GWDG
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

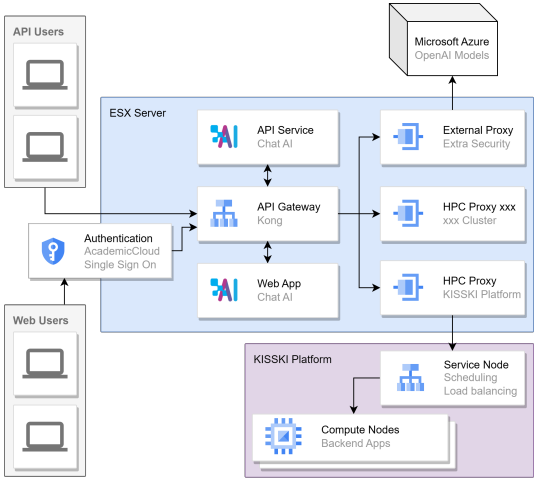
Parallel I/O for Machine-Learning Workflows

Mohammad Hossein Biniaz (mbiniaz@gwdg.de)
Dr. Kevin Lüdemann (kevin.luedemann@gwdg.de)

2024-05-07

Introduction

- HPC trainer / MLOps Engineer
- Scalable AI team in GWDG we do:
  - Serving / hosting LLMs that are HPC-native! vs (cloud)
  - No manageing ML environments / infrustructure!
  - Take care of our own pipelining / infrustructure
  - Finding bugs: dogfooding
  - We try to make such environment for others to use our service!



RAGs

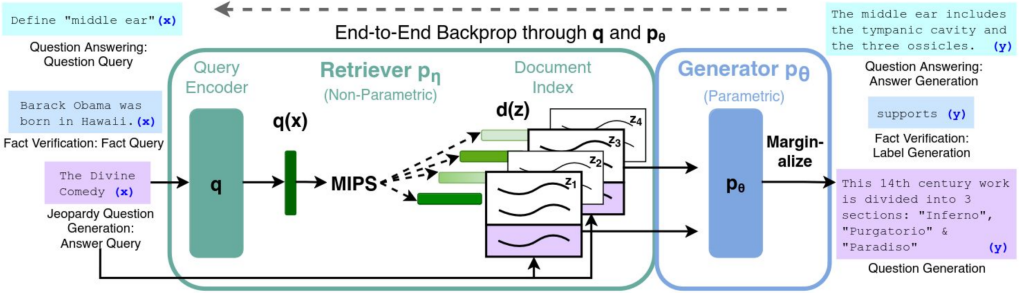
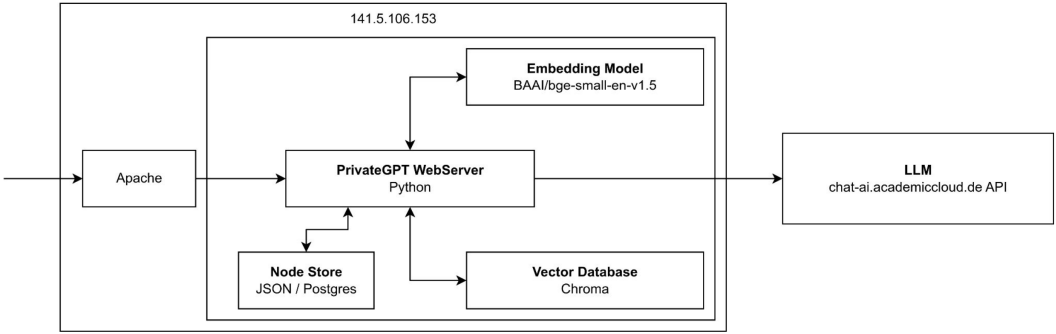


Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder* + *Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x, we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i. For final prediction y, we treat z as a latent variable and marginalize over seq2seq predictions given different documents.

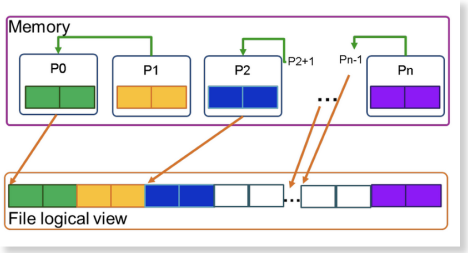
RAGs


141.5.106.153
Embedding Model
BAAI/bge-small-en-v1.5
Apache
PrivateGPT WebServer
Python
LLM
chat-ai.academiccloud.de API
Node Store
JSON / Postgres
Vector Database
Chroma

Motivations / Order of business

"*Lets look at parallel I/O as a part of DL parallelization*"

- Not simply just about ’parallel IO’? Read -> Write (In-place!? Better)
- Where do we parallelize? As we parallelize, how do we handle IO?
- Question: looking at I/O as a tool / segment of parallelziation in DL workflows.

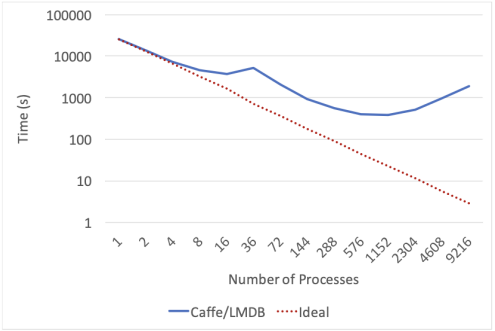


Motivations / Order of business

"*Lets look at parallel I/O as a part of DL parallelization*"

- DL requirements: deep layers are sequential. Backward gradient calculations as well: NOT parallel!
- Question: how would one think about parallelization in this case? (Given hardware requirements) i.o.w: how does one apply parallel IO to *this* problem?
- Parameter (=data) propagation/communication is IO (instead of in-place updates of a memory / array)
- "Data" ingestion is IO: how does i.e. pytorch / cafe *hand over* IO to OS? And why?
- Checkpoints instead of 'writes' in ML!
- What are traditional bottlenecks? Data transfer: Host -> GPU!

Caffe/LMDB Scalability Analysis



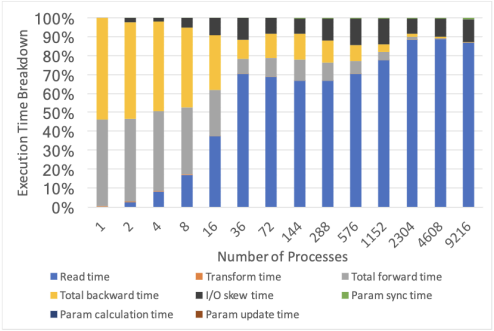


Fig. 3. Caffe/LMDB's strong scaling using CIFAR10-Large on Bebop: (a) total execution time; (b) execution time breakdown

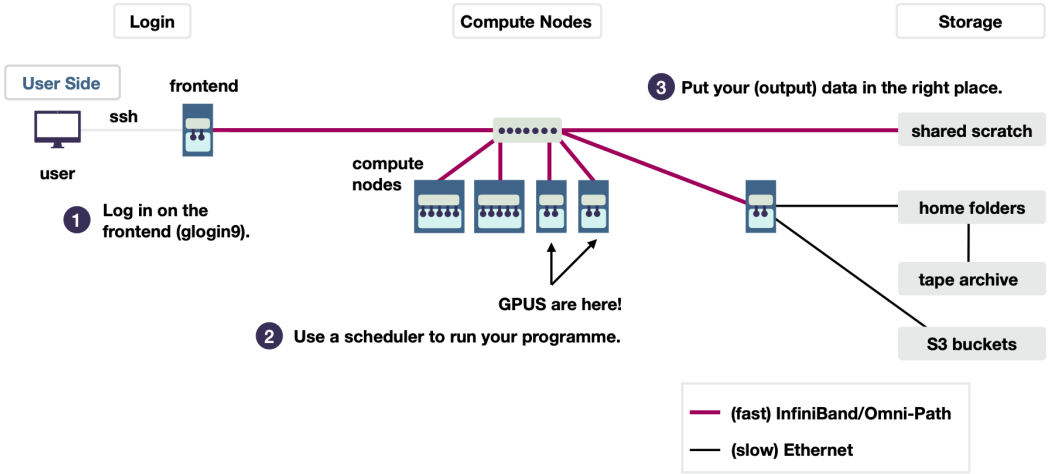
- Compute scales, I/O bottleneck 90%. "I/O skew time" grows with # of proc, raising concerns of load imbalance between the proc's (Alexnet, batch size 18,432 / 512 iter ~9mil)

Motivations / Order of business

"*Lets look at parallel I/O as a part of DL parallelization*"

- Question: how would one think about parallelization in this case? (Given hardware requirements) i.o.w: how does one apply parallel IO to *this* problem?
- Parameter (=data) propagation/communication is IO (instead of in-place updates of a memory / array)
- "Data" ingestion is IO: how does i.e. pytorch / cafe *hand over* IO to OS? And why?
- Checkpoints instead of 'writes' in ML!
- What are traditional bottlenecks? Data transfer: Host -> GPU!

GWDG Cluster



Motivations / Order of business

"*Analyzing the status quo*"

- Existing parallel deep learning libraries and existing frameworks.
- Discuss how they handle parallel I/O.
- Does the paradigm fully overcome the needs for it?
  - I.e. duplicating data? Non-shared memory pattern.
  - Using some sort of message passing to coordinate between the threads containing data?
- And how to manually do this in case we dont like those.

Problems with HDFS

- We can see this in newly proposed solutions
- traditionally; storage / computation paradigms
  - HDFS / mapReduce
  - Data moves to compute
  - Large internal traffic
  - commodity hardware: 3x replication / redundancy
- More money: VAST - storage grade storage over NVMe OF, etc.

HPC vs Cloud

- This presentation is about HPC, not cloud
- Most large supercomputers do not have a local disk on each node
- Thus, I/O typically is performed over the shared filesystem.
- Even if: on-node storage might be present in the form of nonvolatile storage:
  - Such storage is not persistent across the lifetime of the machine: typically wiped clean when a new job is assigned to a node.
  - Thus, data I/O still has to be performed from the global filesystem.
  - Systems that utilize on-node storage technologies in the form of burst buffers:
  - staging data on to these burst buffers requires prior knowledge as to which node would need what segment of the data. Such information is, not readily available in modern deep learning systems

Scalable Deep Neural Networks

- Deep learning optimization: runtime / algorithmic enhancements
- Enhanced hardware
  - Hardware includes processors (NVIDIA GPUs, Intel Xeon Phi, Google TPUs
  - High-speed networks (e.g., Mellanox InfiniBand, Intel OmniPath
  - Memory technologies
- Researchers: developed algorithms to make DL computationally efficient
- Via algorithmic parallelism without losing convergence accuracy.
- Data I/O: major bottleneck!
- LMDB, the most widely used I/O subsystem of deep learning frameworks

Scalable Deep Neural Networks

- Many parallel DL frameworks have been proposed - incorporate the trends
- Caffe, TensorFlow, Theano, Caffe2, PyTorch, Microsoft Cognitive Toolkit, Apache MXNet, Chainer.
- Why is it still such a hard problem to train NNs: "are they inherently hard to solve or is there more optimizations?"
- First principles:
  - For single pass of the data (large data / deep nets)
  - For training models where computation can be easily/efficiently parallelized or offloaded to hardware computational units
  - Moving the data becomes a bigger problem than the computation itself.

VAST storage solutions / paradigm

- We can see this in newly proposed solutions
- traditionally; storage / computation paradigms
  - HDFS / mapReduce
  - Data moves to compute
  - Large internal traffic
  - commodity hardware: 3x replication / redundancy
- More money: VAST - storage grade storage over NVMe OF, etc.

VAST

- Large-scale (tera/petabyte enclosure: order of million$) all-flash file/object-storage
- Traditionally: scale-out shared-nothing model (HDFS)
  - X86 servers: operate on own-media
  - ’Software"/network gives unified big-system overview
  - Big scale: internal traffic overhead is unbareable
- Replaced by: DASE (Disaggregated Shared-Everything)
  - No single point failure
  - 400Gb/s connection
  - Tech: NVMe OF
  - "Services" i.e. frontend runs on "stateless" docker containers
  - Frontend server request (NFS / S3 / SMB)
  - Service node refers to metadata in storage grade memroy -> finds in QLC flash -> retrieves over NVMe OF
  - Performance Evaluation of Object Storages (NHR2022) explain about gwdg!

VAST

- No reassembling data
- No node to node communication
- Metadata in storage class memory: direct access from front-end node
- Enable sharing data across nodes vs caching data
- Complexities of keeping cache coherent go away!

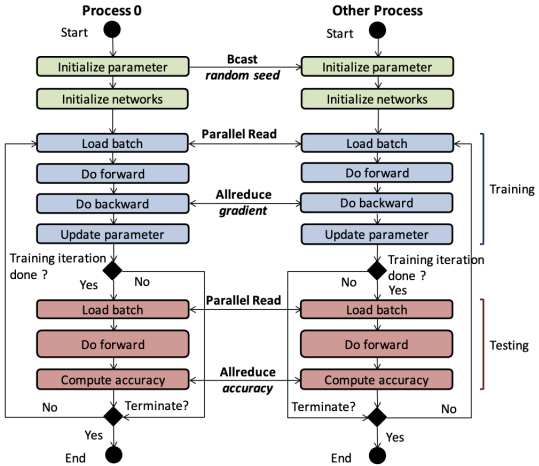
Back to IO bottleneck in DL

- "Moving data from a global filesystem (remember: just a view) for such processing can be a major bottleneck in the overall computation.
- Studies show that even with a small amount of parallelism in such deep learning systems, I/O accounts for a majority of the training time, thus degrading the overall system scalability
- As much as 90% of the execution time may be devoted to data I/O
- Fine-grade:
  - implicit I/O through mmap: relying on a B+-tree database structure for storing data
  - inefficient in I/O management in the context of parallel computing.
  - goal: saturate the system's available I/O bandwidth in deep learning frameworks. Currently just 10% LMDB vs LMDDIO cafee

Deep learning worfklow / Caffe

- Caffe is a well-known deep learning framework developed by Berkeley Vision, with CUDA support.
- Caffe follows the stochastic gradient descent approach to train DNNs.
- Goal of the training is to obtain a set of parameters for the DNN that most accurately represent a given dataset.
- Most training frameworks typically initialize the parameters to random values
- Although a growing number of researchers use better initial approximations of the parameters based on known properties of the input data.

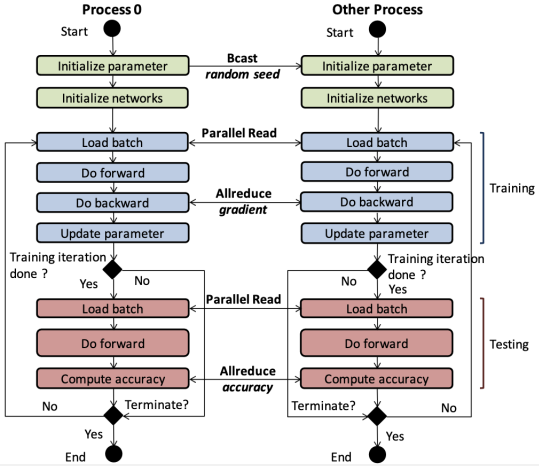
Caffe



Deep learning worfklow / Caffe

Caffe

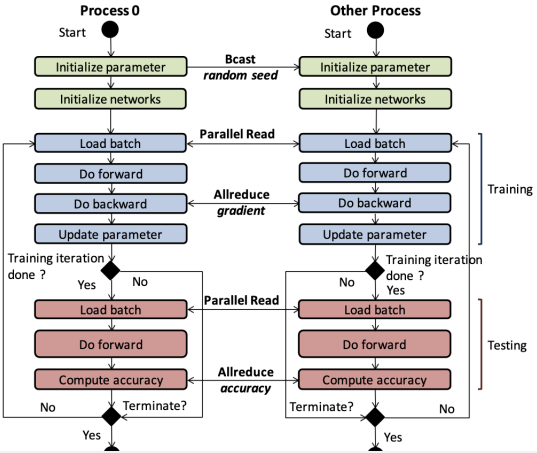
- Training is an iterative process that continues until the parameter set converges to the desired accuracy.
- In each training iteration, a subset of data samples, called a batch, in the database is drawn and used to train the network.
- Goal of the training is to obtain a set of parameters for the DNN that most accurately represent a given dataset.
- Caffe then measures the deviation error between the predicted value from the current DNN parameters and the actual value from the dataset.
- This error is then utilized to improve the



Deep learning worfklow / Caffe

Caffe

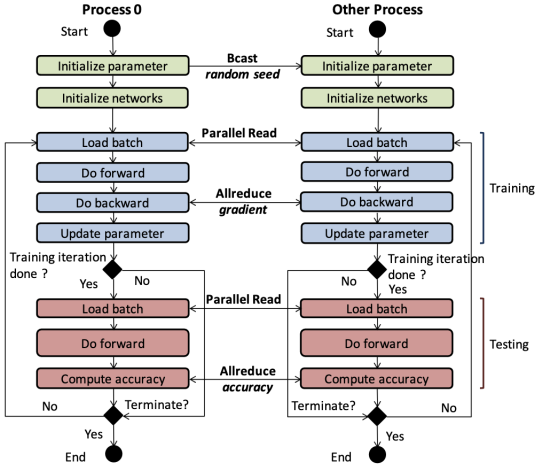
- This error is then utilized to improve the DNN parameters for the next training iteration.
- Once the training converges: final set of DNN parameters is used to generate a mathematical equation
- Can use model for highly accurate classification of new data samples.
- Key to generating - the use of a very large set of (high-quality) training data samples.
- Large organizations usually train DNNs with hundreds of terabytes or even petabytes of data.



Batch training

Caffe

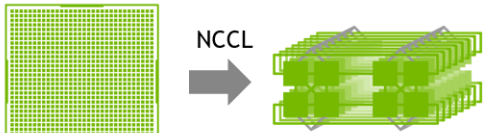
- Sequentially processing each data sample in the training dataset is not practical
- Most modern deep learning frameworks allow for what is called batch training.
- *Question one: "how can this be parallelized to multiple GPU?"*
- *Question two: "MPI"*



NVIDIA Collective Communications Library (NCCL)

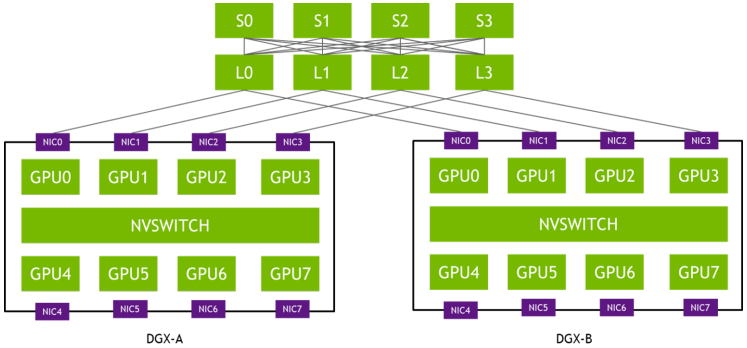
- NCCL implements multi-GPU / multi-node communication primitives optimized for NVIDIA GPUs / Networking
- Provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter point-to-point send and receive

- Over PCIe and NVLink high-speed interconnects within a node
- Over NVIDIA Mellanox Network across nodes
- Caffe2, Chainer, MxNet, PyTorch, TensorFlow have integrated NCCL for multi-GPU multi-node systems.
- NCCL is to GPU what MPI is to CPU



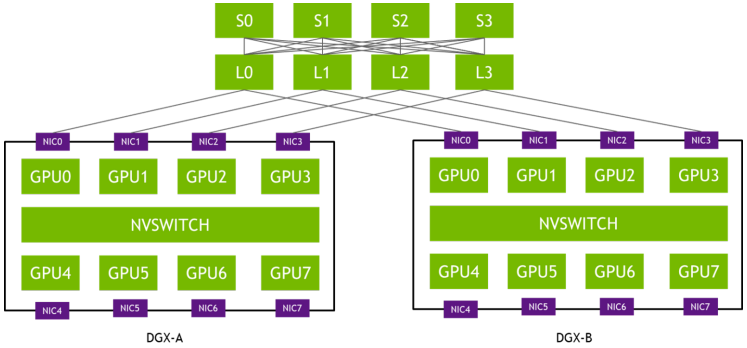
NCCL and I/O: PXN: PCI × NVLink

- New feature introduced in NCCL 2.12 called PXN.
- Enables a GPU to communicate with a NIC on the node through NVLink and then PCI
- This is instead of data going through the CPU using QPI or other inter-CPU protocols
- **Communication is still there!**
- Each GPU still tries to use its local NIC as much as possible, it can reach other NICs if required. topolgy (ring) one GPU close to each NIC



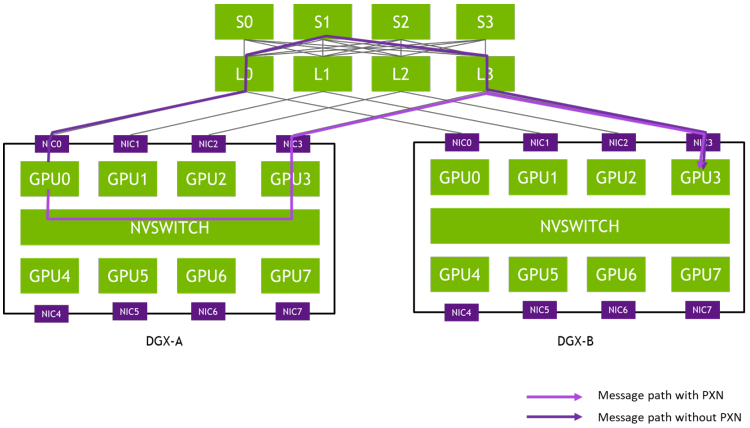
NCCL and I/O: PXN: PCI × NVLink

- Instead of preparing a buffer on its local memory for the local NIC to send, the GPU prepares a buffer on an intermediate GPU, **writing to it through NVLink**.
- It then notifies the CPU proxy managing that NIC that the data is ready, instead of notifying its own CPU proxy.
- GPU-CPU synchronization might be a little slower because it may have to cross CPU sockets



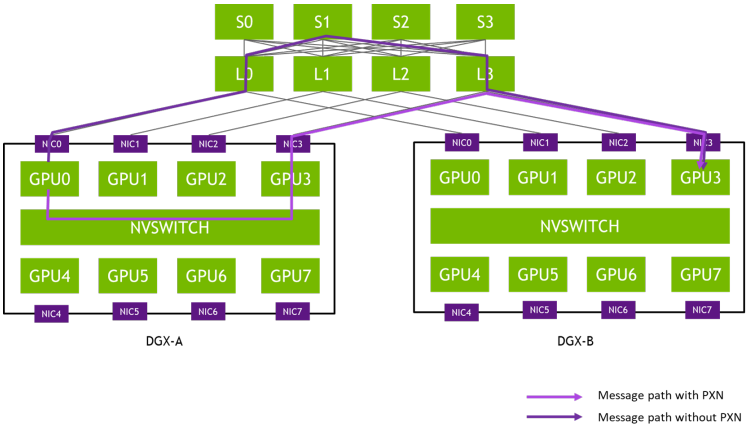
NCCL and I/O: PXN: PCI × NVLink

- But the data itself only uses NVLink and PCI switches: guaranteeing maximum bandwidth.
- Before NCCL 2.12: message would have traversed through three hops of network switches (L0, S1, and L3)
- Potentially causing contention / being slowed down by other traffic.
- Messages passed between the same pair of NICs are aggregated to maximize effective message rate and network bandwidth.



NCCL and I/O: PXN: PCI × NVLink

- But the data itself only uses NVLink and PCI switches: guaranteeing maximum bandwidth.
- Before NCCL 2.12: message would have traversed through three hops of network switches (L0, S1, and L3)
- Potentially causing contention / being slowed down by other traffic.
- Messages passed between the same pair of NICs are aggregated to maximize effective message rate and network bandwidth.



Rebase

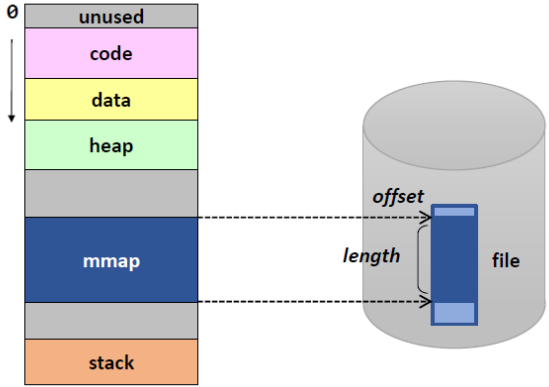
- Cafe: data-parallel model: the overall flow of the training is the same as that of sequential processing
  - Exception: data batch loading
  - forward / backwared pass: parallelized on mutiple processes/threads
  - Parallel network training: comes with additional communication cost where: network parameters must be synchronized across processes/threads.
- Storing and retrieving data samples: so far only Read
- Write is in the forms of:
  - parameters update / communication
  - checkpointing (not covered in this session)
- Widely used database options: LMDB
  - used by Cafe: production oritented VS pytorch: research!

Lightning Memory-Mapped Database Manager (LMDB)

- Database format that arranges its content based on a B+-tree and allows efficient simultaneous read and write access to the database
- LMDB refers to a library that provides the API to access and manipulate the LMDB database
- Makes use of the operating system (OS) memory-mapping mechanism, *mmap*, to enable in-memory database access.
- LMDB is not specific to deep learning
- LMDB data access model? How it lends itself to DL (or not!)
- Dynamic mapping mechanism of *mmap*

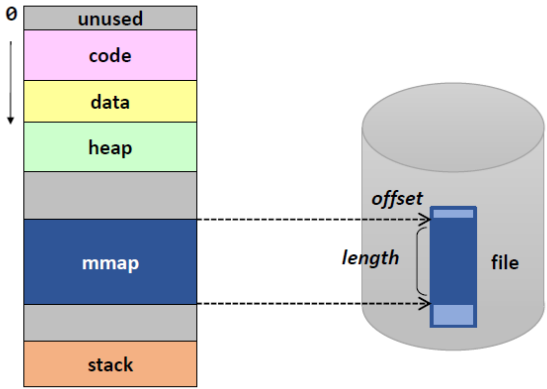
mmap (Memory Mapped File)

- Applications can treat the file data similar to how they treat regular main memory
- *mmap* system call asks OS to create new mapping in virtual address space of the calling process.
- Mapping: file system side resides on disk -> to memory space side
- CPU view: file as in-memory



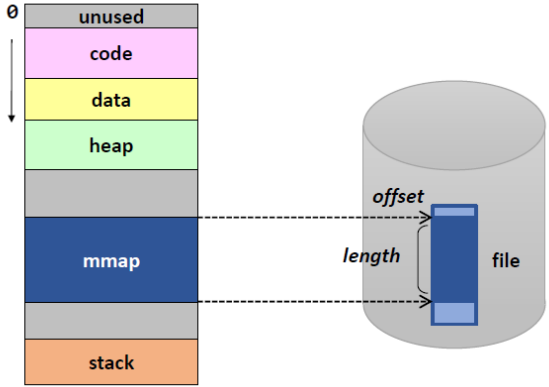
mmap (Memory Mapped File)

- Process perspective: this region of memory is just like any other, except that accessing it will read/write to corresponding file on disk, rather than system RAM.
- Can control visibility of mapping to ***other*** processes: also modifications (or not)
- Updates might not occur immediately depending on factors such as system page caching and the specific flags that were passed to mmap.



mmap (Memory Mapped File)

- Updates might not occur immediately (due to system page caching / specific flags passed to mmap)
- Can improve the performance, especially for large files, as it allows the **OS pageable, virtual memory system** (swap-system) to handle *caching* instead of application-level, read/write system calls
- => offloads I/O to OS!
- How does this help us? (or not?)



LMDB / mmap

- Data access is tracked by the OS at a page-level granularity: everything is 'in memory' (no out-of-memory error: swapped)
- Good bc data is not read until it is required
- mmap does not allow users to provide detailed information about their access pattern.
- fadvise / madvise : hints for simple manipulation of **access patterns**: sequential / random
- We want i.e. **strided access of batches of data** (complex access patterns)

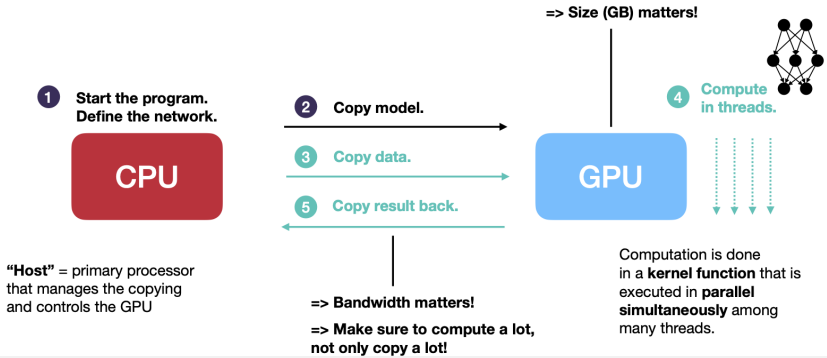
LMDB Cons

- LMDB Database Format: flattened B+-tree data structure as database layout.
- Use pages to represent nodes (i.e., branch and leaf nodes) (each node is stored on filesystem and is block-aware)
- Consists of four types of pages: metadata pages, branch pages, leaf pages, and overflow pages.
- Since LMDB's data format is a complex tree structure, correctly identifying a record requires a complete collection of pointers to all the branch pages in the path to the target data record

What is the biggest bottleneck for GPU training & inference?

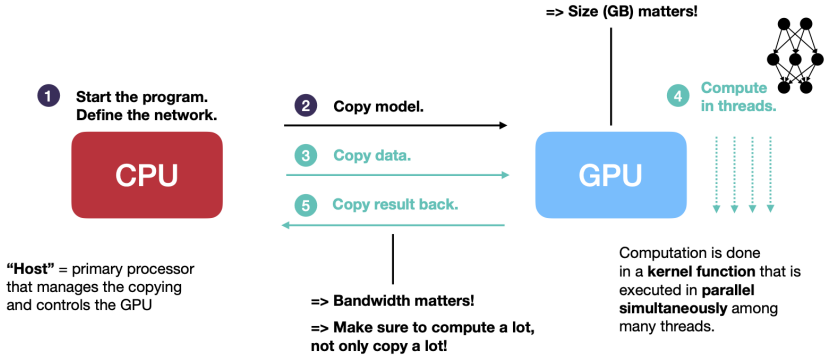
"*Traditionally, where is the bottleneck of GPU training procedure?*"

- Step 1: set batchsize: be careful how much your GPU can handle!
- Step 2: run nvitop to check GPU utilization



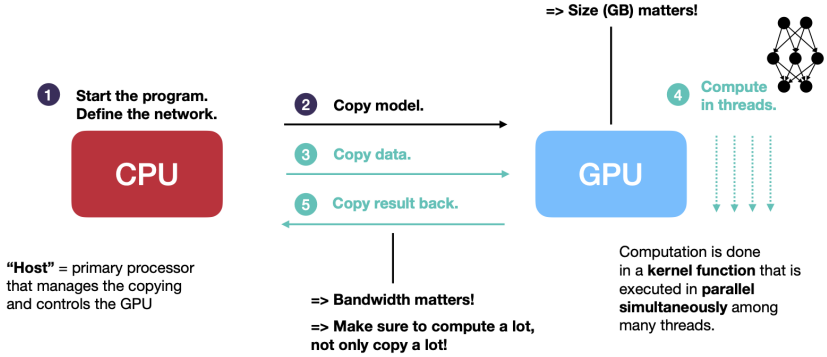
What is the biggest bottleneck for GPU training & inference?

- Observation: "buffering period for loading data from ’Host’ to ’GPU’"
- Step 3: GPU utilization is low? Increase batch size!



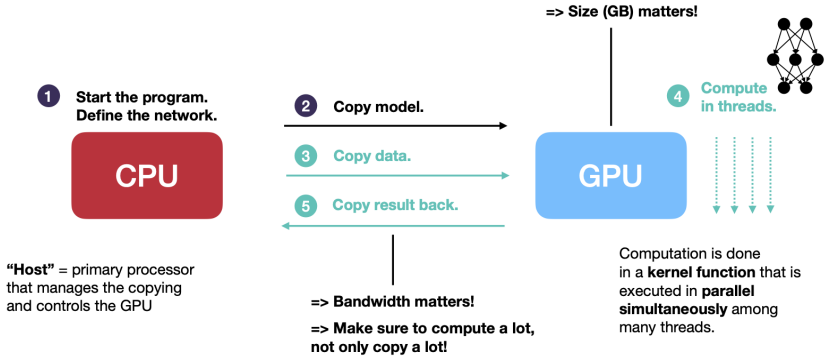
What is the biggest bottleneck for GPU training & inference?

- Step 4: Increase learning rate!
- Step 5: use GPU slicing -> Run nvitop again!



What is the biggest bottleneck for GPU training & inference?

How to avoid making copies to CPU en route GPU?

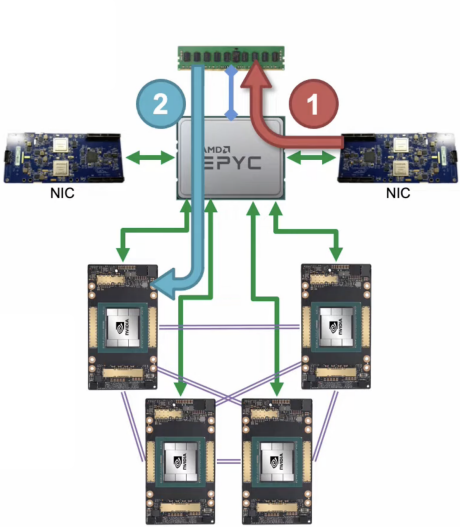


Copy from Host to Device and Vice Versa

```
// used for memory data transfer between the host (CPU) and the device (GPU), or vice versa.
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )

// dst: is a pointer to the destination memory.
// src: is a pointer to the source memory.
// count: is the size (in bytes) of the memory to be copied.
// kind: defines the direction of the copy. It can be one of the following:
    cudaMemcpyHostToHost: data copying happens from the host to the host.
    cudaMemcpyHostToDevice: data copying happens from the host to the device.
    cudaMemcpyDeviceToHost: data copying happens from the device to the host.
    cudaMemcpyDeviceToDevice: data copying happens from the device to the device.

int h_a[5], h_b[5]; // host arrays

int * d_a;          // device array

cudaMalloc((void **) &d_a, sizeof(int) * 5); // allocate device memory

// Copy from host (h_a) to device (d_a)
cudaMemcpy((void *) d_a, (void *) h_a, sizeof(int) * 5, cudaMemcpyHostToDevice);

// ... some operations ...

// Copy from device (d_a) to host (h_b)
cudaMemcpy((void *) h_b, (void *) d_a, sizeof(int) * 5, cudaMemcpyDeviceToHost);
```

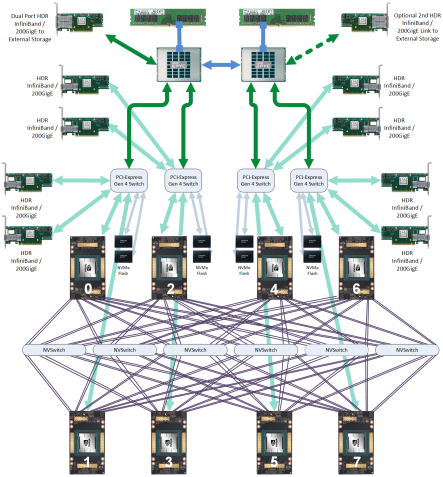
"*Where the process may be slowed down?*"

- If PCIe oversubscribed: we have NIC / GPU connected to PCIe switches
- NIC "competes" with GPU for bandwitch to CPU memory!
- If other memory-bound computation runs on node: IO competes with proc for CPU cycles / bandwidth

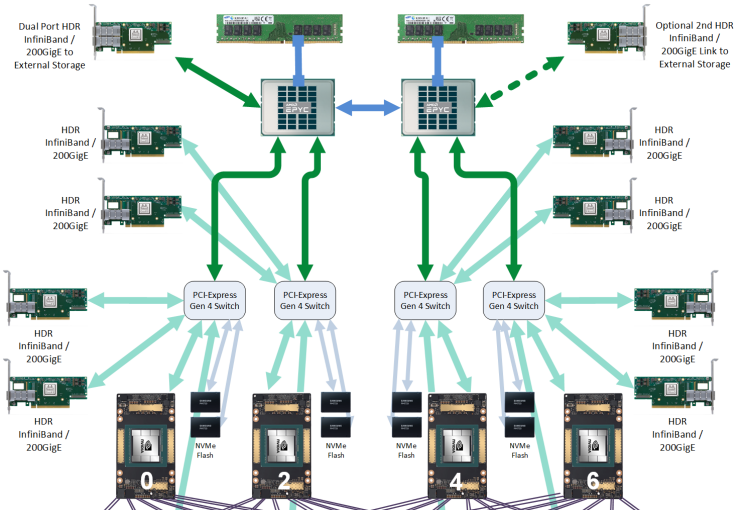


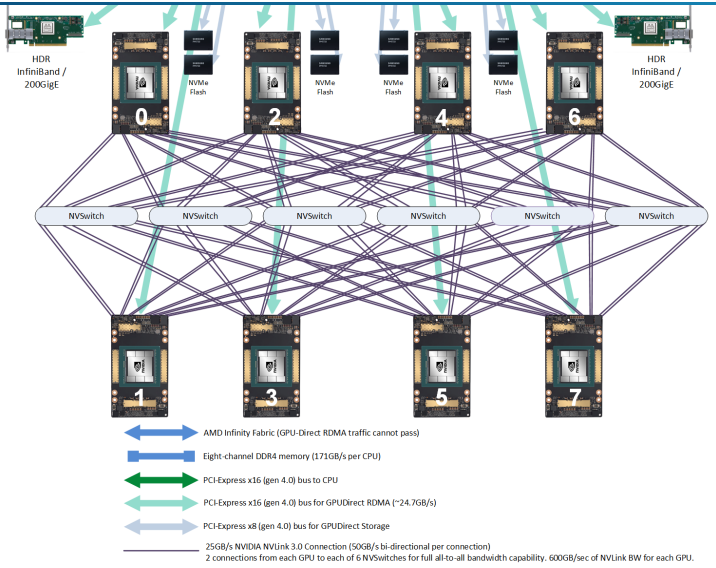
"*Where the process may be slowed down?*"

- Real situation: many more PCIe switches
- High bandwidth between NIC <=> GPUs
- Running everything through CPU: reduces bandwith by half



Server Block Diagram NVIDIA DGX A100




HDR
InfiniBand /
200GigE
NVMe
Flash
NVMe
Flash
NVMe
Flash
NVMe
Flash
HDR
InfiniBand /
200GigE
0
2
4
6
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVSwitch
NVSwitch
1
3
5
7
AMD Infinity Fabric (GPU-Direct RDMA traffic cannot pass)
Eight-channel DDR4 memory (171GB/s per CPU)
PCI-Express x16 (gen 4.0) bus to CPU
PCI-Express x16 (gen 4.0) bus for GPUDirect RDMA (~24.7GB/s)
PCI-Express x8 (gen 4.0) bus for GPUDirect Storage
25GB/s NVIDIA NVLink 3.0 Connection (50GB/s bi-directional per connection)
2 connections from each GPU to each of 6 NVSwitches for full all-to-all bandwidth capability. 600GB/sec of NVLink BW for each GPU.

Observation

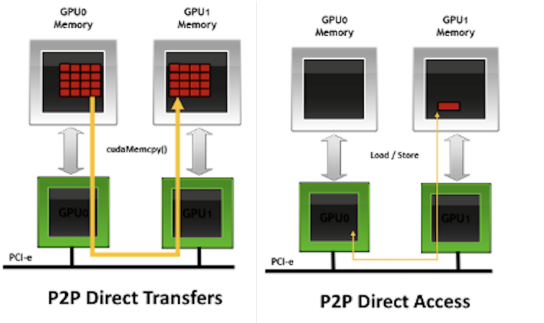
"*What do we learn?*"

- Huge bandwidtch between PCIe devices: NIC & GPU
- Buy running all storage IO through CPU, NIC => GPU bandwidth reduced to half!
- Solution: Nvidia Mellanox (mpi/NCCL): GPUDirect RDMA

Remote direct memory access (RDMA) allows NIC to send / recieve data directly into GPU memory without going through CPU memory! High-throughput, low-latency networking, which is especially useful in massively parallel computer clusters.

Extending P2P RDMA to storage IO

- Underlying technology making this possible is called P2P RDMA
- Instead of using read -> memcp, use cuFile API
- NIC / SSD talks directly to GPU:
- Read from other device GPU memoy visa direct access /transfer



cuFilRead

```
ssize_t cuFileRead(CuFileHandle_t *handle, void *buf, size_t count, off_t offset);
// handle is a pointer to an open file handle
// buf points to a buffer where the read content will be stored
// count is the number of bytes to read
// offset is the position in the file from which to start reading.
```

- No double copy: NIC / SSD => CPU, CPU => GPU
- No competition for CPU memory bandwidth => freeing up resources
- PCIe<=>CPU switch bandwidth independent
- Speed: PICe<=>GPU » PICe<=>GPU

cuFilRead

```
#include <cuda.h>
#include <cufio.h>
int main(){
  CuFileHandle_t fileHandle;
  cuFileHandle_t *pHandle = &fileHandle;
  char* fpath = "/tmp/sample.txt"; // path to your file
  CUfileDescr_t cfr_descr;
  char* gpu_buffer;
  size_t file_offset = 0;
  size_t bytes_to_read = 128; // or as required
  CUDA_CHECK(cudaMalloc((void**)&gpu_buffer, bytes_to_read));
  //Setting according to read for the file
  cfr_descr.type = CUFILE_HANDLE_TYPE_OPAQUE_FD;
  cfr_descr.handle.fd = open(fpath, O_RDONLY);
  CUfileError_t status = cuFileHandleRegister(pHandle, &cfr_descr);
  if (status.err == CU_FILE_SUCCESS) {
    ssize_t ret = cuFileRead(*pHandle, gpu_buffer, bytes_to_read, file_offset);
    if(ret < 0){
      printf("cuFileRead failed.\n");
    }
  }
  cuFileHandleDeregister(*pHandle);
  CUDA_CHECK(cudaFree(gpu_buffer));
  return 0;
}
```

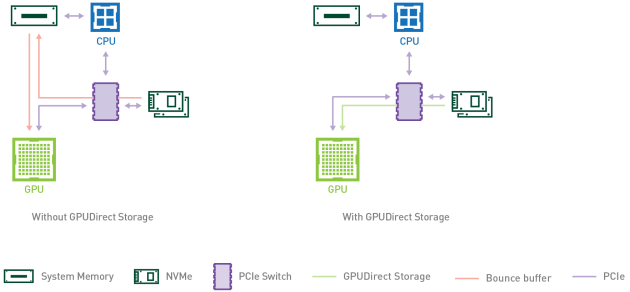
P2P mpi vs P2P File IO

"*GDS Support*"

- No free lunch: file system implemented on OS kernel unlike P2P mpi
- Cannot use OS features: i.e. paged caches
- Cannot use POSIX re-wr (instead use: cuFileRead / cuFileWrite)
  - POSIX API calls OS kernel
  - GDS (Nvidia) non-posix: cuFileRead/Write
  - Code changes: swap POSIX read/writes in code as cuFileRead/Write
- DIY or (like mpi does), use I/O middleware
  - HDF5 middleware already supports GDS
  - Already using HDF5: minimal code change
- Filesystem must support kernel bypass
  - Lustre, VAST, ...

What is storage I/O to GPU?

Nvidia GPUDirect Storage (GDS) or Nvidia Magnum IO provides a direct DMA path between GPU and PCIe attached storage via the `cuFile` API in a Nvidia ConnectX-4+ based fabric.



Source: https://developer.nvidia.com/blog/gpudirect-storage/

Examples for useable storage: local NVME drives, Lustre, BeeGFS, GPFS, WekaFS, VASTData, NetApp ONTAP, RDMA enabled NFS

ML Frameworks

- scikit-learn 

- PyTorch 

- Tensorflow 

- Horovod 

scikit-learn

- Open-source machine learning library for building and training models
- Supports various ML algorithms ranging from decision trees over K nearest neighbour and support vector machines to neural networks
- Only builtin support for CPUs
- Experimentally GPU supported for CuPy or PyTorch arrays
- Simpler compared to other frameworks
- No explicit usage of parallel-io

- Open-source machine learning library developed by Facebook
- Framework for building and training deep neural networks
- Primarily used for applications such as computer vision and natural language processing
- Support fo GPU, CPU and TPU.
- Strong ecosystem for easier optimization of hyperparameter tuning
- Computational graph support at runtime.
- Support for distributed training

Tensorflow

- Open-Source platform for deep learning applications developed by Google
- Support devices ranging from cell phones up to big compute clusters
- Utilizing CPUs, GPUs, TPUs and FPGAs
- Provides with Tensorboard a better data visualization tool for easy debugging
- Compatible with many languages like Python, C++, JavaScript
- Its support of TPUs enables increased training performance

Horovod

- Framework agnostic Supports TensorFlow, MXNet, Torch and Keras
- High Performance features
  NVIDIA’s NCCL allows direct GPU access which obviates the need to move data between GPU and CPU or sending it across the network
- Supports distributed components
  Supports a wide ranges of distributed computing frameworks such as Kubernetes, Spark, Ray, Docker, Singularity

Dataformats

- TFRecord
- Webdataset
- small files
- HDF5
- ADIOS2

TFRecord

- TensorFlow’s internal binary storage format
- uses Protobuf for serialization
- consists of a sequence of serialized protobuf records
- more efficient than text-based formats, especially for large datasets.
- optimized for large-scale training tasks in hpc enviroments.
- avoids anti-pattern of many small files

Webdataset

- implements standard PyTorch IterableDataset interface
- supported by PyTorch DataLoader
- provides efficient access to datasets stored in POSIX tar archives
- limited to sequential/streaming data access for improved performance for large-scale models
- optimized for large-scale training tasks in hpc enviroments.
- avoids anti-pattern of many small files

HDF5

- HDF5 == Hierarchical Data Format, v5
- Versatile data model consisting of groups and datatsets with a wide variety of metadata
- Open source software library running on a wide range of computational platforms, from cell phones to massively parallel systems
- Rich set of integrated performance features allowing access time and storage space optimizations
- Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection
- Completely portable file format with no limit on the number or size of data objects stored

HDF5

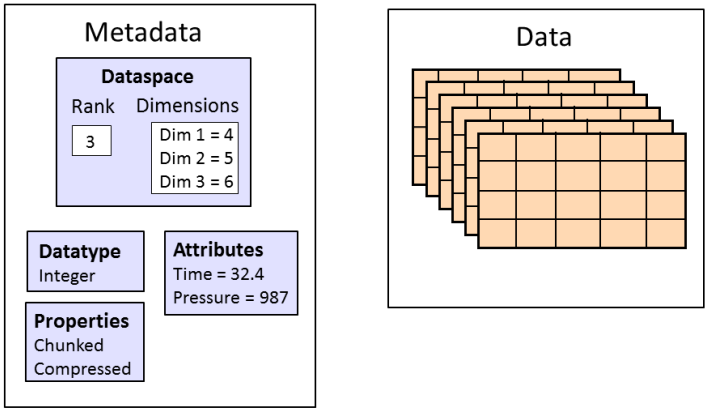


Figure: Dataset in HDF5 (Source: https://docs.hdfgroup.org/hdf5/v1_14/v1_14_4/_intro_h_d_f5.html)

Zarr

- Allows representation of N-dimensional arrays with any NumPy dtype.
- Supports chunk arrays along any dimension.
- Allows to compress and/or filter chunks using any NumCodecs codec.
- Supports to store arrays on many media, e.g. in memory, on disk, within compressed files, using S3
- Enables concurrently reading arrays from multiple threads or processes.
- Supports to write to arrays concurrently from multiple threads or processes.
- Permits to organize arrays into hierarchies via groups.

Small files

Handling small files in parallel I/O can pose several challenges is considered anti-pattern.

- Metadata Overhead: Reading metadata and conducting non-contiguous disk seeks for each file adds up.
- I/O Management: data distribution, cache management, and network overhead can cause low performance.
- Atomicity of Operation: Strong ordering of read and write operations between processes can impact performance.

Application I/O Types

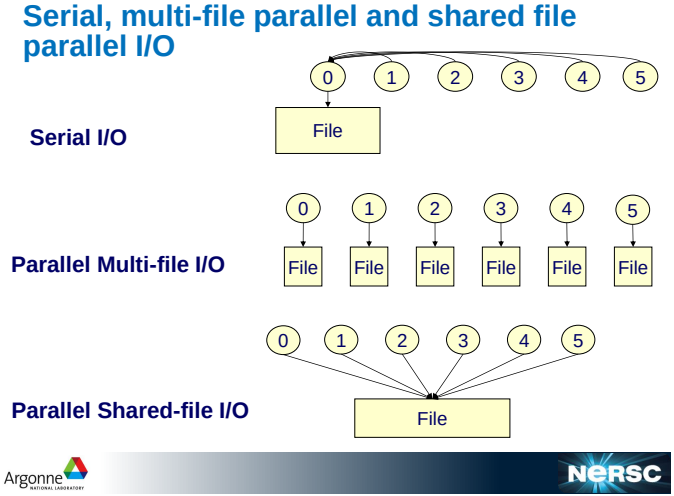


Figure: Source: Lonnie Crosby https://www.nics.utk.edu/sites/www.nics.tennessee.edu/files/pdf/Lonnie.pdf

Application I/O Access Patterns

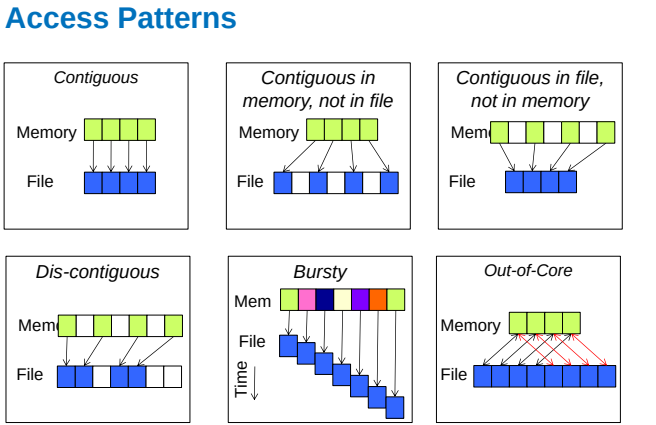


Figure: Source: Lonnie Crosby https://www.nics.utk.edu/sites/www.nics.tennessee.edu/files/pdf/Lonnie.pdf

File Striping: Distributing Data Across Devices

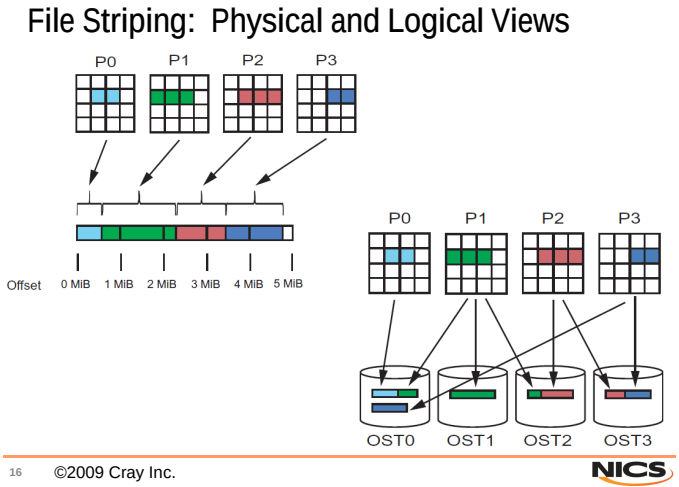


Figure: Source: Lonnie Crosby https://www.nics.utk.edu/sites/www.nics.tennessee.edu/files/pdf/Lonnie.pdf

APIs

- Middleware and low-level
  - pHDF5
  - MPI-IO
  - POSIX-IO
  - S3
- application oriented
  - Dask
  - Spark

pHDF5

- File format identical to previously discussed HDF5 format
- Parallel processed files shareable between different serial or parallel platforms
- Specially compiled libarary to support parallel data access
- Usually based on MPI-IO backend of HDF5
- Additionally direct access to parallel file systems supported, e.g. Lustre, IBM GPFS
- Sychronization needed between different processes for structural modification of metadata by collective operations
- Array data transfer both possible individual and collective

pHDF5 - Code Example

```
/* setup file access template with parallel IO access. */
acc_tpl1 = H5Pcreate (H5P_FILE_ACCESS);
/* set Parallel access with communicator */
H5Pset_fapl_mpio(acc_tpl1, comm, info);

/* create the file collectively */
fid1=H5Fcreate(filename,H5F_ACC_TRUNC,H5P_DEFAULT,acc_tpl1);

/* Release file-access template */
H5Pclose(acc_tpl1);

/*
 * Define the dimensions of the overall datasets and create the dataset
 */
/* setup dimensionality object */
sid1 = H5Screate_simple(SPACE1_RANK, dims1, NULL);

/* create a dataset collectively */
dataset1 = H5Dcreate2(fid1, DATASETNAME1, H5T_NATIVE_INT, sid1, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
```

pHDF5 - Code Example 2

```
/*
 * Set up dimensions of the slab this process accesses.
 */

/* Dataset1: each process takes a block of rows. */
slab_set(start, count, stride, BYROW);

/* create a file dataspace independently */
file_dataspace = H5Dget_space (dataset1);
ret=H5Sselect_hyperslab(file_dataspace, H5S_SELECT_SET, start, stride,
        count, NULL);

/* create a memory dataspace independently */
mem_dataspace = H5Screate_simple (SPACE1_RANK, count, NULL);

/* fill the local slab with some trivial data */
dataset_fill(start, count, stride, &data_array1[0][0]);

/* set up the collective transfer properties list */
xfer_plist = H5Pcreate (H5P_DATASET_XFER);
H5Pset_dxpl_mpio(xfer_plist, H5FD_MPIO_COLLECTIVE);

/* write data collectively */
H5Dwrite(dataset1, H5T_NATIVE_INT, mem_dataspace, file_dataspace, xfer_plist, data_array1);

/* release all temporary handles. */
```

Parallel-IO with pHDF5 in Action - Instructions

```
# Connect to cluster by ssh:
 ssh userid@glogin11.hpc.gwdg.de
# Load hdf5 module:
 module load gcc hdf5
# Copy sample code:
 cp /sw/dataformats/hdf5—parallel/ompi/gcc.9.2.0/1.12.0/skl/share/hdf5_examples/c/ph5example.c ~
# Change directory:
 cd
# Check source in text editor:
 nano ~/ph5example.c
# Compile code:
 gcc —lmpi —lhdf5 ph5example.c —o ph5
# Execute code:
 ./ph5
```

MPI-IO

- First introduced in MPI Version 2.0
- Standard backend for parallel IO in HDF5
- Used in many simulation applications
- possible performance improvements by MPI hints
  - Data Sieving
  - Two-Stage IO
  - Collective buffering
  - Direct IO
- Differences in supported file systems in the most common implementations, e.g. MPICH and OpenMPI

Parallel-IO with MPI-IO in Action - Instructions

```
ssh userid@glogin6.hpc.gwdg.de
module load openmpi
cp my_mpi-io-example.c ~
cd ~
nano ~/my_mpi-io-example.c
mpicc my_mpi-io-example.c -o mpiio
mpirun --mca orte_base_help_aggregate 0 -np 4 ./mpiio
```

MPI-IO - Code Example

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_File fh;
    MPI_Offset offset;
    MPI_Info info;
    MPI_Info_create(&info);
    MPI_Status status;

    /* Enable the collective buffering optimisation */
    MPI_Info_set(info , "romio_cb_write" , "enable") ;

    int data = rank * 10; // Example data to be written

    MPI_File_open(MPI_COMM_WORLD, "output.txt", MPI_MODE_CREATE | MPI_MODE_WRONLY, info, &fh);
    offset = rank * sizeof(int);
    MPI_File_write_at(fh, offset, &data, 1, MPI_INT, &status);
    MPI_File_close(&fh);

    MPI_Finalize();
    return 0;
}
```

POSIX-IO

- Standard IO-API in Unix-based operating systems
- Implementation as VFS in Linux-Kernel generally available in HPC-Systems
- Strong Consistency Semantics to achieve strict read after write consistency
- Requires locking to achive strong consistency and limits performance of parallel filesystems in HPC environments
- Designed with operations on one single node in mind

S3

- S3 wide spread API for accessing object storages
- Originally developed at Amazon for Amazon S3 service
- S3 Data Loaders used for loading data from object storages during model trainings
- FSSpecFileOpener and S3FileLoader for loading a single object implemnted in PyTorch
- S3FileLister for listing existing implemented in PyTorch
- Sharded data access recommended to avoid congestion of storage systm

S3

```python
from torch.utils.data import DataLoader
from torchdata.datapipes.iter import IterableWrapper

s3_shard_urls = IterableWrapper(["s3://bucket/prefix/",]).list_files_by_s3()
s3_shards = s3_shard_urls.load_files_by_s3()

# text data
training_data = s3_shards.readlines(return_path=False)
data_loader = DataLoader(
        training_data,
        batch_size=batch_size,
        num_workers=num_workers,
)
# training loop
for epoch in range(epochs):
        # training step
        for bach_data in data_loader:
            # forward pass, backward pass, model update
```

Dask

- Open-source Python library for parallel and distributed computing
- Compatible with NumPy, Pandas and Scikit-learn
- Two-stage I/O operations: Building the graph of operations and accessing data chunks.
- Optimized computations for efficiency.
- Avoids shuffling data between workers.
- Adjusts partition count based on column subset.
- Binds all tasks involved in I/O operations to workers providing the resource ’io’.

Dask - Code Example

```
import dask.dataframe as dd

# Load the dataset into a Dask DataFrame
# Types being casted. It is not lazy like as spark
df = dd.read_csv('IRAhandle_tweets_*.csv', dtype={'tco2_step1': 'object', 'tco3_step1': 'object'},  blocksize=100000)

ffiltered_df = df[df['language'] == "English"]

# Compute the result
rresult = ffiltered_df.compute()

# Print the result
rresult[:20]
```

Apache Spark

- Originally developped at University of California, later donated to Apache Software Foundation
- Open Source analytics engine for large-scale data processing
- Executable both on single-node machines or clusters
- Several programming languages supported, i.e. Python, Java, Scala, and R
- Interface with implicit data parallelism and fault tolerance

PySpark - Code Example

```
!wget https://github.com/fivethirtyeight/russian-troll-tweets/raw/master/IRAhandle_tweets_1.csv
!wget https://github.com/fivethirtyeight/russian-troll-tweets/raw/master/IRAhandle_tweets_2.csv

from pyspark.sql import SparkSession
import os

# Create a SparkSession
spark = SparkSession.builder.appName('large_dataset_processing').getOrCreate()

# Load the dataset into a DataFrame
df = spark.read.format('csv').option('header', 'true').load('IRAhandle_tweets_*.csv')

# Perform some filtering operations
filtered_df = df.filter(df['language'] == "English")  # Replace 'ColumnName' and 100 with your actual column name and condition

# Show the result
filtered_df.show()
```

Thank you!