Choose the Best Accelerated Technology

Distributed DL/ML Solutions for HPC systems

Shailen Sobhee – AI Engineer

15 June 2021
Agenda

- Types of parallelism
- Distribution strategy for
  - Machine Learning
    - daal4py from oneDAL
  - Deep Learning
    - Horovod with oneCCL
    - torch-ccl example
Types of parallelism

- **SIMD**: Single instruction multiple data (Data Parallel)
  - The same instruction is simultaneously applied on multiple data items

- **MIMD**: Multiple instructions multiple data (Task Parallel)
  - Different instructions on different data

- **SPMD**: Single program multiple data (MPI Parallel)
  - This is the message passing programming on distributed systems
Shared vs distributed memory system

- **Shared memory**
  - There is a unique address space shared between the processors
  - All the processors can access the same memory

- **Distributed memory**
  - Each processor has its own local memory
  - Messages are exchanged between the processors to communicate the data
What is high-performance computing (HPC)?

- Leveraging distributed compute resources to solve complex problems with large datasets
- Terabytes to petabytes to zettabytes of data
- Results in minutes to hours instead of days or weeks

Cluster manager runs workloads on distributed resources, such as CPUs, FPGAs, GPUs and disk drives all interconnected via network.

Submit job to the cluster manager

Get the result back to analyze

Compute resources

Management services

Storage resources
Domain decomposition method for HPC

- The domain decomposition is a technique for dividing a computational problem in several parts (domains) allowing to solve a large problem on the available resources
  - **Partition** the data, assign them to each resource and associate the computation
  - **Communication** happens to eventually exchange intermediate results
  - **Aggregate** the results from the different resources
Distributing strategy for machine learning
From Prototype to Production

Why distributed ML/DL (1/2)

• Most Machine Learning tasks assume the data can be easily accessible, but:
  • Data loading on a single machine can be a bottleneck in case of large amount of data
  • To run production applications large memory systems is required (data not fitting in the local computer RAM)
  • Traditional sequential algorithms are not suitable in case of distributed memory system
• Time to solution is critical on highly competitive market.
Why distributed ML/DL (2/2)

• Deep Learning training takes time:
  • Computational complexity of DL training can be up to 100+ ExaFLOP (1 ExaFLOP = $10^{18}$ op);
  • Typical single node performance is up-to tens of TeraFLOPS (1 TF = $10^{12}$ op/sec);
  • Peak performance of most powerful HPC clusters is up-to tens of PetaFLOPS (1 PF = $10^{15}$ op/sec).

• Time to solution is critical on highly competitive market.
Intel® daal4py

• **daal4py** makes your Machine Learning algorithms in Python lightning fast and easy to use

• For scaling capabilities, daal4py also provides the ability to do distributed machine learning using Intel® MPI library

• daal4py operates in SPMD style (Single Program Multiple Data), which means your program is executed on several processes (e.g. similar to MPI)

• The use of MPI is not required for daal4py's SPMD-mode to work, all necessary communication and synchronization happens under the hood of daal4py

• It is possible to use daal4py and mpi4py in the same program
Scaling Machine Learning Beyond a Single Node

Simple Python* API
Powers scikit-learn*
Powered by Intel® oneDAL

Scalable to multiple nodes

> python -m daal4py <your-scikit-learn-script>

import daal4py.sklearn

daal4py.sklearn.patch_sklearn()

https://intelpython.github.io/daal4py/sklearn.html#
oneAPI Data Analytics Library (oneDAL)

Use directly for:
- Scaling to multiple nodes
- Streaming data
- Non-homogeneous dataframes

### Equivalents
- PCA
- Kmeans
- LinearRegression
- Ridge
- SVC
- pairwise_distances
- Logistic_regression_path

### oneDAL

**Scikit-Learn* Equivalent**
USE_DAL4PY_SKLEARN=YES

**Scikit-Learn* API Compatible**

KNeighborsClassifier
RandomForestClassifier
RandomForestRegressor

daal4py

oneDAL
**Processing Modes**

**Batch Processing**

\[ R = F(D_1, \ldots, D_k) \]

```
import d4p
kmeans_init(10, method="plusPlusDense")
```

**Distributed Processing**

\[ R = F(R_1, \ldots, R_k) \]

```
import d4p
kmeans_init(10, method="plusPlusDense", distributed="True")
```

**Online Processing**

\[ S_{i+1} = T(S_i, D_i) \]

\[ R_{i+1} = F(S_{i+1}) \]

```
import d4p
kmeans_init(10, method="plusPlusDense", streaming="True")
```
Speedup of oneDAL-Powered Scikit-learn* over Original Scikit-learn

Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Your costs and results may vary. Intel technologies may require enabled hardware, software, or service activation.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Configuration: Testing by Intel as of 10/23/2020. Intel® oneAPI Data Analytics Library 2021.1 (oneDAL), Scikit-learn 0.23.1, Intel® Distribution for Python 3.8; Intel® Xeon(R) Platinum 8280L CPU @ 2.70GHz, 2 sockets, 28 cores per socket, 10M samples, 10 features, 100 clusters, 100 iterations, float32.
oneDAL K-Means Fit, Cores Scaling
(10M samples, 10 features, 100 clusters, 100 iterations, float32)

Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.
Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.
Your costs and results may vary. Intel technologies may require enabled hardware, software, or service activation.
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.
Configuration: Testing by Intel as of 10/23/2020. Intel® oneAPI Data Analytics Library 2021.1 (oneDAL); Intel® Xeon® Platinum 8280LCPU @ 2.70GHz, 2 sockets, 28 cores per socket, 10M samples, 10 features, 100 clusters, 100 iterations, float32.
Strong & Weak Scaling via daal4py

On a 32-node cluster (1280 cores) daal4py computed linear regression of 2.15 TB of data in 1.18 seconds and 68.66 GB of data in less than 48 milliseconds.

On a 32-node cluster (1280 cores) daal4py computed K-Means (10 clusters) of 1.12 TB of data in 107.4 seconds and 35.76 GB of data in 4.8 seconds.
Distributed K-Means using daal4py

1) Performs a pixel-wise Vector Quantization (VQ) using K-Means
2) Implemented the domain decomposition according to:
   - `d4p.num_procs()`
   - `d4p.my_procid()`
3) Using the distributed algorithm from Daal4Py
   - `d4p.kmeans_init(n_colors, method="plusPlusDense", distributed=True)`
4) What is the meaning of `d4p.daalinit()` & `d4p.daalfini()`?
5) How does threading compare to multiprocessing in terms of performance?
Distributed K-Means Summary

- Each process (MPI rank) gets a different chunk of data
- Only process #0 reports results
- Inference is using the same routines as training with 0 maximum iterations and centroid assignment
- There is no oversubscription since DAAL only sees the cores “owned” by the corresponding MPI rank
K-Means using daal4py (batch)

```python
import daal4py as d4p

# daal4py accepts data as CSV files, numpy arrays or pandas dataframes
# here we let daal4py load process-local data from csv files
data = "kmeans_dense.csv"

# Create algob object to compute initial centers
init = d4p.kmeans_init(10, method="plusPlusDense")
# compute initial centers
ires = init.compute(data)
# results can have multiple attributes, we need centroids
Centroids = ires.centroids
# compute initial centroids & kmeans clustering
result = d4p.kmeans(10).compute(data, centroids)
```
import daal4py as d4p

# initialize distributed execution environment
d4p.daalinit()

# daal4py accepts data as CSV files, numpy arrays or pandas dataframes
# here we let daal4py load process-local data from csv files
data = "kmeans_dense_{}.csv".format(d4p.my_procid())

# compute initial centroids & kmeans clustering
init = d4p.kmeans_init(10, method="plusPlusDense", distributed=True)
centroids = init.compute(data).centroids
result = d4p.kmeans(10, distributed=True).compute(data, centroids)

mpirun -n 4 python ./kmeans.py
Distribution strategy for deep learning
Deep Learning Training procedure

- **Forward propagation**: calculate loss function based on the input batch and current weights;
- **Backward propagation**: calculate error gradients w.r.t. weights for all layers (using chain rule);
- **Weights update**: use gradients to update weights; there are different algorithms exist – vanilla SGD, Momentum, Adam, etc.

\[ W_{n}^{*} = W_{n} - \alpha \frac{\partial E}{\partial W_{n}} \] or variants
Neural Network parallelism

Data is processed in increments of N. Work on minibatch samples and distributed among the available resources.

The work is divided according to the neurons in each layer. The sample minibatch is copied to all processors which compute part of the DNN.

Multi-node parallelization

- **Data parallelism:**
  - Replicate the model across nodes;
  - Feed each node with its own batch of input data;
  - Communication for gradients is required to get their average across nodes;
  - Can be either
    - *AllReduce* pattern
    - *ReduceScatter* + *AllGather* patterns
Multi-node parallelization

- **Model parallelism:**
  - Model is split across nodes;
  - Feed each node with the same batch of input data;
  - Communication for partial activations is required to gather the result;
Multi-node parallelization

- **What parallelism flavor to use?**
  - Use model parallelism when volume of gradients is much higher than volume of activations or when model doesn’t fit memory;
  - Use data parallelism otherwise;
  - Parallelism choice affects activations/gradients ratio
    - Data parallelism at scale makes activations << weights
    - Model parallelism at scale makes weights << activations
  - There’re also other parallelism flavors – pipelined, spatial, etc.
Parameter Server

Tree using gRPC calls
Horovod

Ring All-Reduce using MPI

Distributed Training for Deep Neural Network

Intel® oneAPI Collective Communications Library (oneCCL)
oneCCL provides optimized communication patterns for high performance on Intel CPUs & GPUs to distribute model training across multiple nodes.

Transparencyly supports many interconnects, such as Intel® Omni-Path Architecture, InfiniBand, & Ethernet.

Built on top of lower-level communication middleware-MPI & libfabrics.

Enables efficient implementations of collectives used for deep learning training-all-gather, all-reduce, & reduce-scatter.
Enables efficient implementations of collectives used for deep learning training – all-gather, all-reduce, and more

oneCCL is designed for easy integration into deep learning (DL) frameworks

Provides C++ API and interoperability with DPC++

Supported Collectives
- Allgatherv
- Allreduce
- Alltoallv
- Broadcast
- Reduce
- ReduceScatter
Intel® oneAPI Collective Communications Library

Key Features (part 2/2)

Deep Learning Optimizations include:
- Asynchronous progress for compute communication overlap
- Dedication of cores to ensure optimal network use
- Message prioritization, persistence, and out-of-order execution
- Collectives in low-precision data types

Supported Collectives
- Allgatherv
- Allreduce
- Alltoallv
- Broadcast
- Reduce
- ReduceScatter
Message Passing Interface (MPI)

$ mpirun -H 192.168.1.100,192.168.1.105 hostname

aipg-infra-07.intel.com

aipg-infra-09.intel.com

$ mpirun -H host1,host2,host3 python hello.py

Hello World!

Hello World!

Hello World!
Changes to TensorFlow

1. `import tensorflow as tf`
2. `import horovod.tensorflow as hvd`
3. `hvd.init()`
4. `opt = tf.train.AdagradOptimizer(0.01 * hvd.size())`
5. `opt = hvd.DistributedOptimizer(opt)`
6. `hooks = [hvd.BroadcastGlobalVariablesHook(0)]`
Sockets & Cores

**SOCKET**

Receptacle on the motherboard for one physically packaged processor.

**CORE**

A complete private set of registers, execution units, and queues to execute a program.
Multiple workers per CPU with OpenMPI

```bash
$ mpirun
  -H hostA,hostB,hostC
  -np 6
  --map-by ppr:1:socket:pe=2
  --oversubscribe
  --report-bindings
  python train_model.py
```
Multiple workers per CPU with Intel MPI

```bash
$ mpirun
   -H hostA, hostB, hostC
   -n 6
   -ppn 2
   -print-rank-map
   -genv I_MPI_PIN_DOMAIN=socket
   -genv OMP_NUM_THREADS=24
   -genv OMP_PROC_BIND=true
   -genv KMP_BLOCKTIME=1
python train_model.py
```
Multiple workers per CPU

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>hostA</td>
<td>[BB/BB/.../...][.../.../.../.../...]</td>
</tr>
<tr>
<td>R1</td>
<td>hostA</td>
<td>[.../.../.../...][BB/BB/.../...]</td>
</tr>
<tr>
<td>R2</td>
<td>hostB</td>
<td>[BB/BB/.../...][.../.../.../...]</td>
</tr>
<tr>
<td>R3</td>
<td>hostB</td>
<td>[.../.../.../...][BB/BB/.../...]</td>
</tr>
<tr>
<td>R4</td>
<td>hostC</td>
<td>[BB/BB/.../...][.../.../.../...]</td>
</tr>
<tr>
<td>R5</td>
<td>hostC</td>
<td>[.../.../.../...][BB/BB/.../...]</td>
</tr>
</tbody>
</table>

```bash
mpirun -H hostA,hostB,hostC -np 6 --map-by ppr:1:socket:pe=2 ...
```
Try it on GitHub

github.com/IntelAI/unet
BKC/BKM for HPC AI


Version: 1.1
Date of Issue: January 2019
Prepared By:
Aishwarya Bhandare1, Deepthi Karkada1, Kushal Datta1, Anupama Kurpad1, Vamsi Sripathi1, Sun Cho1, Vikram Salotra1
1Connectivity Group & 3Al Products Group, Data Center Group
Customer Solutions Technical Enabling, Intel Corporation

- Docker
- SLURM
- Singularity
- NFS
- Lustre
Distributed Training with \texttt{torch-ccl}

- Distributed Training Methods
  - Data Parallel
  - Model Parallel
  - Data + Model Parallel

- Types of Multi-worker communication
  - MPI
  - oneCCL
  - NCCL
  - Gloo
torch-ccl

- Holds PyTorch bindings for the Intel® oneAPI Collective Communications Library (oneCCL).
- Expand Pytorch C10D communication Library, dynamically loaded.
- A Github repository maintained by Intel
  - [https://github.com/intel/torch-ccl](https://github.com/intel/torch-ccl)

**Motivation**
- Speedup PyTorch multi-node training on IA with oneCCL

**Methodology**
- oneCCL is loaded as a PyTorch 3rd party communication library

**Features**
- C10D dynamic loading
- BF16 support
- CMP/COMM overlapping
torch-ccl sample code

```python
import os
import torch
import torch.nn as nn
from torch.nn.parallel import DistributedDataParallel as DDP
import torch.distributed as dist
import torch_ccl

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = nn.Linear(4, 5)

    def forward(self, input):
        return self.linear(input)

if __name__ == "__main__":
    os.environ['RANK'] = os.environ.get('PMI_RANK', -1)
    os.environ['WORLD_SIZE'] = os.environ.get('PMI_SIZE', -1)

    # Initialize the process group with ccl backend
    dist.init_process_group(backend='ccl')

    model = Model()
    if dist.get_world_size() > 1:
        model = DDP(model)

    for i in range(3):
        input = torch.randn(2, 4)
        labels = torch.randn(2, 5)
```

Only 3 changes needed from general torch DDP code

1. import torch_ccl
2. Access PMI_* environment variables
3. Set backend to 'ccl'

https://github.com/intel/optimized-models/tree/master/pytorch/distributed
Distributed Training on multiple sockets

```
source ~/.local/env/setvars.sh
export LD_PRELOAD="${CONDA_PREFIX}/lib/libomp.so"
export MASTER_ADDR="127.0.0.1"
export MASTER_PORT="29500"

# Example:
# Run 2 processes on 2 sockets. (28 cores/socket, 4 cores for CCL, 24 cores for computation)
#
# CCL_WORKER_COUNT means per instance threads used by CCL.
# CCL_WORKER_COUNT, CCL_WORKER_AFFINITY and _MPI_PIN_DOMAIN should be consistent.

export CCL_WORKER_COUNT=4
export CCL_WORKER_AFFINITY="0,1,2,3,28,29,31,32"

mpiexec.hydra -np 2 -ppn 2 -l -genv I_MPI_PIN_DOMAIN=[0x00000000FFFFF0,0xFFFFF0000000] \
   -genv KMP_BLOCKTIME=1 -genv KMP_AFFINITY=granularity=fine,compact,1,0 \
   -genv OMP_NUM_THREADS=24 python -u ut_memory.py
```

https://github.com/intel/optimized-models/tree/master/pytorch/distributed
Distributed Training on multiple nodes

```bash
source ~/.local/env/setvars.sh
export LD_PRELOAD="${CONDA_PREFIX}/lib/mpiomp5.so"
export MASTER_ADDR="10.xxx.xxx.xxx"  # IP address on which users launch MPI command
export MASTER_PORT="29500"

# Example:
# Run 4 processes on 2 Nodes, 2 sockets/Node (28 cores/socket, 4 cores for CCL, 24 cores for computation)
#
# CCL_WORKER_COUNT means per instance threads used by CCL.
# CCL_WORKER_COUNT, CCL_WORKER_AFFINITY and I_MPI_PIN_DOMAIN should be consistent.
#
# `hostfile`: add all Nodes' IP into this file

export CCL_WORKER_COUNT=4
export CCL_WORKER_AFFINITY="0,1,2,3,28,29,31,32"

mpiexec.hydra -f hostfile -np 4 -ppn 2 -l -genv I_MPI_PIN_DOMAIN=[0x00000000FFFFF0,0xFFFFFFFF00000000] \
    -genv KMP_BLOCKTIME=1 -genv KMP_AFFINITY=granularity=fine,compact,1,0 \n    -genv OMP_NUM_THREADS=24 python -u ut_memory.py
```

https://github.com/intel/optimized-models/tree/master/pytorch/distributed
Installation Guide

1. Build PyTorch from source
   • git clone https://github.com/pytorch/pytorch.git
   • git checkout 762270c

2. Build oneCCL from source
   • git clone https://github.com/oneapi-src/oneCCL.git

3. Build torch-ccl from source
   • git clone https://github.com/intel/torch-ccl.git

Installation

To install torch-ccl:

1. Install PyTorch.
2. Install Intel oneCCL (please refer to this page).
3. Source the oneCCL environment.

   $ source <ccl_install_path>/env/setvars.sh

4. Install the torch-ccl pip package.

   $ python setup.py install
HANDS-ON
Delete the checkpoint if needed, otherwise TF won't train any further

- rm -rf checkpoints

Let's start changing the number of MPI tasks, what performance difference would you expect?

- mpirun -prepend-rank -geny OMP_NUM_THREADS=2 -geny I_MPI_DEBUG=5 -n 2 python -u cnn_mnist-hvd.py
- mpirun -prepend-rank -geny OMP_NUM_THREADS=2 -geny I_MPI_DEBUG=5 -n 4 python -u cnn_mnist-hvd.py
- check the size of the dataset:
  - ls -lha ~/.keras/datasets/

Intel Python and Optimized Tensorflow

- source activate hvd-mpi
- pip show tensorflow | grep Location
  - useful to locate the TF installation for see the library linked: ldd $Location/tensorflow/libtensorflow...so
- rm-rf /tmp/*
- export MAKELDNN_VERBOSE=1
Tensorflow+Horovod/cnn_mnist-hvd.ipynb

1) How to initialize Horovod and why is it necessary?
2) Why is it necessary to adept the learning rate with larger batches?
3) How can you dynamically adept the learning rate?
4) How to identify rank #1 (0)?
5) Why is it necessary to adept the number of training steps according to the number of workers / larger batches?
6) How can you dynamically adept the number of training steps?
7) How is the single process performance vs 2 ranks vs 4 ranks?
MNIST CNN Horovod Demo Summary

- Horovod initializes the MPI communication, and therefore defines rank() and size()
- In order to reduce the Time To Train with multiple workers, the batch size needs to scale
- Same for the number of steps for training
- 4 ranks can be faster since less threading is required in small convolutions
Legal Disclaimer & Optimization Notice

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

- INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

- Copyright © 2021, Intel Corporation. All rights reserved. Intel, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
CASE-STUDY
Engagement overview on the ASPIRE project

Problem Statement: AI Algorithm to **segment** the Metabolic Tumour Volume (MTV) in oesophageal cancer

* **Segmentation:** Find the contour of the tumor on the CT scan
AI solution to for tumor segmentation:

- **Datacenter**
  - Labelled Data
  - CT scan
  - Radiologist’s mask
  - Deep Neural Network (U-Net model)
  - Training

- **On-Premise**
  - Trained model
  - New patient
  - Inferencing

- New patient
- CT scan
Training Platform configuration

AI Software stack Configuration
- Intel® Distribution for Python 3.7
- Intel-optimized TensorFlow 1.12
- OpenMPI 4.0.2
- OpenMP (gcc 4.8.5)
- Horovod 0.18.2

Hardware
- 2 Intel® Xeon® 6152 2S nodes (22 cores per socket) => 88 cores total
- 188 GB RAM / node (376 GB RAM total)
- 300 GB SSD / node (600 GB total)
- Omni-Path interconnect fabric
Model accuracy improvement

- Using CT scans alone gave low accuracy*
  - We included an additional PET channel
- U-Net Neural Network optimizations
  - Custom dropout rates on individual layers
  - Custom loss function (Dice Score Metric, Jaccard Metric and Tversky Loss)
  - Attention-Gating (available in TF 2.0)
Training performance improvement

- Intel-optimized TensorFlow with node-level optimizations
  - OMP_NUM_THREADS = #physical cores
  - KMP_BLOCKTIME=1
  - KMP_AFFINITY=granularity=fine,compact
  - INTER and INTRA THREADS

- Scaling-out: Horovod – MPI parameter optimizations
  - 72 hours -> 39 hours
Performance results (Brain Tumor)

Run configurations

- Tensorflow (Stock)
  - 1 worker
  - 1 node (28 cores)
  - Training time (hours) 76.2

- Intel-optimized Tensorflow
  - 1 worker
  - 1 node (28 cores)
  - Training time (hours) 43.75

- Horovod
  - 1 worker
  - 1 node (28 cores)
  - Training time (hours) 44.5

- Horovod
  - 2 workers
  - 1 node (56 cores)
  - Training time (hours) 24.2

- Horovod
  - 8 workers
  - 4 nodes (224 cores)
  - Training time (hours) 7.5

- Horovod
  - 16 workers
  - 4 nodes (224 cores)
  - Training time (hours) 5.5

Notes:
- Training time (hours) - lower is better