Advanced Workshop on Parallel Programming Models

Architecture, Environment and Frameworks

Dr Malik M Barakathullah

Code Optimizer
Fujitsu Asia Pte Ltd, Singapore

April 2025

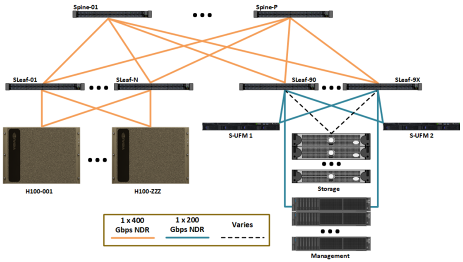
Contents

ASPIRE 2A+ Architecture

DGX H100 SuperPOD: Overview

- Nodes: 40 compute nodes, 2 login nodes, 2 admin nodes
- Parallel filesystem storage:
  - /home + /data/projects: 25PB
  - /scratch: 2.4 PB
- Network topology: **Non-blocking Leaf-Spine**. Leaf switches connect all compute nodes. Spine switches connect all Leaf switches
- Built for AI requirements:
  - Support for NGC containers through Enroot
  - FP8 and transformer engine
  - Large memory GPU
  - Fast intra-node GPU comm.
  - Tensor cores with more throughput than A100.

Non-blocking Leaf-Spine Topology:



Nodes and Interconnects

Overview:

- Infiniband interconnect with 400 GB/s HCAs (ConnectX-7) for inter-node communications.
- These HCA's allow GPUDirect RDMA through Infiniband fabric for inter-node GPU-GPU comm.
- These HCA's also allow MPI communications via CPU
- NVLINK 4.0 for intra-node GPU-GPU communications at 900GB/s
- Nvlinks are connected by NVSwitch which has several TB/s bandwidth.
- 28 TB local SSD storage (\raid) on each compute node.

DGX H100 node:

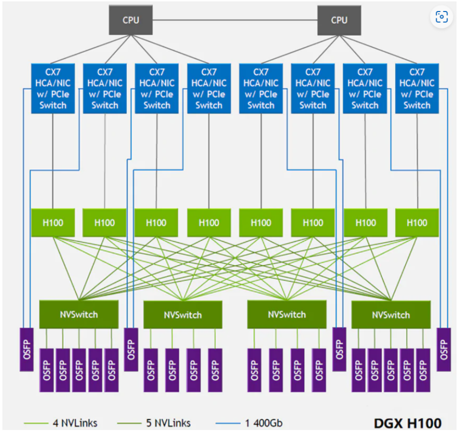


Image credit:

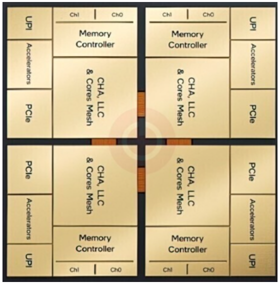
https://www.naddod.com/blog/unveiling-the-evolution-of-nvlink

Compute Node CPU

Overview:

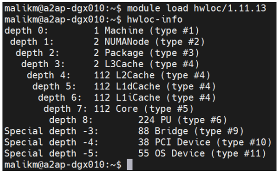
- Intel(R) Xeon(R) Platinum 8480C
- 2TB Memory
- 112 physical cores (3.8 GHz max, 2.0 GHz base)
- 224 hardware threads
- 2 NUMA nodes
- 2 threads per core
- 2 sockets (56 cores per socket)
- Sapphire Rapids-SP architecture
- Caches:
  - L1d: 5.3 MiB (112 instances)
  - L1i: 3.5 MiB (112 instances)
  - L2: 224 MiB (112 instances)
  - L3: 210 MiB (2 instances)
  - L1 & L2 – per core
  - L3 – per socket

A CPU in one socket:



(CHA & LLC – Caches, UPI – Interconnect for sockets)

The output from "hwloc-info":



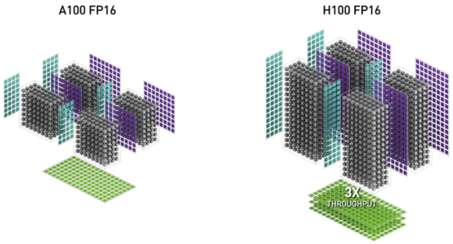
```
malikm@a2ap-dgx010:~$ module load hwloc/1.11.13
malikm@a2ap-dgx010:~$ hwloc-info
depth 0:         1 Machine (type #1)
 depth 1:        2 NUMANode (type #2)
  depth 2:       2 Package (type #3)
   depth 3:      2 L3Cache (type #4)
    depth 4:     112 L2Cache (type #4)
     depth 5:    112 L1dCache (type #4)
      depth 6:   112 L1iCache (type #4)
       depth 7:  112 Core (type #5)
        depth 8:         224 PU (type #6)
Special depth -3:        88 Bridge (type #9)
Special depth -4:        38 PCI Device (type #10)
Special depth -5:        55 OS Device (type #11)
malikm@a2ap-dgx010:~$
```

Compute Node GPU

Overview:

- NVIDIA H100 80GB (HBM3)
- 8 GPU's per node
- 80 GB VRAM
- 132 SM's
- 128 CUDA 32 bit cores per SM (or 64 CUDA 64 bit cores)
- 16,896 cores per GPU
- 4 warp schedulers, 64 warps/SM
- maximum 2048 threads per SM
- Compute Capability 9.0
- Caches:
  - L1 + shared memory 256 KB
  - L2 Cache 50MB
- Clock frequency: 1.5-1.8 GHz

Tensor core:



- Specialized cores for matrix multiplication and accumulation
- Useful in Deep Learning
- 3 times more through put than A100 in FP16

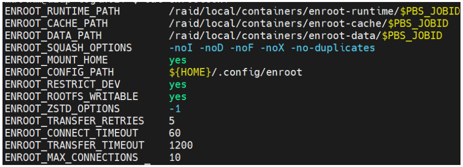
Enroot Containers

Introduction and Configuration

Introduction:

- Enroot gives an isolated filesystem and environment variables for the applications.
- Similar to "**chroot**" command of Linux, but Enroot comes with ability to import and export containers.
- It has builtin GPU and Infiniband support through hooks and libraries.
- Mounts the user's home directory, (and hence scratch directory too).
- Cgroups of the host are transparent inside the container, enabling scheduler to control the resources.
- Enables unprivileged users (non-root accounts) to install their packages as part of the container, and distribute the image.

Configuration in ASPIRE 2A+:



```
ENROOT_RUNTIME_PATH          /raid/local/containers/enroot-runtime/$PBS_JOBID
ENROOT_CACHE_PATH            /raid/local/containers/enroot-cache/$PBS_JOBID
ENROOT_DATA_PATH             /raid/local/containers/enroot-data/$PBS_JOBID
ENROOT_SQUASH_OPTIONS        -noI -noD -noF -noX -no-duplicates
ENROOT_MOUNT_HOME            yes
ENROOT_CONFIG_PATH           ${HOME}/.config/enroot
ENROOT_RESTRICT_DEV          yes
ENROOT_ROOTFS_WRITABLE       yes
ENROOT_ZSTD_OPTIONS          -1
ENROOT_TRANSFER_RETRIES      5
ENROOT_CONNECT_TIMEOUT       60
ENROOT_TRANSFER_TIMEOUT      1200
ENROOT_MAX_CONNECTIONS       10
```

- It shows that the enroot is mounted on "`/raid/local/containers`".
- The scheduler has a hook that creates mount points under the job-id.
- These folders with job-ids as names will be deleted by the scheduler after the job.
- Configured by default to mount $HOME (hence $HOME/scratch too).

Importing an Enroot Image

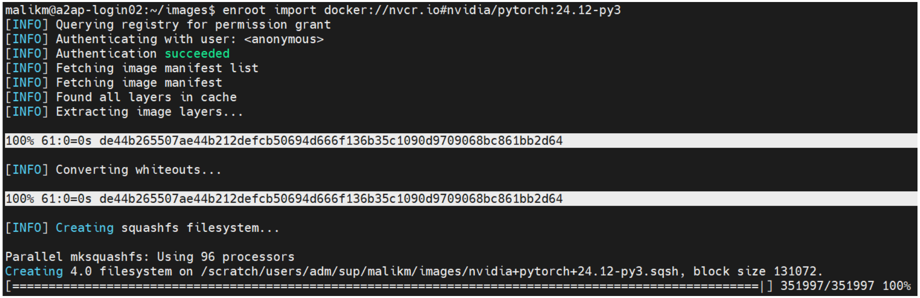
Importing

The common strategy to build an image is to start from a base-image that could be to import a docker image from NVIDIA NGC Catalog: https://catalog.ngc.nvidia.com/.

- **Get the URI:**
  - > On a browser visit https://catalog.ngc.nvidia.com/
  - > Go to "containers"
  - > On the search field enter the package name, for example, PyTorch.
  - > Click the PyTorch link followed by "tags"
  - > Copy the link of the image. For example, nvcr.io/nvidia/pytorch:24.12-py3.
- **Get the command prompt:** Use the login node if there not going to be heavy compilation or installation, else get a compute node by submitting an interactive job through **qsub -I**.
- Execute: "`enroot import docker://nvcr.io#nvidia/pytorch:24.12-py3`". Note the change of a "/" to "#".

Importing an Enroot Image and Creating the Container

This gives the image as squashfile as shown below:



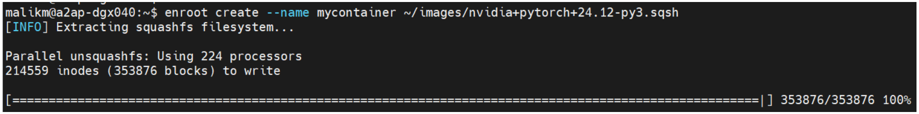
```
malikm@a2ap-login02:~/images$ enroot import docker://nvcr.io#nvidia/pytorch:24.12-py3
[INFO] Querying registry for permission grant
[INFO] Authenticating with user: <anonymous>
[INFO] Authentication succeeded
[INFO] Fetching image manifest list
[INFO] Fetching image manifest
[INFO] Found all layers in cache
[INFO] Extracting image layers...

100% 61:0=0s de44b265507ae44b212defcb50694d666f136b35c1090d9709068bc861bb2d64

[INFO] Converting whiteouts...

100% 61:0=0s de44b265507ae44b212defcb50694d666f136b35c1090d9709068bc861bb2d64

[INFO] Creating squashfs filesystem...

Parallel mksquashfs: Using 96 processors
Creating 4.0 filesystem on /scratch/users/adm/sup/malikm/images/nvidia+pytorch+24.12-py3.sqsh, block size 131072.
[===========================================================================================|] 351997/351997 100%
```

Creating the container

- Get a compute node by submitting an interactive job.
- Then, the container can be created from the squash file using the following synopsis: "`enroot create --name mycontainer pytorch:24.12-py3.sqsh`".

Creating and Starting the Enroot Container

This gives the image as squashfile as shown below:



```
malikm@a2ap-dgx040:~$ enroot create --name mycontainer ~/images/nvidia+pytorch+24.12-py3.sqsh
[INFO] Extracting squashfs filesystem...

Parallel unsquashfs: Using 224 processors
214559 inodes (353876 blocks) to write

[=============================================================================================|] 353876/353876 100%
```

The command "`enroot list`" will show "mycontainer" as its result.

Starting the container as a root with write permission

- Execute "`enroot start --root --rw mycontainer`"
- The options "`--mount folder/path/in/host:folder/path/in/container`" can be used to mount any other folders.
- The options "`--env VAR_NAME=VALUE`" can be used to export an environment variable, "VAR_NAME" with value, "VALUE".
- Execution of "`enroot start --root --rw mycontainer`" results in the command prompt inside the container as in: `root@a2ap-dgx040:/workspace#`

Now we are nearly all set for installation in the system directories of the container.

Proxy Setting and Installation Using APT in Container

Set the following proxies when you are in the command prompt of the container:

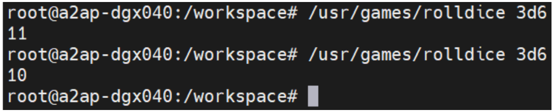
```
export no_proxy=localhost,127.0.0.1,10.104.0.0/21
export https_proxy=http://10.104.4.124:10104
export http_proxy=http://10.104.4.124:10104
```

Now let us proceed to install a game, "rolldice".

- Execute:

```
apt update && apt upgrade
apt install rolldice
/usr/games/rolldice 3d6
```

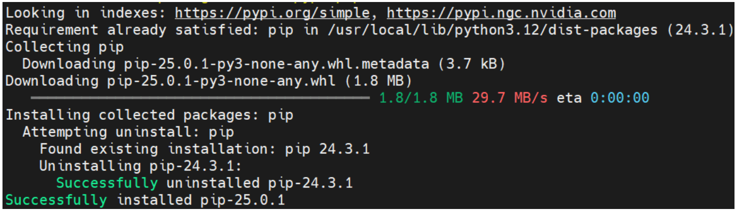
- This gives (a random number from 3 to 24) during my each run as follows:



```
root@a2ap-dgx040:/workspace# /usr/games/rolldice 3d6
11
root@a2ap-dgx040:/workspace# /usr/games/rolldice 3d6
10
root@a2ap-dgx040:/workspace# 
```

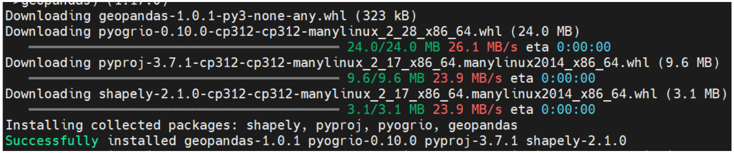
Upgrading PIP and Installing a Python Package

Executing `python -m pip install --upgrade pip` gives:



```
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Requirement already satisfied: pip in /usr/local/lib/python3.12/dist-packages (24.3.1)
Collecting pip
  Downloading pip-25.0.1-py3-none-any.whl.metadata (3.7 kB)
Downloading pip-25.0.1-py3-none-any.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 29.7 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 24.3.1
    Uninstalling pip-24.3.1:
      Successfully uninstalled pip-24.3.1
Successfully installed pip-25.0.1
```

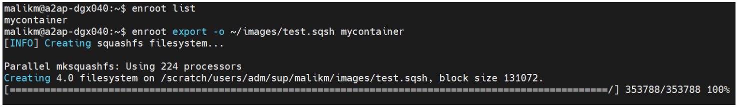
Executing `pip install geopandas` installs the package GeoPandas:



```
Downloading geopandas-1.0.1-py3-none-any.whl (323 kB)
Downloading pyogrio-0.10.0-cp312-cp312-manylinux_2_28_x86_64.whl (24.0 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.0/24.0 MB 26.1 MB/s eta 0:00:00
Downloading pyproj-3.7.1-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (9.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.6/9.6 MB 23.9 MB/s eta 0:00:00
Downloading shapely-2.1.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.1/3.1 MB 23.9 MB/s eta 0:00:00
Installing collected packages: shapely, pyproj, pyogrio, geopandas
Successfully installed geopandas-1.0.1 pyogrio-0.10.0 pyproj-3.7.1 shapely-2.1.0
```

Exporting the Container as a Squash Filesystem

- To export the container after installation, first exit the container, by executing `exit`. The will take us back to the command prompt of the host compute node.
- The command `enroot list` will show the container, which is `mycontainer` in our case.
- Executing `enroot export -o  /images/test.sqsh mycontainer` gives:



```
malikm@a2ap-dgx040:~$ enroot list
mycontainer
malikm@a2ap-dgx040:~$ enroot export -o ~/images/test.sqsh mycontainer
[INFO] Creating squashfs filesystem...

Parallel mksquashfs: Using 224 processors
Creating 4.0 filesystem on /scratch/users/adm/sup/malikm/images/test.sqsh, block size 131072.
[=============================================================================================/] 353788/353788 100%
```

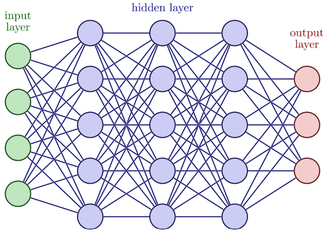
For the material on how to use Enroot for running commands in batch jobs, please see the slides of Introductory workshop.

PyTorch Distributed Data Parallel

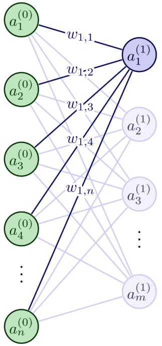
Introduction

- **DDP** class from the module **torch·Distributed** is used for parallelizing training in multiple GPU's.
- Uses multiple processes like MPI. Recommended: One process for each GPU.
- Each process does deep-learning using a replica of the model: Hence, Data Parallelism via Single Program Multiple Data (SPMD) paradigm.
- The model should be small-enough to fit into each single GPU.
- Uses collective-communication functions from Torch•Distributed module, which is generally chosen to be NCCL in the backend.
- The collective communications are used during back-propagation to synchronize gradient across all processes.
- Faster than Torch•Dataparallel, since the latter uses only threads, and thus suffers from locking to avoid race-condition.
- Pytorch DDP is launched using "torchrun", which spans the specified number of processes.

Basics of Deep Learning (DL)



- A DL model could have 100's of layers.
- Input layer data are features of each sample.
- samples can be bunched together as batches.
- The output layer predicts.
- Loss = A positive norm of (prediction - targe label).



$$= \sigma\left(w_{1,1}a_1^{(0)} + w_{1,2}a_2^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{a}^{(0)} + \mathbf{b}^{(0)}\right)$$

- The **W**'s and **b**'s are the learnable parameters.
- The $\sigma()$ is the activation function or "switch".
- Various **paradigms:** MLP, CNN, RNN, & GNN.
- **Problems:** Classification, regression, segmentation & generation

Image credit: https://tikz.net/neural_networks/

Basics of Deep Learning (DL) (continued · · ·)

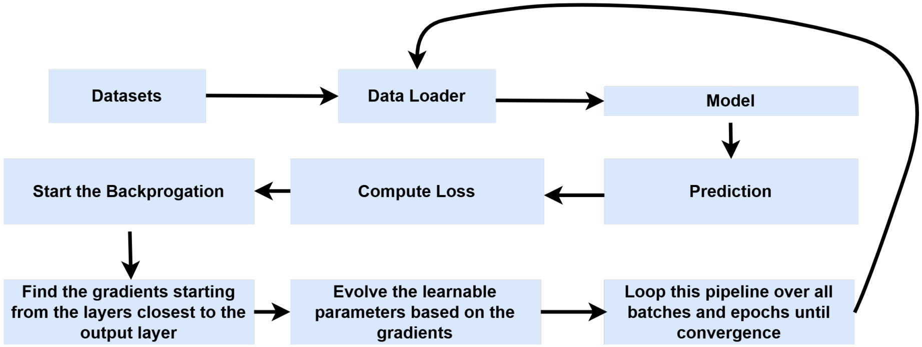
Different types of layers:

- 1D,2D,3D convolution and transposed convolution layers
- Pooling (eg, maxpool, maxunpool,average in 1-3D) and upsampling and padding (eg. zero padding, reflection) layers.
- Normalization (eg. batchnorm) layers.
- Activation functions (eg. sigmoid, tanh, ReLU)
- Recurrent layers (eg. LSTM, GRU)
- Linear and dropout layer
- various loss functions (e.g. MAE, MSE, cross entropy)

Applications:

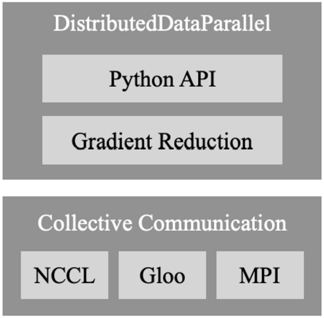
- **MLP:** Suitable for table data: forecasting, estimation (interpolation in multi-dimension), classification, pattern recognition. [Reference: http://www.lx.it.pt/ lbalmeida/papers/AlmeidaHNC.pdf]
- **CNN:**Image classification, segmentation, video and audio analysis, time series analysis
- **RNN:** Time series prediction, natural language processing. [Reference: (latest review article) https://www.sciencedirect.com/science/article/pii/S1319157824001575].
- **GNN:** Analysis of connected data, network analysis such as social networks, urban planning, fraud detection, business forecasting, etc. [Reference: https://arxiv.org/pdf/2504.07645 and references thereof.].

End-to-end Machine Learning Pipeline


Datasets
Data Loader
Model
Prediction
Compute Loss
Start the Backprogation
Find the gradients starting from the layers closest to the output layer
Evolve the learnable parameters based on the gradients
Loop this pipeline over all batches and epochs until convergence

- Backpropagation (termed as backward pass) in PyTorch is the crucial step that determines how much the learnable parameters changed.
- Typically training is based on a subset of dataset, training set.
- Epochs should be rightly set to avoid over-fitting.

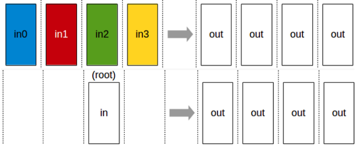
PyTorch DDP and NCCL



Pytorch DDP can use NCCL, Gloo or MPI for communication.

In ASPIRE 2A+, NCCL library is available, and recommended by PyTorch.

- NCCL offers both point-to-point and **collective communication** functions.
- PyTorch DDP uses **all-reduce** and **broadcast** functions of NCCL.
- Broadcast is used for **state-dictionary** in rank 0.
- All-reduce is used for synchronizing **gradients** during back propagation.
- The reduction operation is **mean** on the gradients in all GPU's.



All-reduce

Broadcast

PyTorch DDP Design: Step-by-Step Operation Method

- DDP uses Pytorch library **c10d**.
- c10d could use NCCL backend.
- c10d forms the process group.
- DDP version of the model is created as instance by passing the model as an argument to the DDP class's constructor.
- Process ranked "0" broadcasts the "state_dict()" of the model at each step of every epoch.
- Forward pass in DDP model is same as that of the original model
- During back propagation, the gradients are bucketed before reduction by mean across all GPU's. This is to minimize the number of communications.

```python
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
import os
from torch.nn.parallel import DistributedDataParallel as DDP


def example(rank, world_size):
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)
    # create local model
    model = nn.Linear(10, 10).to(rank)
    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # forward pass
    outputs = ddp_model(torch.randn(20, 10).to(rank))
    labels = torch.randn(20, 10).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()
    # update parameters
    optimizer.step()

def main():
    world_size = 2
    mp.spawn(example,
        args=(world_size,),
        nprocs=world_size,
        join=True)

if __name__=="__main__":
    # Environment variables which need to be
    # set when using c10d's default "env"
    # initialization mode.
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "29500"
    main()
```

Torchrun: An Utility to Launch PyTorch DDP Workload

The previous example had used the Python module **Multiprocessing** which spawned the processes. Now we will see **Torchrun**.

Advantages of Torchrun:

- Ranks are allocated automatically.
- If processes in a node fail, the processes in other nodes can keep running without killing the whole job. This is when the "elasticity" property is enabled by giving minimum and maximum for the number of nodes. An useful feature, since the budget to run in GPU is expensive, and one can't afford to waste it.
- when the node becomes available again, the exited processes are automatically restarted.

Common usage:

```
torchrun
    --nnodes=$NUM_NODES
    --nproc-per-node=$NUM_TRAINERS
    --max-restarts=3
    --rdzv-id=$JOB_ID
    --rdzv-backend=c10d
    --rdzv-endpoint=$HOST_NODE_ADDR
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

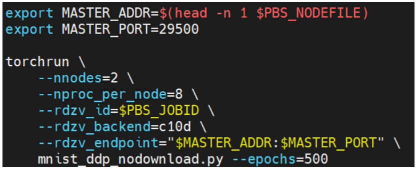
- This command is passed through job script in ASPIRE 2A+ unless run in interactive mode.
- –max-restarts=3 means that the exited groups could be started maximum three times.
- –nprocs-per-node is set to the number of GPU's asked for in the job script per node.
- –nnodes=2, for example, means that the 2 nodes are going to be used.

Torchrun: DDP Launcher (continued· · ·)

In the context of environment in ASPIRE 2A+, the following options apply.

- –rdzv-backend=c10d means that c10d has been chosen as the process-group former.
- –rdzv-id=$PBS_JOBID. This gives an id for the process group that takes part in the cooperation.
- –rdzv-endpoint=”$(head -n 1 $PBS_NODEFILE):29555” The rendezvous end-point decides where to base the backend, c10d. Here it has been chosen as the hostname appearing in the first line of the node-file generated by the scheduler for the job. The number following the ”:” refers to the port number of choice for communication.

This is usually done as shown in the figure below.



```
export MASTER_ADDR=$(head -n 1 $PBS_NODEFILE)
export MASTER_PORT=29500

torchrun \
    --nnodes=2 \
    --nproc_per_node=8 \
    --rdzv_id=$PBS_JOBID \
    --rdzv_backend=c10d \
    --rdzv_endpoint="$MASTER_ADDR:$MASTER_PORT" \
    mnist_ddp_nodownload.py --epochs=500
```

The following environment variables will be made available by `torchrun` for the programs to access.

- RANK Global rank.
- LOCAL_RANK local rank within the node
- WORLD_SIZE total number of processes in the group.
- LOCAL_WORLD_SIZE number processes in the node.

Torchrun: DDP Launcher (continued· · ·)

For an example, the usage of the environment variable LOCAL_RANK:

```
local_rank = int(os.environ["LOCAL_RANK"])
model = torch.nn.parallel.DistributedDataParallel(model,
                                                   device_ids=[local_rank],
                                                   output_device=local_rank)
```

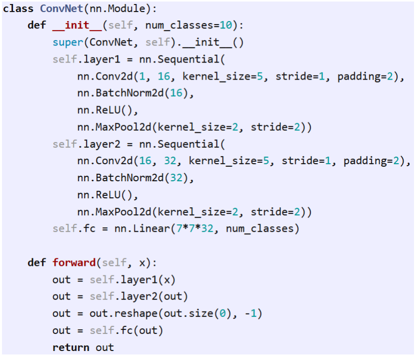
Best practices

- Batch size could be highest possible for making use of a larger SM occupancy and utilization
- Dataloader workers: This number can be sufficiently larger for minimizing the overhead of loading.
- Use prefetch for efficient loading.
- NCCL backend is always preferable for c10d process group's communications.
- Use Torchrun and choose c10d as the backend for rendezvous.
- Never kill using `qdel` command. Let the jobs finish by itself. If you wish to terminate the job by your will, make provisions for it in the code to periodically check for an existence of a file, and exit (after calling dist.destroy_process_group()) when it exists. Such a file can be created by you by **touch** command when you wish to terminate. Our experience suggest that NCCL communications reach a deadlock, causing the scheduler to get the node offline.

PyTorch DDP Exercise: MNIST Image Classification

Download a version of CNN model for image classification from https://yangkky.github.io/2019/07/08/distributed-pytorch-tutorial.html (shown on the right) and run it on 8 GPU's

- Increase the batchsize, and compare the loss at the end of 40 epochs
- Analyze the GPU resource output from the command "qstat -xf" in both cases.
- Increase the learn rate and observe the change in the loss at the end of 40 epochs.
- How do you know that you are not over-fitting?

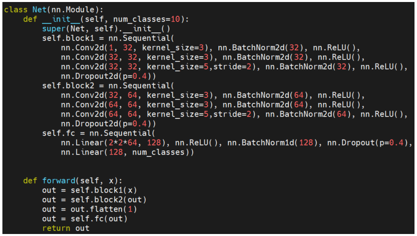


```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(7*7*32, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        return out
```

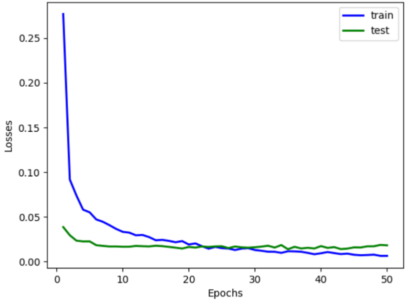
(Use the case folder provided for this exercise. The curves of training and test losses could be obtained on login nodes by executing "%run plot.py" while inside IPython.)

PyTorch DDP Exercise: MNIST with a Different Model

Change the model to the one shown below and plot the performance on the test set. compare it with the loss curves for the previous model. (Use the provided codes.)



```python
class Net(nn.Module):
    def __init__(self, num_classes=10):
        super(Net, self).__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3), nn.BatchNorm2d(32), nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3), nn.BatchNorm2d(32), nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=5,stride=2), nn.BatchNorm2d(32), nn.ReLU(),
            nn.Dropout2d(p=0.4))
        self.block2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3), nn.BatchNorm2d(64), nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3), nn.BatchNorm2d(64), nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=5,stride=2), nn.BatchNorm2d(64), nn.ReLU(),
            nn.Dropout2d(p=0.4))
        self.fc = nn.Sequential(
            nn.Linear(2*2*64, 128), nn.ReLU(), nn.BatchNorm1d(128), nn.Dropout(p=0.4),
            nn.Linear(128, num_classes))


    def forward(self, x):
        out = self.block1(x)
        out = self.block2(out)
        out = out.flatten(1)
        out = self.fc(out)
        return out
```



Experiment by changing the batch size and by diminishing learning-rate over epochs.

PyTorch DDP: Closing Remarks

Some more best practices:

- Always address the GPU using the LOCAL_RANK environment variable. For example, as in

```
local_rank=int(os.environ["LOCAL_RANK"])
device = torch.device(f"cuda:{local_rank}")
```

- Always use DistributedSampler class as in

```
from torch.utils.data.distributed import DistributedSampler
sampler = DistributedSampler(dataset, shuffle=True)
dataloader = DataLoader(dataset, batch_size=100, sampler=sampler
```

If the model is too big for GPU memory:

- The DDP is not ideal for this. PyTorch offers another distributed-execution framework for such large models: Fully Sharded Data Parallel(**FSDP**).
- In this frame work, the each GPU takes part in a serial pipeline, where each operates on several different layers of the model.
- Less efficient than DDP when the whole model can be fit into each GPU.
- For an example of using this in an MNIST classification task, see: https://pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Further learning: Follow another version of MNIST classification using DDP in this link: https://github.com/yqhu/profiler-workshop/blob/main/mnist_ddp.py

Transformers, LLM and FSDP Parallelization

LLM's: Short Introduction

Introduction

- LLM's have revolutionized the world of AI resulting in a new paradigm of genrative AI.
- Tasks: Quesion answering, sentiment analysis, language translations, and sentence completion.
- Made possible due to advances in GPU performance.

Model characteristics

- Mostly encoder-decoder type.
- Sequence to scalar model applied recursively to generate a sequence.

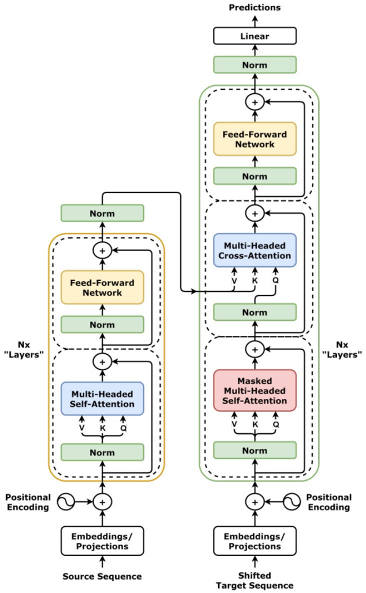
- Since 2017, the LLM's are based on Transformers instead of pure RNN based blocks that contains LSTM or GRU layers.
- The models are pre-trained in self-supervised manner on large quantity of texts using the methods such as masked-token prediction and next-sentence prediction).
- They are then further trained slightly in a supervised manner for domain-specific tasks, and for alignment with socially accepted norms, ethics and beliefs.
- The model sizes varies from few billions to the order of a trillion learnable parameters.

Various LLM's and Their Sizes

LLM	# of Parameters	Depth L	Width d	# of Heads (Q/KV)
GPT-1 [Radford et al., 2018]	0.117B	12	768	12/12
GPT-2 [Radford et al., 2019]	1.5B	48	1,600	25/25
GPT-3 [Brown et al., 2020]	175B	96	12,288	96/96
LLaMA2 [Touvron et al., 2023b]	7B	32	4,096	32/32
	13B	40	5,120	40/40
	70B	80	8,192	64/64
LLaMA3/3.1 [Dubey et al., 2024]	8B	32	4,096	32/8
	70B	80	8,192	64/8
	405B	126	16,384	128/8
Gemma2 [Team et al., 2024]	2B	26	2,304	8/4
	9B	42	3,584	16/8
	37B	46	4,608	32/16
Qwen2.5 [Yang et al., 2024]	0.5B	24	896	14/2
	7B	28	3,584	28/4
	72B	80	8,192	64/8
DeepSeek-V3 [Liu et al., 2024a]	671B	61	7,168	128/128
Falcon [Penedo et al., 2023]	7B	32	4,544	71/71
	40B	60	8,192	128/128
	180B	80	14,848	232/232
Mistral [Jiang et al., 2023a]	7B	32	4,096	32/32

Table credit: https://arxiv.org/pdf/2501.09223

Transformer Model

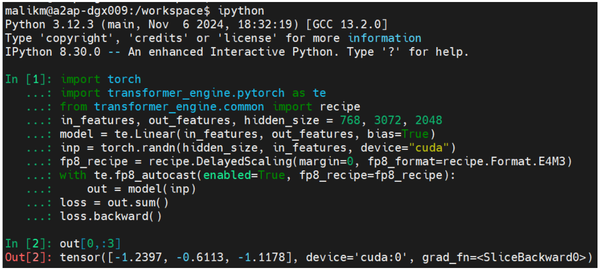


- A path-breaking framework ("Attention is all you need" Vaswani et al. (2017))
- Full model comprises encoder and decoder.
- The encoder contains token and position embeddings and a series of transformer blocks.
- Each transformer block contains normalization, multiheaded attention and feed-forward layers.
- Decoder uses the last time-step's output as input.
- Then subsequently undergoes self attention with causality enabled. (masked self attention)
- The encoder output is used for cross attention.
- skip connections prevent zero-gradient issue.
- After series of the transformer blocks, the next token in the sequence is predicted by a softmax activation.

H100 GPU: Transformer Engine with FP8 Support

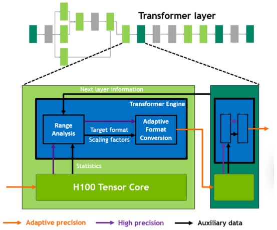
- H100 provides support to Transformer Engine (TE) and FP8 arithmetic.
- TE module provides different types of neural-network layers capable with FP8.
- Useful for Generative-AI tasks that use transformer-architecture blocks (containing multi-headed attention and feed-forward networks).

Example:

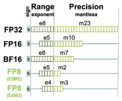


```
malikm@a2ap-dgx009:/workspace$ ipython
Python 3.12.3 (main, Nov  6 2024, 18:32:19) [GCC 13.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.30.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import torch
   ...: import transformer_engine.pytorch as te
   ...: from transformer_engine.common import recipe
   ...: in_features, out_features, hidden_size = 768, 3072, 2048
   ...: model = te.Linear(in_features, out_features, bias=True)
   ...: inp = torch.randn(hidden_size, in_features, device="cuda")
   ...: fp8_recipe = recipe.DelayedScaling(margin=0, fp8_format=recipe.Format.E4M3)
   ...: with te.fp8_autocast(enabled=True, fp8_recipe=fp8_recipe):
   ...:     out = model(inp)
   ...: loss = out.sum()
   ...: loss.backward()

In [2]: out[0,:3]
Out[2]: tensor([-1.2397, -0.6113, -1.1178], device='cuda:0', grad_fn=<SliceBackward0>)
```

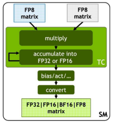
Transformer layer modules:



FP8 Support:



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

LLM: Exercise on Loading a Model, Qwen3-32B-FP8, and Running Inference

We will load a thinking-model that uses the H100 GPU card's specialty to use FP8 data-type, and run inference. One such model is **Qwen3-32B-FP8**, which is small enough to load into a single GPU. This model is known to be a highest performer for code generation from prompts.

Step 1: Create a directory under your scratch folder, and change to that directory as below:

```
mkdir -p  /scratch/llm/Qwen332BFP8
cd  /scratch/llm/
```

Step 2: Install Hugging Face hub's command-line interface using the following command, and download the model:

```
pip install -U "huggingface_hub[cli]"
huggingface-cli download --local-dir ./Qwen332BFP8 Qwen/Qwen3-32B-FP8
```

Wait for the model to be saved in your directory.

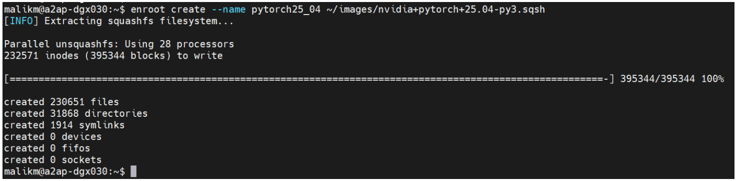
LLM: Exercise on Loading a Model, Qwen3-32B-FP8, and Running Inference (Continues · · ·)

Step 3: Submit a request for an interactive job with two GPU's. Two GPU's are not needed for this exercise, but needed later when running FSDP 2 (coverered in later slides): `qsub -I -l select=1:ngpus=2:mem=400gb -l walltime=3:00:00 -P <project id>`

The next exercise on FSDP 2 requires latest PyTorch version, so we will use th latest version so that we can reuse it later.

Step 4: Use "enroot import" to download a Docker image of latest PyTorch from NGC website and save it as a squash-file system: enroot import docker://nvcr.io#nvidia/pytorch:25.04-py3

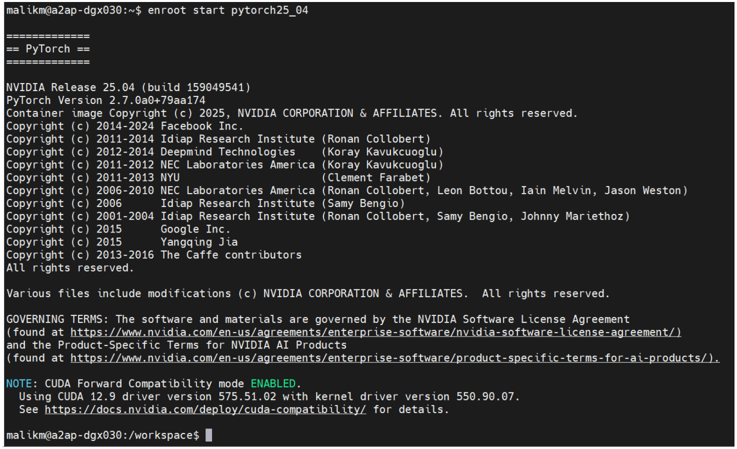
Step 5: Create and start container from the squash-file as below:



```
malikm@a2ap-dgx030:~$ enroot create --name pytorch25_04 ~/images/nvidia+pytorch+25.04-py3.sqsh
[INFO] Extracting squashfs filesystem...

Parallel unsquashfs: Using 28 processors
232571 inodes (395344 blocks) to write

[===========================================================================================-] 395344/395344 100%

created 230651 files
created 31868 directories
created 1914 symlinks
created 0 devices
created 0 fifos
created 0 sockets
malikm@a2ap-dgx030:~$
```

LLM: Exercise on Loading a Model and Running Inference (Continues · · ·)

Step 6: start the container shown in the picture.



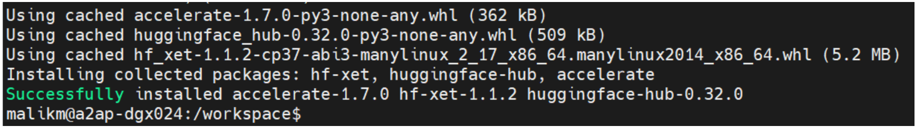
```
malikm@a2ap-dgx030:~$ enroot start pytorch25_04

=============
== PyTorch ==
=============

NVIDIA Release 25.04 (build 159049541)
PyTorch Version 2.7.0a0+79aa174
Container image Copyright (c) 2025, NVIDIA CORPORATION & AFFILIATES. All rights reserved.
Copyright (c) 2014-2024 Facebook Inc.
Copyright (c) 2011-2014 Idiap Research Institute (Ronan Collobert)
Copyright (c) 2012-2014 Deepmind Technologies    (Koray Kavukcuoglu)
Copyright (c) 2011-2012 NEC Laboratories America (Koray Kavukcuoglu)
Copyright (c) 2011-2013 NYU                      (Clement Farabet)
Copyright (c) 2006-2010 NEC Laboratories America (Ronan Collobert, Leon Bottou, Iain Melvin, Jason Weston)
Copyright (c) 2006      Idiap Research Institute (Samy Bengio)
Copyright (c) 2001-2004 Idiap Research Institute (Ronan Collobert, Samy Bengio, Johnny Mariethoz)
Copyright (c) 2015      Google Inc.
Copyright (c) 2015      Yangqing Jia
Copyright (c) 2013-2016 The Caffe contributors
All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION & AFFILIATES.  All rights reserved.

GOVERNING TERMS: The software and materials are governed by the NVIDIA Software License Agreement
(found at https://www.nvidia.com/en-us/agreements/enterprise-software/nvidia-software-license-agreement/)
and the Product-Specific Terms for NVIDIA AI Products
(found at https://www.nvidia.com/en-us/agreements/enterprise-software/product-specific-terms-for-ai-products/).

NOTE: CUDA Forward Compatibility mode ENABLED.
  Using CUDA 12.9 driver version 575.51.02 with kernel driver version 550.90.07.
  See https://docs.nvidia.com/deploy/cuda-compatibility/ for details.

malikm@a2ap-dgx030:/workspace$ 
```

LLM: Exercise on Loading a Model and Running Inference (Continues · · ·)

Step 7: Set proxies to get internet connection inside the container:

```
export no_proxy=localhost,127.0.0.1,10.104.0.0/21
export https_proxy=http://10.104.4.124:10104
export http_proxy=http://10.104.4.124:10104
```

Step 8: Install transformers and accelerate: First, `pip install transformers`. Then, since the model uses pipeline from transformers library with automatics GPU mapping, we need to install the Python package `accelerate`: `pip install accelerate`. This will install `accelerate` as shown below:



```
Using cached accelerate-1.7.0-py3-none-any.whl (362 kB)
Using cached huggingface_hub-0.32.0-py3-none-any.whl (509 kB)
Using cached hf_xet-1.1.2-cp37-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (5.2 MB)
Installing collected packages: hf-xet, huggingface-hub, accelerate
Successfully installed accelerate-1.7.0 hf-xet-1.1.2 huggingface-hub-0.32.0
malikm@a2ap-dgx024:/workspace$
```

Step 9: Since we would like to rn inference by chatting, go into interactive Python: `ipython`.

LLM: Exercise on Loading a Model and Running Inference (Continues · · ·)

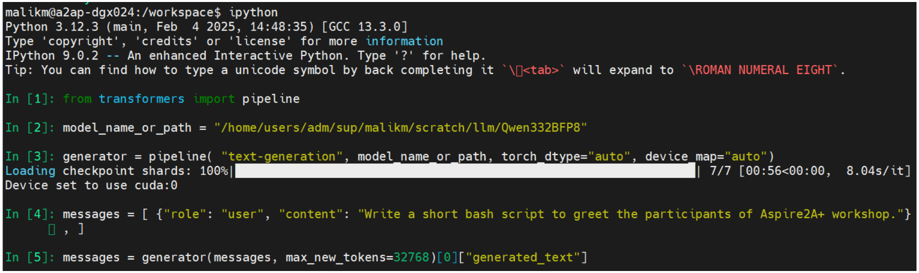
Step 10: In Ipython, given the following sequence of commands to see the response from Qwen3-32B-FP8:

```
from transformers import pipeline

model_name_or_path = "/home/users/adm/sup/malikm/scratch/llm/Qwen332BFP8"

generator = pipeline( "text-generation", model_name_or_path,
        torch_dtype="auto", device_map="auto")

messages = [ "role":  "user", "content":  "Write a short bash script
        to greet the participants of ASPIRE 2A+ workshop.", ]

messages = generator(messages, max_new_tokens=32768)[0]["generated_text"]

messages.append("role":  "user", "content":  "Write the Python version
        of the same")

messages[-1]['content']
```

This outputs shown in the next two slides.

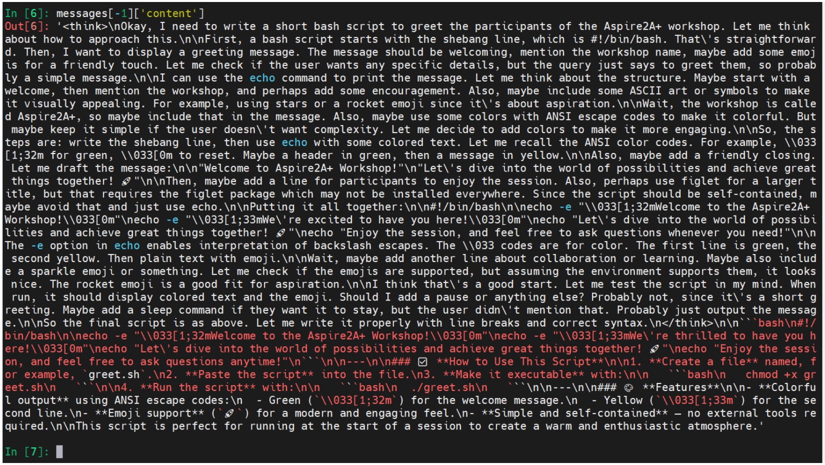
FUJITSU NSCC National Supercomputing Centre SINGAPORE

LLM: Exercise on Loading a Model and Running inference (Continues · · ·)



```
malikm@a2ap-dgx024:/workspace$ ipython
Python 3.12.3 (main, Feb  4 2025, 14:48:35) [GCC 13.3.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 9.0.2 -- An enhanced Interactive Python. Type '?' for help.
Tip: You can find how to type a unicode symbol by back completing it `\Ⅷ<tab>` will expand to `\ROMAN NUMERAL EIGHT`.

In [1]: from transformers import pipeline

In [2]: model_name_or_path = "/home/users/adm/sup/malikm/scratch/llm/Qwen332BFP8"

In [3]: generator = pipeline( "text-generation", model_name_or_path, torch_dtype="auto", device_map="auto")
Loading checkpoint shards: 100%|████████████████████████████████████████| 7/7 [00:56<00:00,  8.04s/it]
Device set to use cuda:0

In [4]: messages = [ {"role": "user", "content": "Write a short bash script to greet the participants of Aspire2A+ workshop."}
     , ]

In [5]: messages = generator(messages, max_new_tokens=32768)[0]["generated_text"]
```

LLM: Exercise on loading a model and running inference (Continues · · ·)



```
In [6]: messages[-1]['content']
Out[6]: '<think>\nOkay, I need to write a short bash script to greet the participants of the Aspire2A+ workshop. Let me think
about how to approach this.\n\nFirst, a bash script starts with the shebang line, which is #!/bin/bash. That\'s straightforwar
d. Then, I want to display a greeting message. The message should be welcoming, mention the workshop name, maybe add some emoj
is for a friendly touch. Let me check if the user wants any specific details, but the query just says to greet them, so probab
ly a simple message.\n\nI can use the echo command to print the message. Let me think about the structure. Maybe start with a
welcome, then mention the workshop, and perhaps add some encouragement. Also, maybe include some ASCII art or symbols to make
it visually appealing. For example, using stars or a rocket emoji since it\'s about aspiration.\n\nWait, the workshop is calle
d Aspire2A+, so maybe include that in the message. Also, maybe use some colors with ANSI escape codes to make it colorful. But
 maybe keep it simple if the user doesn\'t want complexity. Let me decide to add colors to make it more engaging.\n\nSo, the s
teps are: write the shebang line, then use echo with some colored text. Let me recall the ANSI color codes. For example, \\033
[1;32m for green, \\033[0m to reset. Maybe a header in green, then a message in yellow.\n\nAlso, maybe add a friendly closing.
 Let me draft the message:\n\n"Welcome to Aspire2A+ Workshop!"\n"Let\'s dive into the world of possibilities and achieve great
 things together! 🚀"\n\nThen, maybe add a line for participants to enjoy the session. Also, perhaps use figlet for a larger t
itle, but that requires the figlet package which may not be installed everywhere. Since the script should be self-contained, m
aybe avoid that and just use echo.\n\nPutting it all together:\n\n#!/bin/bash\n\necho -e "\\033[1;32mWelcome to the Aspire2A+
Workshop!\\033[0m"\necho -e "\\033[1;33mWe\'re excited to have you here!\\033[0m"\necho "Let\'s dive into the world of possibi
lities and achieve great things together! 🚀"\necho "Enjoy the session, and feel free to ask questions whenever you need!"\n\n
The -e option in echo enables interpretation of backslash escapes. The \\033 codes are for color. The first line is green, the
 second yellow. Then plain text with emoji.\n\nWait, maybe add another line about collaboration or learning. Maybe also includ
e a sparkle emoji or something. Let me check if the emojis are supported, but assuming the environment supports them, it looks
 nice. The rocket emoji is a good fit for aspiration.\n\nI think that\'s a good start. Let me test the script in my mind. When
 run, it should display colored text and the emoji. Should I add a pause or anything else? Probably not, since it\'s a short g
reeting. Maybe add a sleep command if they want it to stay, but the user didn\'t mention that. Probably just output the messag
e.\n\nSo the final script is as above. Let me write it properly with line breaks and correct syntax.\n</think>\n\n```bash\n#!/
bin/bash\n\necho -e "\\033[1;32mWelcome to the Aspire2A+ Workshop!\\033[0m"\necho -e "\\033[1;33mWe\'re thrilled to have you h
ere!\\033[0m"\necho "Let\'s dive into the world of possibilities and achieve great things together! 🚀"\necho "Enjoy the sessi
on, and feel free to ask questions anytime!"\n```\n\n---\n\n### ☑ **How to Use This Script**\n\n1. **Create a file** named, f
or example, `greet.sh`.\n2. **Paste the script** into the file.\n3. **Make it executable** with:\n\n   ```bash\n   chmod +x gr
eet.sh\n   ```\n\n4. **Run the script** with:\n\n   ```bash\n  ./greet.sh\n   ```\n\n---\n\n### 🎨 **Features**\n\n- **Colorfu
l output** using ANSI escape codes:\n  - Green (`\\033[1;32m`) for the welcome message.\n  - Yellow (`\\033[1;33m`) for the se
cond line.\n- **Emoji support** (`🚀`) for a modern and engaging feel.\n- **Simple and self-contained** – no external tools re
quired.\n\nThis script is perfect for running at the start of a session to create a warm and enthusiastic atmosphere.'

In [7]: 
```

SGLang: Introduction and Installation

Introduction:

- SGLang is a tool to serve an LLM. It is also useful to **run benchmarks without serving the model**.
- We will use here for **benchmarking Qwen3-32B-FP8 model's latency and throughput**.
- This model is already with FP8 datatype. But SGLang model can be used for quantization for FP8 eventhough the model does not explicitly use it.

Installation:

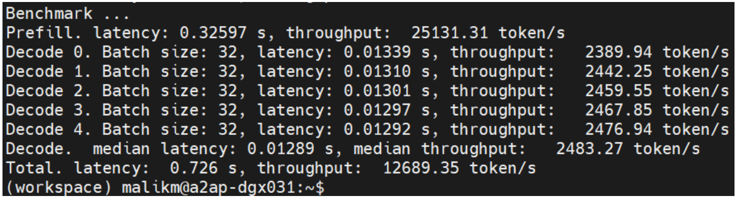
- Start the latest PyTorch container as before.
- UV package in a virtual environment will help accelerating Pip based installations. Issue the following sequence of commands:

```
pip install --upgrade pip
pip install uv
pip install transformers
pip install accelerate
uv venv
source .venv/bin/activate
uv pip install "sglang[all]>=0.4.6.post5"
```

SGLang: Benchmarking an LLM Model

Enter the following command to benchmark a single batch's latency and throughput on two GPU's:

```
python -m sglang.bench_one_batch --model-path
/scratch/llm/Qwen332BFP8 --tokenizer-path
/scratch/llm/Qwen332BFP8 --batch 32 --input-len 256 --output-len
32 --tp-size 2
```

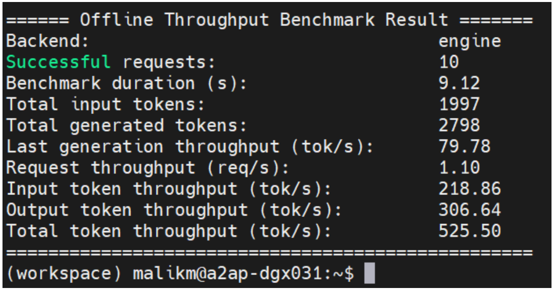


```
Benchmark ...
Prefill. latency: 0.32597 s, throughput:  25131.31 token/s
Decode 0. Batch size: 32, latency: 0.01339 s, throughput:   2389.94 token/s
Decode 1. Batch size: 32, latency: 0.01310 s, throughput:   2442.25 token/s
Decode 2. Batch size: 32, latency: 0.01301 s, throughput:   2459.55 token/s
Decode 3. Batch size: 32, latency: 0.01297 s, throughput:   2467.85 token/s
Decode 4. Batch size: 32, latency: 0.01292 s, throughput:   2476.94 token/s
Decode.  median latency: 0.01289 s, median throughput:   2483.27 token/s
Total. latency:  0.726 s, throughput:  12689.35 token/s
(workspace) malikm@a2ap-dgx031:~$
```

SGLang: Benchmarking an LLM Model (Continued · · ·)

Enter the following command to benchmark the offline throughput on two GPU's:

```
python -m sglang.bench_offline_throughput --model-path
/scratch/llm/Qwen332BFP8 --tokenizer-path
/scratch/llm/Qwen332BFP8 --num-prompts 10 --tp-size 2
```

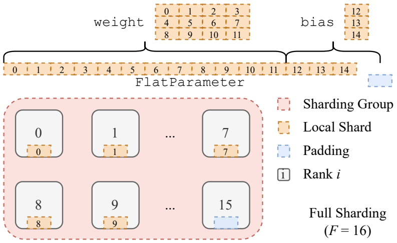


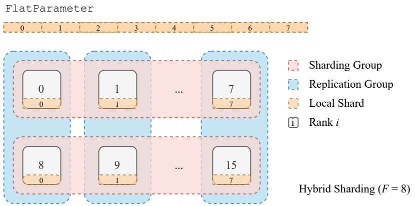
```
====== Offline Throughput Benchmark Result =======
Backend:                                 engine
Successful requests:                     10
Benchmark duration (s):                  9.12
Total input tokens:                      1997
Total generated tokens:                  2798
Last generation throughput (tok/s):      79.78
Request throughput (req/s):              1.10
Input token throughput (tok/s):          218.86
Output token throughput (tok/s):         306.64
Total token throughput (tok/s):          525.50
==================================================
(workspace) malikm@a2ap-dgx031:~$
```

Introduction to FSDP

- **FSDP** class from the module **torch·Distributed** is another tool for process based parallelization on multiple GPU's besides DDP.
- Unlike DDP, the replicas of the model do not reside in each process or GPU in this method.
- Each GPU has a shard of the model.
- Suitable when the model is too big to fit into a GPU.
- Like DDP, it too uses collective-communication functions like NCCL.
- Like DDP, FSDP too is launched using "torchrun", which spans the specified number of processes.
- Introductory materials:
  https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html
  https://docs.pytorch.org/docs/stable/distributed.fsdp.fully_shard.html
  https://docs.pytorch.org/docs/stable/distributed.tensor.html
  https://github.com/pytorch/torchtitan/blob/main/docs/fsdp.md
  https://github.com/pytorch/examples/tree/main/distributed/FSDP2
  https://arxiv.org/abs/2304.11277
  https://www.youtube.com/watch?v=By_O0k102PY

FSDP Sharding



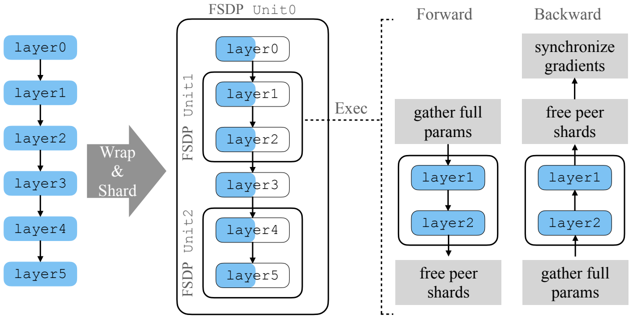


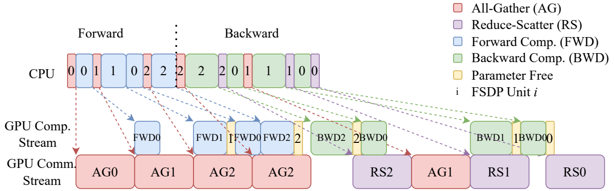
- Sharding reduces memory footprint.
- Increases communication.
- Some layers can be chosen to not be sharded if small in size. In this case the layer gets replicated across the GPU's.
- Sharding means communication overhead.
- All-gather and reduce-scatter are the collective communications used.

- Two methods: FSDP1 and FSDP2
- FSDP1 may become deprecated.
- FSDP2 uses DTensors (distributed)
- No sharding policies in FSDP2: use "if" conditions on the type of layers.
- More on diff. between FSDP 1 and 2: https://github.com/pytorch/torchtitan/blob/main/docs/fsdp.md
- FSDP2 uses `fully_shard()` as the constructor.

Image credit: https://arxiv.org/abs/2304.11277

FSDP: Sharding, Unsharding and Computation Overlap





FSDP is DDP + more communications.

All-gather makes it possible to make each shard equivalent to the full model momentarily for the layer of active computation at a chosen time.

After computations in a layer, it is resharded.

Communication between processes **overlap** in time with the computation in them. See the figure on left bottom. Image credit: https://arxiv.org/abs/2304.11277

FSDP 2: fully_shard()

fully_shard(module, *, mesh=None, reshard_after_forward=True, shard_placement_fn=None, mp_policy=MixedPrecisionPolicy(param_dtype=None, reduce_dtype=None, output_dtype=None, cast_forward_inputs=True), offload_policy=OffloadPolicy(), ignored_params=None)

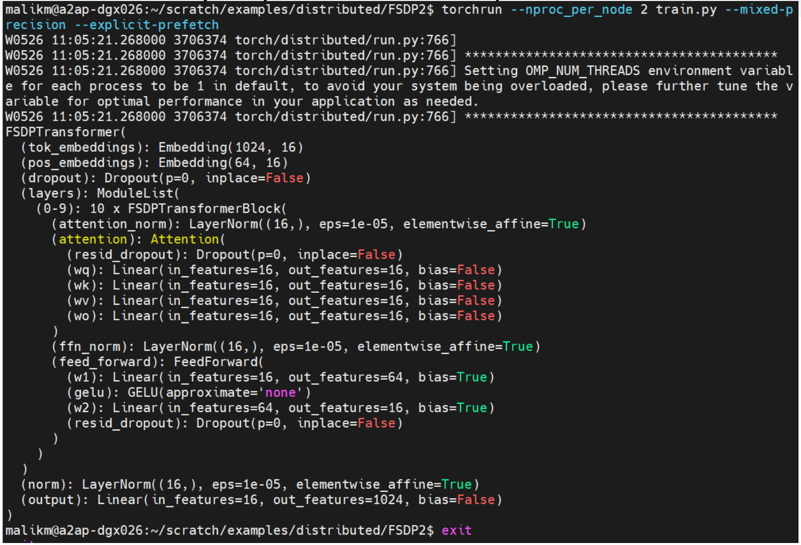
- **Bottom-up sharding**: If a module contains nested submodules, the `fully_shard()` should be applied to the inner-most submodules first, and then progressing to parent modules until the root (outer-most) module.
- **prefetching**: During the forward and backward pass, the next-layers' parameters need to be all-gathered while the computation on current layer is going on. There are methods available for specifying the number of layers.
- **Hybrid- and full-sharding**: The "mesh" argument above in the constructor can be used to specify which rank should contain shards, and which ranks should contain the replica of existing shards.
- **Mixed-precision policy**: This argument in the constructor can modify the data-types of parameters and gradients.
- **Offload policy**: The default setting "offload_policy=OffloadPolicy()" disables CPU offloading. offload_policy=CPUOffloadPolicy() enables it. It uses CPU RAM as a swap space for storing the sharded parameters of inactive layers.

FSDP: Example

As a demonstration, let us do an example available on PyTorch's official tutorial.

- Create and Start the latest PyTorch container on two GPU's through an interactive job as we did earlier for running an inference on Qwen3-32B-FP8. But, make sure to start with the environment variable CUDA_VISIBLE_DEVICES as in `enroot start --env CUDA_VISIBLE_DEVICES=$CUDA_VISIBLE_DEVICES pytorch25_04`.
- Change to scratch folder and download the examples from PyTorch's Github page: git clone https://github.com/pytorch/examples
- Change to the directory ∼/scratch/examples/distributed/FSDP2.
- Note the following in train.py:
  - Customize the functions for pre-fetching
  - bottom-top way of applying `fsdp()` on module layers and the model. (This example does not use mesh to create replicated FSDP units.)
  - The option for mixed precision allows the use of different precisions for parameters (Bfloat 16 bit) and reduce operation (32 bit).
- Run the workload using torchrun: `torchrun --nproc_per_node 2 train.py --mixed-precision --explicit-prefetch`
  The output is shown in the next slide

FSDP: Example (Continues · · ·)



```
malikm@a2ap-dgx026:~/scratch/examples/distributed/FSDP2$ torchrun --nproc_per_node 2 train.py --mixed-p
recision --explicit-prefetch
W0526 11:05:21.268000 3706374 torch/distributed/run.py:766]
W0526 11:05:21.268000 3706374 torch/distributed/run.py:766] *****************************************
W0526 11:05:21.268000 3706374 torch/distributed/run.py:766] Setting OMP_NUM_THREADS environment variabl
e for each process to be 1 in default, to avoid your system being overloaded, please further tune the v
ariable for optimal performance in your application as needed.
W0526 11:05:21.268000 3706374 torch/distributed/run.py:766] *****************************************
FSDPTransformer(
  (tok_embeddings): Embedding(1024, 16)
  (pos_embeddings): Embedding(64, 16)
  (dropout): Dropout(p=0, inplace=False)
  (layers): ModuleList(
    (0-9): 10 x FSDPTransformerBlock(
      (attention_norm): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
      (attention): Attention(
        (resid_dropout): Dropout(p=0, inplace=False)
        (wq): Linear(in_features=16, out_features=16, bias=False)
        (wk): Linear(in_features=16, out_features=16, bias=False)
        (wv): Linear(in_features=16, out_features=16, bias=False)
        (wo): Linear(in_features=16, out_features=16, bias=False)
      )
      (ffn_norm): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
      (feed_forward): FeedForward(
        (w1): Linear(in_features=16, out_features=64, bias=True)
        (gelu): GELU(approximate='none')
        (w2): Linear(in_features=64, out_features=16, bias=True)
        (resid_dropout): Dropout(p=0, inplace=False)
      )
    )
  )
  (norm): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
  (output): Linear(in_features=16, out_features=1024, bias=False)
)
malikm@a2ap-dgx026:~/scratch/examples/distributed/FSDP2$ exit
```

The model printed from `rank=0`.

FSDP: Closing Remarks

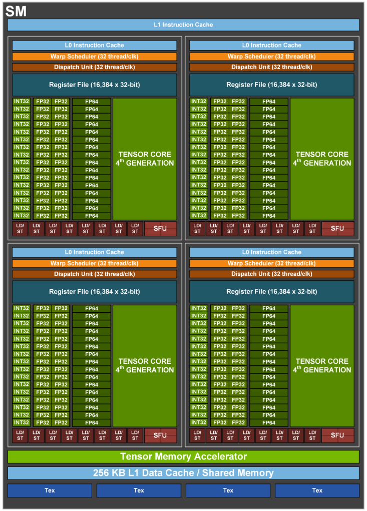
- If the model is small to fit into a GPU, use DDP. FSDP is inefficient for such cases.
- If Using multidimensional `mesh` in `fully_shard()`, note that the sharding happens on ranks given by `dim = 0` of the mesh array, and replicated on the other dimension ranks correspondingly. If using this feature, allowing the intranode ranks to have shards, and the ranks across the nodes to be corresponding replicas would enhance performance. This is because, the inter-node `all-gather()` communications are absent.
- CPU-offloading can greatly reduce the GPU memory footprint of the workload, thought it will increase the H2D and D2H communications.

Computations in GPU Using CUDA

Introduction

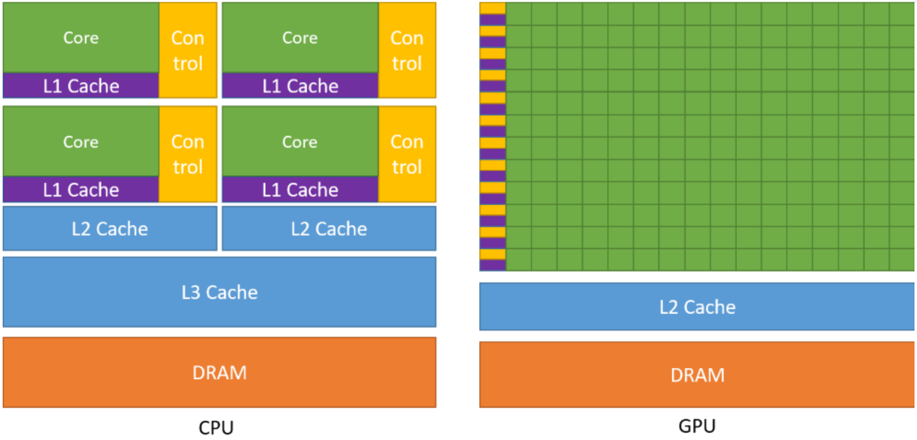
- Massive parallelization. High throughput calculations
- More than 250,000 threads in H100. 138 SM's, each with 128 CUDA cores.
- CUDA is an extension to C/C++ (by header file cuda_runtime.h and library).
- CUDA gives a framework to define the functions that need to be executed in GPU. These functions are called **CUDA kernels**
- Memory management functions: allocation, movement between GPU and CPU, and releasing the memory in GPU.
- Provision of thread and block specific indices to identify them. threadIdx.x, blockIdx.x, blockDim.x, gridDim.x
- Libraries for scientific computations. cuBLAS, cuDNN and cuFFT.

Introduction (continues · · ·)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

- Qualifiers for kernels callable from CPU and GPU: __global__ and __device__.
- Qualifiers for allocations of sharable variables within the block (i.e. within SM). __sharable__.
- Device selection:
  `cudaSetDevice(0) // Sets to device to 0 GPU`

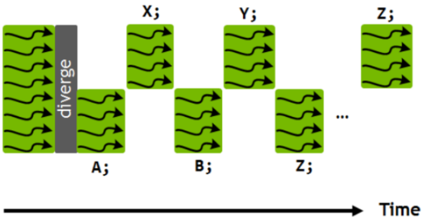
Introduction (continues · · ·)



- CPU: Low Latency, GPU: High throughput.
- GPU needs more time to spawn threads, but it can spawn thousands.
- H100 has more than 16,000 cores.

SIMT vs SIMD

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



- SIMT and SIMD both are data parallelism
- Since CUDA cores lacks exclusive caches for each, the threads are operated in locked steps.
- If there is no `if` conditions, SIMT is most efficient.
- When there are `if` conditions, the branching is handled through *masking*.
- The threads are executed in CUDA cores grouped into a size of 32. Each group is known as a warp.
- If the `if` condition is true for all threads in a warp, and false for all threads in different warp, the idling time of threads in each warp is avoided.
- If a warp would be idle waiting for memory access, it will be replaced in their CUDA cores by a different warp.

Image credit: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Common Flow of CUDA Programs

1. Define the kernel function outside "`main()`" function.
2. In the "`main()`" function,
    1. Declare CPU variables and pointers to the memory in GPU.
    2. Allocate memory in GPU using the pointers.
    3. Copy from CPU to GPU the contents of variables required by kernel.
    4. Launch the kernel in GPU from CPU.
    5. After the calculations in GPU, copy the result from GPU memory to CPU memory.
    6. De-allocate the GPU memory to avoid any memory-leaks.
    7. continue with CPU workload if any.
    8. Finally, de-allocate the dynamically allocated memories in CPU if any.
3. Load the CUDA module: `module load cuda/12.2.2`
4. Compile using the command: `nvcc -gencode arch=compute_90, code=sm_90 -o executable_name cude_filename.cu`
5. Run: `./executable_name`

Memory Management in GPU

cudaMalloc((void**)&a, size_in_bytes)	Allocates memory in GPU, and stores the GPU memory pointer in a CPU memory pointer. Note, The "a" should already be a pointer variable. The "(void**)" could be avoided in modern versions of nvcc compiler.
cudaMemcpy(ptr_in_device, ptr_in_host, size_in_bytes, cudaMemcpyHostToDevice)	Copies the content of size_in_bytes number of bytes starting from the memory location ptr_in_host to the GPU memory location that starts at ptr_in_device.
cudaMemcpy(ptr_in_host, ptr_in_device, size_in_bytes, cudaMemcpyDeviceToHost)	Same as above, but vice versa.
cudaMemset(ptr_to_integer_in_dev, 0, N * sizeof(int));	Initialize in GPU. In this example, an array given by its name, "ptr_to_integer_in_dev" (remember, array names are already a pointer) gets initialized by zero.
cudaFree(ptr_in_device)	Frees the memory in GPU to avoid memory-leaks.

Important Keywords that Acts as Specifiers or Qualifiers

__global__	GPU kernel declaration callable from CPU
__device__	GPU kernel declaration callable from GPU
__host__	CPU functions (default)
__shared__	Memory shared by threads within a "block" (see next slide)

Examples

- __global__ double* kernel_callable_from_CPU (double* arr) { ··· } . A function that takes in a double precision array and returns another such.
- __device__ double* kernel_callable_from_GPU (double* arr) { ··· } . Same as the above but callable only inside another GPU kernel.
- __shared__ double Array_accessible_by_all_thread[10,000]. All threads can access, but possible for race. Use synchronization when needed.

Memory Allocation for Functions for 2D and 3D Arrays

2D array: cudaMallocPitch()

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Here, "pitch" has the same meaning of "stride". The memory size that need to be traversed when an index of an array in a direction is incremented by 1. In this example, it refers to the case when row is incremented.

3D array: cudaMalloc3D()

```
// Host code
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                    height, depth);
cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

// Device code
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr,
                         int width, int height, int depth)
{
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
            }
        }
    }
}
```

"slicePitch" refers to the amount of memory to be traversed when the z-direction index increments by one.

The code is from:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Threads, Blocks, Clusters in grid

- A "block" is a group of threads. Blocks binds to different SM's by default. H100 can contain up to 2048 threads in a single block.
- Thread and block indices are 3-tuples to facilitate working with 3D arrays. Example: threadIdx.x, blockIdx.x. The "x" here can also be "y" or "z".
- There is an in-built 3-tuple datatype for declaring number of threads and blocks for the kernel functions in each direction: **dim3** . Example: dim3 blocks(4, 4). Here the no. of blocks in "z" direction defaults to 1.
- Dimensions of a block are the number of threads in each direction. blockDim.x, blockDim.y and blockDim.z
- Computations in each block is independent of other blocks. Mostly data parallelism: SPMD
- It is best to synchronize the threads whenever a new set of calculations are going to begin on the shared memory. Intrinsic function __syncthreads() ("Intrinsic" means low-level instruction with high performance.)

Launching CUDA Kernels and Grid

- Kernels are launched in a grid using triple angular brackets containing the block and thread dimensions. Example:

```
dim3 blocks(4, 4);
dim3 threads(16, 16);
myKernel<<<blocks, threads>>>(...);
```

- The grid dimension in each direction gridDim.x, gridDim.y, gridDim.z (blocks in the grid).
- The grid dimension is not used in kernel launch.
- H100 also supports "clusters": A grid of blocks. Grid dimensions needed to launch kernels in clusters.

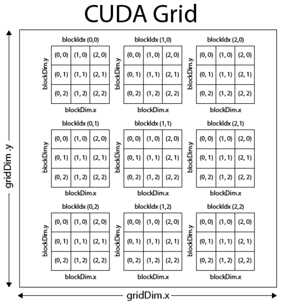
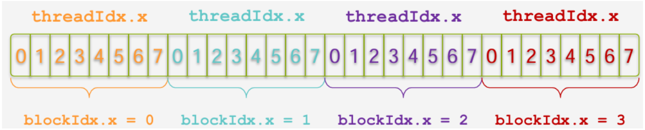


Image credit:

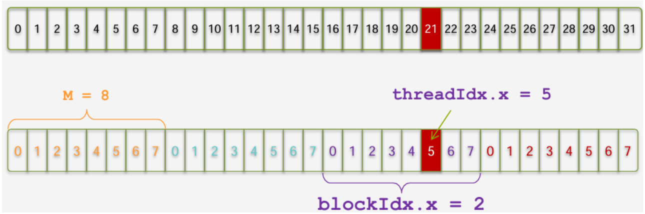
https://www.3dgep.com/cuda-thread-execution-model

A vector as a 1D Thread Block

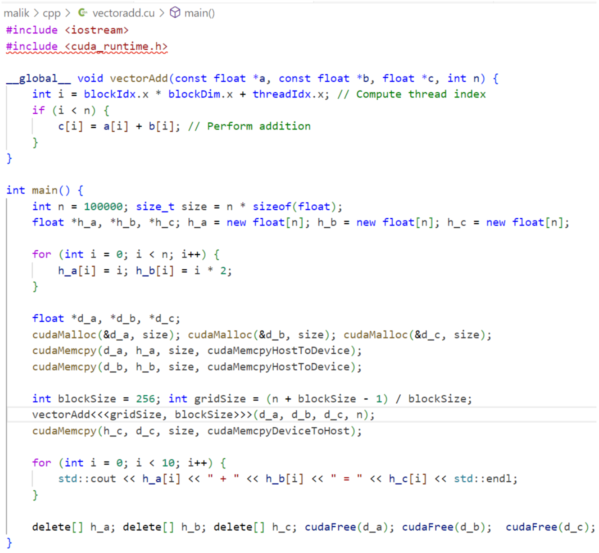
- Consider a vector **v** of 32 elements.
- Split as below:
  - 4 blocks Therefore, gridDim.x $=$ 4 and 0 $\leq$ blockIdx.x $\leq$ 3
  - 8 threads in each block. Therefore, blockDim.x $=$ 8 and 0 $\leq$ threadIdx.x $\leq$ 7



- Then, indexing in eqch block:
  int index $=$ threadIdx.x $+$ blockIdx.x * blockDim.x



Vector Addition Using CUDA



malik > cpp > vectoradd.cu > main()

```
#include <iostream>
#include <cuda_runtime.h>

__global__ void vectorAdd(const float *a, const float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute thread index
    if (i < n) {
        c[i] = a[i] + b[i]; // Perform addition
    }
}

int main() {
    int n = 100000; size_t size = n * sizeof(float);
    float *h_a, *h_b, *h_c; h_a = new float[n]; h_b = new float[n]; h_c = new float[n];

    for (int i = 0; i < n; i++) {
        h_a[i] = i; h_b[i] = i * 2;
    }

    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, size); cudaMalloc(&d_b, size); cudaMalloc(&d_c, size);
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    int blockSize = 256; int gridSize = (n + blockSize - 1) / blockSize;
    vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < 10; i++) {
        std::cout << h_a[i] << " + " << h_b[i] << " = " << h_c[i] << std::endl;
    }

    delete[] h_a; delete[] h_b; delete[] h_c; cudaFree(d_a); cudaFree(d_b);  cudaFree(d_c);
}
```

- Defines the kernel to perform vector addition by using thread and block ids.
- Allocates memory in GPU from the host function.
- Copies buffers
- calls the kernel
- copy back to CPU
- Frees memory

Exercise: Compile and run this code.

CUDA Functions for Performance Measuring

Snippet for measuring GPU kernel time:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
vectorAdd <<< nblk, nthrd >>> (···);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float ms;// ms stores time in "ms"
cudaEventElapsedTime(&ms, start,
stop);
```

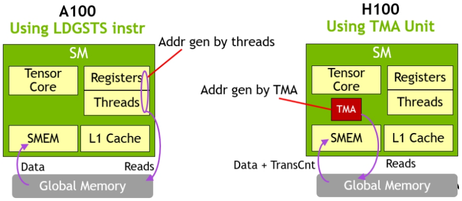
Measuring occupancy:

```
int numBlocks;        // Occupancy in terms of active blocks
int blockSize = 32;

// These variables are used to convert occupancy to warps
int device;
cudaDeviceProp prop;
int activeWarps;
int maxWarps;

cudaGetDevice(&device);
cudaGetDeviceProperties(&prop, device);

cudaOccupancyMaxActiveBlocksPerMultiprocessor(
    &numBlocks,
    MyKernel,
    blockSize,
    0);

activeWarps = numBlocks * blockSize / prop.warpSize;
maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;
```

Occupancy = 100 x no. of active warps / max warp

H100 Specific CUDA Enhancements: TMA

- H100 cards have specific hardware to carry out memory movements from global memory to the shared memory of thread blocks.
- From the global memory of each GPU, copying the rows or columns of array (i.e., the data section of the object "Tensor") to the shared location within a block requires strided copying. (Strides are briefly introduced in the section for MPI).
- H100 GPU has a special method known as Tensor Memory Accelerator to do these strided copying to each block's shared locations through dedicated hardware different from CUDA cores.
- These operations are asynchronous.



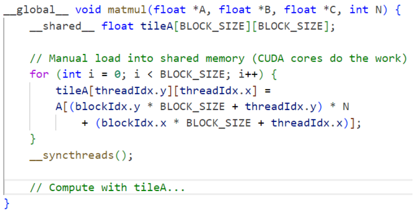
- Advantages:
  1. CUDA cores not used for this.
  2. Better overlap of compute and copy.
- Use cases:
  - Deep Learning (cuDNN library)
  - Computations on a stencil of grids
  - Linear Algebra (cuBLAS) and spectral problems (cuFFT)

Figure credit: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

H100 Specific CUDA Enhancements: TMA Example

Loading a tile (i.e. a rectangular section) of a matrix from global memory to shared memory:

Without TMA:

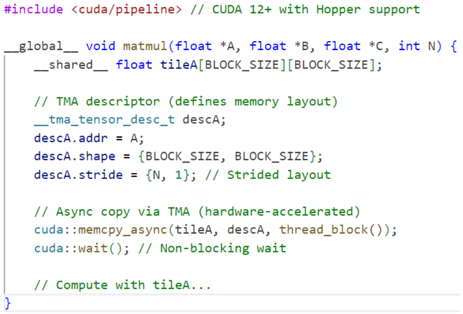


```
__global__ void matmul(float *A, float *B, float *C, int N) {
    __shared__ float tileA[BLOCK_SIZE][BLOCK_SIZE];

    // Manual load into shared memory (CUDA cores do the work)
    for (int i = 0; i < BLOCK_SIZE; i++) {
        tileA[threadIdx.y][threadIdx.x] =
        A[(blockIdx.y * BLOCK_SIZE + threadIdx.y) * N
            + (blockIdx.x * BLOCK_SIZE + threadIdx.x)];
    }
    __syncthreads();

    // Compute with tileA...
}
```

TMA: Requirement and enabling:

- Use -arch=sm_90a as the compiler flag.
- Needs CUDA versions 12.x

With TMA:



```
#include <cuda/pipeline> // CUDA 12+ with Hopper support

__global__ void matmul(float *A, float *B, float *C, int N) {
    __shared__ float tileA[BLOCK_SIZE][BLOCK_SIZE];

    // TMA descriptor (defines memory layout)
    __tma_tensor_desc_t descA;
    descA.addr = A;
    descA.shape = {BLOCK_SIZE, BLOCK_SIZE};
    descA.stride = {N, 1}; // Strided layout

    // Async copy via TMA (hardware-accelerated)
    cuda::memcpy_async(tileA, descA, thread_block());
    cuda::wait(); // Non-blocking wait

    // Compute with tileA...
}
```

Best practices

- De-allocate all unwanted memories.
- Memory Pooling: If reusing a memory of one variable is possible for a new variable, it is better than a new allocation. This avoids overhead and fragmentation of address space.
- Coalesced access: Make sure to get the adjacent threads act on contiguous parts of memory with "stride=1". This will help with the cache memory for each thread block.
- Use shared memories of thread-block for faster access.
- Pin the CPU memory and use asynchronous methods for memory movements.
- Use multiple CUDA streams for a better overlap between compute and copy operations.
- Use CUDA-optimized libraries such as cuDNN, cuBLAS, cuFFT, cuSOLVER, cuRAND, and NCCL.
- If doing Checkpointing, use a separate stream to have a better overlap with the streams for compute kernels. It is best to use /raid storage via GPUDirect for this purpose. This will reduce the overhead of memory movements through CPU.

Closing Remarks for the Workshop

- Avoid CPU only jobs. ASPIRE2A+ requires GPU's to be used in all job submissions.
- It is recommended to use most of the GPU memory. In deep-learning workloads, this possible by increasing the batch size, beside the model size. Increasing the batch size could allow you to use large learning rates initially, saving the GPU time considerably.
- Make sure to reach a highest possible SM-utilization rate (reported by the job output). This can be achieved by increasing the model size in the case of deep-learning applications. In pure CUDA codes, this can be achieved by increasing the problem size, by using several streams, and by getting the memcpy to overlap in time with kernel jobs.
- Do not mention a container in the job script if it is not used by the job. This will increase unnecessary IO on the /raid storage.
- Make sure to load the right modules and mention the path sequence in a manner right versions of the software are used. For example, you may want to use your local Python, but the system Python may be preceding in the PATH variable.
- If not using the obtained interactive session through `qsub`, make sure to release it to facilitate the usage by other users.
- Avoid submitting sleep jobs to hog the node. It will reduce the efficiency the system, besides depleting the SU allocation for the project.

OpenMP: Introduction and Processor Bindings

Introduction

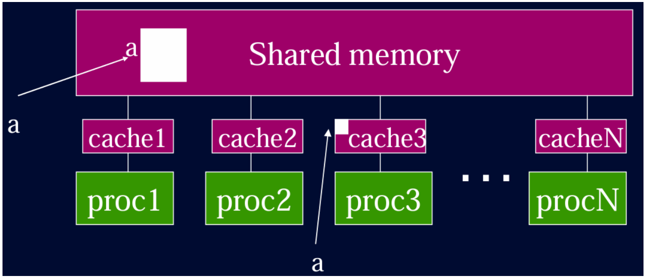
Features

- Parallelization using threads: Threads share single process
- Works on shared memory: All threads can access all variables, but some of them could be thread-private
- Uses compiler directives: Switch between parallel & serial versions at compile time without a need to change the codes.
- Environment can controls the number of threads unless overridden by directives

Drawbacks

- Not suitable for distributed memory
- Race condition by threads to access same address location if programmed badly
- Thread synchronization could slow-down executions.

Shared Memory

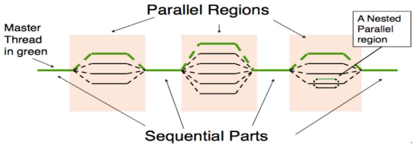


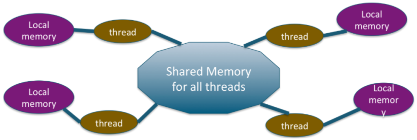
- OpenMP exploits shared-memory architecture of multiple CPU's in a single node
- When the number of threads (T) is less than number of physical cores (P), i.e., $T \leq P$, each threads binds to the physical cores. If $2P \geq T > P$, then the threads binds to logical cores (hardware threads).

Fork & Join and Shared & Private Memories

Single Program Multiple Data (SPMD) parallel is most common with OpenMP.

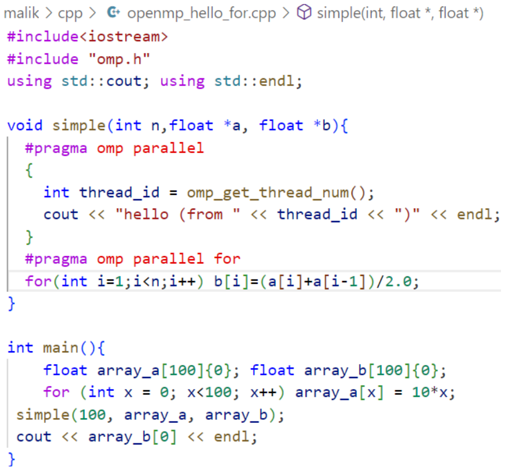
- Master thread: Thread id $= 0$
- Forks in parallel region and joins as it finishes
- Forking can be nested





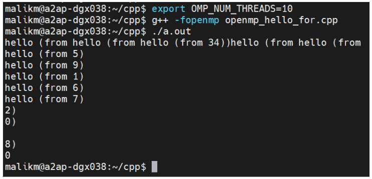
- Threads can have local memories
- Mostly uses shared memory
- synchronization is needed when accessing shared data

Sample Code: Pragma, Parallel & Parallel For



malik > cpp > openmp_hello_for.cpp > simple(int, float *, float *)

```cpp
#include<iostream>
#include "omp.h"
using std::cout; using std::endl;

void simple(int n,float *a, float *b){
  #pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    cout << "hello (from " << thread_id << ")" << endl;
  }
  #pragma omp parallel for
  for(int i=1;i<n;i++) b[i]=(a[i]+a[i-1])/2.0;
}

int main(){
    float array_a[100]{0}; float array_b[100]{0};
    for (int x = 0; x<100; x++) array_a[x] = 10*x;
 simple(100, array_a, array_b);
 cout << array_b[0] << endl;
}
```

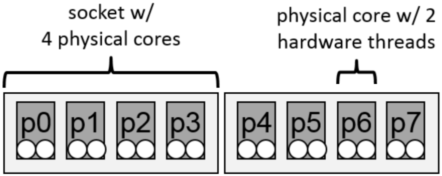


```
malikm@a2ap-dgx038:~/cpp$ export OMP_NUM_THREADS=10
malikm@a2ap-dgx038:~/cpp$ g++ -fopenmp openmp_hello_for.cpp
malikm@a2ap-dgx038:~/cpp$ ./a.out
hello (from hello (from hello (from 34))hello (from hello (from
hello (from 5)
hello (from 9)
hello (from 1)
hello (from 6)
hello (from 7)
2)
0)

8)
0
malikm@a2ap-dgx038:~/cpp$
```

The iteration index is thread-private by default.

OpenMP Processor Binding to Enhance Performance

- Set the environment variable, OMP_PROC_BIND=true
- The threads can be bounded to be closer to each other, or spread out to all physical cores.



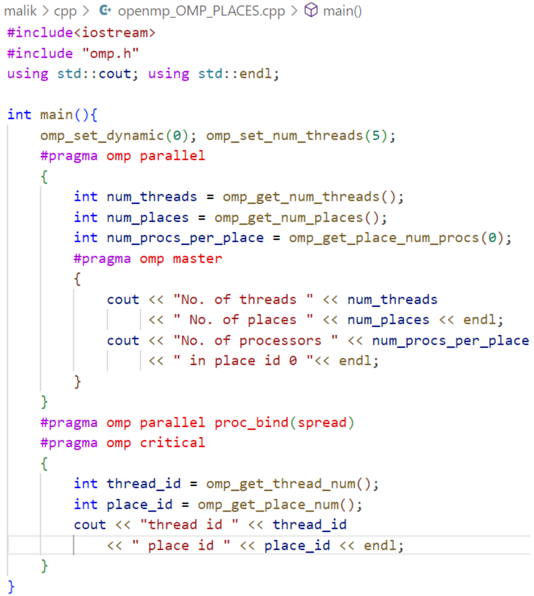
socket w/ 4 physical cores
physical core w/ 2 hardware threads
p0
p1
p2
p3
p4
p5
p6
p7

- #pragma omp parallel proc_bind(master) binds all threads to the single core
- #pragma omp parallel proc_bind(close) binds threads such that the the threads with adjacent thread-id's are closer to each other
- #pragma omp parallel proc_bind(spread) binds threads such that the the threads with adjacent thread-id's are spread to the whole range of available CPU core id's.
- If the data that each thread works on is closer to memory, `proc_bind(close)` could benefit by avoiding cache misses.

OpenMP Processor Binding: OMP_PLACES

- OMP_PLACES controls how the CPU's including hardware threads are numbered
- OMP_PLACES options: threads, cores, sockets, or list convention
- These following options are equivalent for cores having 4 hardware threads
  - export OMP_PLACES=threads
  - export OMP_PLACES="threads(4)"
  - export OMP_PLACES="0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15"
  - export OMP_PLACES="0:4,4:4,8:4,12:4"
  - export OMP_PLACES="0:4:4:4"
- #pragma omp parallel proc_bind(spread) binds threads such that the the threads with adjacent thread-id's are spread to the whole range of available CPU core id's.
- If the data that each thread works on is closer to memory, `proc_bind(close)` could benefit by avoiding cache misses.

Using OMP_PLACES

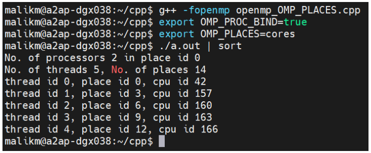


malik > cpp > openmp_OMP_PLACES.cpp > main()

```cpp
#include<iostream>
#include "omp.h"
using std::cout; using std::endl;

int main(){
    omp_set_dynamic(0); omp_set_num_threads(5);
    #pragma omp parallel
    {
        int num_threads = omp_get_num_threads();
        int num_places = omp_get_num_places();
        int num_procs_per_place = omp_get_place_num_procs(0);
        #pragma omp master
        {
            cout << "No. of threads " << num_threads
                << " No. of places " << num_places << endl;
            cout << "No. of processors " << num_procs_per_place
                << " in place id 0 "<< endl;
        }
    }
    #pragma omp parallel proc_bind(spread)
    #pragma omp critical
    {
        int thread_id = omp_get_thread_num();
        int place_id = omp_get_place_num();
        cout << "thread id " << thread_id
            << " place id " << place_id << endl;
    }
}
```

Chosen parameters:

- Five threads
- proc_bind(spread)
- OMP_PLACES=cores



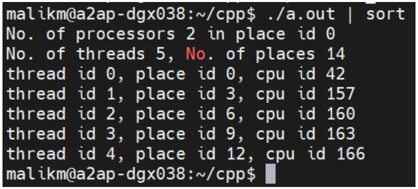
```
malikm@a2ap-dgx038:~/cpp$ g++ -fopenmp openmp_OMP_PLACES.cpp
malikm@a2ap-dgx038:~/cpp$ export OMP_PROC_BIND=true
malikm@a2ap-dgx038:~/cpp$ export OMP_PLACES=cores
malikm@a2ap-dgx038:~/cpp$ ./a.out | sort
No. of processors 2 in place id 0
No. of threads 5, No. of places 14
thread id 0, place id 0, cpu id 42
thread id 1, place id 3, cpu id 157
thread id 2, place id 6, cpu id 160
thread id 3, place id 9, cpu id 163
thread id 4, place id 12, cpu id 166
malikm@a2ap-dgx038:~/cpp$
```

Output on a compute node with:
1 socket, 14 cores, and
2 hardware-threads in each core

proc_bind() vs OMP_PLACES

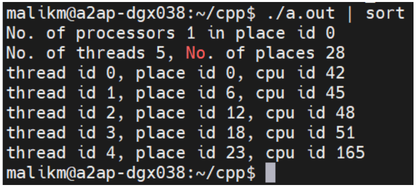
Threads = 5; cores = 14; hardware-threads = 2; socket=1

proc_bind(spread); OMP_PLACES=cores



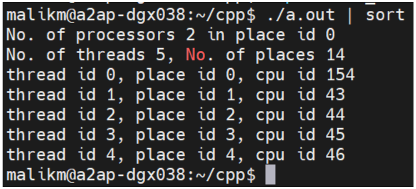
```
malikm@a2ap-dgx038:~/cpp$ ./a.out | sort
No. of processors 2 in place id 0
No. of threads 5, No. of places 14
thread id 0, place id 0, cpu id 42
thread id 1, place id 3, cpu id 157
thread id 2, place id 6, cpu id 160
thread id 3, place id 9, cpu id 163
thread id 4, place id 12, cpu id 166
malikm@a2ap-dgx038:~/cpp$
```

proc_bind(spread); OMP_PLACES=threads



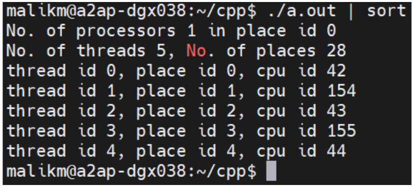
```
malikm@a2ap-dgx038:~/cpp$ ./a.out | sort
No. of processors 1 in place id 0
No. of threads 5, No. of places 28
thread id 0, place id 0, cpu id 42
thread id 1, place id 6, cpu id 45
thread id 2, place id 12, cpu id 48
thread id 3, place id 18, cpu id 51
thread id 4, place id 23, cpu id 165
malikm@a2ap-dgx038:~/cpp$
```

proc_bind(close); OMP_PLACES=cores



```
malikm@a2ap-dgx038:~/cpp$ ./a.out | sort
No. of processors 2 in place id 0
No. of threads 5, No. of places 14
thread id 0, place id 0, cpu id 154
thread id 1, place id 1, cpu id 43
thread id 2, place id 2, cpu id 44
thread id 3, place id 3, cpu id 45
thread id 4, place id 4, cpu id 46
malikm@a2ap-dgx038:~/cpp$
```

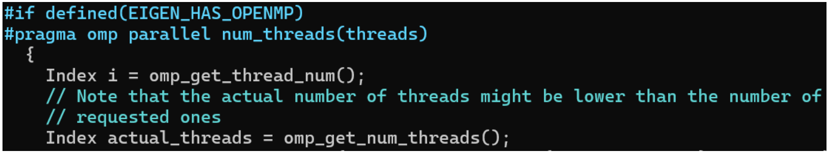
proc_bind(close); OMP_PLACES=threads



```
malikm@a2ap-dgx038:~/cpp$ ./a.out | sort
No. of processors 1 in place id 0
No. of threads 5, No. of places 28
thread id 0, place id 0, cpu id 42
thread id 1, place id 1, cpu id 154
thread id 2, place id 2, cpu id 43
thread id 3, place id 3, cpu id 155
thread id 4, place id 4, cpu id 44
malikm@a2ap-dgx038:~/cpp$
```

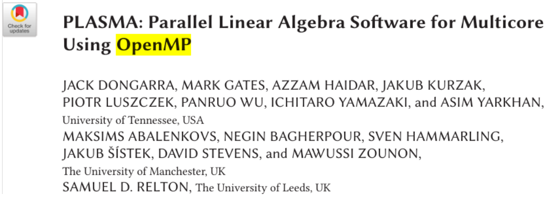
Sample Application Using OpenMP: Eigen and PLASMA

Eigen package uses OpenMP for matrix multiplications. Following is a snippet from "Eigen/src/Core/products/Parallelizer.h".



```
#if defined(EIGEN_HAS_OPENMP)
#pragma omp parallel num_threads(threads)
  {
    Index i = omp_get_thread_num();
    // Note that the actual number of threads might be lower than the number of
    // requested ones
    Index actual_threads = omp_get_num_threads();
```

Another linear-algebra package that uses OpenMP is PLASMA:



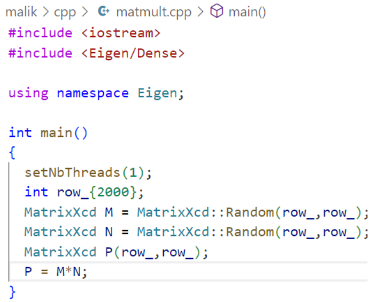
PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP

JACK DONGARRA, MARK GATES, AZZAM HAIDAR, JAKUB KURZAK, PIOTR LUSZCZEK, PANRUO WU, ICHITARO YAMAZAKI, and ASIM YARKHAN,
University of Tennessee, USA
MAKSIMS ABALENKOVS, NEGIN BAGHERPOUR, SVEN HAMMARLING, JAKUB ŠÍSTEK, DAVID STEVENS, and MAWUSSI ZOUNON,
The University of Manchester, UK
SAMUEL D. RELTON, The University of Leeds, UK

Scalability of thread parallelization using Eigen package

Matrix multiplication using Eigen:

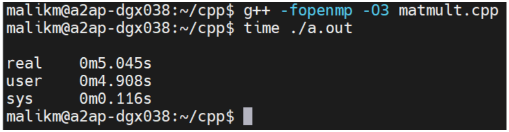
Data type: complex double



```
malik > cpp > matmult.cpp > main()
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;

int main()
{
   setNbThreads(1);
   int row_{2000};
   MatrixXcd M = MatrixXcd::Random(row_,row_);
   MatrixXcd N = MatrixXcd::Random(row_,row_);
   MatrixXcd P(row_,row_);
   P = M*N;
}
```

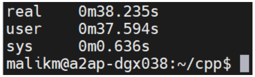
No. of flops. $N_{\mathrm{flops}} \sim O(n^3)$ for multiplying two $n \times n$ matrices. In general, the time, $t \sim N_{\mathrm{flops}}$.

Case 1: $n = 2000,\ N_{\mathrm{threads}} = 1$



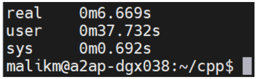
```
malikm@a2ap-dgx038:~/cpp$ g++ -fopenmp -O3 matmult.cpp
malikm@a2ap-dgx038:~/cpp$ time ./a.out

real    0m5.045s
user    0m4.908s
sys     0m0.116s
malikm@a2ap-dgx038:~/cpp$
```

Case 2: $n = 4000,\ N_{\mathrm{threads}} = 1$



```
real    0m38.235s
user    0m37.594s
sys     0m0.636s
malikm@a2ap-dgx038:~/cpp$
```

Case 3: $n = 4000,\ N_{\mathrm{threads}} = 8$



```
real    0m6.669s
user    0m37.732s
sys     0m0.692s
malikm@a2ap-dgx038:~/cpp$
```

When n is doubled, time $t \sim 2^3$, since $N_{\mathrm{flops}} \sim O(n^3)$. But when N_{threads} is multiplied by 2^3 in case 3, the time does not reduce to that of Case 1. This shows the **sub-linear scaling**.

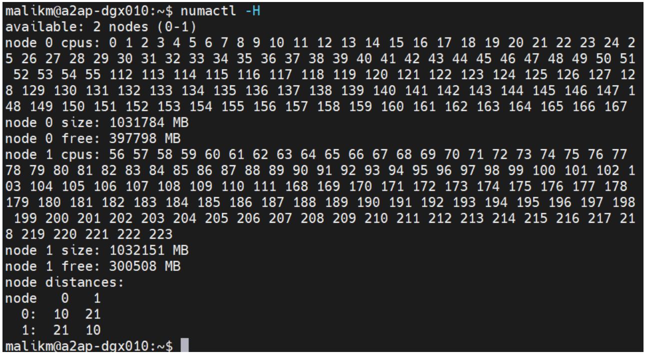
OpenMP: Closing Remarks

- Race-condition and conflicting memory updates are common issues while using threads.
- ASPIRE 2A+ compute nodes have 112 physical cores each with 2 hardware threads. Each node also have 2 sockets. These should be considered when declaring OMP_PLACES.
- Binding threads that access data located closer in memory enhances performance.
- compiler optimization flags "-O3" and "-O2" can improve the thread-parallelized codes better than MPI-parallelized codes, since former mostly uses threads on loops, which are highly susceptible for optimization by these flags.

OpenMP: Closing Remarks (Continued · · ·)

- if the data of two threads are different objects altogether, best to spread them out to enhance the performance. In this spreading to far enough cores will make use of NUMA feature. (ASPIRE 2A+ has two NUMA regions in each compute nodes, one for each socket.)

Cores' NUMA bindings:



```
malikm@a2ap-dgx010:~$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 2
5 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
 52 53 54 55 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 12
8 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 1
48 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167
node 0 size: 1031784 MB
node 0 free: 397798 MB
node 1 cpus: 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 1
03 104 105 106 107 108 109 110 111 168 169 170 171 172 173 174 175 176 177 178
179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198
 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 21
8 219 220 221 222 223
node 1 size: 1032151 MB
node 1 free: 300508 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
malikm@a2ap-dgx010:~$
```

MPI: Introduction and Processor Bindings

Introduction

Features

- Parallelization using multiple processes (like separate applications) instead of process-id-sharing threads: Processes can be on different nodes
- Commonly used in distributed-memory clusters, though it works with shared memory as well: Variables are exclusively accessible by their respective processes; Sharing is only possible by message passing.
- Uses libraries, not directives: Example implementations: Open MPI, MPICH, MVAPICH, Cray MPICH, etc.
- ASPIRE 2A+ uses Open MPI (default version 4.1.2. Alternative: 5.0.5)

Drawbacks

- Extensive code modification on a serial code, if decided to parallelize using MPI.
- In shared-memory machines, communications increase overhead when compared to OpenMP.

MPI Communication

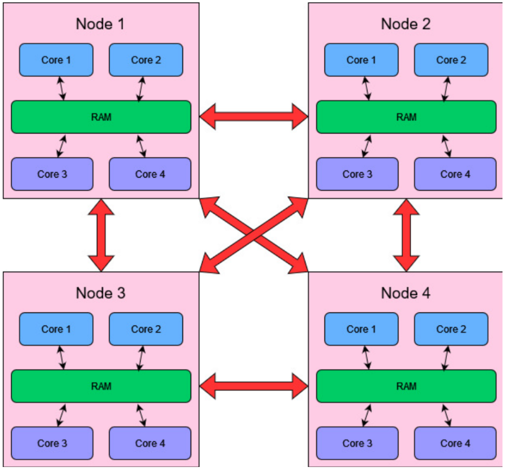


Image credit: https://github-pages.ucl.ac.uk/research-computing-with-cpp

- MPI communications (red arrows) move copies of data between nodes via high-speed interconnect.
- The interconnect is **Infiniband** in ASPIRE 2A+.
- Three types:
  1. Point to point,
  2. Collective, and
  3. One-sided (through RMA window)

Point-to-Point and Collective Communications

Point-to-Point:

- Uses send/receive API calls.
- Destination and Source ranks are mentioned
- Each call is communication between only a pair of processes
- It can be a blocking or non-blocking communication.

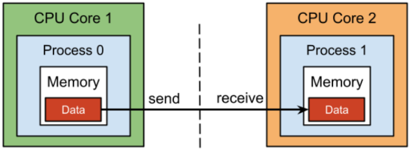


Image credit: https://hpc.nmsu.edu/discovery/mpi/introduction/

Collective Communication:

- Synchronization by a barrier
- Broadcasting, gathering and scattering operations.
- Reduction across the processes via a reduction operation.

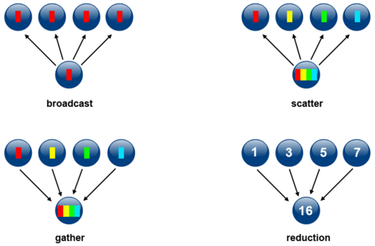


Image credit:

https://hpc-tutorials.llnl.gov/mpi/collective_communication_routines/

Defined Datatypes for Sending Sections of Arrays

- Sending non-contiguous parts of memory will need loops with primitive datatypes.
- This will result in communication overhead hitting the performance.
- Solution: Group non-contiguous parts of memory by "**Defined Datatypes**".

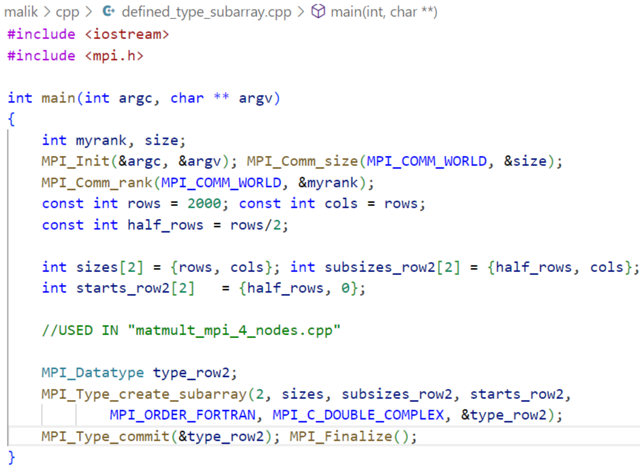
For example, the defined-datatypes can be used to define the each half of a matrix:

$$\mathbf{M} = \left(\begin{array}{ccc} M_{1,1} & \cdots & M_{1,n} \\ \vdots & \ddots & \vdots \\ M_{(n/2),1} & \cdots & M_{(n/2),n} \\ \hline M_{(n/2+1),1} & \cdots & M_{(n/2+1),n} \\ \vdots & \ddots & \vdots \\ M_{n,1} & \cdots & M_{n,n} \end{array}\right)$$

This type of "defined-types" are used in the code "**matmult_mpi_4_nodes.cpp**" and in "**matmult_mpi_16_nodes.cpp**". Relevant portions are shown as a snippet in the next slide.

Note: Each half shown here are contiguous in memory in "row major" storage convention (a default in C/C++), but not in column-major storage convention (which is default in Fortran and in a C++ package, **Eigen**).

Defined Datatypes for a Sub-array: Example



malik > cpp > defined_type_subarray.cpp > main(int, char **)

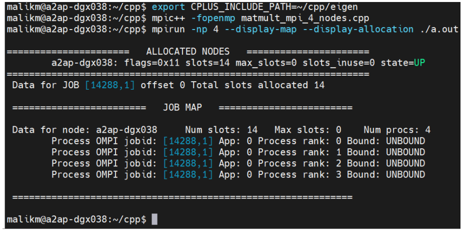
```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char ** argv)
{
    int myrank, size;
    MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    const int rows = 2000; const int cols = rows;
    const int half_rows = rows/2;

    int sizes[2] = {rows, cols}; int subsizes_row2[2] = {half_rows, cols};
    int starts_row2[2]   = {half_rows, 0};

    //USED IN "matmult_mpi_4_nodes.cpp"

    MPI_Datatype type_row2;
    MPI_Type_create_subarray(2, sizes, subsizes_row2, starts_row2,
            MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_row2);
    MPI_Type_commit(&type_row2); MPI_Finalize();
}
```

This example uses column-major storage convention.

See the codes "**mat-mult_mpi_4_nodes.cpp**" and in "**mat-mult_mpi_16_nodes.cpp**" for how they are used.

MPI Process: Slots and Processing Elements(PE)

- Slot: Unit of allocation of resources.
  - Default: cores. If "-use-hwthread-cpus" flag used: hardware threads
  - If "-np" is omitted, the number process will be same as the number of MPI processes mentioned in the job script.
  - The No. of slots on a node can be greater or less than the number of cores. Determined by scheduler by "mpiprocs" value in the PBS job script.
  - slot information is given by - -display-map or - -display-allocation flags.



```
malikm@a2ap-dgx038:~/cpp$ export CPLUS_INCLUDE_PATH=~/cpp/eigen
malikm@a2ap-dgx038:~/cpp$ mpic++ -fopenmp matmult_mpi_4_nodes.cpp
malikm@a2ap-dgx038:~/cpp$ mpirun -np 4 --display-map --display-allocation ./a.out

======================   ALLOCATED NODES   ======================
        a2ap-dgx038: flags=0x11 slots=14 max_slots=0 slots_inuse=0 state=UP
=================================================================
 Data for JOB [14288,1] offset 0 Total slots allocated 14

 ========================   JOB MAP   ========================

 Data for node: a2ap-dgx038     Num slots: 14    Max slots: 0    Num procs: 4
        Process OMPI jobid: [14288,1] App: 0 Process rank: 0 Bound: UNBOUND
        Process OMPI jobid: [14288,1] App: 0 Process rank: 1 Bound: UNBOUND
        Process OMPI jobid: [14288,1] App: 0 Process rank: 2 Bound: UNBOUND
        Process OMPI jobid: [14288,1] App: 0 Process rank: 3 Bound: UNBOUND

 =============================================================

malikm@a2ap-dgx038:~/cpp$ 
```

- Process Element (PE):
  - Default: core
  - If "-use-hwthread-cpus" flag used: hardware threads

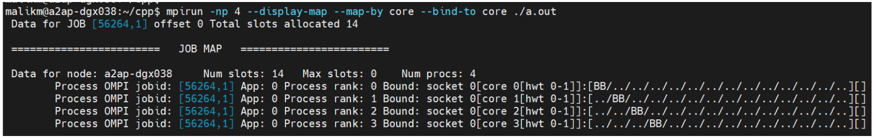
MPI Process Binding Using - -map-by and - -bind-to

- - -map-by The category (core, hwthread, or node, socket, etc) by which the location is changed when the process rank is changed.
- - -map-by core, for example, changes by core-id when process rank is changed by one.
- - -map-by hwthread changes by hardware thread when the process rank is changed.
- - -bind-to Instructs which collection of hardware need to be allotted for each process rank.
- - -bind-to core tells the process rank to use all hardware threads of the core where it is located.
- - -bind-to hwthread tells the process rank to use only one hwthread at the location it has been put by the "- -map-by" flag.

The example usage is, mpirun -np 4 –display-map –map-by core –bind-to core ./a.out

MPI Process Binding Using - -map-by and - -bind-to (continued · · ·)

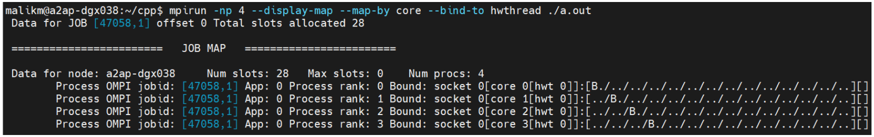
- mpirun -np 4 –display-map –map-by core –bind-to core ./a.out.



```
malikm@a2ap-dgx038:~/cpp$ mpirun -np 4 --display-map --map-by core --bind-to core ./a.out
 Data for JOB [56264,1] offset 0 Total slots allocated 14

 ========================   JOB MAP   ========================

 Data for node: a2ap-dgx038     Num slots: 14   Max slots: 0    Num procs: 4
        Process OMPI jobid: [56264,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0-1]]:[BB/../../../../../../../../../../../../..][]
        Process OMPI jobid: [56264,1] App: 0 Process rank: 1 Bound: socket 0[core 1[hwt 0-1]]:[../BB/../../../../../../../../../../../..][]
        Process OMPI jobid: [56264,1] App: 0 Process rank: 2 Bound: socket 0[core 2[hwt 0-1]]:[../../BB/../../../../../../../../../../..][]
        Process OMPI jobid: [56264,1] App: 0 Process rank: 3 Bound: socket 0[core 3[hwt 0-1]]:[../../../BB/../../../../../../../../../..][]
```

In this screenshot, the letter "B" signifies the "Bound" region of hardware threads for each rank.

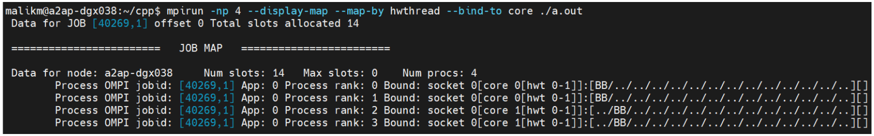
- mpirun -np 4 –display-map –map-by core –bind-to hwthread ./a.out.



```
malikm@a2ap-dgx038:~/cpp$ mpirun -np 4 --display-map --map-by core --bind-to hwthread ./a.out
 Data for JOB [47058,1] offset 0 Total slots allocated 28

 ========================   JOB MAP   ========================

 Data for node: a2ap-dgx038     Num slots: 28   Max slots: 0    Num procs: 4
        Process OMPI jobid: [47058,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0]]:[B./../../../../../../../../../../../../..][]
        Process OMPI jobid: [47058,1] App: 0 Process rank: 1 Bound: socket 0[core 1[hwt 0]]:[../B./../../../../../../../../../../../..][]
        Process OMPI jobid: [47058,1] App: 0 Process rank: 2 Bound: socket 0[core 2[hwt 0]]:[../../B./../../../../../../../../../../..][]
        Process OMPI jobid: [47058,1] App: 0 Process rank: 3 Bound: socket 0[core 3[hwt 0]]:[../../../B./../../../../../../../../../..][]
```

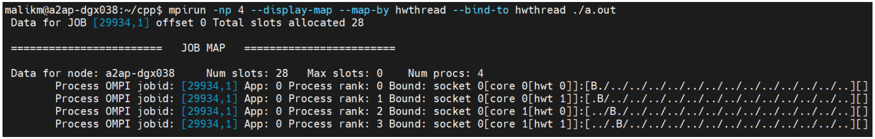
MPI Process Binding Using - -map-by and - -bind-to (continued · · ·)

- mpirun -np 4 –display-map –map-by hwthread –bind-to core ./a.out.



```
malikm@a2ap-dgx038:~/cpp$ mpirun -np 4 --display-map --map-by hwthread --bind-to core ./a.out
 Data for JOB [40269,1] offset 0 Total slots allocated 14

 ========================   JOB MAP   ========================

 Data for node: a2ap-dgx038     Num slots: 14   Max slots: 0    Num procs: 4
        Process OMPI jobid: [40269,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0-1]]:[BB/../../../../../../../../../../../../..][]
        Process OMPI jobid: [40269,1] App: 0 Process rank: 1 Bound: socket 0[core 0[hwt 0-1]]:[BB/../../../../../../../../../../../../..][]
        Process OMPI jobid: [40269,1] App: 0 Process rank: 2 Bound: socket 0[core 1[hwt 0-1]]:[../BB/../../../../../../../../../../../..][]
        Process OMPI jobid: [40269,1] App: 0 Process rank: 3 Bound: socket 0[core 1[hwt 0-1]]:[../BB/../../../../../../../../../../../..][]
```

- mpirun -np 4 –display-map –map-by hwthread –bind-to hwthread ./a.out.



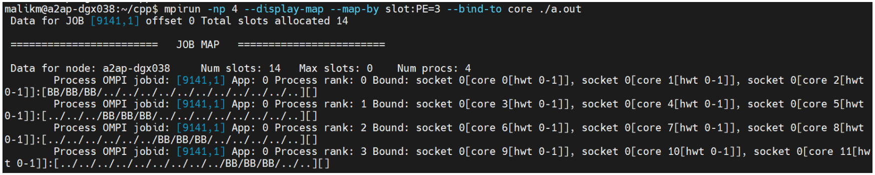
```
malikm@a2ap-dgx038:~/cpp$ mpirun -np 4 --display-map --map-by hwthread --bind-to hwthread ./a.out
 Data for JOB [29934,1] offset 0 Total slots allocated 28

 ========================   JOB MAP   ========================

 Data for node: a2ap-dgx038     Num slots: 28   Max slots: 0    Num procs: 4
        Process OMPI jobid: [29934,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0]]:[B./../../../../../../../../../../../../..][]
        Process OMPI jobid: [29934,1] App: 0 Process rank: 1 Bound: socket 0[core 0[hwt 1]]:[.B/../../../../../../../../../../../../..][]
        Process OMPI jobid: [29934,1] App: 0 Process rank: 2 Bound: socket 0[core 1[hwt 0]]:[../B./../../../../../../../../../../../..][]
        Process OMPI jobid: [29934,1] App: 0 Process rank: 3 Bound: socket 0[core 1[hwt 1]]:[../.B/../../../../../../../../../../../..][]
```

These options can also be performed with respect to NUMA nodes. However, in ASPIRE 2A+, Each node has only 2 NUMA nodes (opposed to 8 in ASPIRE 2A). Therefore, using "numa" as the parameter is in effect same as that of using "socket".

MPI Process Binding using - -map-by and - -bind-to (continued · · ·)

- mpirun -np 4 –display-map –map-by slot:PE=3 –bind-to core ./a.out.

  The program uses two threads for matrix multiplication. None of the above combinations allotted two physical cores for these threads. This will become possible only when each MPI process is given two
  physical cores. This is achieved by this combination of "map-by" and "bind-to"



```
malikm@a2ap-dgx038:~/cpp$ mpirun -np 4 --display-map --map-by slot:PE=3 --bind-to core ./a.out
 Data for JOB [9141,1] offset 0 Total slots allocated 14

 ========================   JOB MAP   ========================

 Data for node: a2ap-dgx038     Num slots: 14   Max slots: 0    Num procs: 4
        Process OMPI jobid: [9141,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt
0-1]]:[BB/BB/BB/../../../../../../../../../../..][]
        Process OMPI jobid: [9141,1] App: 0 Process rank: 1 Bound: socket 0[core 3[hwt 0-1]], socket 0[core 4[hwt 0-1]], socket 0[core 5[hwt
0-1]]:[../../../BB/BB/BB/../../../../../../../..][]
        Process OMPI jobid: [9141,1] App: 0 Process rank: 2 Bound: socket 0[core 6[hwt 0-1]], socket 0[core 7[hwt 0-1]], socket 0[core 8[hwt
0-1]]:[../../../../../../BB/BB/BB/../../../../..][]
        Process OMPI jobid: [9141,1] App: 0 Process rank: 3 Bound: socket 0[core 9[hwt 0-1]], socket 0[core 10[hwt 0-1]], socket 0[core 11[hw
t 0-1]]:[../../../../../../../../../BB/BB/BB/../..][]
```

- **exercise:** Compile **matmult_mpi_4_nodes.cpp** and **matmult_mpi_4_nodes.cpp** using mpic++ and run using mpirun with various combinations of "–map-by" and "–bind-to" options. Also try with "–cpu-list <comma separated list>" option.

MPI: Closing Remarks

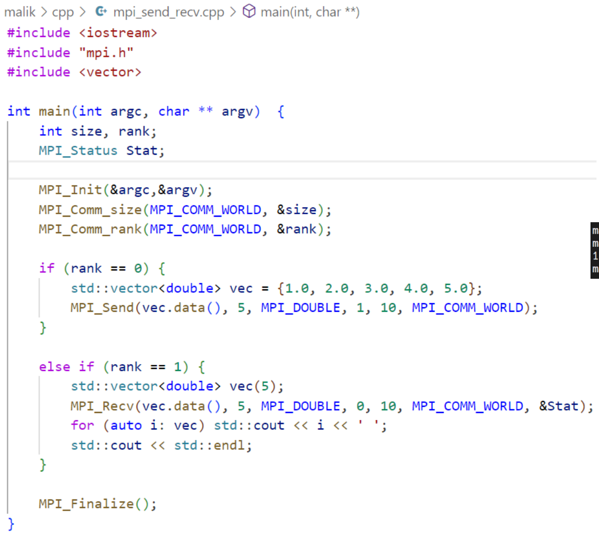
- Extra care is required with hybrid code that uses MPI and OpenMP. The MPI processes must be allocated at enough distance to make way for threads in each MPI process.
- Non-blocking communications must be used with care. Always call MPI_Wait() or similar functions before working on the received data.
- The commands
  - lstopo –pid $(pgrep <executable> | head -<number from 1 to n>), or
  - ps -eo pid,psr,comm | grep <executable>

  could show/list the CPU's being used. (The command "lstopo" would not work on compute nodes a present since the X11 forwarding is forbidden).

Thank You

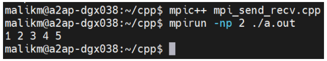
Appendix

Sample MPI Send & Receive Code



malik > cpp > mpi_send_recv.cpp > main(int, char **)

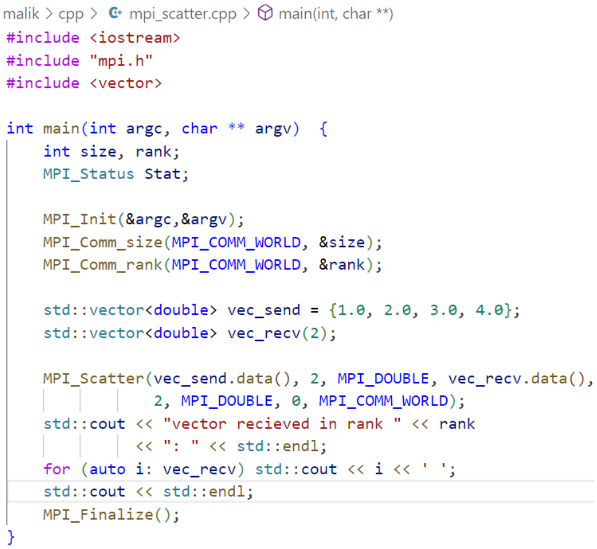
```cpp
#include <iostream>
#include "mpi.h"
#include <vector>

int main(int argc, char ** argv)  {
    int size, rank;
    MPI_Status Stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        std::vector<double> vec = {1.0, 2.0, 3.0, 4.0, 5.0};
        MPI_Send(vec.data(), 5, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD);
    }

    else if (rank == 1) {
        std::vector<double> vec(5);
        MPI_Recv(vec.data(), 5, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, &Stat);
        for (auto i: vec) std::cout << i << ' ';
        std::cout << std::endl;
    }

    MPI_Finalize();
}
```



```
malikm@a2ap-dgx038:~/cpp$ mpic++ mpi_send_recv.cpp
malikm@a2ap-dgx038:~/cpp$ mpirun -np 2 ./a.out
1 2 3 4 5
malikm@a2ap-dgx038:~/cpp$ 
```

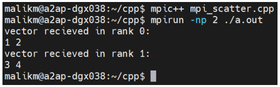
Compilation and Running

Sample Code for Collective Communication



```cpp
#include <iostream>
#include "mpi.h"
#include <vector>

int main(int argc, char ** argv)  {
    int size, rank;
    MPI_Status Stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    std::vector<double> vec_send = {1.0, 2.0, 3.0, 4.0};
    std::vector<double> vec_recv(2);

    MPI_Scatter(vec_send.data(), 2, MPI_DOUBLE, vec_recv.data(),
                  2, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::cout << "vector recieved in rank " << rank
                << ": " << std::endl;
    for (auto i: vec_recv) std::cout << i << ' ';
    std::cout << std::endl;
    MPI_Finalize();
}
```

In this code a vector of four elements is split into two halves and scattered to two nodes. Therefore, it needs to be run with 2 MPI processes as shown below.



```
malikm@a2ap-dgx038:~/cpp$ mpic++ mpi_scatter.cpp
malikm@a2ap-dgx038:~/cpp$ mpirun -np 2 ./a.out
vector recieved in rank 0:
1 2
vector recieved in rank 1:
3 4
malikm@a2ap-dgx038:~/cpp$
```

Compilation and Running

OpenMP: Important API Functions, Directives, Environment Variables

OMP_NUM_THREADS=10	Sets number of threads
omp_set_num_threads(10)	Sets number of threads; Supersedes $OMP_NUM_THREADS
omp_get_num_threads()	Gets the number of threads
omp_get_thread_num()	Gets the thread-id
omp_get_wtime()	Gets wall-time from an arbitrary reference instance. May vary between threads
#pragma omp parallel	The scope that follows will be executed on all threads
#pragma omp parallel private(var1, var2)	var1 and var2 becomes thread-private
#pragma omp parallel shared(var1, var2)	All threads shares "var1" and "var2"
#pragma omp parallel for	The for-loop that follows will be split ad executed on all threads

OpenMP: Important API Functions, Directives, Environment Variables

#pragma omp critical	Threads will execute the scope in a queue; appears inside a parallel scope
#pragma omp atomic	Threads will execute the following value update to a shared variable in a queue; appears inside a parallel scope
#pragma omp master	Execution only on thread-id=0
#pragma omp barrier	explicit barrier point in the code for all threads to arrive before proceeding to next statement
#pragma omp single	Execution by one thread only. Implicit barrier implied at the end of its scope
#pragma omp ordered	When mentioned inside a parallel region, the region marked by this directive will be run sequentially

OpenMP: Important API Functions, Directives, Environment Variables

#pragma omp parallel default(shared)	All variables in the threads are shared
#pragma omp sections	A non-iterative parallel section where sub-sections marked "section" are executed in each thread
#pragma omp section	Section that need to be executed by one thread. Placed inside the scope for "#pragma omp sections"
#pragma omp task	Task that need to be run by a thread in the parallel scope started earlier

- The "tasks" could be executed in any order and at anytime by the processor
- During iteration over array elements group the operations related to each object in one or fewer locations in the code if possible. **This will help avoid cache misses.**

Clauses for "**omp parallel for**"

nowait	Removes the implied barrier at the end of the "`for`" loop
schedule(static) nowait	Same as the above, but the iteration index of the "`for`" loops becomes global among threads
reduction(+var)	variable var at the end of each thread will be summed. (*, -, || and && are other allowed reduction operations)
private(var1, var2))	var1 and var2 are thread-private initialized randomly
firstprivate(var1, var2))	var1 and var2 are thread-private initialized with values in the master thread before forking
lastprivate(var1, var2))	var1 and var2 are thread-private with final value in the master thread equal to that of the last executed thread
collapse(2) private(i,j,k)	collapse the next two nested for-loops into a single loop and parallelize

Important MPI Setup and Query Functions

MPI_Init(&argc, &argv)	Initialize MPI
MPI_Finalize()	Call the destructor of MPI
MPI_COMM_WORLD	Default communicator with all ranks
MPI_Comm_rank(MPI_COMM_WORLD, &r)	Get the rank in variable "r"
MPI_Comm_size(MPI_COMM_WORLD, &n)	Get the no. of processes in variable "n"
MPI_initialized(&flag), MPI_finalized(&flag)	Check whether MPI initialized or finalized
MPI_Type_size(datatype, &size)	Get the size in bytes
MPI_Wtime()	Get the time from a reference time as double
MPI_Abort(MPI_COMM_WORLD, 1)	Abort with error code 1

Important MPI Point-to-Point Functions

MPI_Send(&buf, count, type, dest, tag, MPI_COMM_WORLD)	Send "count" number of data of type "type" to the rank, "dest" with tag, "tag".
MPI_Recv(&buf, count, type, source, tag, MPI_COMM_WORLD, &Stat)	Similar to the above, but from the rank "source"
MPI_Sendrecv(&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf, recvcount, recvtype, source, recvtag, MPI_COMM_WORLD, &status)	MPI_Send and MPI_Recv combined
MPI_Isend(&buf, count, type, dest, tag, MPI_COMM_WORLD, &request)	Nonblocking send; Does not wait for the re-usability of send-buffer
MPI_Irecv(&buf, count, type, source, tag, MPI_COMM_WORLD, &request)	Nonblocking receive; Does not wait for the receive to begin or complete
MPI_Wait(&request,&status)	Used in conjunction with non-blocking send or receive. This blocks until they complete.
MPI_Waitall(count, &array_of_requests, &array_of_statuses)	Same as above but used for many non-blocking send or receive requests.

MPI: Collective Communication Functions

MPI_Barrier (MPI_WORLD_COMM)	Block until all processes call this function.
MPI_Bcast (&buffer, count, type, root, MPI_WORLD_COMM)	Send to all processes.
MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, MPI_WORLD_COMM)	Divide the memory into parts and send to the processes one each.
MPI_Gather (&sendbuf, sendcnt, sendtype,&recvbuf, recvcount, recvtype, root, MPI_WORLD_COMM)	The opposite of MPI_Scatter mentioned. The "sendbuf" from each process are concatenated in the "root" process.
MPI_Allgather (&sendbuf,sendcount, sendtype, &recvbuf, recvcount, recvtype, MPI_WORLD_COMM)	Same as MPI_Gather, but the effect is as if the result is broadcast to all processes.
MPI_Reduce (&sendbuf, &recvbuf, count, type, op, root, MPI_WORLD_COMM)	Same as the MPI_Gather, but the resulting array elements are reduced by the reduction operation, "op".
MPI_Allreduce (&sendbuf, &recvbuf, count, type, op, MPI_WORLD_COMM)	Same as MPI_Reduce but the effect is as if its result is broadcast to all processes.

Matrix Multiplication by Domain Decomposition

Let us consider the matrix multiplication, $\mathbf{P} = \mathbf{MN}$, where $\mathbf{M}$ and $\mathbf{M}$ are $n \times n$ matrices such that n is a multiple of 4. Then $\mathbf{M}$ and $\mathbf{N}$ can be split into four row and four column matrices, respectively. Then, we can write $\mathbf{P}$ as:

$$\mathbf{P} = \left(\begin{array}{c|c|c|c} \mathbf{P}_{11} & \mathbf{P}_{11} & \mathbf{P}_{11} & \mathbf{P}_{11} \\ \hline \mathbf{P}_{12} & \mathbf{P}_{22} & \mathbf{P}_{32} & \mathbf{P}_{42} \\ \hline \mathbf{P}_{13} & \mathbf{P}_{23} & \mathbf{P}_{33} & \mathbf{P}_{43} \\ \hline \mathbf{P}_{14} & \mathbf{P}_{24} & \mathbf{P}_{34} & \mathbf{P}_{44} \end{array}\right) = \left(\begin{array}{c} \mathbf{M}_1 \\ \hline \mathbf{M}_2 \\ \hline \mathbf{M}_3 \\ \hline \mathbf{M}_4 \end{array}\right) \left(\begin{array}{c|c|c|c} \mathbf{N}_1 & \mathbf{N}_2 & \mathbf{N}_3 & \mathbf{N}_4 \end{array}\right),$$

where each of $\mathbf{P}_{ij}$ blocks with $i \in \{1 \cdots 4\}$ and $j \in \{1 \cdots 4\}$ is given by $\mathbf{P}_{ij} = \mathbf{M}_i \mathbf{N}_j$. ($\mathbf{P}_{ij}$, $\mathbf{M}_i$ and $\mathbf{N}_j$ are $(n/4) \times (n/4)$, $(n/4) \times n$ and $n \times (n/4)$ in sizes, respectively.)

Matrix Multiplication by Domain Decomposition (continues)

Remarks:

- Each of $\mathbf{P}_{ij}$ can be computed in an MPI process with rank, $r = 4(i-1) + (j-1)$.
- Therefore, it can be performed in 16 MPI processes.
- Defined datatypes need to be created for Each $\mathbf{M}_i$ and $\mathbf{N}_j$ and sent to the rank, $r = 4(i-1) + (j-1)$ from rank 0.
- After multiplication each of $\mathbf{P}_{ij}$ need to be received from rank $r = 4(i-1) + (j-1)$ into the rank 0. For this the defined datatypes for each of $\mathbf{P}_{ij}$ need to be created as well.

Exercises:

- Compile **matmult_mpi_4_nodes.cpp** using "mpic++" compiler and run via a job script with 4 MPI processes on a single node and note down the time of execution
- Repeat it with **matmult_mpi_16_nodes.cpp** on 16 nodes.
- Since these codes are hybrid, compile with "-fopenmp" and repeat.
- Analyze the execution times.