Choose the Best Accelerated Technology

Distributed DL/ML Solutions for HPC systems

Shailen Sobhee – AI Engineer

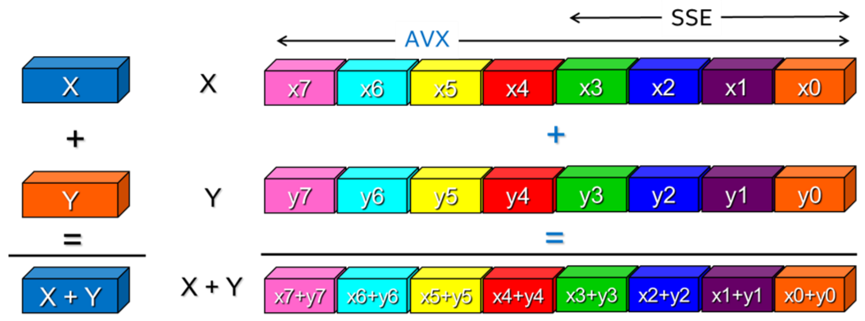
15 June 2021

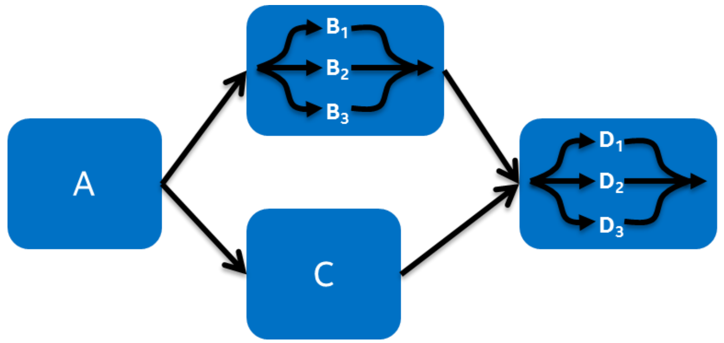
Agenda

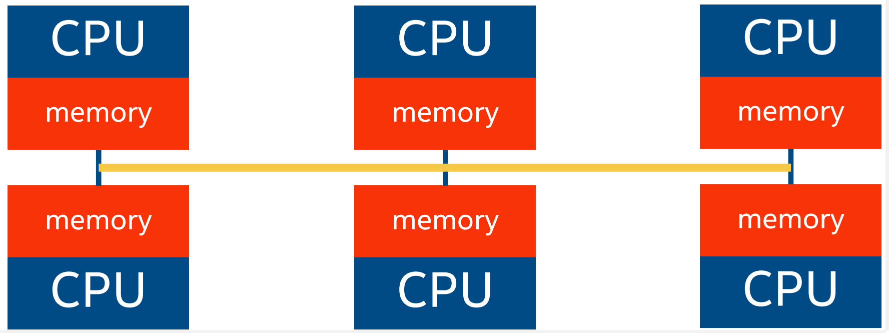
- Types of parallelism
- Distribution strategy for
  - Machine Learning
    - daal4py from oneDAL
  - Deep Learning
    - Horovod with oneCCL
    - torch-ccl example

Types of parallelism

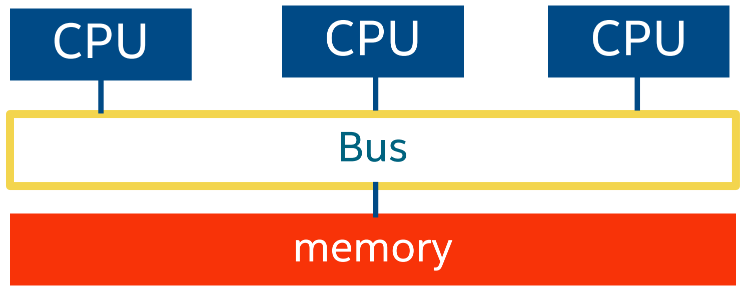
- **SIMD**: Single instruction multiple data (Data Parallel)
  - The same instruction is simultaneously applied on multiple data items
- **MIMD**: Multiple instructions multiple data (Task Parallel)
  - Different instructions on different data
- **SPMD**: Single program multiple data (MPI Parallel)
  - This is the message passing programming on distributed systems



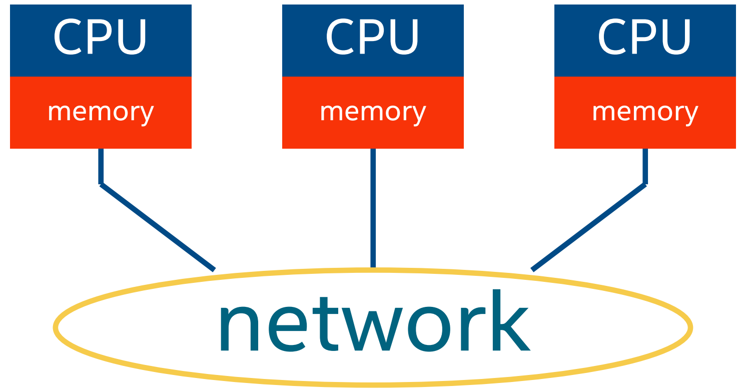




Shared vs distributed memory system



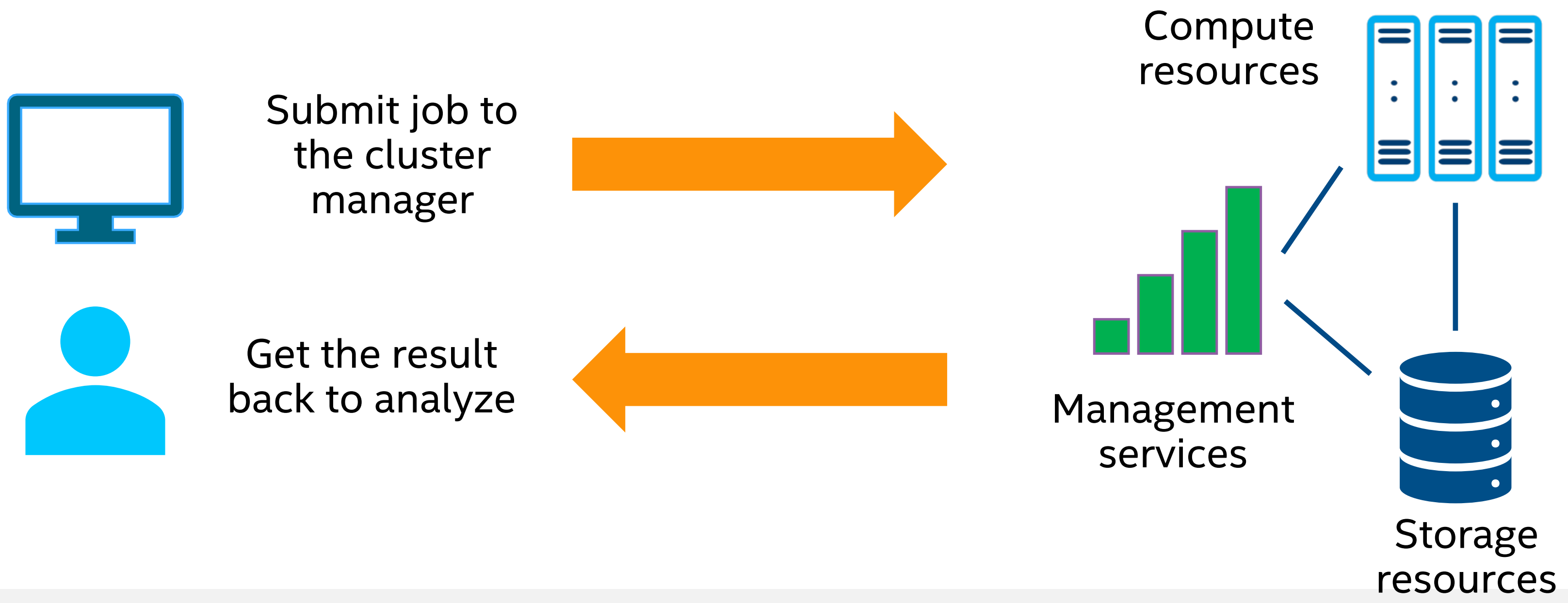
- Shared memory
  - There is a unique address space shared between the processors
  - All the processors can access the same memory



- 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)?

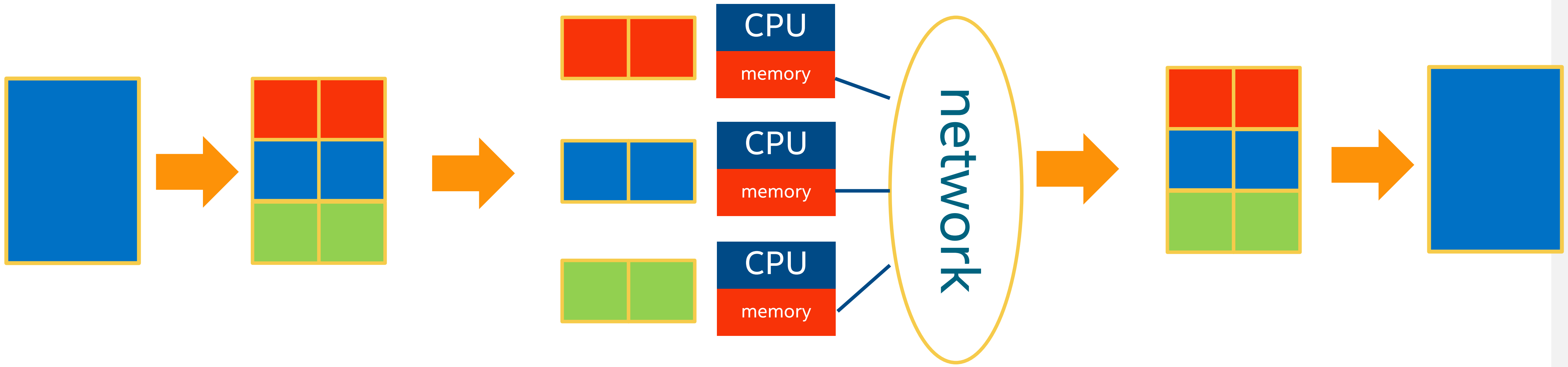
- Leveraging distributed compute resources to solve complex problems with large datasets
- Terabytes to petabytes to zettabytes of data
- Results in minutes to hours instead of days or weeks


Submit job to the cluster manager
Get the result back to analyze
Compute resources
Management services
Storage resources

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

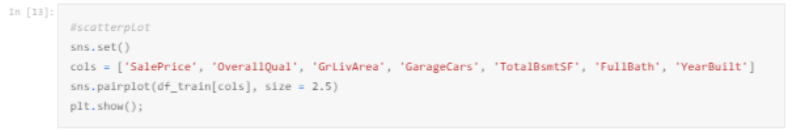
Domain decomposition method for HPC

- 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

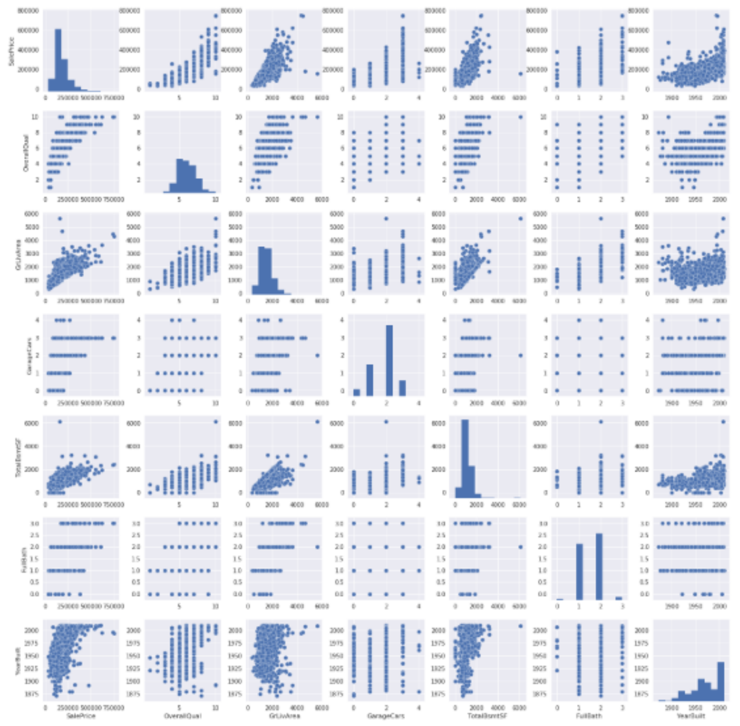


Distributing strategy for machine learning

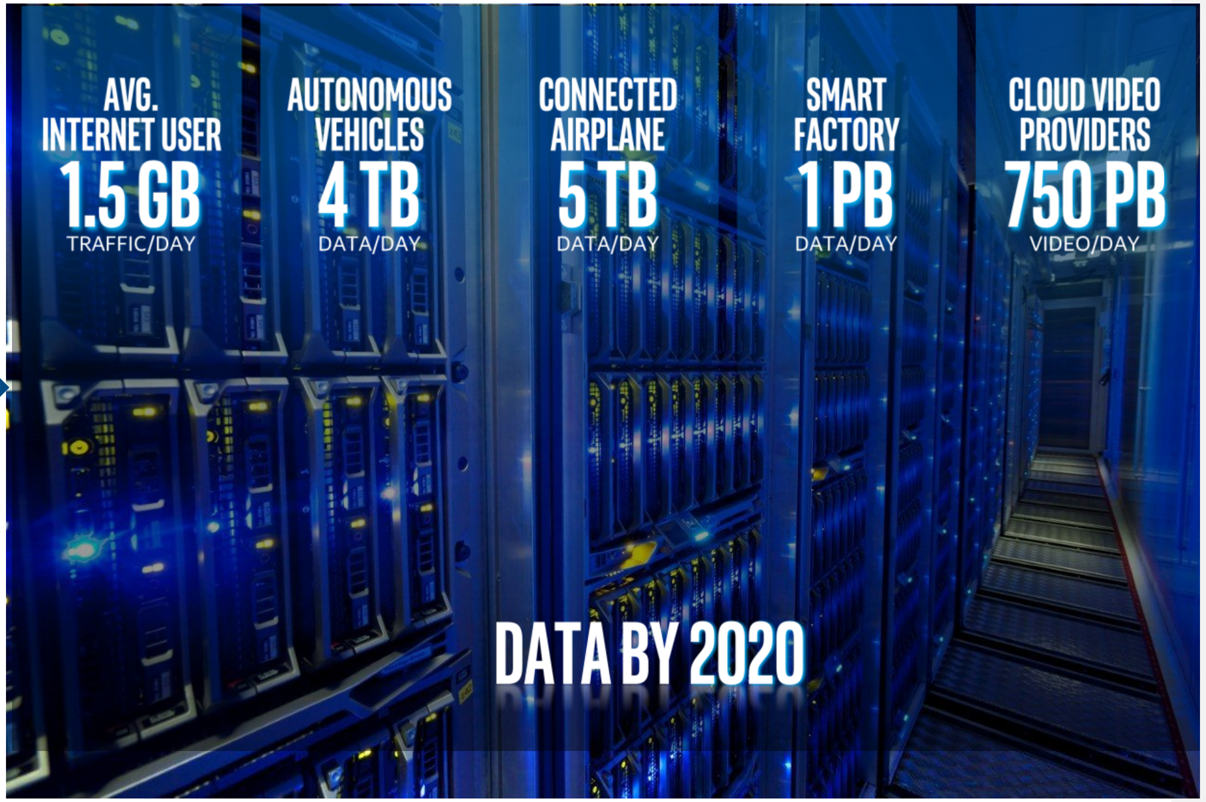
From Prototype to Production



```
In [13]:
#scatterplot
sns.set()
cols = ['SalePrice', 'OverallQual', 'GrLivArea', 'GarageCars', 'TotalBsmtSF', 'FullBath', 'YearBuilt']
sns.pairplot(df_train[cols], size = 2.5)
plt.show();
```



PERFORMANCE



https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python

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.

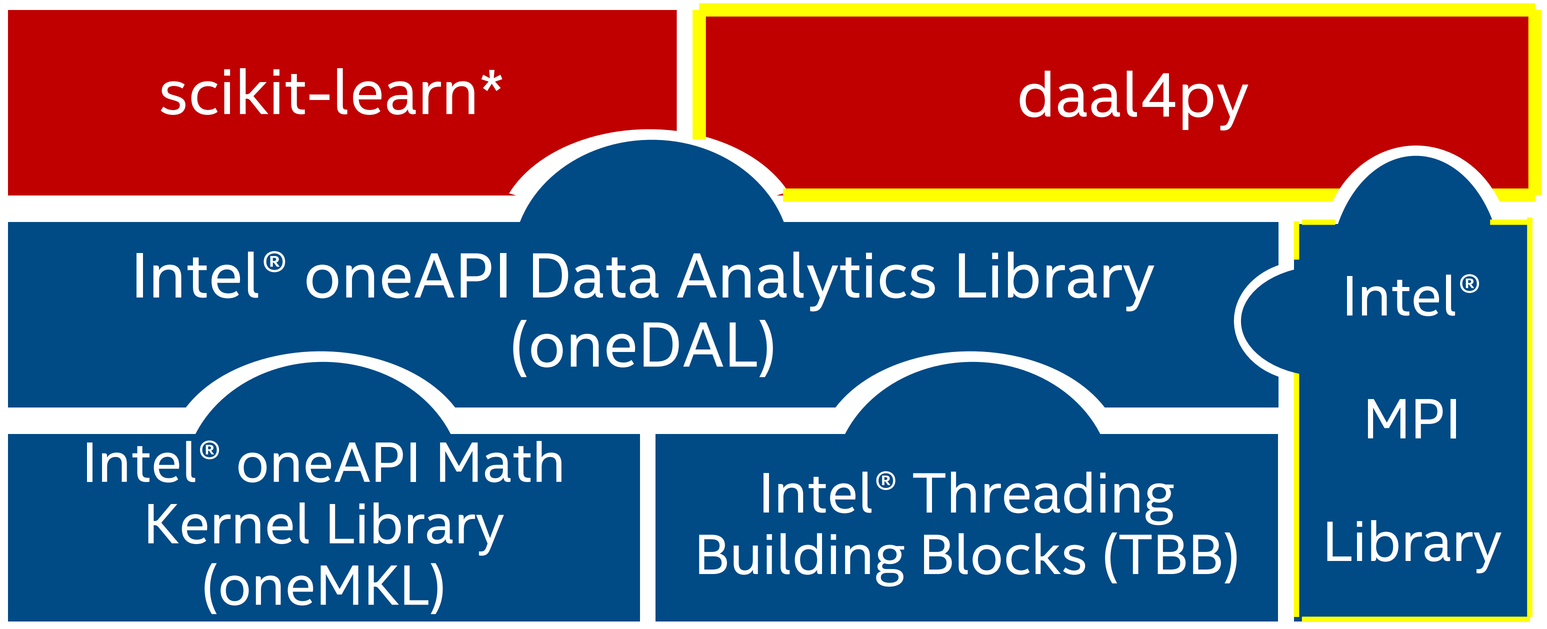
Why distributed ML/DL (2/2)

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


scikit-learn*
daal4py
Intel® oneAPI Data Analytics Library (oneDAL)
Intel® MPI Library
Intel® oneAPI Math Kernel Library (oneMKL)
Intel® Threading Building Blocks (TBB)

Simple Python* API

Powers scikit-learn*

Powered by Intel® oneDAL

Scalable to multiple nodes

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

Monkey-patch any scikit-learn* on the command-line

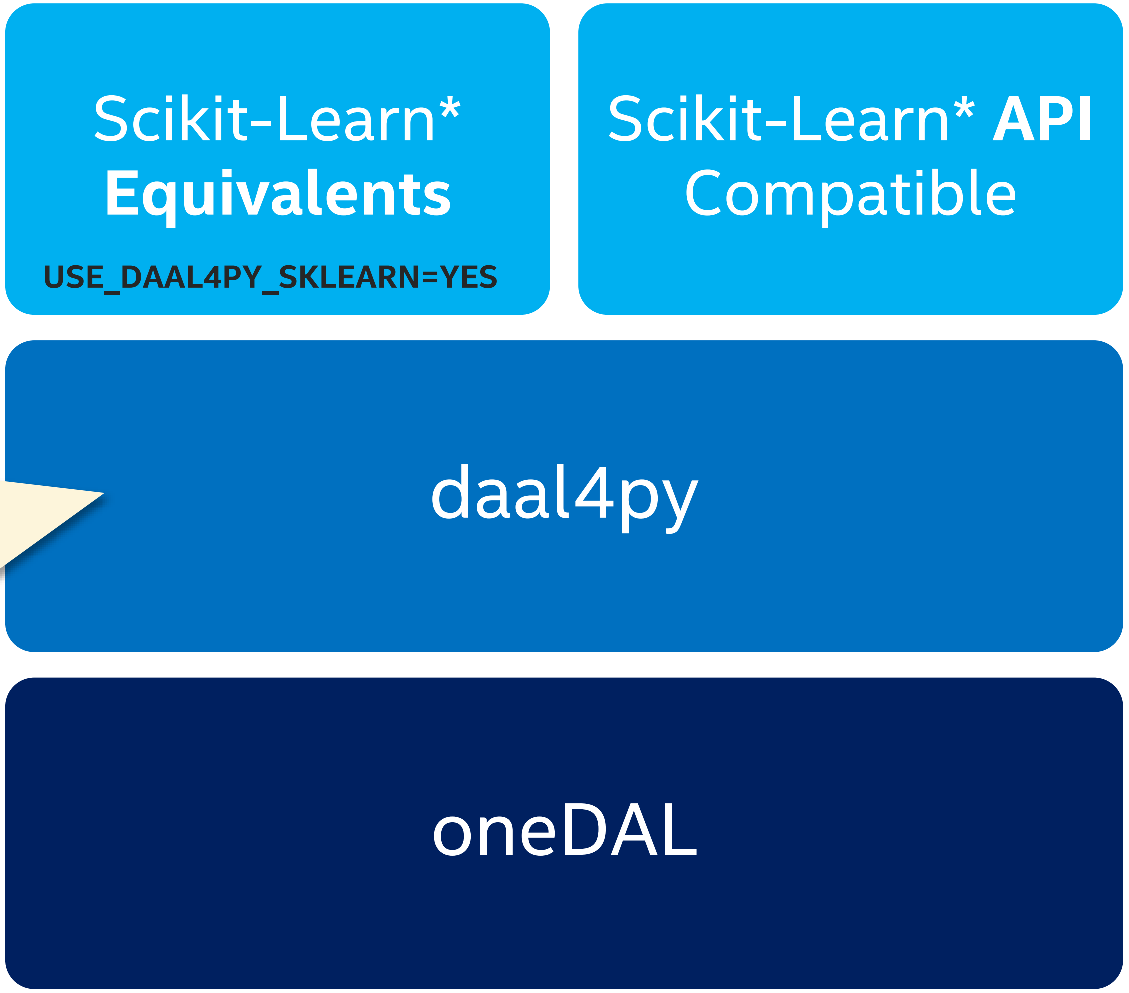
```
import daal4py.sklearn
daal4py.sklearn.patch_sklearn()
```

Monkey-patch any scikit-learn* programmatically

https://intelpython.github.io/daal4py/sklearn.html#

oneAPI Data Analytics Library (oneDAL)

PCA
Kmeans
LinearRegression
Ridge
SVC
pairwise_distances
Logistic_regression_path


Scikit-Learn* **Equivalents**
USE_DAAL4PY_SKLEARN=YES
Scikit-Learn* **API** Compatible
daal4py
oneDAL

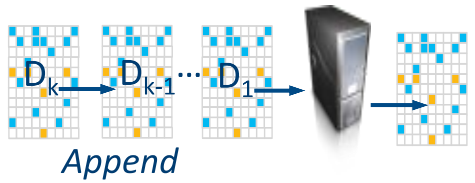
KNeighborsClassifier
RandomForestClassifier
RandomForestRegressor

Use directly for

- Scaling to multiple nodes
- Streaming data
- Non-homogeneous dataframes

Processing Modes

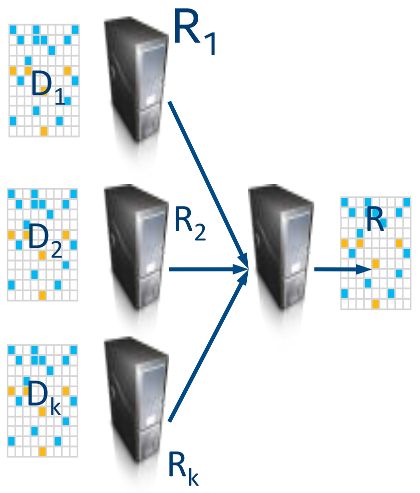
Batch Processing



$R = F(D_1,\ldots,D_k)$

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

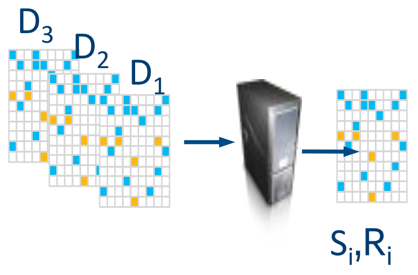
Distributed Processing



$R = F(R_1,\ldots,R_k)$

```
d4p.kmeans_init(10, method="plusPlusDense“, 
distributed=“True”)
```

Online Processing

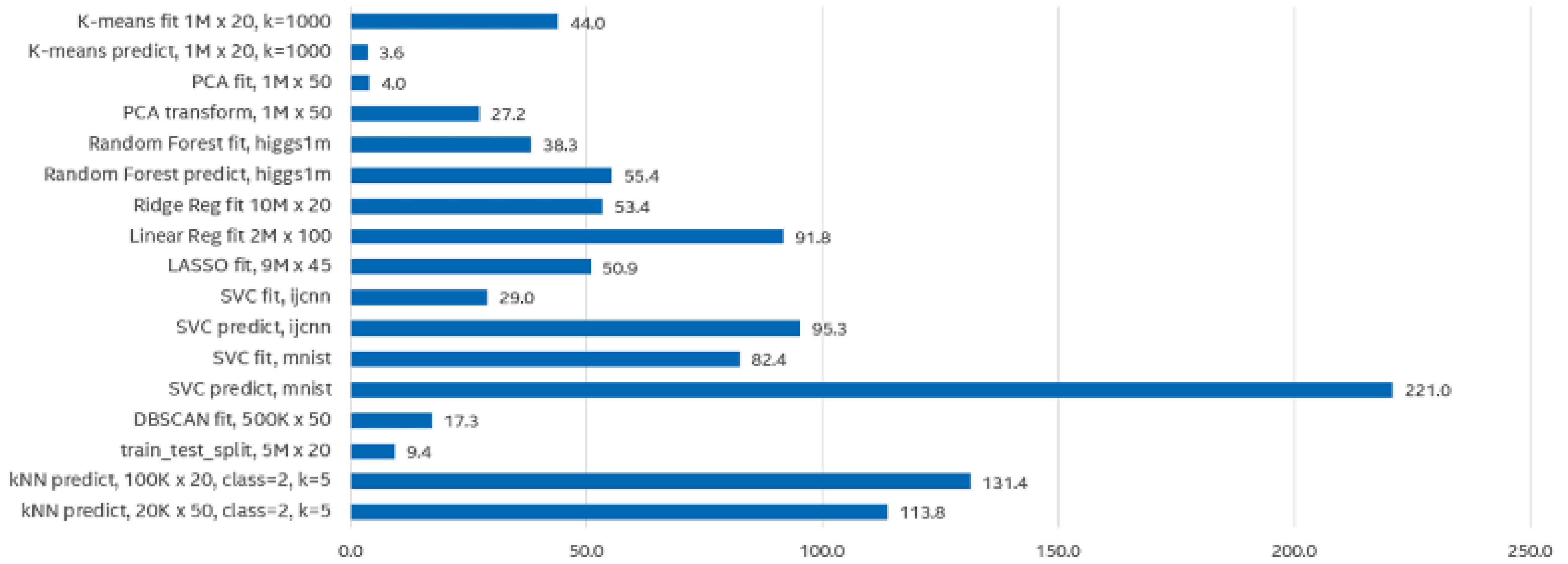


$S_{i+1} = T(S_i,D_i)$

$R_{i+1} = F(S_{i+1})$

```
d4p.kmeans_init(10, method="plusPlusDense“, 
streaming=“True”)
```

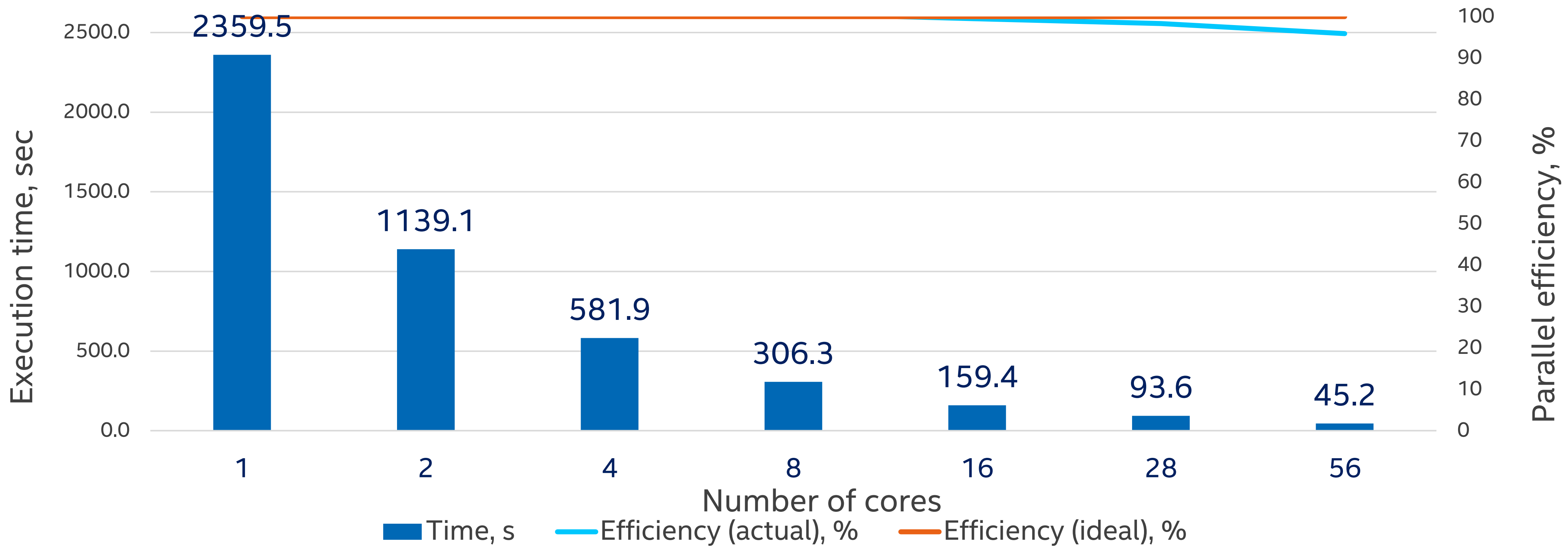
Speedup of oneDAL-Powered Scikit-learn* over Original Scikit-learn



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(R) Xeon(R) Platinum 8280LCPU @ 2.70GHz, 2 sockets, 28 cores per socket, 10M samples, 10 features, 100 clusters, 100 iterations, float32.

oneDAL K-Means Fit, Cores Scaling

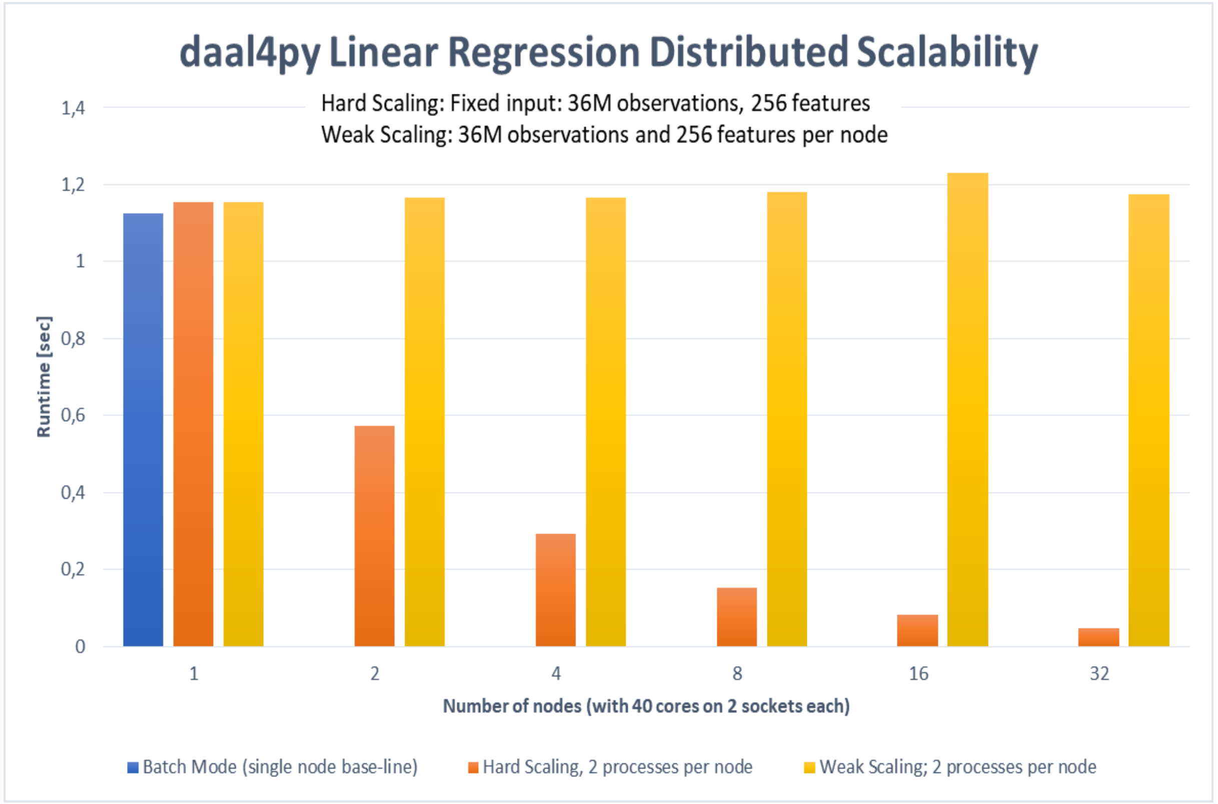
(10M samples, 10 features, 100 clusters, 100 iterations, float32)

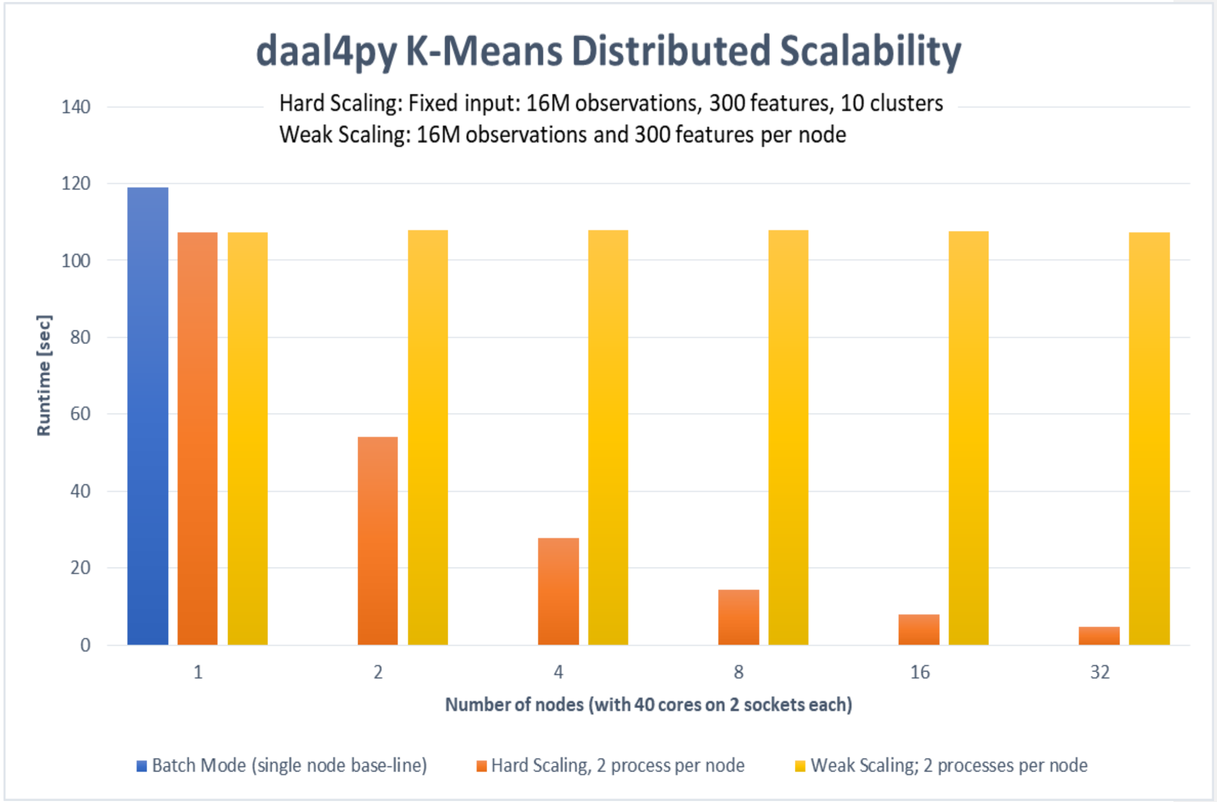


Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.
Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component 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

Hardware	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz, EIST/Turbo on
	2 sockets, 20 Cores per socket
	192 GB RAM
	16 nodes connected with Infiniband
Operating System	Oracle Linux Server release 7.4
Data Type	double





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.

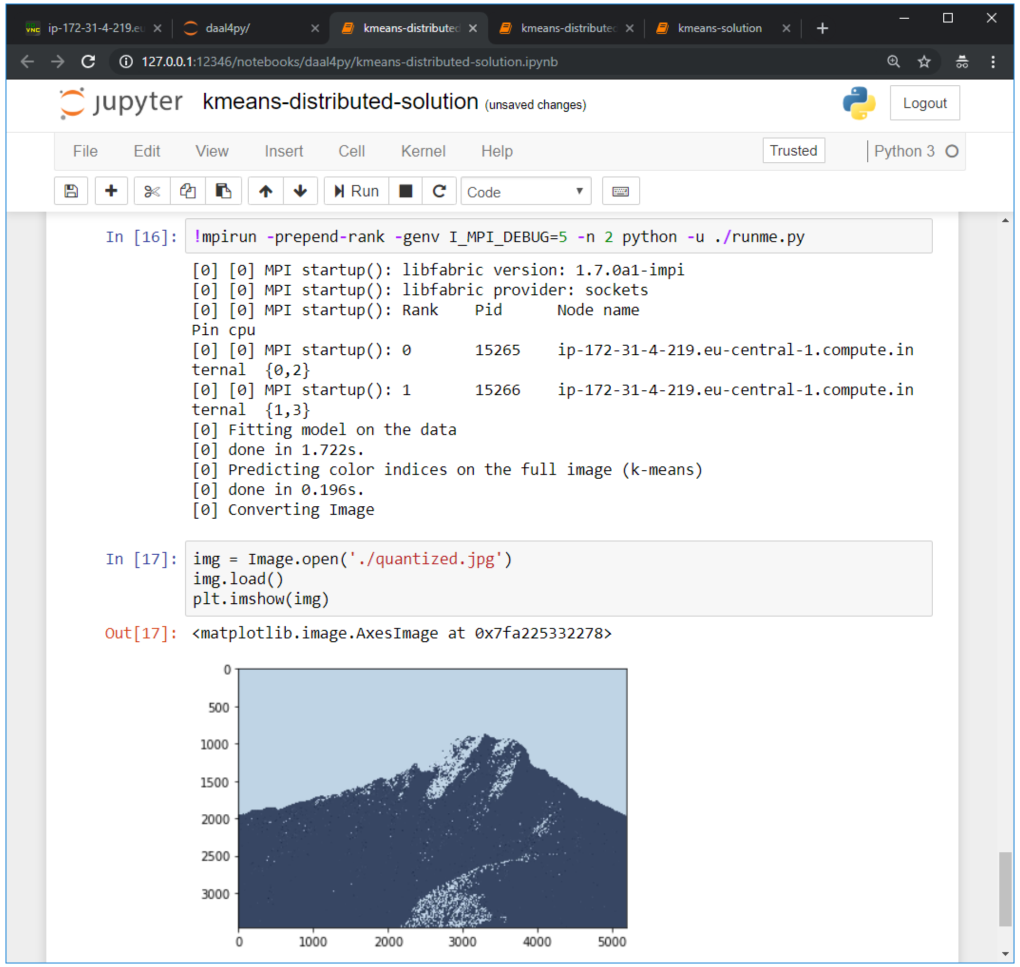
HANDS-ON

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) get's 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



ip-172-31-4-219.e
daal4py/
kmeans-distribute
kmeans-distribute
kmeans-solution
127.0.0.1:12346/notebooks/daal4py/kmeans-distributed-solution.ipynb
jupyter kmeans-distributed-solution (unsaved changes)
Logout
File Edit View Insert Cell Kernel Help
Trusted
Python 3
Run
Code

```
In [16]: !mpirun -prepend-rank -genv I_MPI_DEBUG=5 -n 2 python -u ./runme.py
[0] [0] MPI startup(): libfabric version: 1.7.0a1-impi
[0] [0] MPI startup(): libfabric provider: sockets
[0] [0] MPI startup(): Rank    Pid      Node name
Pin cpu
[0] [0] MPI startup(): 0       15265    ip-172-31-4-219.eu-central-1.compute.in
ternal  {0,2}
[0] [0] MPI startup(): 1       15266    ip-172-31-4-219.eu-central-1.compute.in
ternal  {1,3}
[0] Fitting model on the data
[0] done in 1.722s.
[0] Predicting color indices on the full image (k-means)
[0] done in 0.196s.
[0] Converting Image

In [17]: img = Image.open('./quantized.jpg')
         img.load()
         plt.imshow(img)

Out[17]: <matplotlib.image.AxesImage at 0x7fa225332278>
```

K-Means using daal4py (batch)

```
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)
```

Distributed K-Means using daal4py

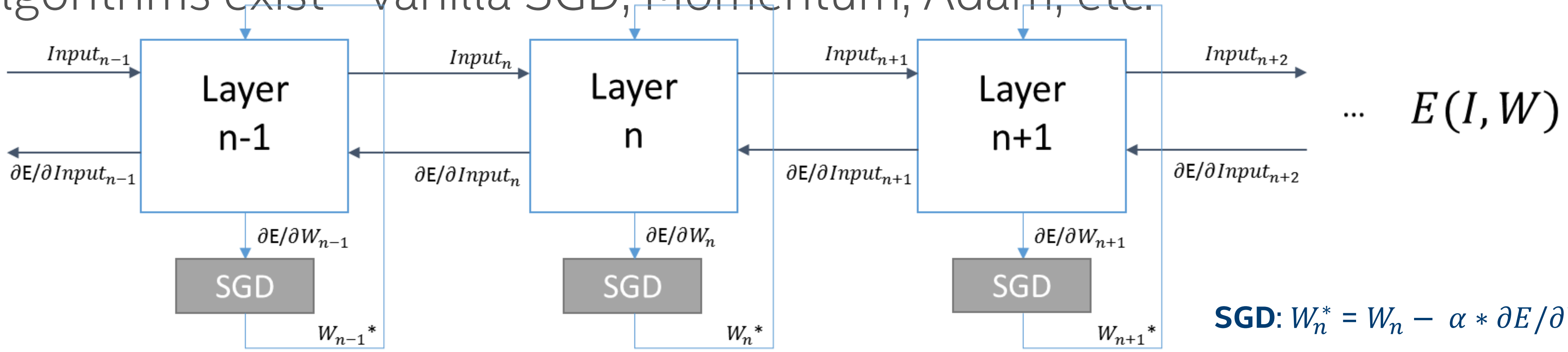
```python
import daal4py as d4p

# 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

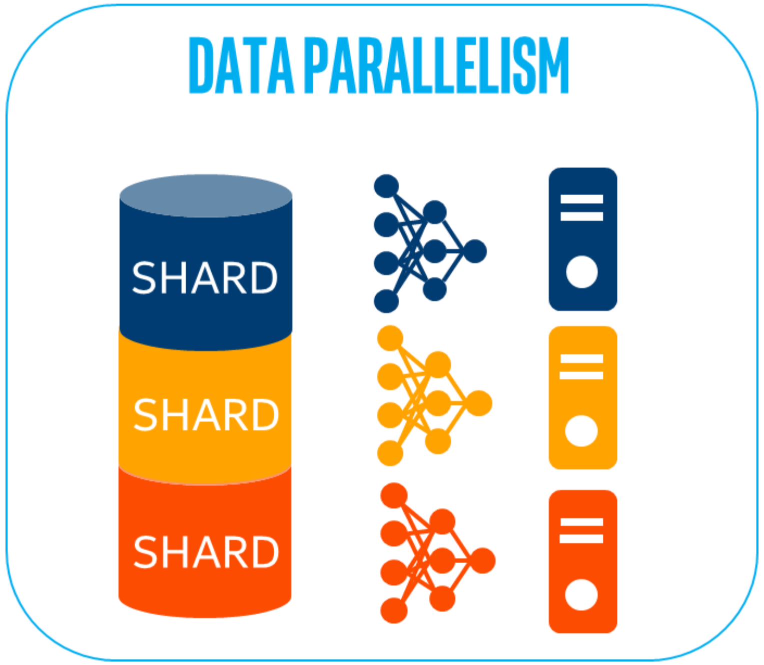
- **Forward propagation**: calculate loss function based on the input batch and current weights;
- **Backward propagation**: calculate error gradients w.r.t. weights for all layers (using chain rule);
- **Weights update**: use gradients to update weights; there are different algorithms exist - vanilla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

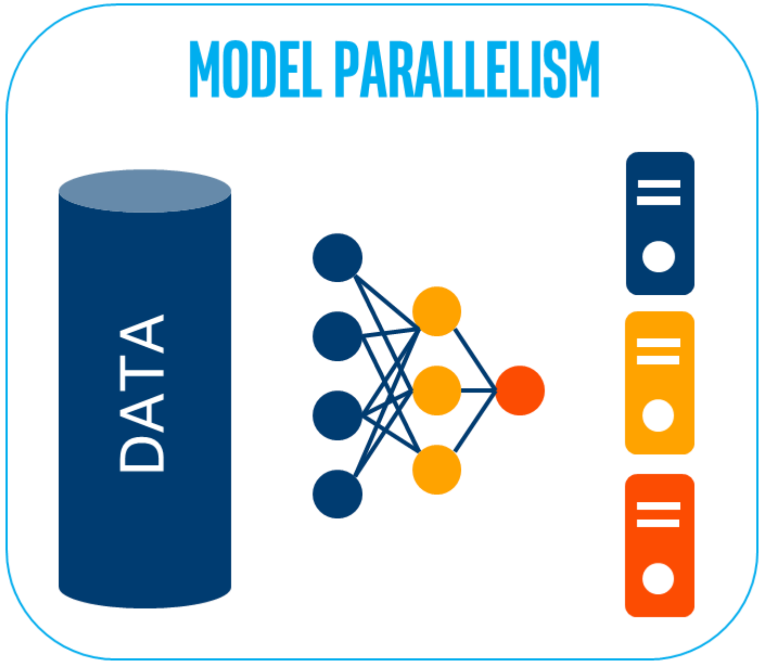


$Input_{n-1}$
Layer n-1
$Input_n$
Layer n
$Input_{n+1}$
Layer n+1
$Input_{n+2}$
...
$E(I,W)$
$\partial E/\partial Input_{n-1}$
$\partial E/\partial Input_n$
$\partial E/\partial Input_{n+1}$
$\partial E/\partial Input_{n+2}$
$\partial E/\partial W_{n-1}$
$\partial E/\partial W_n$
$\partial E/\partial W_{n+1}$
SGD
SGD
SGD
$W_{n-1}{}^*$
$W_n{}^*$
$W_{n+1}{}^*$

SGD: $W_n^* = W_n - \alpha * \partial E/\partial W_n$ or variants

Neural Network parallelism





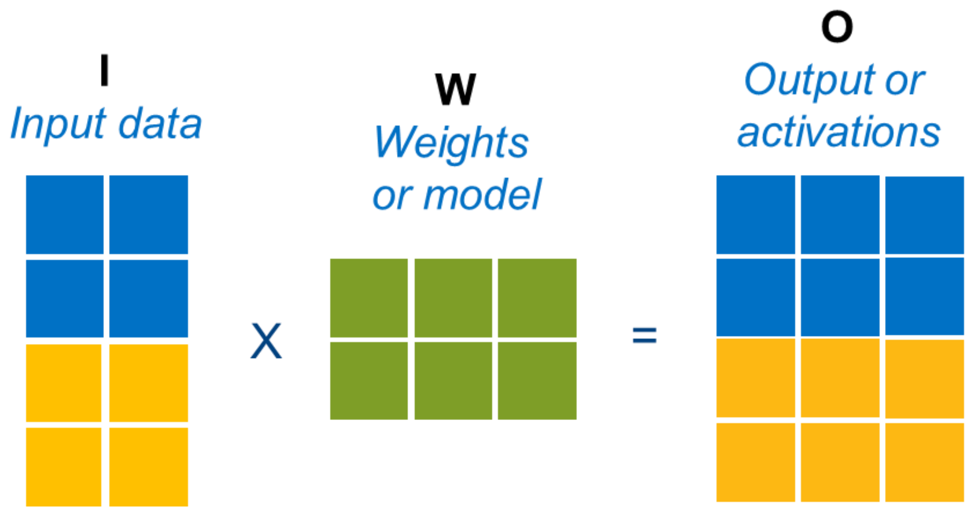
Data is processed in increments of N. Work on minibatch samples and distributed among the available resources.

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.

source: https://arxiv.org/pdf/1802.09941.pdf

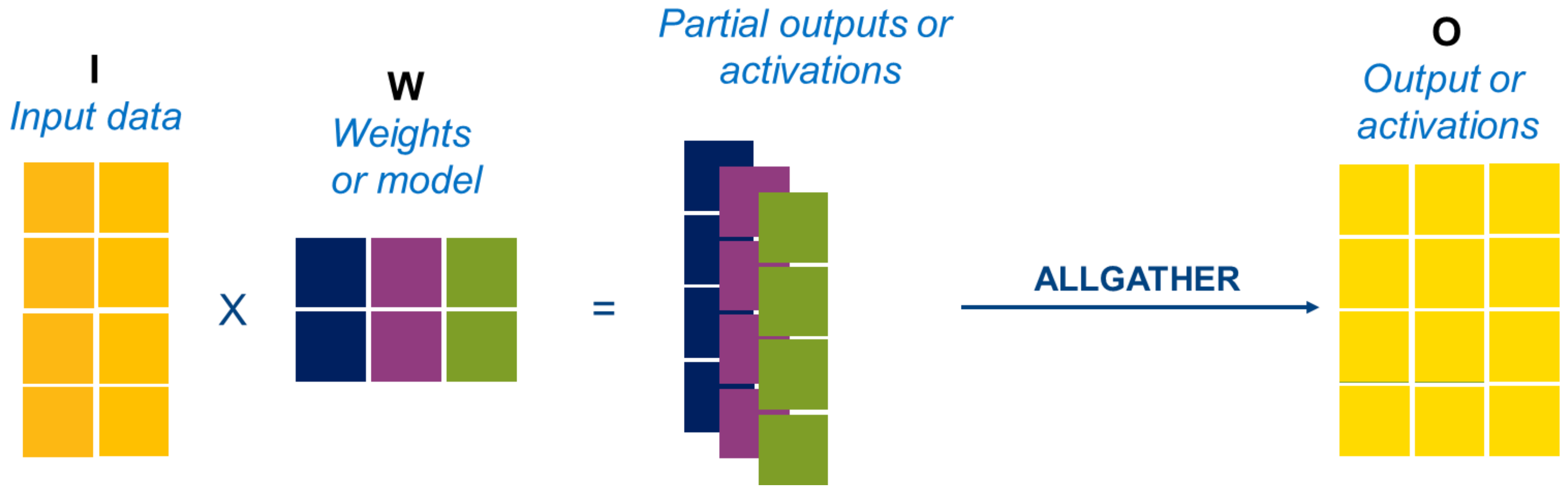
Multi-node parallelization

- **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

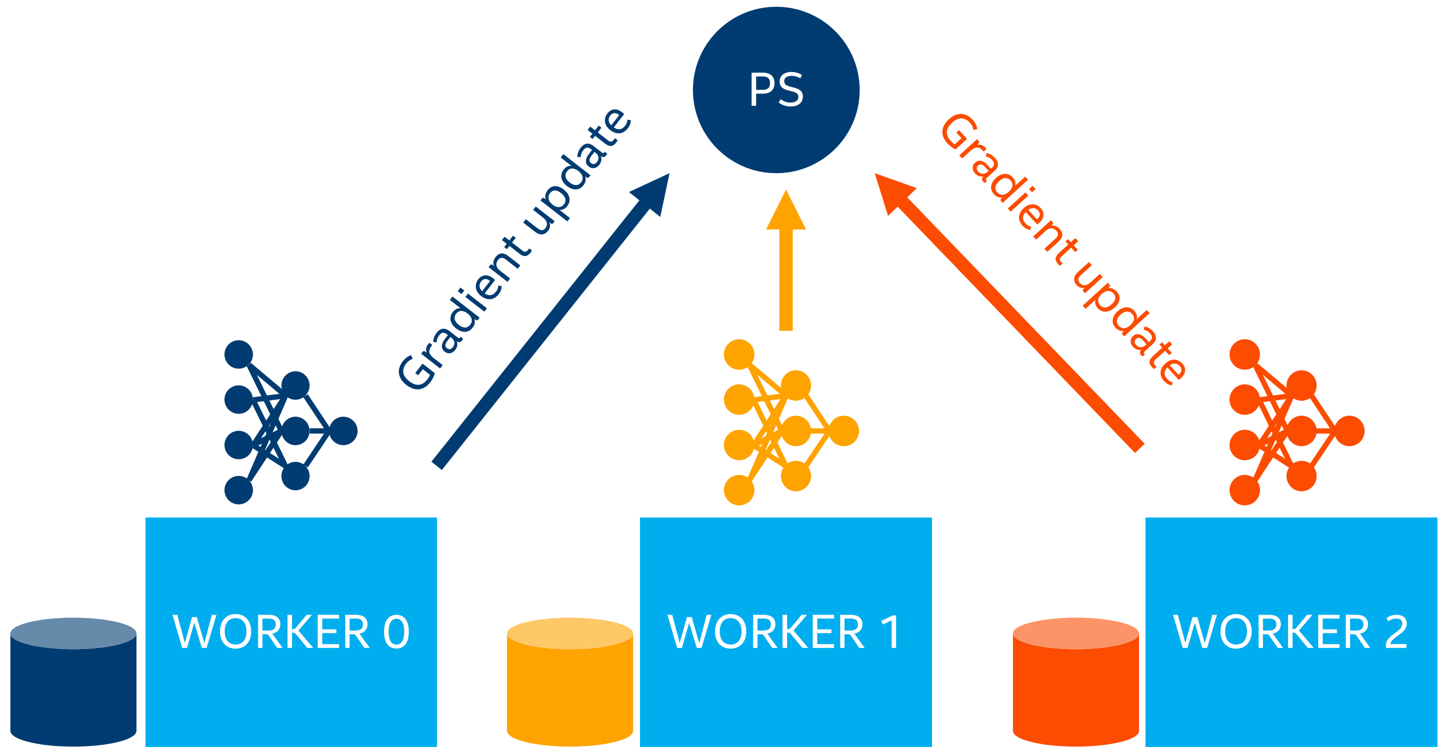
- **Model parallelism**:
  - Model is split across nodes;
  - Feed each node with the same batch of input data;
  - Communication for partial activations is required to gather the result;



Multi-node parallelization

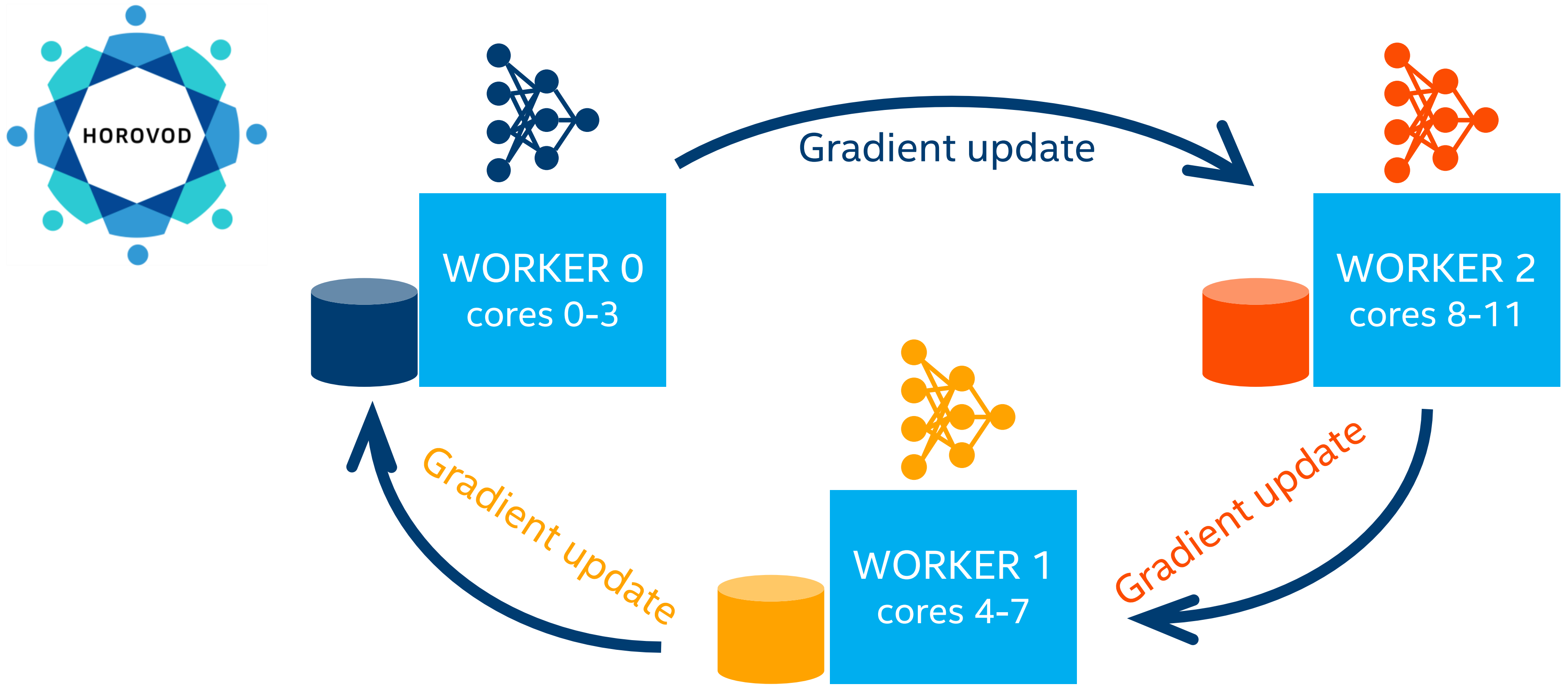
- **What parallelism flavor to use?**
  - Use model parallelism when volume of gradients is much higher than volume of activations or when model doesn’t fit memory;
  - Use data parallelism otherwise;
  - Parallelism choice affects activations/gradients ratio
    - Data parallelism at scale makes activations << 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

https://arxiv.org/abs/1802.05799v3

Distributed Training for Deep Neural Network

Intel® oneAPI Collective Communications Library (oneCCL)

intel®

Intel® oneAPI Collective Communications Library

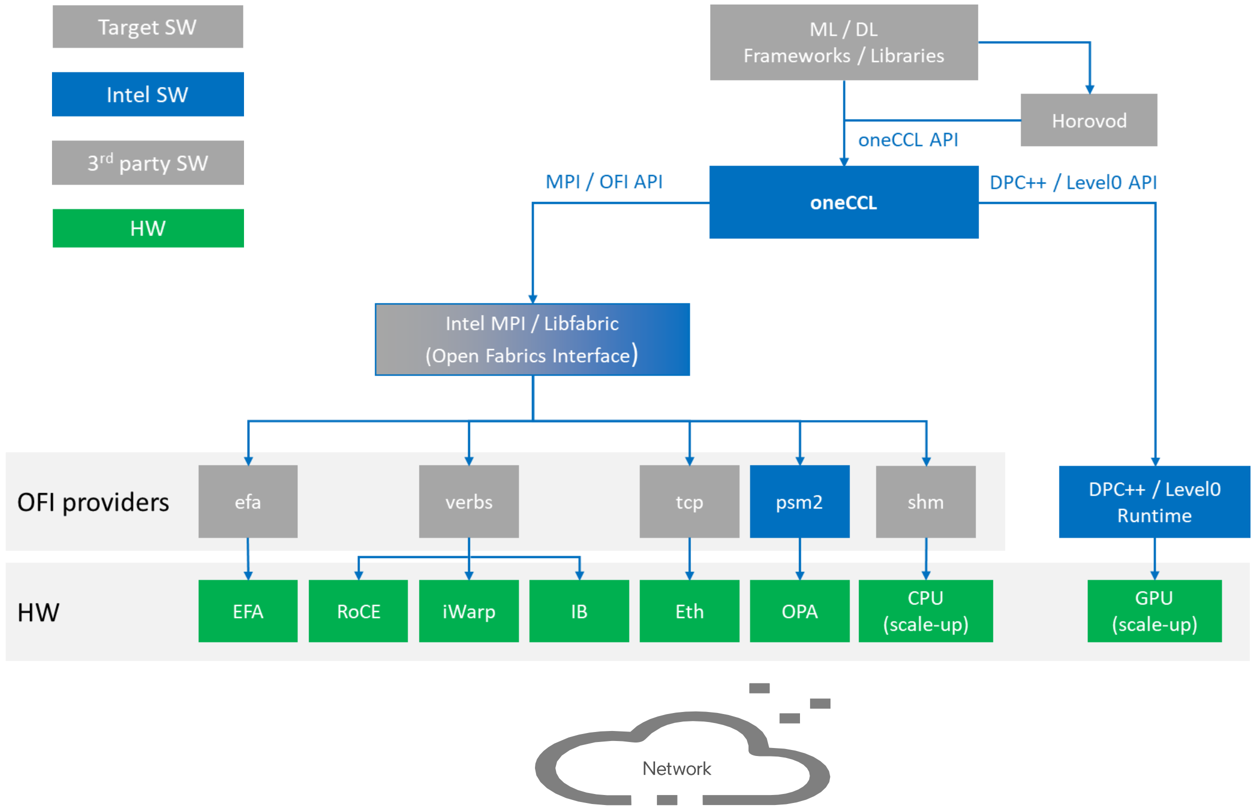
Optimize Communication Patterns

oneCCL provides optimized communication patterns for high performance on Intel CPUs & GPUs to distribute model training across multiple nodes

Transparently 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



Intel® oneAPI Collective Communications Library

Key Features (part 1/2)

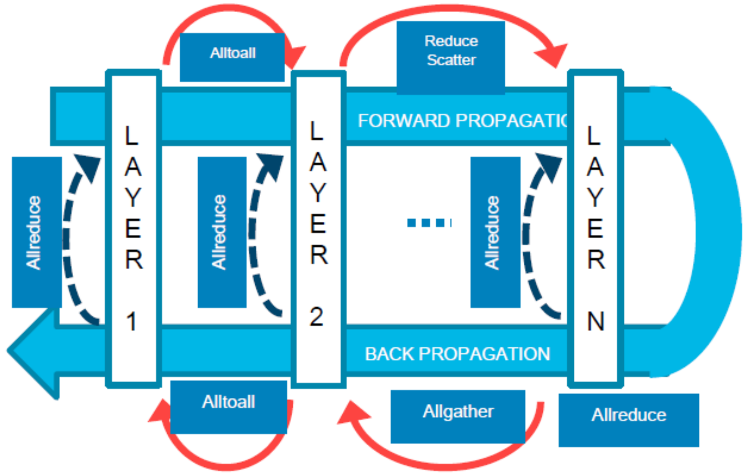
Enables efficient implementations of collectives used for deep learning training – all-gather, all-reduce, and more

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



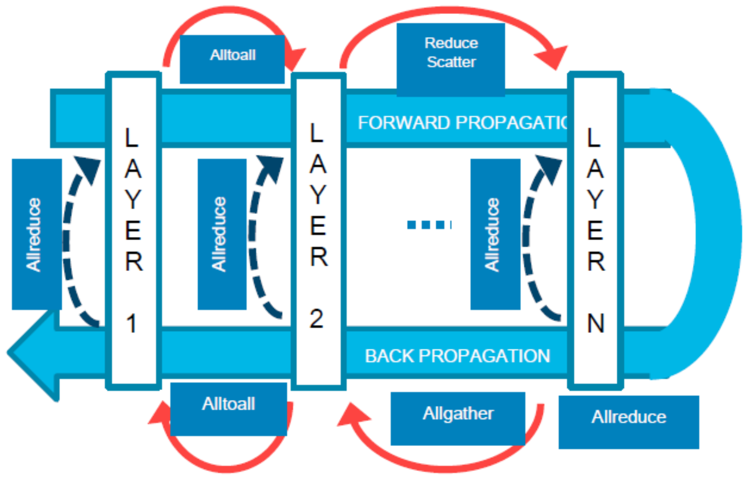
Intel® oneAPI Collective Communications Library

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

import horovod.tensorflow as hvd
```

2

```
hvd.init()
```

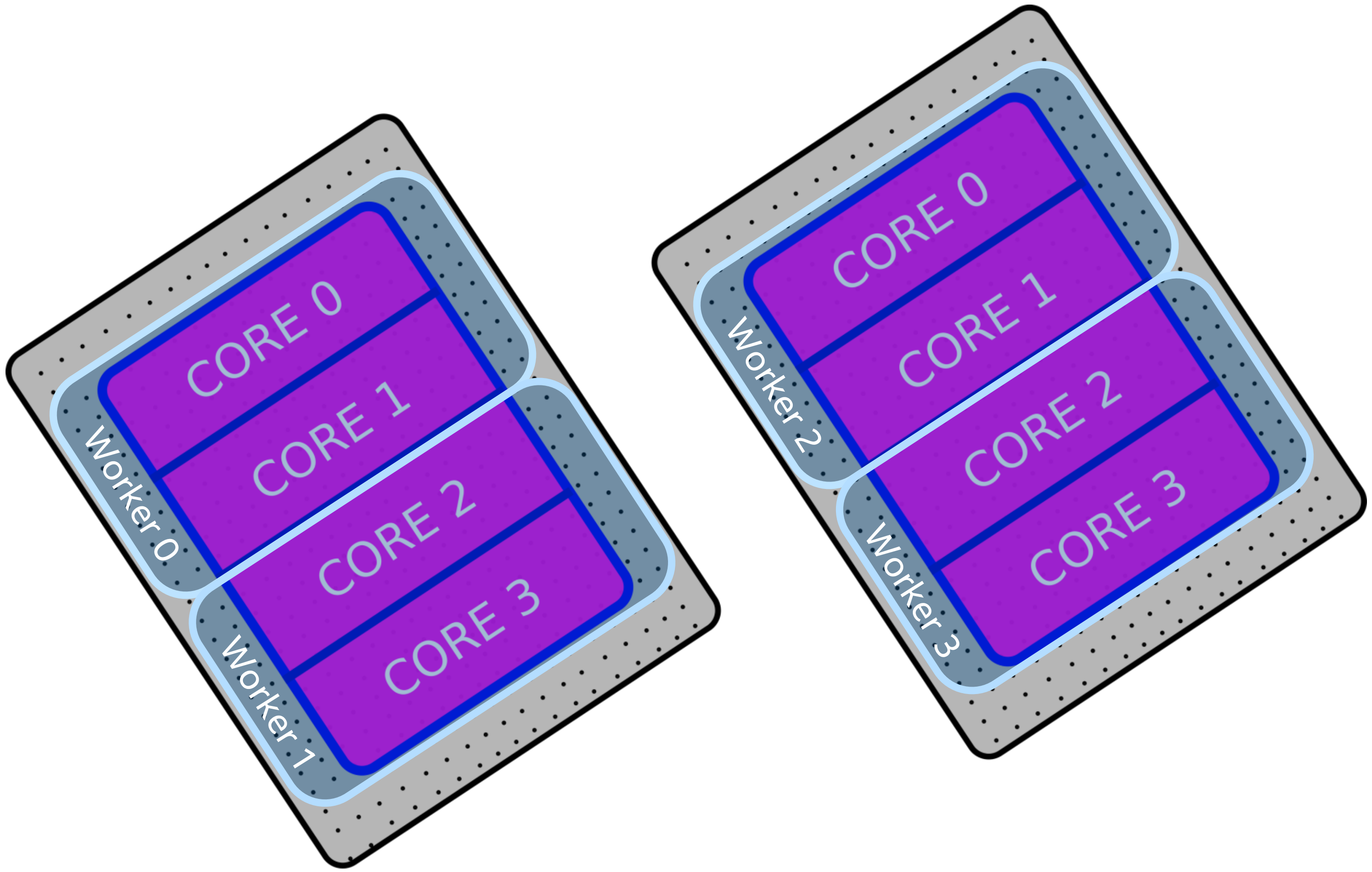
3

```
opt = tf.train.AdagradOptimizer(0.01 * hvd.size())

opt = hvd.DistributedOptimizer(opt)
```

4

```
hooks = [hvd.BroadcastGlobalVariablesHook(0)]
```

Sockets & Cores



SOCKET 0

SOCKET 1

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

```
$  mpirun
-H hostA,hostB,hostC
-np 6
--map-by ppr:1:socket:pe=2
--oversubscribe
--report-bindings
python train_model.py
```

OpenMPI

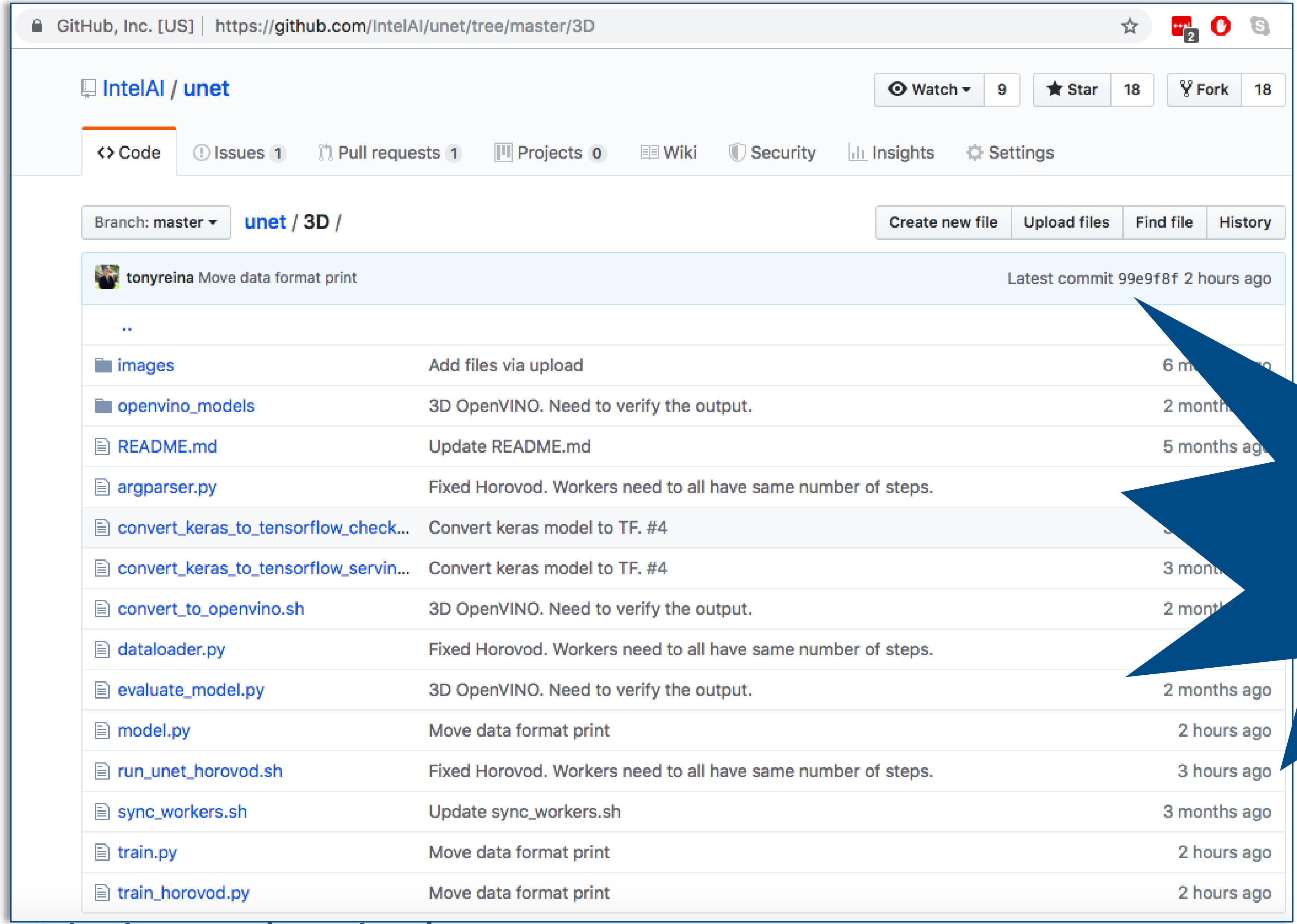
Multiple workers per CPU with Intel MPI

```
$ 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

```
                    SOCKET 0     SOCKET 1
R0   hostA   [BB/BB/../..][../../../..]
R1   hostA   [../../../..][BB/BB/../..]
R2   hostB   [BB/BB/../..][../../../..]
R3   hostB   [../../../..][BB/BB/../..]
R4   hostC   [BB/BB/../..][../../../..]
R5   hostC   [../../../..][BB/BB/../..]
```

```
mpirun -H hostA,hostB,hostC   -np 6  --map-by ppr:1:socket:pe=2 …
```



GitHub, Inc. [US] | https://github.com/IntelAI/unet/tree/master/3D

IntelAI / unet

unet / 3D /

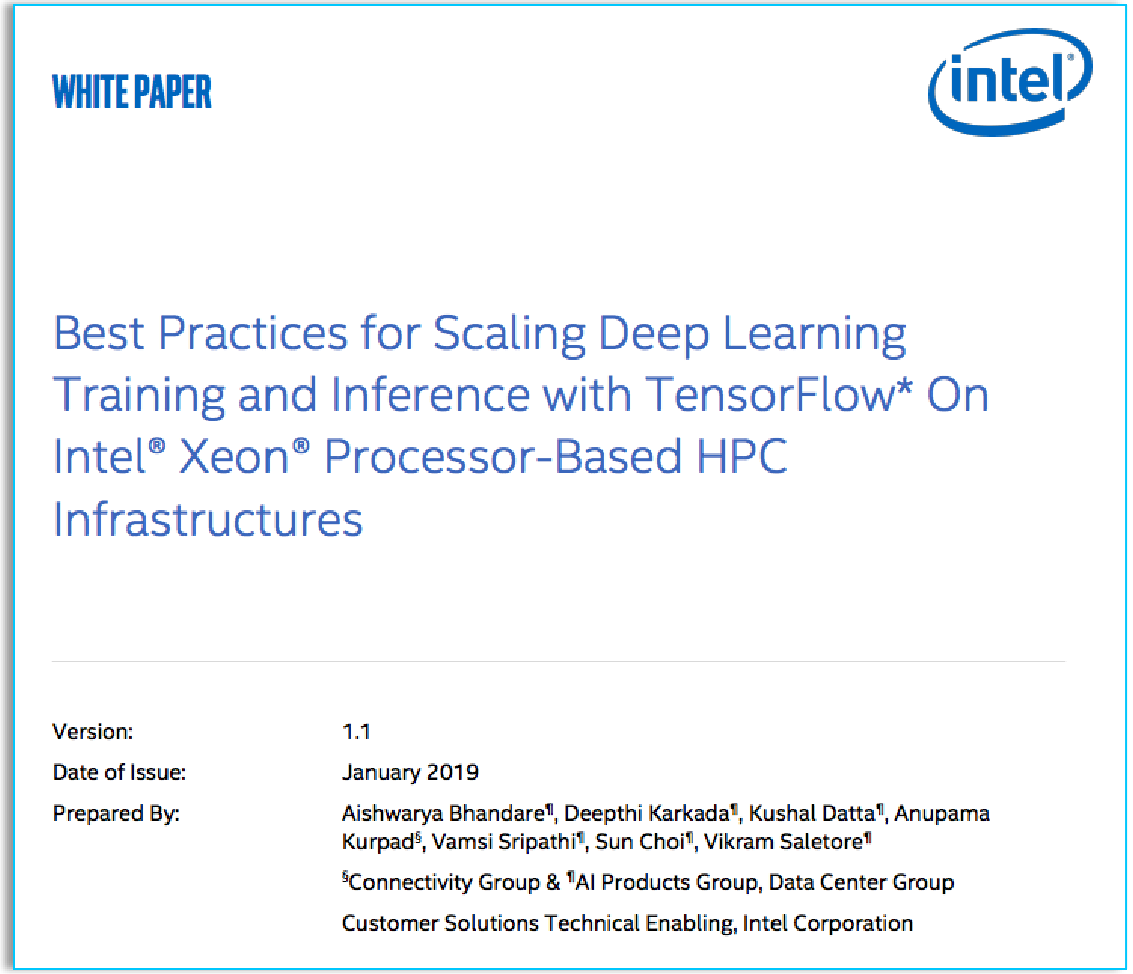
tonyreina Move data format print — Latest commit 99e9f8f 2 hours ago

..		
images	Add files via upload	6 m
openvino_models	3D OpenVINO. Need to verify the output.	2 month
README.md	Update README.md	5 months ag
argparser.py	Fixed Horovod. Workers need to all have same number of steps.	
convert_keras_to_tensorflow_check...	Convert keras model to TF. #4	
convert_keras_to_tensorflow_servin...	Convert keras model to TF. #4	3 mont
convert_to_openvino.sh	3D OpenVINO. Need to verify the output.	2 mont
dataloader.py	Fixed Horovod. Workers need to all have same number of steps.	
evaluate_model.py	3D OpenVINO. Need to verify the output.	2 months ago
model.py	Move data format print	2 hours ago
run_unet_horovod.sh	Fixed Horovod. Workers need to all have same number of steps.	3 hours ago
sync_workers.sh	Update sync_workers.sh	3 months ago
train.py	Move data format print	2 hours ago
train_horovod.py	Move data format print	2 hours ago

Try it on GitHub

github.com/IntelAI/unet

BKC/BKM for HPC AI



WHITE PAPER

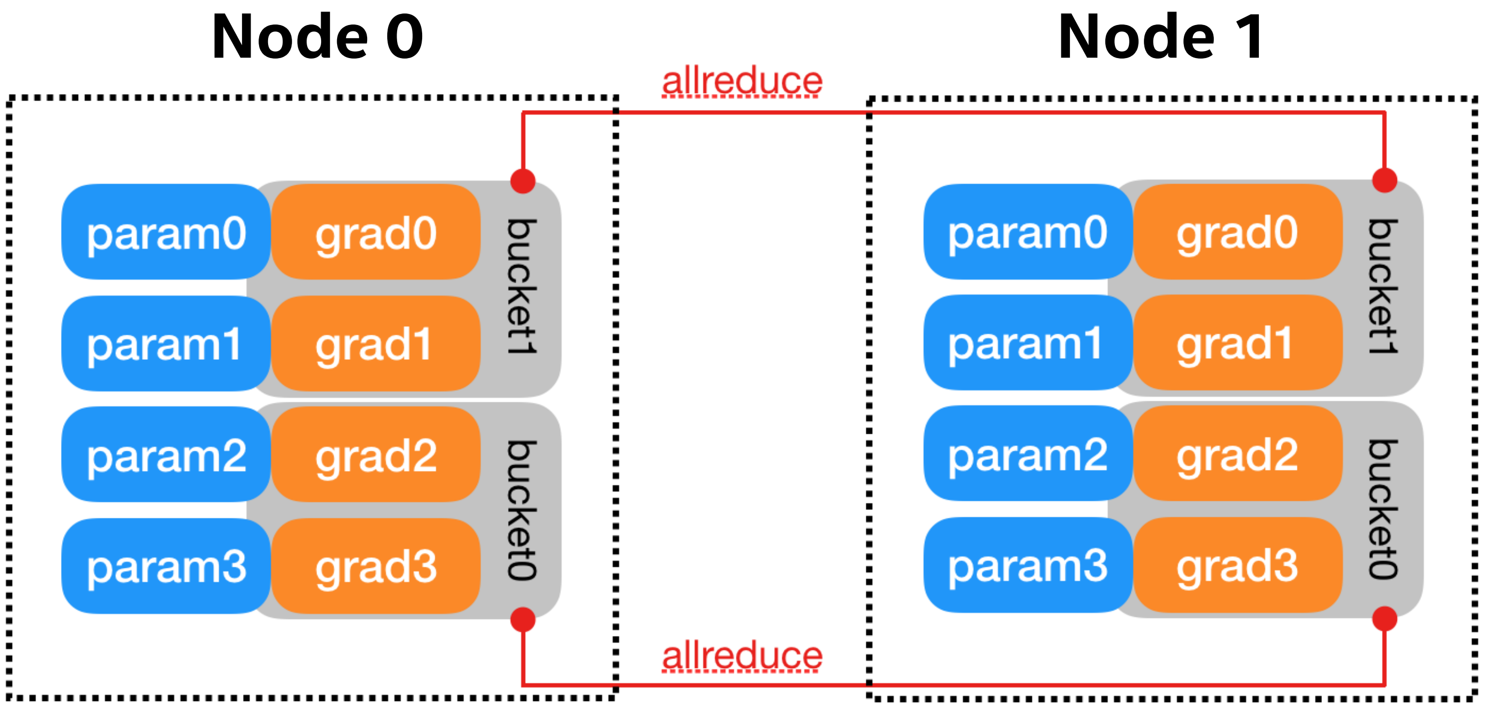
Best Practices for Scaling Deep Learning Training and Inference with TensorFlow* On Intel® Xeon® Processor-Based HPC Infrastructures

Version:	1.1
Date of Issue:	January 2019
Prepared By:	Aishwarya Bhandare[¶], Deepthi Karkada[¶], Kushal Datta[¶], Anupama Kurpad[§], Vamsi Sripathi[¶], Sun Choi[¶], Vikram Saletore[¶]
	[§]Connectivity Group & [¶]AI Products Group, Data Center Group
	Customer Solutions Technical Enabling, Intel Corporation

- Docker
- SLURM
- Singularity
- NFS
- Lustre

https://www.intel.ai/best-practices-for-tensorflow-on-intel-xeon-processor-based-hpc-infrastructures

Distributed Training with `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

Motivation	Methodology	Features
• Speedup PyTorch multi-node training on IA with oneCCL	• oneCCL is loaded as a PyTorch 3rd party communication library	• 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/libiomp5.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 I_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=[0x0000000FFFFFF0,0xFFFFFF00000000] \
              -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

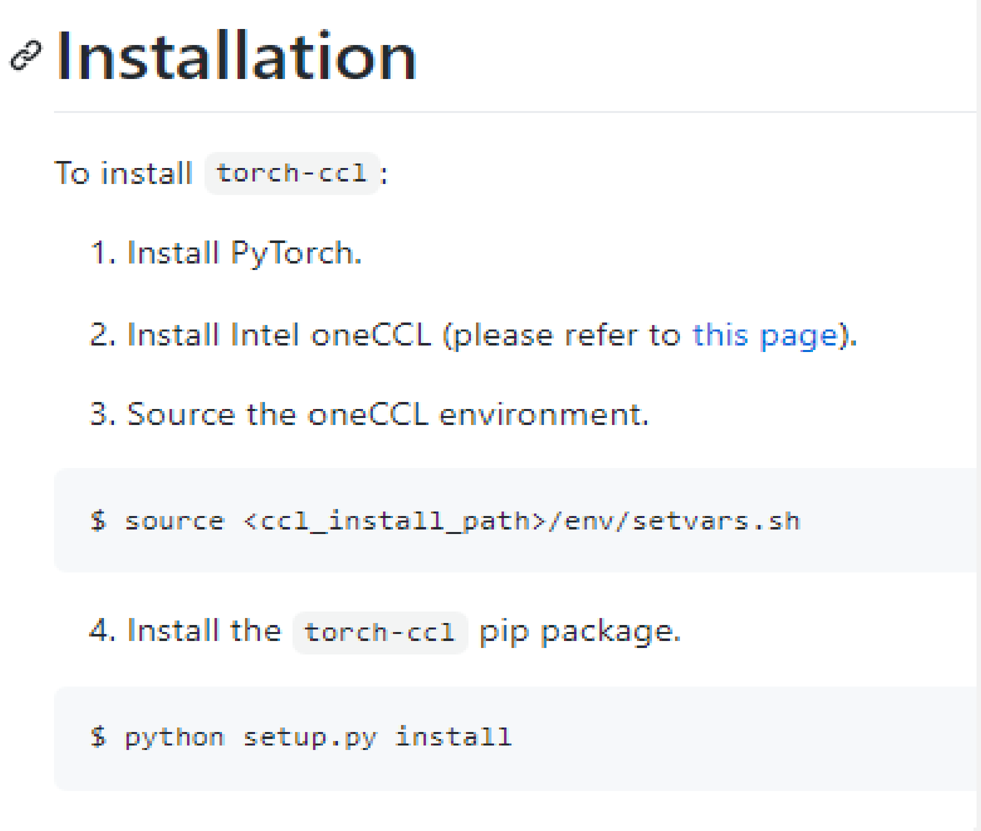
Distributed Training on multiple nodes

```
source ~/.local/env/setvars.sh
export LD_PRELOAD="${CONDA_PREFIX}/lib/libiomp5.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=[0x0000000FFFFFF0,0xFFFFFF00000000] \
              -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

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.

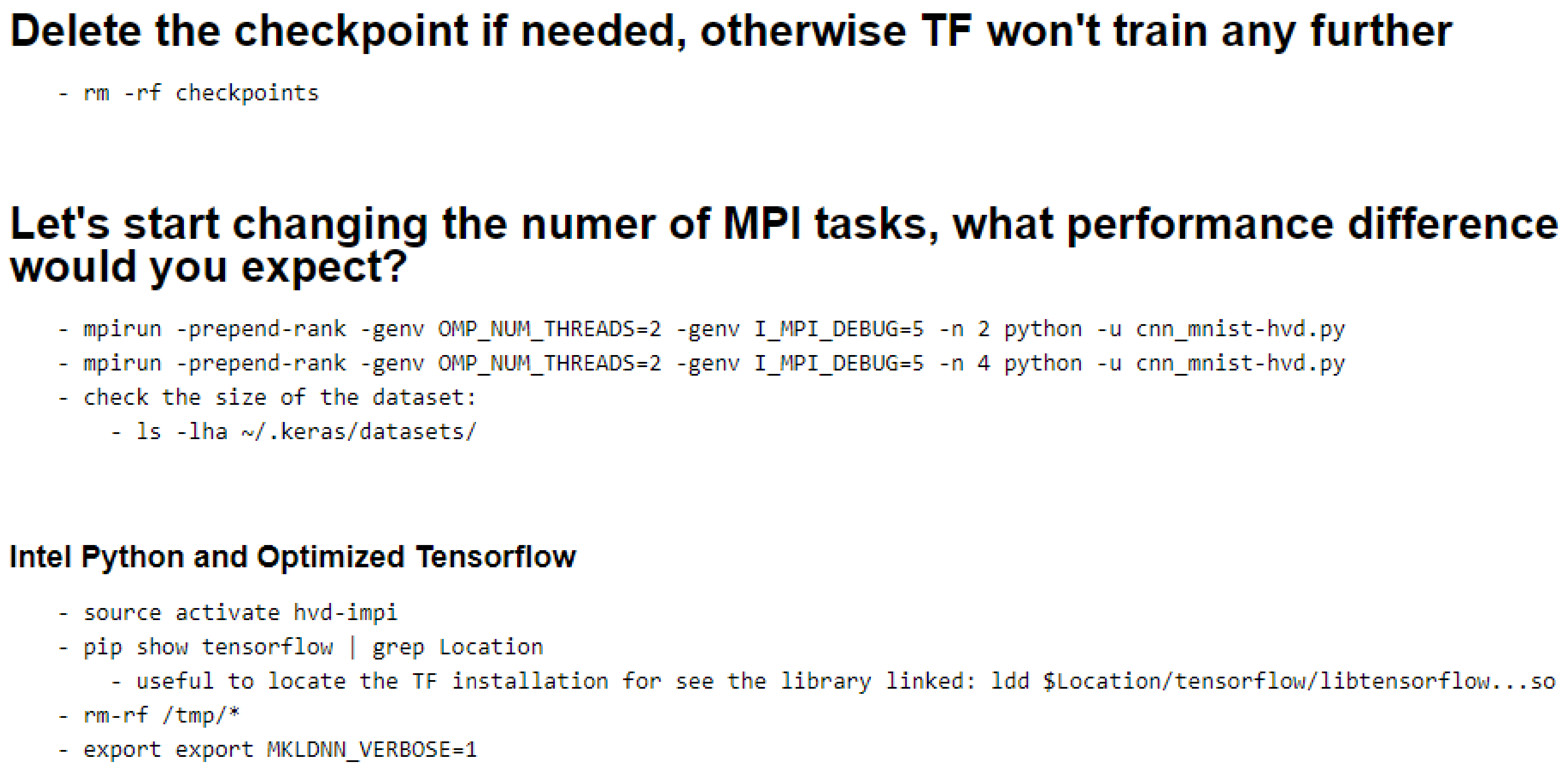
```
$ source <ccl_install_path>/env/setvars.sh
```

4. Install the `torch-ccl` pip package.

```
$ python setup.py install
```

HANDS-ON

Tensorflow+Horovod/cnn_mnist-hvd.ipynb



Delete the checkpoint if needed, otherwise TF won't train any further

```
- rm -rf checkpoints
```

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

```
- mpirun -prepend-rank -genv OMP_NUM_THREADS=2 -genv I_MPI_DEBUG=5 -n 2 python -u cnn_mnist-hvd.py
- mpirun -prepend-rank -genv OMP_NUM_THREADS=2 -genv 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-impi
- 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 export MKLDNN_VERBOSE=1
```

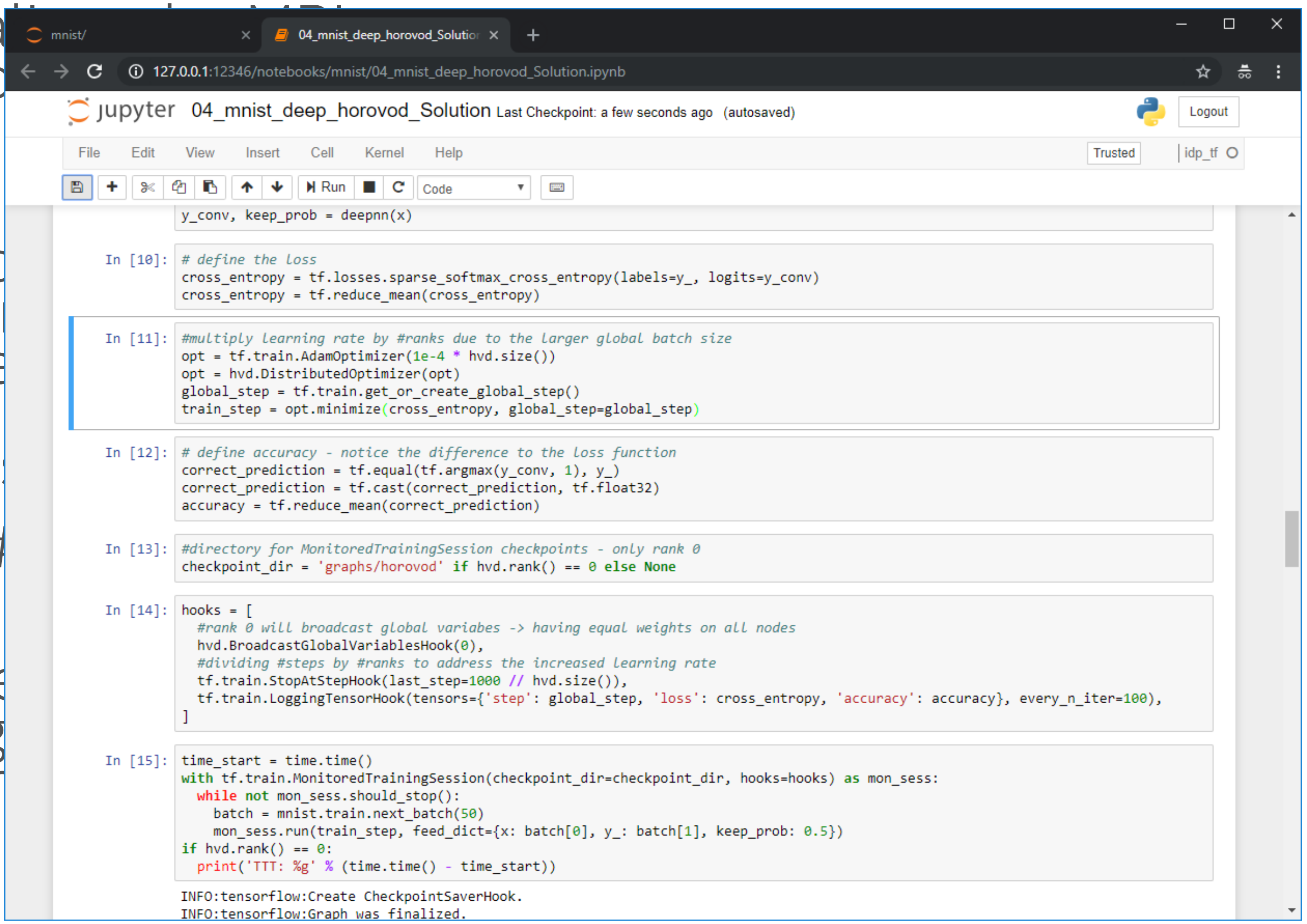
Tensorflow+Horovod/cnn_mnist-hvd.ipynb

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 initia
  communicatio
  and therefore
  and size()
- In order to red
  To Train with
  workers, there
  the batch size
  rate needs to
- Same for the
  training
- 4 ranks can be
  less threading
  required in sm
  convolutions


mnist/
04_mnist_deep_horovod_Solutio
127.0.0.1:12346/notebooks/mnist/04_mnist_deep_horovod_Solution.ipynb
jupyter 04_mnist_deep_horovod_Solution Last Checkpoint: a few seconds ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Help
Trusted
idp_tf
Run
Code

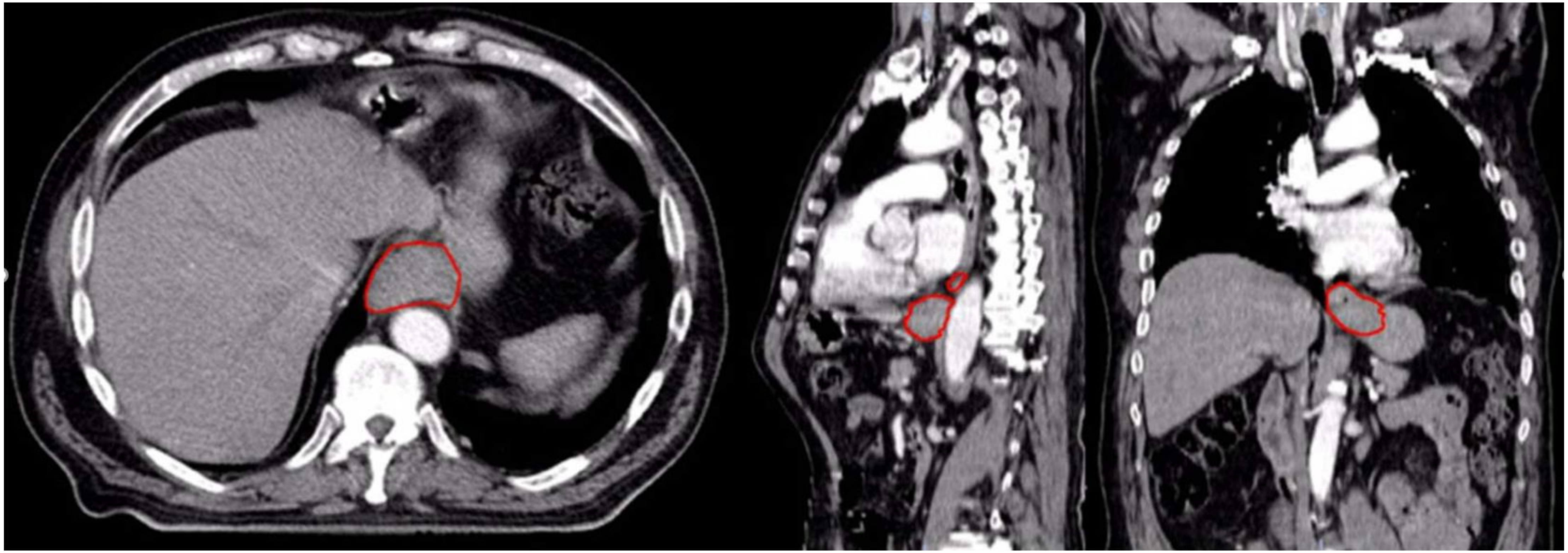
```
y_conv, keep_prob = deepnn(x)

In [10]: # define the loss
         cross_entropy = tf.losses.sparse_softmax_cross_entropy(labels=y_, logits=y_conv)
         cross_entropy = tf.reduce_mean(cross_entropy)

In [11]: #multiply learning rate by #ranks due to the larger global batch size
         opt = tf.train.AdamOptimizer(1e-4 * hvd.size())
         opt = hvd.DistributedOptimizer(opt)
         global_step = tf.train.get_or_create_global_step()
         train_step = opt.minimize(cross_entropy, global_step=global_step)

In [12]: # define accuracy - notice the difference to the loss function
         correct_prediction = tf.equal(tf.argmax(y_conv, 1), y_)
         correct_prediction = tf.cast(correct_prediction, tf.float32)
         accuracy = tf.reduce_mean(correct_prediction)

In [13]: #directory for MonitoredTrainingSession checkpoints - only rank 0
         checkpoint_dir = 'graphs/horovod' if hvd.rank() == 0 else None

In [14]: hooks = [
           #rank 0 will broadcast global variabes -> having equal weights on all nodes
           hvd.BroadcastGlobalVariablesHook(0),
           #dividing #steps by #ranks to address the increased learning rate
           tf.train.StopAtStepHook(last_step=1000 // hvd.size()),
           tf.train.LoggingTensorHook(tensors={'step': global_step, 'loss': cross_entropy, 'accuracy': accuracy}, every_n_iter=100),
         ]

In [15]: time_start = time.time()
         with tf.train.MonitoredTrainingSession(checkpoint_dir=checkpoint_dir, hooks=hooks) as mon_sess:
           while not mon_sess.should_stop():
             batch = mnist.train.next_batch(50)
             mon_sess.run(train_step, feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
         if hvd.rank() == 0:
           print('TTT: %g' % (time.time() - time_start))

         INFO:tensorflow:Create CheckpointSaverHook.
         INFO:tensorflow:Graph was finalized.
```

Legal Disclaimer & Optimization Notice

CASE-STUDY

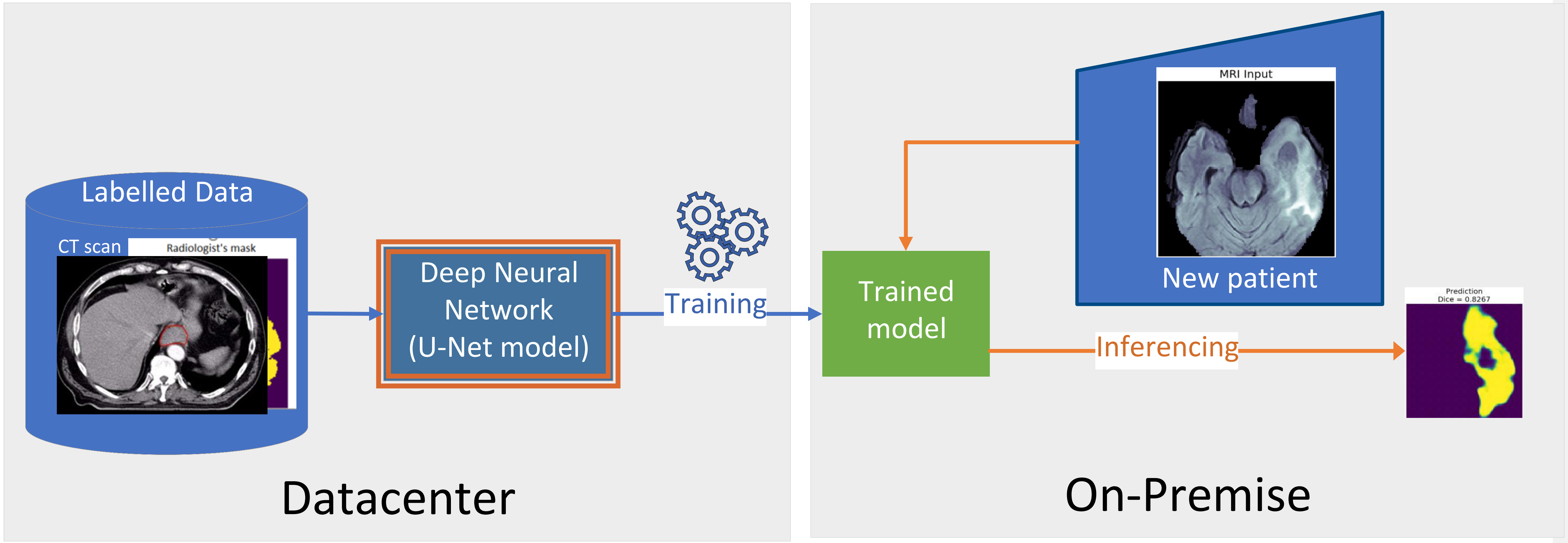
Engagement overview on the ASPIRE project

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

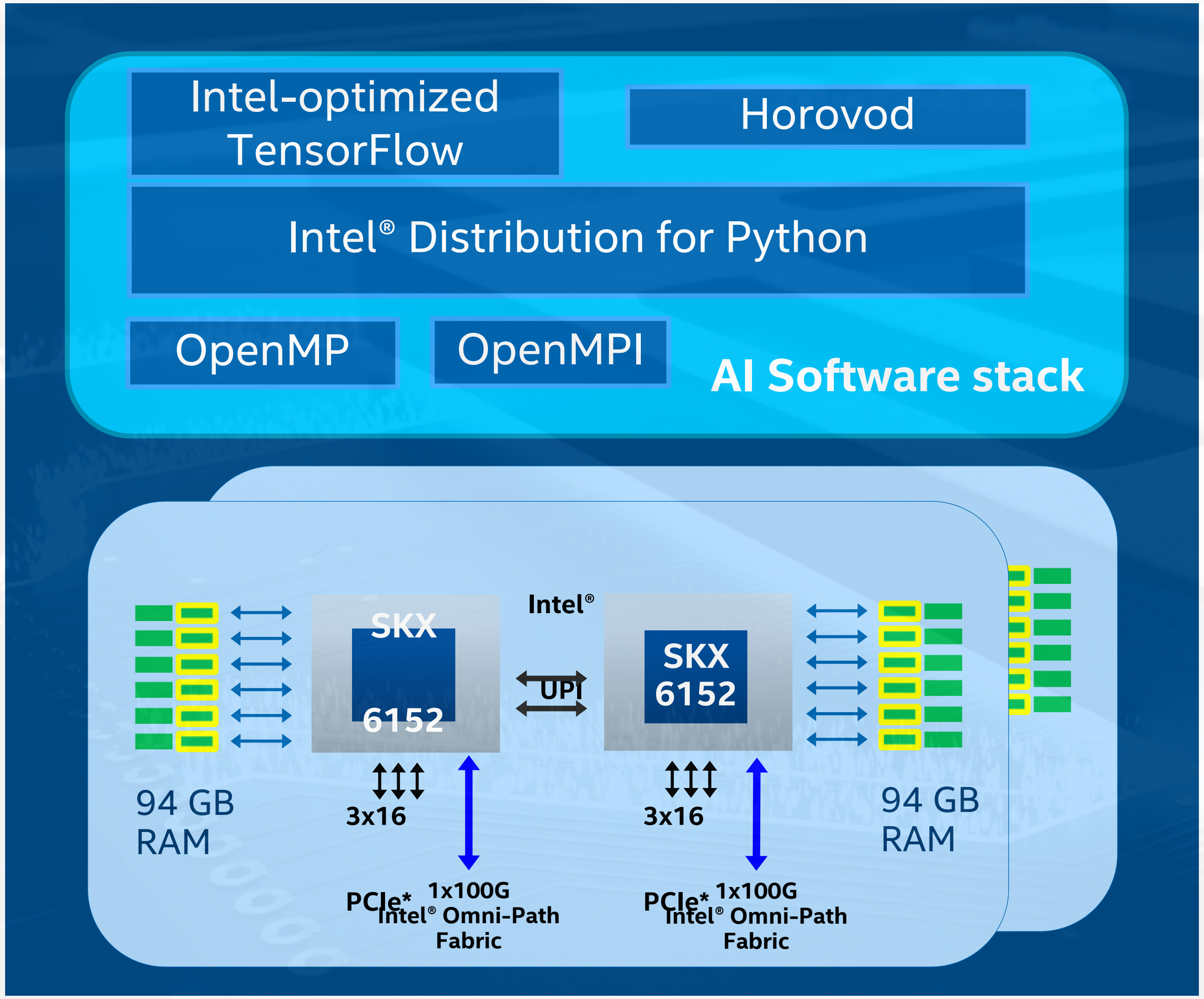


*** Segmentation**: Find the contour of the tumor on the CT scan

AI solution to for tumor segmentation:



Training Platform configuration


Intel-optimized TensorFlow
Horovod
Intel® Distribution for Python
OpenMP
OpenMPI
AI Software stack
SKX 6152
Intel® UPI
SKX 6152
94 GB RAM
3x16
PCIe*
1x100G Intel® Omni-Path Fabric
3x16
PCIe*
1x100G Intel® Omni-Path Fabric
94 GB RAM

AI Software stack Configuration

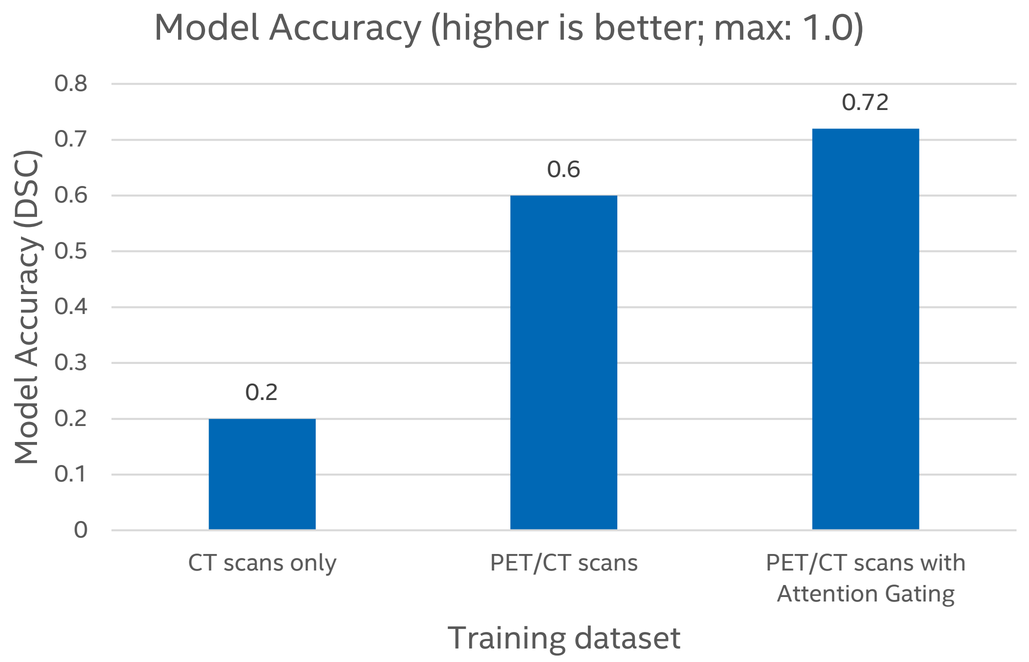
- Intel® Distribution for Python **3.7**
- Intel-optimized TensorFlow **1.12**
- OpenMPI **4.0.2**
- OpenMP **(gcc 4.8.5)**
- Horovod **0.18.2**

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

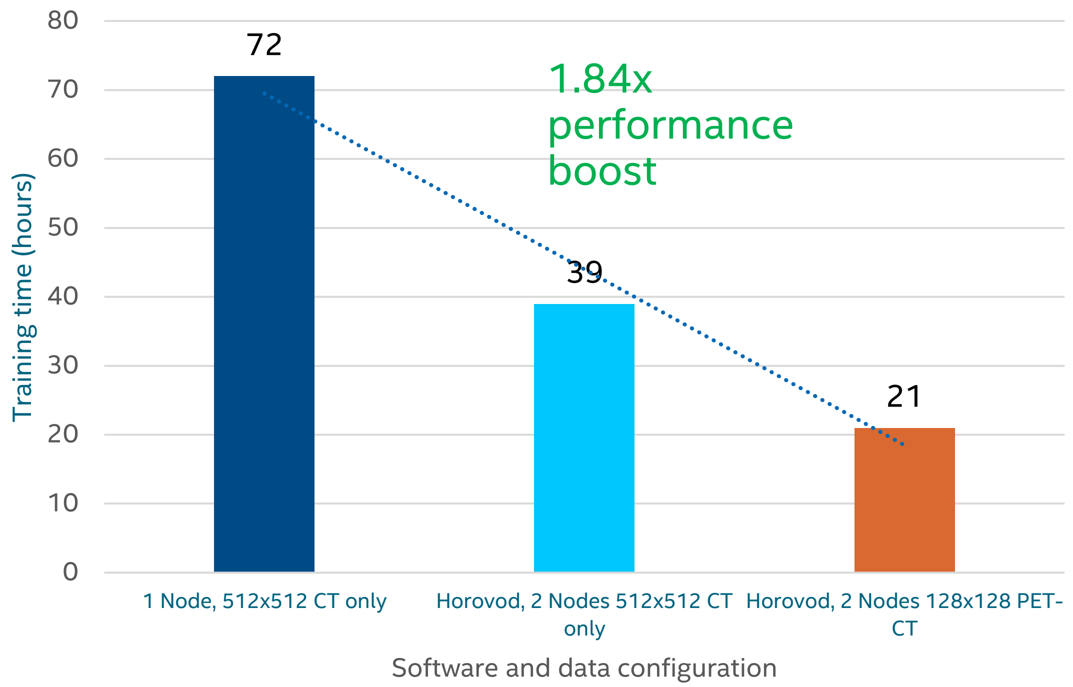
Model accuracy improvement

- 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)



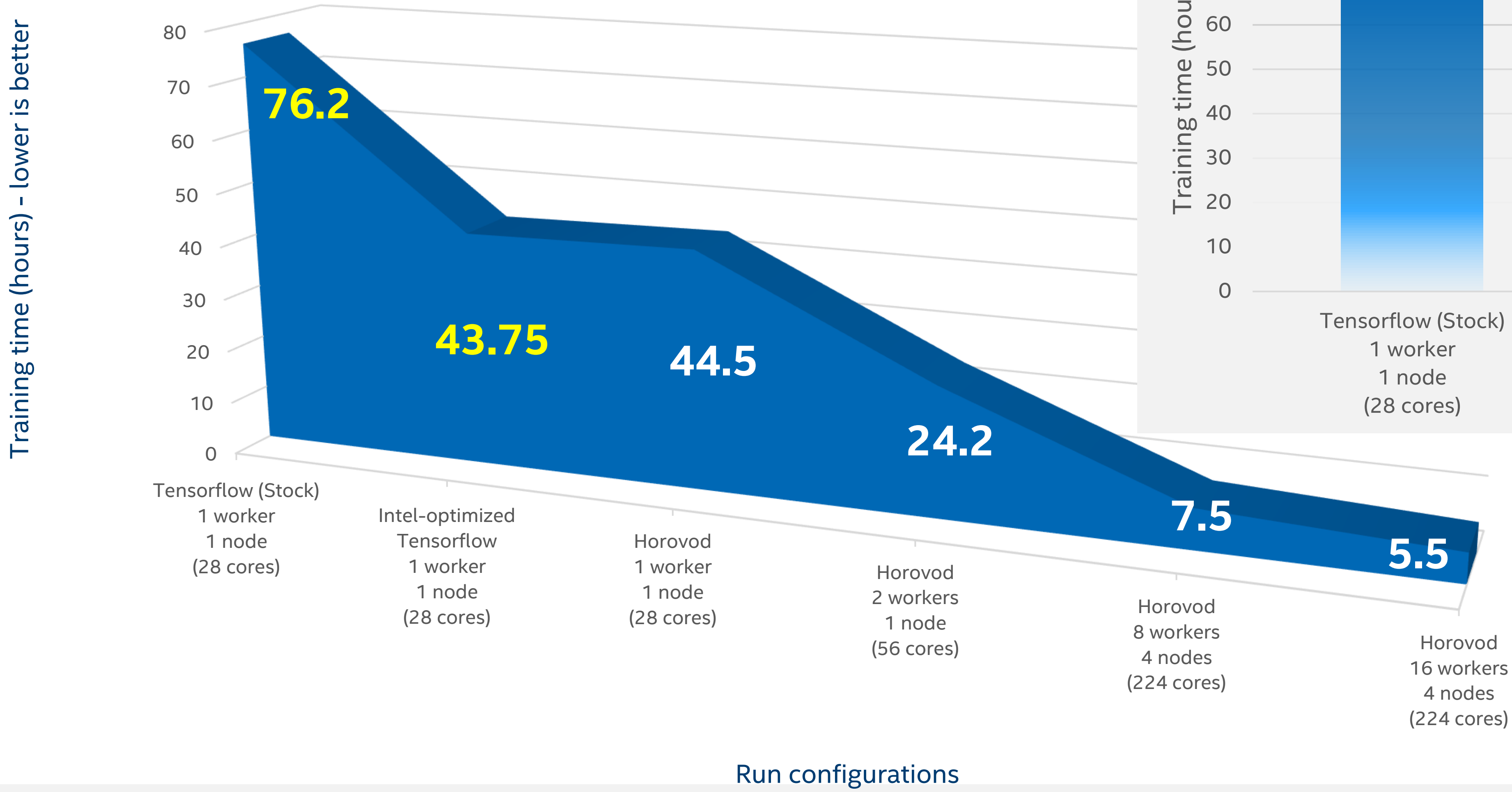
Training performance improvement

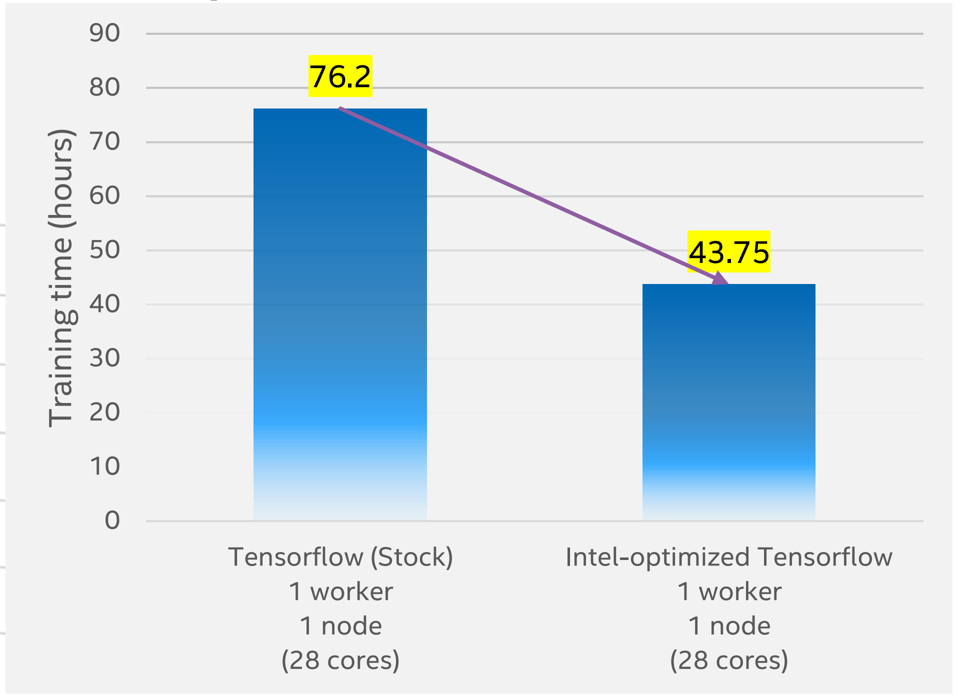
- Intel-optimized TensorFlow with node-level optimizations
  - OMP_NUM_THREADS = #physical cores
  - KMP_BLOCKTIME=1
  - KMP_AFFINITY=granularity=fine,compact
  - INTER and INTRA THREADS
- **Scaling-out:** Horovod – MPI parameter optimizations
  - 72 hours -> 39 hours



Training dataset:
3489 CT images, 3489 PET images

Performance results (Brain Tumor)


Training time (hours) - lower is better
Tensorflow (Stock) 1 worker 1 node (28 cores)
Intel-optimized Tensorflow 1 worker 1 node (28 cores)
Horovod 1 worker 1 node (28 cores)
Horovod 2 workers 1 node (56 cores)
Horovod 8 workers 4 nodes (224 cores)
Horovod 16 workers 4 nodes (224 cores)
Run configurations


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