/
Deep Learning Frameworks

• Many Deep Learning frameworks
  – Berkeley Caffe
  – Facebook Caffe2
  – Google TensorFlow
  – Microsoft CNTK
  – Facebook Torch/PyTorch
  – Chainer/ChainerMN
  – Intel Neon/Nervana Graph

• Open Neural Net eXchange (ONNX) Format
Scale-up and Scale-out

- **Scale-up**: Intra-node Communication
  - Many improvements like:
    - NVIDIA cuDNN, cuBLAS, NCCL, etc.
    - CUDA 9 Co-operative Groups
- **Scale-out**: Inter-node Communication
  - DL Frameworks – most are optimized for single-node only
  - Distributed (Parallel) Training is an emerging trend
    - OSU-Caffe – MPI-based
    - Microsoft CNTK – MPI/NCCL2
    - Google TensorFlow – gRPC-based/MPI/NCCL2
    - Facebook Caffe2 – Hybrid (NCCL2/Gloo/MPI)
Conventional Execution on GPUs and CPUs

- My framework is faster than your framework!
- This needs to be understood in a holistic way.
- Performance depends on the entire execution environment (the full stack)
- Isolated view of performance is not helpful

The Need for Parallel and Distributed Training

• Why do we need Parallel Training?
• Larger and Deeper models are being proposed
  – AlexNet to ResNet to Neural Machine Translation (NMT)
  – DNNs require a lot of memory
  – Larger models cannot fit a GPU’s memory
• Single GPU training became a bottleneck
• As mentioned earlier, community has already moved to multi-GPU training
• Multi-GPU in one node is good but there is a limit to Scale-up (8 GPUs)
• **Multi-node (Distributed or Parallel) Training is necessary!!**
Data Parallel Deep Learning and MPI Collectives

- **Major MPI Collectives** involved in Designing distributed frameworks
- **MPI_Bcast** – required for DNN parameter exchange
- **MPI_Reduce** – needed for gradient accumulation from multiple solvers
- **MPI_Allreduce** – use just one Allreduce instead of Reduce and Broadcast

Overview of the MVAPICH2 Project

- High Performance open-source MPI Library for InfiniBand, Omni-Path, Ethernet/iWARP, and RDMA over Converged Ethernet (RoCE)
  - MVAPICH (MPI-1), MVAPICH2 (MPI-2.2 and MPI-3.1), Started in 2001, First version available in 2002
  - MVAPICH2-X (MPI + PGAS), Available since 2011
  - Support for GPGPUs (MVAPICH2-GDR) and MIC (MVAPICH2-MIC), Available since 2014
  - Support for Virtualization (MVAPICH2-Virt), Available since 2015
  - Support for Energy-Awareness (MVAPICH2-EA), Available since 2015
  - Support for InfiniBand Network Analysis and Monitoring (OSU INAM) since 2015
- Used by more than 2,975 organizations in 88 countries
- More than 529,000 (> 0.5 million) downloads from the OSU site directly
- Empowering many TOP500 clusters (Nov ‘18 ranking)
  - 3rd ranked 10,649,640-core cluster (Sunway TaihuLight) at NSC, Wuxi, China
  - 14th, 556,104 cores (Oakforest-PACS) in Japan
  - 17th, 367,024 cores (Stampede2) at TACC
  - 27th, 241,108-core (Pleiades) at NASA and many others
- Available with software stacks of many vendors and Linux Distros (RedHat, SuSE, and OpenHPC)
- [http://mvapich.cse.ohio-state.edu](http://mvapich.cse.ohio-state.edu) Partner in the upcoming TACC Frontera System
- Empowering Top500 systems for over a decade
Overview

- Introduction

- Setting up the Lab
  - Understanding CUDA-Aware MPI
  - Lab 1 - Distributed Training with TensorFlow
  - Lab 2 - High-Performance Particle Simulation with HOOMD-blue
  - Lab 3 - OSU Micro-Benchmarks using MVAPICH2
  - Lab 4 - Distributed Training using OSU-Caffe

- Homeworks

- Conclusion
Getting Set-up with the Lab

• You will run the experiments on the OSU RI2 cluster
• Please use the account name and password given to you
  – E.g. ssh ri2tut01@ri2.cse.ohio-state.edu
  – Password is tutorial1
• Once on the shell, go to /opt/tutorials
  $ cd /opt/tutorials/dl-tutorial/
• There are four folders for labs and one for homework
• Take a look at the README.md file for all scripts
• Try one of the script (next slide)
Lab Setup (Example Run)

$ srun -N 2 --reservation=dltutorial1 run_tf_bench_multi_mv2.sh

+ /opt/tutorials/mv2/bin/mpirun_rsh -np 2 gpu29 gpu30 MV2_USE_CUDA=1 MV2_ENABLE_AFFINITY=0
/opt/tutorials/mconda-mv2/bin/python
/opt/tutorials/benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=resnet50 --num_gpus=1 --num_batches=50 --variable_update=horovod

Step Img/sec total_loss
1 images/sec: 54.1 +/- 0.0 (jitter = 0.0) 8.394
1 images/sec: 54.1 +/- 0.0 (jitter = 0.0) 8.346

30 images/sec: 53.9 +/-0.0 (jitter = 0.1) 8.361
40 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.455
40 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.126
50 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.191

----------------------------------------------------------------
total images/sec: 107.79
----------------------------------------------------------------
Overview

- Introduction
- Setting up the Lab
  - **CUDA-Aware MPI: Designs and Benefits**
    - Lab 1 - Distributed Training with TensorFlow
    - Lab 2 - High-Performance Particle Simulation with HOOMD-blue
    - Lab 3 - OSU Micro-Benchmarks using MVAPICH2
    - Lab 4 - Distributed Training using OSU-Caffe
- Homeworks
- Conclusion
MPI + CUDA - Naive

• Data movement in applications with standard MPI and CUDA interfaces

At Sender:

cudA_Memcopy(s_hostbuf, s_devbuf, . . .);
MPI_Send(s_hostbuf, size, . . .);

At Receiver:

MPI_Recv(r_hostbuf, size, . . .);
cudA_Memcopy(r_devbuf, r_hostbuf, . . .);

High Productivity and Low Performance
MPI + CUDA - Advanced

- Pipelining at user level with non-blocking MPI and CUDA interfaces

**At Sender:**

```c
for (j = 0; j < pipeline_len; j++)
    cudaMemcpyAsync(s_hostbuf + j * blk, s_devbuf + j * blksz, ...);

for (j = 0; j < pipeline_len; j++) {
    while (result != cudaSuccess) {
        result = cudaStreamQuery(...);
        if(j > 0) MPI_Test(...);
    }
    MPI_Isend(s_hostbuf + j * block_sz, blksz ...);
}
MPI_Waitall();
```

<<Similar at receiver>>

*Low Productivity and High Performance*
CUDA-Aware MPI Library: MVAPICH2-GPU

- Standard MPI interfaces used for unified data movement
- Takes advantage of Unified Virtual Addressing (>= CUDA 4.0)
- Overlaps data movement from GPU with RDMA transfers

**At Sender:**

\[
\text{MPI\_Send}(s\_devbuf, \text{size, ...});
\]

**At Receiver:**

\[
\text{MPI\_Recv}(r\_devbuf, \text{size, ...});
\]

*High Performance and High Productivity*
Enhanced Inter-node MPI Designs with GPUDirect RDMA (GDR)

- Current MPI design using GPUDirect RDMA uses Rendezvous protocol
  - Has higher latency for small messages
- Can eager protocol be supported to improve performance for small messages?
- Two schemes proposed and used
  - Loopback (using network adapter to copy data)
  - Fastcopy/GDRCOPY (using CPU to copy data)

Performance Benefits of MVAPICH2-GDR Designs (OSU Micro-Benchmarks)

GPU-GPU Inter-node Latency

- **Latency (us)**
  - MV2-(NO-GDR)
  - MV2-GDR 2.3

- **Message Size (Bytes)**: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1K, 2K, 4K

- **Latency**, e.g., 1.85us, 10x

GPU-GPU Inter-node Bandwidth

- **Bandwidth (MB/s)**
  - MV2-(NO-GDR)
  - MV2-GDR-2.3

- **Message Size (Bytes)**: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1K, 2K, 4K

- **Bandwidth**, e.g., 9x

**Notes**:
- **Intel Haswell (E5-2687W @ 3.10 GHz) node - 20 cores**
- **NVIDIA Volta V100 GPU**
- **Mellanox Connect-X4 EDR HCA**
- **CUDA 10.0**
- **Mellanox OFED 4.0 with GPU-Direct-RDMA**
## Tuning GPUDirect RDMA (GDR) Designs in MVAPICH2-GDR

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Significance</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV2_USE_GPUDIRECT</td>
<td>• Enable / Disable GDR-based designs</td>
<td>1 (Enabled)</td>
<td>• Always enable</td>
</tr>
<tr>
<td>MV2_GPUDIRECT_LIMIT</td>
<td>• Controls messages size until which GPUDirect RDMA is used</td>
<td>8 KByte</td>
<td>• Tune for your system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• GPU type, host architecture and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CUDA version: impact pipelining overheads and P2P bandwidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bottlenecks</td>
</tr>
<tr>
<td>MV2_USE_GPUDIRECT_RECEIVE_LIMIT</td>
<td>• Controls messages size until which 1 hop design is used (GDR Write at the</td>
<td>256KBytes</td>
<td>• Tune for your system</td>
</tr>
<tr>
<td></td>
<td>receiver)</td>
<td></td>
<td>• GPU type, HCA type and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>configuration</td>
</tr>
</tbody>
</table>

- Refer to Tuning and Usage Parameters section of MVAPICH2-GDR user guide for more information
- http://mvapich.cse.ohio-state.edu/userguide/gdr/#_tuning_and_usage_parameters
### Tuning GDRCOPY and LOOPBACK Designs in MVAPICH2-GDR

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Significance</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV2_USE_GPUDIRECT_GDRCOPY</td>
<td>• Enable / Disable GDRCOPY-based designs</td>
<td>1 (Enabled)</td>
<td>• Always enable</td>
</tr>
<tr>
<td>MV2_GPUDIRECT_GDRCOPY_LIMIT</td>
<td>• Controls messages size until which GDRCOPY is used</td>
<td>8 KByte</td>
<td>• Tune for your system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• GPU type, host architecture. Impacts the eager performance</td>
</tr>
<tr>
<td>MV2_USE_GPUDIRECT_D2H_GDRCOPY_LIMIT</td>
<td>• Controls messages size until which GDRCOPY is used at sender</td>
<td>16Bytes</td>
<td>• Tune for your systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CPU and GPU type</td>
</tr>
<tr>
<td>MV2_USE_GPUDIRECT_LOOPBACK</td>
<td>• Enable / Disable LOOPBACK-based designs</td>
<td>1 (Enabled)</td>
<td>• Always enable</td>
</tr>
<tr>
<td>MV2_GPUDIRECT_LOOPBACK_LIMIT</td>
<td>• Controls messages size until which LOOPBACK is used</td>
<td>8 KByte</td>
<td>• Tune for your system</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• GPU type, host architecture and HCA.</td>
</tr>
</tbody>
</table>

- Refer to Tuning and Usage Parameters section of MVAPICH2-GDR user guide for more information
- [http://mvapich.cse.ohio-state.edu/userguide/gdr/#_tuning_and_usage_parameters](http://mvapich.cse.ohio-state.edu/userguide/gdr/#_tuning_and_usage_parameters)
Intra-node Designs for Multi-GPU Configurations

- Multi-GPU node architectures are becoming common
- Until CUDA 3.2
  - Communication between processes staged through the host
  - Shared Memory (pipelined)
  - Network Loopback [asynchronous]
- CUDA 4.0 and later
  - Inter-Process Communication (IPC)
  - Host bypass
  - Handled by a DMA Engine
  - Low latency and Asynchronous
  - Requires creation, exchange and mapping of memory handles
  - Overhead
Tuning IPC designs in MVAPICH2-GDR

- Works between GPUs within the same socket or IOH
- Leads to significant benefits in appropriate scenarios

![Intra-node Small Message Latency Graph](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Significance</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV2_CUDA_IPC</td>
<td>• Enable / Disable CUDA IPC-based designs</td>
<td>1 (Enabled)</td>
<td>• Always leave set to 1</td>
</tr>
<tr>
<td>MV2_CUDA_SMP_IPC</td>
<td>• Enable / Disable CUDA IPC fastpath design for short messages</td>
<td>0 (Disabled)</td>
<td>• Benefits Device-to-Device transfers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Hurts Device-to-Host/Host-to-Device transfers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Always set to 1 if application involves only Device-to-Device transfers</td>
</tr>
<tr>
<td>MV2_IPC_THRESHOLD</td>
<td>• Message size where IPC code path will be used</td>
<td>16 KBytes</td>
<td>• Tune for your system</td>
</tr>
</tbody>
</table>
Overview

• Introduction
• Setting up the Lab
• Understanding CUDA-Aware MPI

• Lab 1 - Distributed Training with TensorFlow
• Lab 2 - High-Performance Particle Simulation with HOOMD-blue
• Lab 3 - OSU Micro-Benchmarks using MVAPICH2
• Lab 4 - Distributed Training using OSU-Caffe
• Homeworks
• Conclusion
Lab 1 - Overview

• Objectives
  – How to train a TensorFlow model on a single NVIDIA GPU?
  – How to perform distributed training of a TensorFlow model on multiple GPUs using InfiniBand and NVIDIA GPUs?

• Tasks
  – Run TensorFlow on a Single GPU
  – Run TensorFlow on two nodes with 1 GPU/node (using MVAPICH2)
  – Run TensorFlow on two nodes with 1 GPU/node (using MVAPICH2-GDR)
  – Run TensorFlow on two nodes with 1 GPU/node (using NCCL2)
Distributed Training using TensorFlow (TF)

- TensorFlow is the most popular DL framework
- gRPC is the official distributed training runtime
  - Many problems for HPC use-cases
- Community efforts - Baidu and Uber’s Horovod have added MPI support to TF across nodes
- Need to understand several options currently available →

https://arxiv.org/abs/1810.11112
Scalable TensorFlow using Horovod, MPI, and NCCL

- Efficient Allreduce is crucial for Horovod’s overall training performance
  - Both MPI and NCCL designs are available
- We have evaluated Horovod extensively and compared across a wide range of designs using gRPC and gRPC extensions
- MVAPICH2-GDR achieved up to 90% scaling efficiency for ResNet-50 Training on 64 Pascal GPUs

https://arxiv.org/abs/1810.11112
Distributed Training with TensorFlow and MVAPICH2-GDR

- ResNet-50 Training using TensorFlow benchmark on 1 DGX-2 node (8 Volta GPUs)

Platform: Nvidia DGX-2 system (16 Nvidia Volta GPUs connected with NVSwitch), CUDA 9.2
Lab1-Task1: Run TensorFlow on a Single GPU

$ srun -N 1 --reservation=dltutorial1 run_tf_bench_single.sh

+ /opt/tutorials/mconda2-mv2gdr/bin/python
/opt/tutorials/benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=resnet50 --num_gpus=1 --num_batches=50

Step Img/sec total_loss
1 images/sec: 59.4 +/- 0.0 (jitter = 0.0) 8.348
10 images/sec: 59.4 +/- 0.1 (jitter = 0.2) 8.143
20 images/sec: 59.4 +/- 0.0 (jitter = 0.2) 8.437
30 images/sec: 59.3 +/- 0.0 (jitter = 0.2) 8.154
40 images/sec: 59.3 +/- 0.0 (jitter = 0.2) 8.460
50 images/sec: 59.3 +/- 0.0 (jitter = 0.2) 8.319

-------------------------------------------------------------
total images/sec: 59.26
-------------------------------------------------------------
Lab1-Task2: Run TensorFlow on two nodes with 1 GPU/node (using MVAPICH2)

$ srun -N 2 --reservation=dltutorial1 run_tf_bench_multi_mv2.sh

+ /opt/tutorials/mv2/bin/mpirun_rsh -np 2 gpu29 gpu30 MV2_USE_CUDA=1 MV2_ENABLE_AFFINITY=0
/opt/tutorials/mconda-mv2/bin/python
/opt/tutorials/benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=resnet50 --num_gpus=1 --num_batches=50 --variable_update=horovod

Step Img/sec total_loss
1 images/sec: 54.1 +/- 0.0 (jitter = 0.0) 8.394
1 images/sec: 54.1 +/- 0.0 (jitter = 0.0) 8.346

30 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.361
40 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.455
40 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.126
50 images/sec: 53.9 +/- 0.0 (jitter = 0.1) 8.191

----------------------------------------------------------------
total images/sec: 107.79
----------------------------------------------------------------

1.8X on 2 GPUs
Lab1-Task3: Run TensorFlow on two nodes with 1 GPU/node (using MVAPICH2-GDR)

$ srun -N 2 --reservation=dltutorial1 run_tf_bench_multi_mv2gdr.sh

+ /opt/tutorials/mv2gdr/bin/mpirun_rsh -np 2 gpu27 gpu28 MV2_USE_CUDA=1 MV2_ENABLE_AFFINITY=0 MV2_USE_GPUDIRECT_GDRCOPY=0 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy9.0/libgdrcopi.so LD_PRELOAD=/opt/tutorials/mv2gdr/lib/libmpi.so /opt/tutorials/mconda2-mv2gdr/bin/python /opt/tutorials/benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=resnet50 --num_gpus=1 --num_batches=50 --variable_update=horovod.

Done warm up
Step Img/sec total_loss
1 images/sec: 59.5 +/- 0.0 (jitter = 0.0) 8.396
1 images/sec: 59.5 +/- 0.0 (jitter = 0.0) 8.355
10 images/sec: 59.4 +/- 0.0 (jitter = 0.1) 8.359
.

40 images/sec: 59.3 +/- 0.0 (jitter = 0.1) 8.143
40 images/sec: 59.3 +/- 0.0 (jitter = 0.1) 8.487
50 images/sec: 59.3 +/- 0.0 (jitter = 0.1) 8.303
----------------------------------------------------------------
total images/sec: 118.64
----------------------------------------------------------------
~2X on 2 GPUs
Near ideal scaling!
Lab1-Task4: Run TensorFlow on two nodes with 1 GPU/node (using NCCL2)

$ srun -N 2 --reservation=dltutorial1 run_tf_bench_multi_nccl2.sh

+ /opt/tutorials/mv2/bin/mpirun_rsh -np 2 gpu27 gpu28 MV2_USE_CUDA=1 NCCL_DEBUG=INFO /opt/tutorial/mconda-nccl2/bin/python /opt/tutorials/benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=resnet50 --num_gpus=1 --num_batches=50 --variable_update=horovod

1 images/sec: 59.2 +/- 0.0 (jitter = 0.0) 8.396

30 images/sec: 58.3 +/- 0.0 (jitter = 0.1) 8.125
30 images/sec: 58.3 +/- 0.1 (jitter = 0.6) 8.346
40 images/sec: 58.2 +/- 0.0 (jitter = 0.2) 8.487
40 images/sec: 58.2 +/- 0.1 (jitter = 0.7) 8.149
50 images/sec: 58.2 +/- 0.0 (jitter = 0.2) 8.325

----------------------------------------------------------------
total images/sec: 116.49
----------------------------------------------------------------

~1.96X on 2 GPUs
Overview

- Introduction
- Setting up the Lab
- Understanding CUDA-Aware MPI
- Lab 1 - Distributed Training with TensorFlow
  - Lab 2 - High-Performance Particle Simulation with HOOMD-blue
    - Lab 3 - OSU Micro-Benchmarks using MVAPICH2
    - Lab 4 - Distributed Training using OSU-Caffe
- Homeworks
- Conclusion
Lab 2 - Overview

• Objectives
  – How to run HOOMD-blue* on a single NVIDIA GPU?
  – How to perform distributed experiments of HOOMD-blue on multiple GPUs using InfiniBand and NVIDIA GPUs?

• Tasks
  – Run HOOMD-blue on a Single GPU
  – Run HOOMD-blue on two nodes with 1 GPUs/node (using MVAPICH2-GDR w/o GPUDirect features)
  – Run HOOMD-blue on two nodes with 2 GPUs/node (using MVAPICH2-GDR w/o GPUDirect features)
  – Run HOOMD-blue on two nodes with 2 GPUs/node (using MVAPICH2-GDR with GPUDirect features)

*http://glotzerlab.engin.umich.edu/hoomd-blue/
HOOMD-blue: Highly-Optimized, Object-oriented Many-particle Dynamics, blue edition

- General Purpose Particle Simulation Toolkit
  - Open-source GPU-accelerated, written in CUDA/C++/Python, use CUDA-Aware MPI
    
    http://glotzerlab.engin.umich.edu/hoomd-blue/

Courtesy:
http://mug.mvapich.cse.ohio-state.edu/static/media/mug/presentations/2014/glaser.pdf
Particle Simulations using HOOMD-blue and MVAPICH2-GDR

**64K Particles**

![Graph showing performance comparison between MV2 and MV2+GDR for 64K particles across different process counts.]

**256K Particles**

![Graph showing performance comparison between MV2 and MV2+GDR for 256K particles across different process counts.]

- **Platform**: Wilkes (Intel Ivy Bridge + NVIDIA Tesla K20c + Mellanox Connect-IB)
- **HoomdBlue Version 1.0.5**
  - GDRCOPY enabled: `MV2_USE_CUDA=1 MV2_IBA_HCA=mlx5_0 MV2_IBA_EAGER_THRESHOLD=32768 MV2_VBUF_TOTAL_SIZE=32768 MV2_USE_GPUDIRECT_LOOPBACK_LIMIT=32768 MV2_USE_GPUDIRECT_GDRCOPY=1 MV2_USE_GPUDIRECT_GDRCOPY_LIMIT=16384`
Lab2-Task1: Run HOOMD-blue on a Single GPU

$ srun -N 1 --reservation=dltutorial1 run_hoomd_single.sh

HOOMD-blue v2.5.0 CUDA (9.2) DOUBLE HPMC_MIXED MPI MPI_CUDA SSE SSE2 SSE3 SSE4_1 SSE4_2 AVX AVX2

HOOMD-blue is running on the following GPU(s):

[0] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM

** starting run **

Time 00:00:10 | Step 205892 / 230000 | TPS 589.069 | ETA 00:00:40
Time 00:00:20 | Step 212852 / 230000 | TPS 695.872 | ETA 00:00:24
Time 00:00:30 | Step 220223 / 230000 | TPS 736.702 | ETA 00:00:13
Time 00:00:40 | Step 227637 / 230000 | TPS 741.385 | ETA 00:00:03
Time 00:00:43 | Step 230000 / 230000 | TPS 734.48 | ETA 00:00:00

Average TPS: 694.011

** run complete **

Hours to complete 10e6 steps: 3.7547008864
Lab2-Task2: Run HOOMD-blue on two nodes with 1 GPUs/node
(using MVAPICH2-GDR w/o GPUDirect features)

```bash
$ srun -N 2 --reservation=dltutorial1 run_hoomd_2gpus_nogdr.sh
```

HOOMD-blue v2.5.0 CUDA (9.2) DOUBLE HPMC_MIXED MPI MPI_CUDA SSE SSE2 SSE3 SSE4_1 SSE4_2 AVX AVX2

HOOMD-blue is running on the following GPU(s):
Rank 0: [0] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
Rank 1: [1] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM

** starting run **
Time 00:00:10 | Step 207682 / 230000 | TPS 767.853 | ETA 00:00:29
Time 00:00:20 | Step 216961 / 230000 | TPS 927.522 | ETA 00:00:14
Time 00:00:30 | Step 226565 / 230000 | TPS 960.37 | ETA 00:00:03
Time 00:00:33 | Step 230000 / 230000 | TPS 958.069 | ETA 00:00:00
Average TPS: 893.01

** run complete **
Hours to complete 10e6 steps: 2.91995390681

1.3X on 2 GPUs
Lab2-Task3: Run HOOMD-blue on two nodes with 2 GPUs/node (using MVAPICH2-GDR w/o GPUDirect features)

$ srun -N 2 --reservation=dltutorial1 run_hoomd_4gpus_nogdr.sh

HOOMD-blue v2.5.0 CUDA (9.2) DOUBLE HPMC_MIXED MPI_CUDA SSE2 SSE3 MPI_CUDA SSE4_1 SSE4_2 AVX AVX2

HOOMD-blue is running on the following GPU(s):
Rank 0: [0] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
Rank 1: [1] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
Rank 2: [0] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
Rank 3: [1] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM

** starting run **
Time 00:00:10 | Step 210832 / 230000 | TPS 1082.89 | ETA 00:00:17
Time 00:00:20 | Step 223604 / 230000 | TPS 1277.09 | ETA 00:00:05
Time 00:00:25 | Step 230000 / 230000 | TPS 1264.74 | ETA 00:00:00
Average TPS: 1197.08

** run complete **
Hours to complete 10e6 steps: 2.16988177809

1.7X on 4 GPUs
Lab2-Task4: Run HOOMD-blue on two nodes with 2 GPUs/node (using MVAPICH2-GDR with GPUDirect features)

$ srun -N 2 --reservation=dltutorial1 run_hoomd_4gpus_gdr.sh

HOOMD-blue v2.5.0 CUDA (9.2) DOUBLE HPMC_MIXED MPI CUDA SSE SSE2 SSE3 SSE4_1 SSE4_2 AVX AVX2

HOOMD-blue is running on the following GPU(s):

- Rank 0: [0] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
- Rank 1: [1] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
- Rank 2: [0] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM
- Rank 3: [1] Tesla K80 13 SM_3.7 @ 0.823 GHz, 11441 MiB DRAM

** starting run **

Time 00:00:10 | Step 210990 / 230000 | TPS 1098.84 | ETA 00:00:17
Time 00:00:20 | Step 224165 / 230000 | TPS 1317.39 | ETA 00:00:04
Time 00:00:24 | Step 230000 / 230000 | TPS 1323.27 | ETA 00:00:00

Average TPS: **1228.91**

** run complete **

Hours to complete 10^6 steps: 2.10824754229
Overview

• Introduction
• Setting up the Lab
• Understanding CUDA-Aware MPI
• Lab 1 - Distributed Training with TensorFlow
• Lab 2 - High-Performance Particle Simulation with HOOMD-blue
• **Lab 3 - OSU Micro-Benchmarks using MVAPICH2**
• Lab 4 - Distributed Training using OSU-Caffe
• Conclusion
• Homeworks
Lab 3 - Overview

• Objectives
  – How to evaluate MPI performance using OSU Micro-Benchmarks?
  – How to understand the impact of IPC designs on Intra-node performance?

• Tasks
  – Run OSU Micro-benchmarks (Default)
  – Run OSU Micro-benchmarks (Intra-node with IPC)
  – Run OSU Micro-benchmarks (Intra-node without IPC)
OSU Micro-benchmarks

- OSU Micro-benchmarks are widely used to compare performance of different MPI stacks and networks
- Enhancements to measure performance of MPI communication from GPU memory
- Support for CUDA and OpenACC
- Flexible selection of data movement between CPU(H) and GPU(D):
  - D->D, D->H, H->D, and H->H
- Available from [http://mvapich.cse.ohio-state.edu/benchmarks](http://mvapich.cse.ohio-state.edu/benchmarks)
- Available in an integrated manner with MVAPICH2-GDR stack
Lab3-Task1: Run OSU Micro-benchmarks (Default)

```
$ srun -N 2 --reservation=dltutorial1 run_omb.sh
+ /opt/tutorials/mv2gdr//bin/mpirun_rsh -np 2 gpu22 gpu23 MV2_USE_CUDA=1 MV2_GPUDIRECT_GDRCOPY_LIBRARY=/opt/gdrcopy9.0/libgdrcopy.so LD_PRELOAD=/opt/tutorials/mv2gdr//lib/libmpi.so /opt/tutorials/mv2gdr//libexec/osu-micro-benchmarks/mpi/pt2pt/osu_latency
# OSU MPI-CUDA Latency Test v5.6
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size    Latency (us)
  0       1.17
  1       2.31
  2       2.30
  4       2.30
  8       2.31
  2048    5.02
  4096    7.61
  8192    11.36
  16384   20.60
  32768   25.49
  65536   33.40
 131072  51.82
```

Lab3-Task1: Run OSU Micro-benchmarks (Default)
Lab3-Task2: Run OSU Micro-benchmarks (Intra-node with IPC)

$$ \text{\$ srun -N 1 --reservation=dltutorial1 run_omb_intra.sh} $$

+ /opt/tutorials/mv2gdr/bin/mpirun_rsh -np 2 gpu22 gpu22 MV2_USE_CUDA=1 MV2_GPUDIRECT_GDRCOPY_LIBRARY=/opt/gdrcopy9.0/libgdrcopy.so
LD_PRELOAD=/opt/tutorials/mv2gdr/lib/libmpi.so /opt/tutorials/mv2gdr/libexec/osu-micro-benchmarks/mpi/pt2pt/osu_latency

# OSU MPI-CUDA Latency Test v5.6
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)

<table>
<thead>
<tr>
<th>Size</th>
<th>Latency (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.23</td>
</tr>
<tr>
<td>1</td>
<td>1.31</td>
</tr>
<tr>
<td>2</td>
<td>1.32</td>
</tr>
<tr>
<td>4</td>
<td>1.33</td>
</tr>
<tr>
<td>8</td>
<td>1.32</td>
</tr>
<tr>
<td>16</td>
<td>1.58</td>
</tr>
<tr>
<td>4096</td>
<td>12.50</td>
</tr>
<tr>
<td>8192</td>
<td>16.94</td>
</tr>
<tr>
<td>16384</td>
<td>17.58</td>
</tr>
<tr>
<td>32768</td>
<td>19.10</td>
</tr>
<tr>
<td>65536</td>
<td>21.85</td>
</tr>
<tr>
<td>131072</td>
<td>27.43</td>
</tr>
<tr>
<td>262144</td>
<td>38.57</td>
</tr>
<tr>
<td>524288</td>
<td>60.91</td>
</tr>
</tbody>
</table>
Lab3-Task3: Run OSU Micro-benchmarks (Intra-node without IPC)

```
$ srun -N 1 --reservation=dltutorial1 run_omb_intra_noipc.sh
+ /opt/tutorials/mv2gdr//bin/mpirun_rsh -np 2 gpu22 gpu22 MV2_USE_CUDA=1 MV2_CUDA_IPC=0 MV2_GPUDIRECT_GDRCOPY_LIB=/opt/gdrcopy9.0/libgdrcopy.so
LD_PRELOAD=/opt/tutorials/mv2gdr//lib/libmpi.so /opt/tutorials/mv2gdr//libexec/osu-micro-benchmarks/mpi/pt2pt/osu_latency D D
# OSU MPI-CUDA Latency Test v5.6
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
<table>
<thead>
<tr>
<th>Size</th>
<th>Latency (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.23</td>
</tr>
<tr>
<td>1</td>
<td>1.43</td>
</tr>
<tr>
<td>2</td>
<td>1.44</td>
</tr>
<tr>
<td>4</td>
<td>1.44</td>
</tr>
<tr>
<td>8</td>
<td>1.45</td>
</tr>
<tr>
<td>16</td>
<td>1.69</td>
</tr>
<tr>
<td>32</td>
<td>1.70</td>
</tr>
<tr>
<td>64</td>
<td>1.93</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>131072</td>
<td>50.28</td>
</tr>
<tr>
<td>262144</td>
<td>70.73</td>
</tr>
<tr>
<td>524288</td>
<td>115.31</td>
</tr>
<tr>
<td>1048576</td>
<td>182.89</td>
</tr>
<tr>
<td>2097152</td>
<td>331.26</td>
</tr>
</tbody>
</table>
```

~2X slower w/o IPC!
Overview

• Introduction
• Setting up the Lab
• Understanding CUDA-Aware MPI
• Lab 1 - Distributed Training with TensorFlow
• Lab 2 - High-Performance Particle Simulation with HOOMD-blue
• Lab 3 - OSU Micro-Benchmarks using MVAPICH2

• **Lab 4 - Distributed Training using OSU-Caffe**
• Homeworks
• Conclusion
Lab 4 - Overview

• Objectives
  – How to train DNNs using OSU-Caffe?
  – How to perform distributed training of a Caffe model on multiple GPUs using InfiniBand and NVIDIA GPUs?

• Tasks
  – Running OSU-Caffe to train AlexNet on Single GPU
  – Run Strong Scaling Experiment for AlexNet Training (BatchSize 128)
  – Run using Larger Batch Sizes to scale better (BatchSize 256)
Scalable Deep Learning with OSU-Caffe and MVAPICH2-GDR

- **Caffe**: A flexible and layered Deep Learning framework.
- **Benefits and Weaknesses**
  - Multi-GPU Training within a single node
  - Performance degradation for GPUs across different sockets
  - Limited Scale-out
- **OSU-Caffe**: MPI-based Parallel Training
  - Enable Scale-up (within a node) and Scale-out (across multi-GPU nodes)
  - Scale-out on 64 GPUs for training CIFAR-10 network on CIFAR-10 dataset
  - Scale-out on 128 GPUs for training GoogLeNet network on ImageNet dataset

OSU-Caffe publicly available from [http://hidl.cse.ohio-state.edu/](http://hidl.cse.ohio-state.edu/)
Setup OSU-Caffe

• Download and setup OSU-Caffe
• OSU-Caffe uses the same build/tools/Caffe binary with some modifications
• ImageNet training will require a separate download and setup. Please consult default Caffe setup
• Auxiliary dataset can be downloaded using ‘get_ilsvrc_aux.sh’ script
• Do not use the download-and-setup.sh script – it is for review only
• Execute the following to setup the libraries

$ cd /opt/tutorials/dl-tutorial/lab4
$ source setup.sh
Testing the Caffe executable

Execute the following:

```bash
$ srun -N 1 --reservation=dltutorial1 caffe
```

- Please note new options in OSU-Caffe
  - `-scal weak`
  - `-scal strong (default)`
  - `-amode 1,2, and 3`
  - `-pmode 1,2, and 3`

By default, OSU-Caffe automatically selects them for you
- Provided for advanced tuning for newer unknown models
Lab4-Task1: Running OSU-Caffe to train AlexNet (Batch Size 128)

$ srun -N 1 --reservation=dltutorial1 run_osucaffe_bench_single.sh 128

• I0228 16:43:30.976338 22344 solver.cpp:434] Solving AlexNet
• I0228 16:43:30.976341 22344 solver.cpp:435] Learning Rate Policy: step
• I0228 16:43:31.490958 22344 solver.cpp:323] Iteration 0, loss = 6.91506
• I0228 16:43:42.189394 22344 solver.cpp:758] Iteration 40, lr = 0.01
• I0228 16:43:47.506415 22344 solver.cpp:323] Iteration 60, loss = 6.9324
• I0228 16:43:47.506441 22344 solver.cpp:341] Rank : 0  Train net output #0: loss = 6.9324 (* 1 = 6.9324 loss)
• I0228 16:43:52.827987 22344 solver.cpp:323] Iteration 80, loss = 6.91919
• I0228 16:43:52.828023 22344 solver.cpp:758] Iteration 80, lr = 0.01
• I0228 16:43:57.895300 22344 solver.cpp:636] Snapshotting to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.caffemodel
• I0228 16:43:59.005717 22344 solver.cpp:921] Snapshotting solver state to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.solverstate
• I0228 16:43:59.557248 22344 solver.cpp:478] Iteration 100, loss = 6.89235, rank = 0
• I0228 16:43:59.557557 22344 caffe.cpp:351]  Avg. Time Taken: 30.635  Time Taken
Lab4-Task1: Running OSU-Caffe to train AlexNet (Batch Size 256)

$ srun -N 1 --reservation=dltutorial1 run_osucaffe_bench_single.sh 256

- I0228 16:53:00.356703 23082 solver.cpp:434] Solving AlexNet
- I0228 16:53:00.356704 23082 solver.cpp:435] Learning Rate Policy: step
- ...
- I0228 16:53:20.587594 23082 solver.cpp:758] Iteration 40, lr = 0.01
- I0228 16:53:30.146497 23082 solver.cpp:758] Iteration 60, lr = 0.01
- I0228 16:53:39.713945 23082 solver.cpp:758] Iteration 80, lr = 0.01
- I0228 16:53:49.935045 23082 solver.cpp:921] Snapshotting solver state to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.solverstate
- I0228 16:53:50.581598 23082 solver.cpp:478] Iteration 100, loss = 6.90535, rank = 0
- I0228 16:53:50.581756 23082 caffe.cpp:351] Avg. Time Taken: 52.4766  Approx. 2X more time than B.S 128
Lab4-Task1: Running OSU-Caffe to train AlexNet (Batch Size 512)

$ srun -N 1 --reservation=dltutorial1 run_osucaffe_bench_single.sh 512

- I0228 17:42:34.857414 24323 solver.cpp:434 Solving AlexNet
- I0228 17:42:34.857417 24323 solver.cpp:435 Learning Rate Policy: step
- I0228 17:42:36.816328 24323 solver.cpp:323 Iteration 0, loss = 6.9177
- ...
- I0228 17:43:55.889017 24323 solver.cpp:323 Iteration 60, loss = 6.91425
- I0228 17:43:55.889230 24323 solver.cpp:341 Rank : 0  Train net output #0: loss = 6.91425 (* 1 = 6.91425 loss)
- I0228 17:43:55.889253 24323 solver.cpp:758 Iteration 60, lr = 0.01
- I0228 17:44:22.362699 24323 solver.cpp:323 Iteration 80, loss = 6.91336
- I0228 17:44:22.362742 24323 solver.cpp:341 Rank : 0  Train net output #0: loss = 6.91336 (* 1 = 6.91336 loss)
- I0228 17:44:22.362761 24323 solver.cpp:758 Iteration 80, lr = 0.01
- I0228 17:44:47.501485 24323 solver.cpp:636 Snapshotting to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.caffemodel
- I0228 17:44:48.634145 24323 solver.cpp:921 Snapshotting solver state to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.solverstate
- I0228 17:44:49.529901 24323 solver.cpp:478 Iteration 100, loss = 6.90982, rank = 0
- I0228 17:44:49.529937 24323 solver.cpp:485 Optimization Done.
- I0228 17:44:49.529944 24323 caffe.cpp:329 Optimization Done.
- I0228 17:44:49.530027 24323 caffe.cpp:351 Avg. Time Taken: 137.213 2.6X more time than B.S 256

Additional consideration: I/O bottleneck!
Lab4-Task2: Strong Scaling for AlexNet Training (BatchSize 128)

$ srun -N 2 --reservation=dltutorial1 run_osucaffe_bench_multi_mv2gdr_strong.sh 128

- I0228 16:47:07.413712 22587 solver.cpp:434] Solving AlexNet
- I0228 16:47:07.898988 22587 solver.cpp:323] Iteration 0, loss = 6.92946
- I0228 16:47:08.013036 22587 solver.cpp:758] Iteration 0, lr = 0.01
  - ...
- I0228 16:47:22.952098 22587 solver.cpp:758] Iteration 80, lr = 0.01

1.3X on 2 GPUs
Lab4-Task3: Larger Batch Size to scale better! (BatchSize 256)

$ srun -N 2 --reservation=dltutorial1 run_osucaffe_bench_multi_mv2gdr_strong.sh 256

- I0228 16:52:08.204085 22856 solver.cpp:434] Solving AlexNet
- I0228 16:52:08.848337 22856 solver.cpp:323] Iteration 0, loss = 6.89747
- I0228 16:52:08.848385 22856 solver.cpp:341] Rank : 0  Train net output #0: loss = 6.89747 (* 1 = 6.89747 loss)
- ...
- I0228 16:52:27.134052 22856 solver.cpp:758] Iteration 60, lr = 0.01
- I0228 16:52:33.130359 22856 solver.cpp:758] Iteration 80, lr = 0.01
- I0228 16:52:38.891939 22856 solver.cpp:636] Snapshotting to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.caffemodel
- I0228 16:52:40.020539 22856 solver.cpp:921] Snapshotting solver state to binary proto file models/bvlc_alexnet/caffe_alexnet_train_iter_100.solverstate
- I0228 16:52:40.576818 22856 solver.cpp:478] Iteration 100, loss = 6.90319, rank = 0

1.5X on 2 GPUs
Lab4-Task3: Larger Batch Size to scale better! (BatchSize 512)

$ srun -N 2 --reservation=dltutorial1 run_osucaffe_bench_multi_mv2gdr_strong.sh 512

• I0228 21:28:35.449717 32383 solver.cpp:434] Solving AlexNet
• I0228 21:28:36.611455 32383 solver.cpp:323] Iteration 0, loss = 6.91593
• I0228 21:28:57.882804 32383 solver.cpp:758] Iteration 40, lr = 0.01
• I0228 21:29:08.401996 32383 solver.cpp:323] Iteration 60, loss = 6.91009
• I0228 21:29:08.402086 32383 solver.cpp:341] Rank : 0  Train net output #0: loss = 6.91009 (* 1 = 6.91009 loss)
• I0228 21:29:08.455415 32383 solver.cpp:758] Iteration 60, lr = 0.01
• I0228 21:29:19.028206 32383 solver.cpp:758] Iteration 80, lr = 0.01
• I0228 21:29:29.078667 32383 solver.cpp:636] Snapshotting to binary proto file /tmp/caffe_iter_100.caffemodel
• I0228 21:29:30.425010 32383 solver.cpp:478] Iteration 100, loss = 6.91818, rank = 0

2.4X on 2 GPUs

Super Linear Speedup!

Effect of better I/O from two nodes
Overview

• Introduction
• Setting up the Lab
• Understanding CUDA-Aware MPI
• Lab 1 - Distributed Training with TensorFlow
• Lab 2 - High-Performance Particle Simulation with HOOMD-blue
• Lab 3 - OSU Micro-Benchmarks using MVAPICH2
• Lab 4 - Distributed Training using OSU-Caffe

• **Homeworks**
  – HW 1 - Benchmarking InfiniBand Latency
  – HW 2 - Benchmarking InfiniBand Bandwidth

• Conclusion
Benchmarking InfiniBand Latency

```sh
dsrun --N 2 --reservation=dltutorial1 run_perftest_lat.sh
```

Executing `/usr/bin/ib_send_lat -d mlx4_0 -a' on the server
Executing `/usr/bin/ib_send_lat -d mlx4_0 -a node008' on the client

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#iterations</th>
<th>t_min[usec]</th>
<th>t_max[usec]</th>
<th>t_typical[usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1000</td>
<td>1.48</td>
<td>11.14</td>
<td>1.51</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>1.48</td>
<td>6.54</td>
<td>1.52</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1.51</td>
<td>6.55</td>
<td>1.54</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>1.50</td>
<td>6.56</td>
<td>1.53</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>1.46</td>
<td>6.59</td>
<td>1.55</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>1.53</td>
<td>6.64</td>
<td>1.57</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>1.66</td>
<td>6.86</td>
<td>1.70</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>2.20</td>
<td>7.67</td>
<td>2.24</td>
</tr>
</tbody>
</table>
## Benchmarking InfiniBand Latency

<table>
<thead>
<tr>
<th>#bytes</th>
<th>ib_send_lat</th>
<th>ib_write_lat</th>
<th>ib_read_lat</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.53</td>
<td>1.44</td>
<td>2.79</td>
</tr>
<tr>
<td>4</td>
<td>1.53</td>
<td>1.44</td>
<td>2.79</td>
</tr>
<tr>
<td>8</td>
<td>1.56</td>
<td>1.45</td>
<td>2.79</td>
</tr>
<tr>
<td>16</td>
<td>1.55</td>
<td>1.45</td>
<td>2.77</td>
</tr>
<tr>
<td>32</td>
<td>1.57</td>
<td>1.49</td>
<td>2.78</td>
</tr>
<tr>
<td>64</td>
<td>1.60</td>
<td>1.50</td>
<td>2.79</td>
</tr>
<tr>
<td>128</td>
<td>1.72</td>
<td>1.64</td>
<td>2.82</td>
</tr>
<tr>
<td>256</td>
<td>2.28</td>
<td>2.31</td>
<td>2.98</td>
</tr>
<tr>
<td>512</td>
<td>2.51</td>
<td>2.55</td>
<td>3.21</td>
</tr>
<tr>
<td>1024</td>
<td>2.94</td>
<td>2.99</td>
<td>3.65</td>
</tr>
<tr>
<td>2048</td>
<td>3.78</td>
<td>3.82</td>
<td>4.48</td>
</tr>
<tr>
<td>4096</td>
<td>5.44</td>
<td>5.48</td>
<td>6.15</td>
</tr>
<tr>
<td>8192</td>
<td>6.63</td>
<td>6.66</td>
<td>7.33</td>
</tr>
<tr>
<td>16384</td>
<td>9.03</td>
<td>9.03</td>
<td>9.66</td>
</tr>
<tr>
<td>32768</td>
<td>14.37</td>
<td>13.69</td>
<td>14.94</td>
</tr>
</tbody>
</table>

*write is faster*

Latency increases with message size.
## Benchmarking InfiniBand Bandwidth

```bash
$ srun --N 2 --reservation=dltutorial1 run_perftest_bw.sh
```

Executing `/usr/bin/ib_send_bw -d mlx4_0 -a' on the server
Executing `/usr/bin/ib_send_bw -d mlx4_0 -a node008' on the client

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#iterations</th>
<th>BW peak[MB/sec]</th>
<th>BW average[MB/sec]</th>
<th>MsgRate[Mpps]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1000</td>
<td>9.29</td>
<td>7.35</td>
<td>3.852803</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>17.93</td>
<td>16.32</td>
<td>4.279208</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>37.24</td>
<td>33.28</td>
<td>4.362247</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>74.48</td>
<td>66.82</td>
<td>4.379319</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>148.68</td>
<td>133.50</td>
<td>4.374631</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>297.94</td>
<td>270.74</td>
<td>4.435800</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>594.73</td>
<td>534.51</td>
<td>4.378744</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>1087.02</td>
<td>1012.27</td>
<td>4.146262</td>
</tr>
</tbody>
</table>
# Benchmarking InfiniBand Bandwidth

<table>
<thead>
<tr>
<th>#Bytes</th>
<th>ib_send_bw</th>
<th>ib_write_bw</th>
<th>ib_read_bw</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7.55</td>
<td>9.27</td>
<td>7.51</td>
</tr>
<tr>
<td>4</td>
<td>16.19</td>
<td>18.76</td>
<td>15.46</td>
</tr>
<tr>
<td>8</td>
<td>32.51</td>
<td>37.48</td>
<td>30.90</td>
</tr>
<tr>
<td>16</td>
<td>65.76</td>
<td>75.04</td>
<td>62.81</td>
</tr>
<tr>
<td>32</td>
<td>131.55</td>
<td>150.10</td>
<td>122.95</td>
</tr>
<tr>
<td>64</td>
<td>258.75</td>
<td>299.91</td>
<td>243.88</td>
</tr>
<tr>
<td>128</td>
<td>518.18</td>
<td>599.93</td>
<td>490.39</td>
</tr>
<tr>
<td>256</td>
<td>985.51</td>
<td>1199.01</td>
<td>943.38</td>
</tr>
<tr>
<td>512</td>
<td>1944.43</td>
<td>2384.70</td>
<td>1929.01</td>
</tr>
<tr>
<td>1024</td>
<td>2546.32</td>
<td>2846.36</td>
<td>3030.27</td>
</tr>
<tr>
<td>2048</td>
<td>2842.59</td>
<td>3035.32</td>
<td>3188.36</td>
</tr>
<tr>
<td>4096</td>
<td>3010.86</td>
<td>3126.79</td>
<td>3170.41</td>
</tr>
<tr>
<td>8192</td>
<td>3114.94</td>
<td>3180.60</td>
<td>3185.02</td>
</tr>
<tr>
<td>16384</td>
<td>3170.75</td>
<td>3208.95</td>
<td>3212.28</td>
</tr>
<tr>
<td>32768</td>
<td>3203.25</td>
<td>3222.59</td>
<td>3201.07</td>
</tr>
</tbody>
</table>

- **write is faster**
- Bandwidth Increases until Link is Saturated
Overview

• Introduction
• Setting up the Lab
• Understanding CUDA-Aware MPI
• Lab 1 - Distributed Training with TensorFlow
• Lab 2 - High-Performance Particle Simulation with HOOMD-blue
• Lab 3 - OSU Micro-Benchmarks using MVAPICH2
• Lab 4 - Distributed Training using OSU-Caffe
• Homeworks

• Conclusion
Conclusion

- Introduced the trends in HPC and AI
- Covered basics of CUDA-Aware MPI and its benefits
- Performed four Lab tasks in HPC/AI applications
- Additional Homework tasks are provided for experimentation
Thank You!

{panda,subramon}@cse.ohio-state.edu

{awan.10,ching.368}@osu.edu

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

The High-Performance MPI/PGAS Project
http://mvapich.cse.ohio-state.edu/

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