Escape Parallel Rooms

Some Fun with Parallelization

03-07-2024 I

Think Parallel!

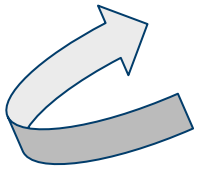
- **Process Parallelization**
- **MPI Parallelization**
- **CUDA Parallelization**
- **CUDA aware MPI Parallelization**
- **Data Parallel vs Distributed DP**
- **DDP Parallelization**

Parallel Evolution

Multiple-Threads

Multiple-Processes

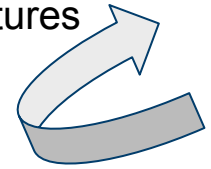
Message Passing Interface MPI

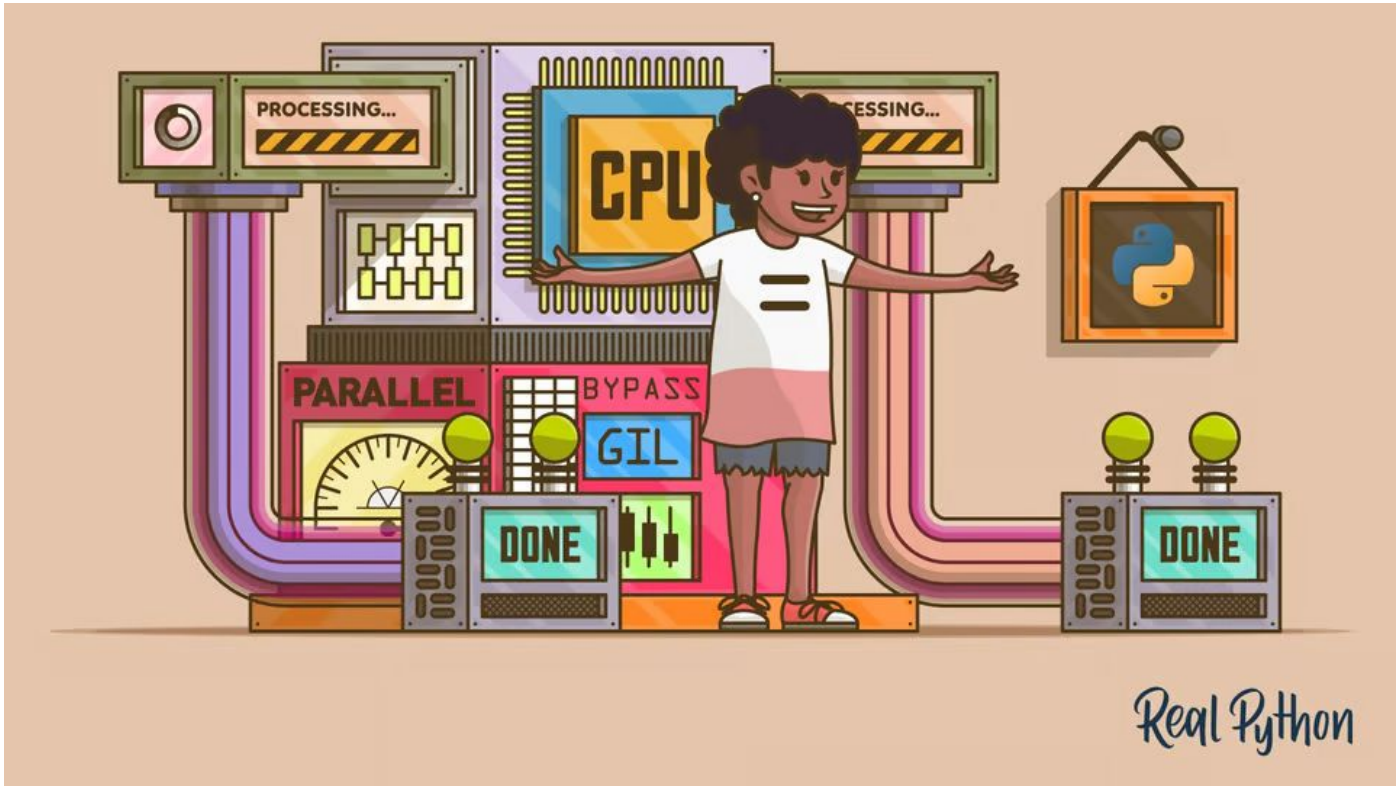


Multiple-Cores

Single Instruction Multiple Data (SIMD)

Architectures CUDA



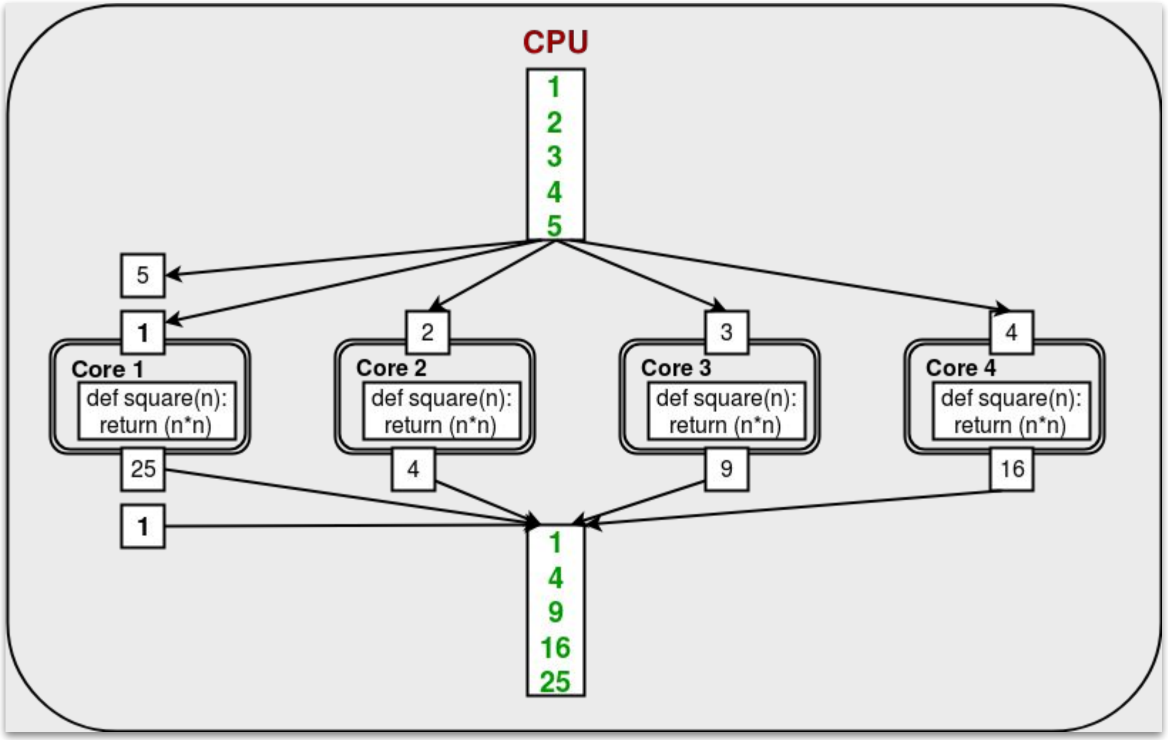


Should we jump to Highest Parallelization always for all tasks? if not how to decide?

Process Parallelization bypassing GIL

Some example snippets using default multiprocessing library:

```
def method_to_parallelize: # Square list of numbers
# Example 1: using starmap
with Pool(num_workers) as pool:
    results = pool.starmap(method_to_parallelize, ranges)

# Example 2: using map
jobs = []
for i in range(0, num_workers):
    jobs.append((i*ranges+1, (i+1)*ranges))
pool = Pool(num_workers).map(method_to_parallelize, jobs)

# Example 3: using apply_async
pool = Pool(num_workers)
for arg in zip([x+1 for x in ranges],ranges[1:]):
    results.append(pool.apply_async(method_to_parallelize, arg))
```

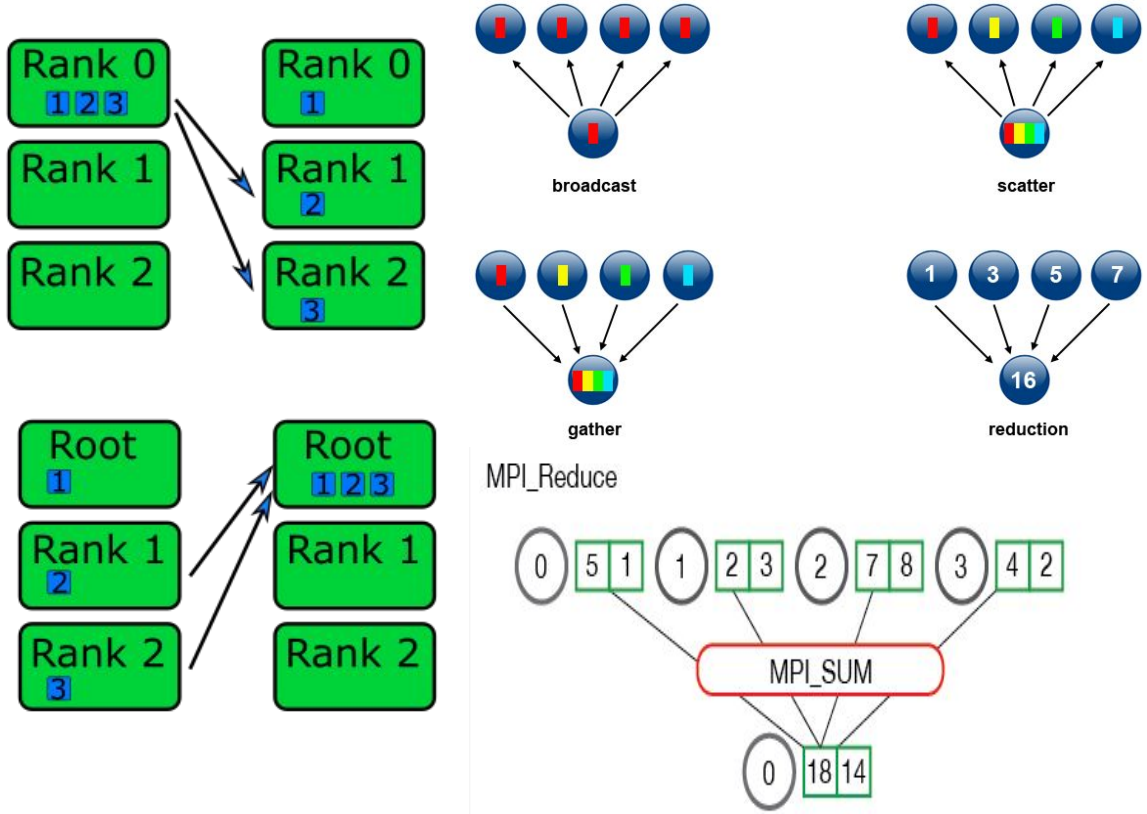


https://docs.python.org/3/library/multiprocessing.html

MPI Parallelization

Example snippets using mpi4py python library:

```
# Example 1: sum partial arrays with reduce (Blocking)
    local_sum = sum(i * 0.001 for i in range(rank *
local_partial_array, (rank + 1) * local_partial_array))

    total_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

    return total_sum

# Example 2: using all reduce (Non - blocking)
    for i in range(len(vector)):
        local_sum += vector[i]

        total_sum = MPI.COMM_WORLD.allreduce(sum, op=MPI.SUM)

    return total_sum
```

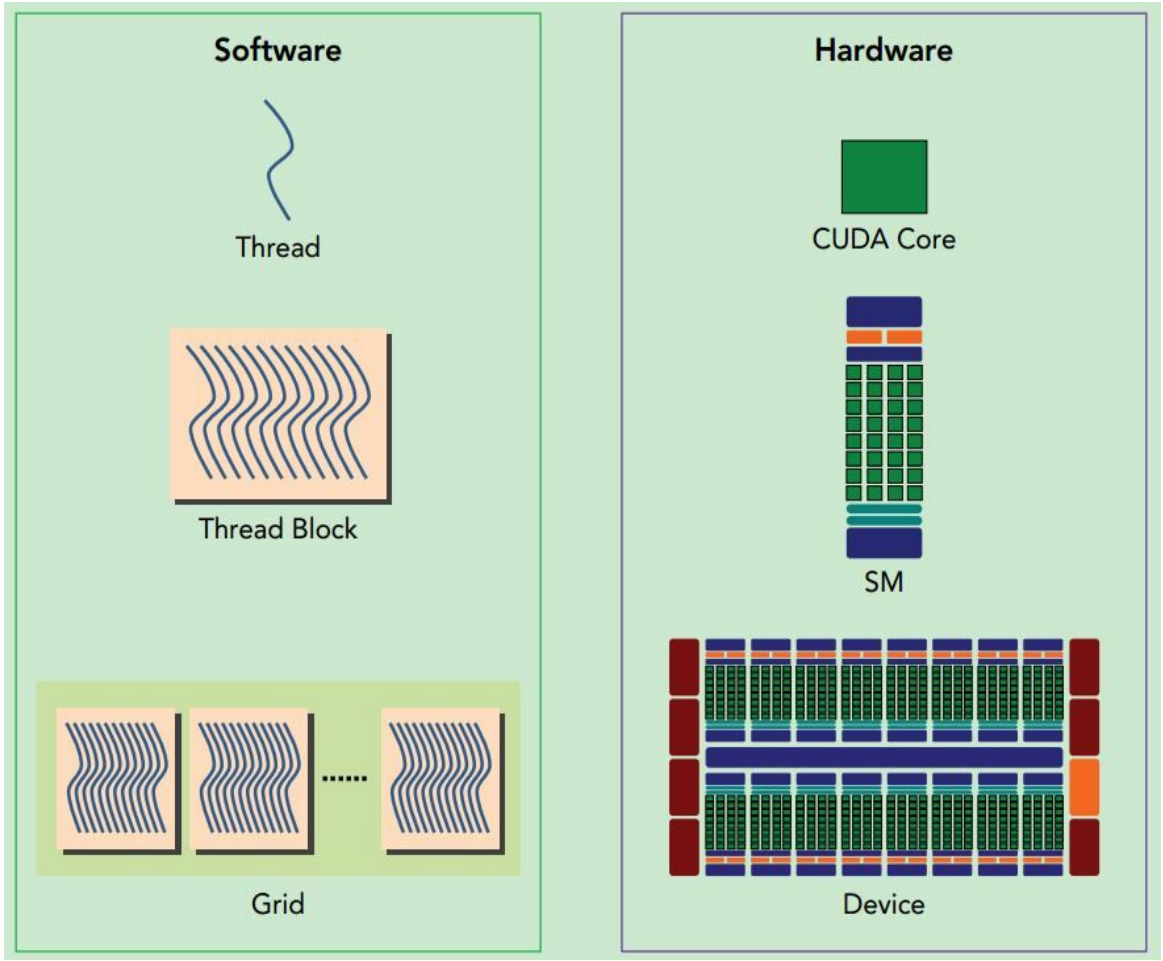


https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/
https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.SUM.html
https://hpc-tutorials.llnl.gov/mpi/collective_communication_routines/

CUDA Parallelization

Example snippet using numba python library's cuda:

```
# Example Kernel function
TPB = 16  # Number of threads per block

# Kernel function
@cuda.jit
def parallel_sum(arr, result):
    sdata = cuda.shared.array(shape=TPB, dtype=float32)
    tid = cuda.threadIdx.x
    i = cuda.grid(1)
    if i < arr.size:
        sdata[tid] = arr[i]
    else:
        sdata[tid] = 0.0

    cuda.syncthreads()
# Ensure enough blocks to cover the array size
    block = TPB
    grid = (n + block - 1) // block
    parallel_sum[grid, block](d_arr, d_result)
```

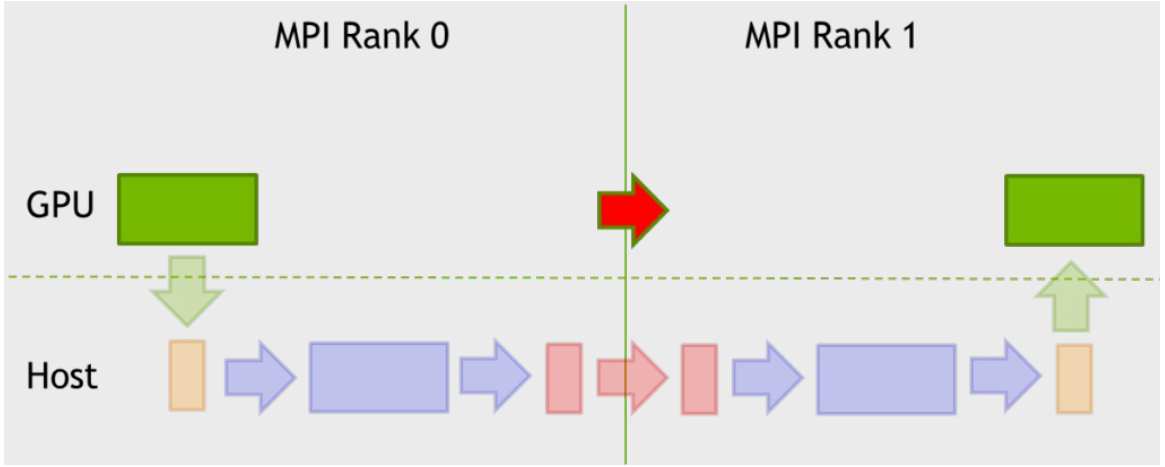


https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

CUDA aware MPI Parallelization

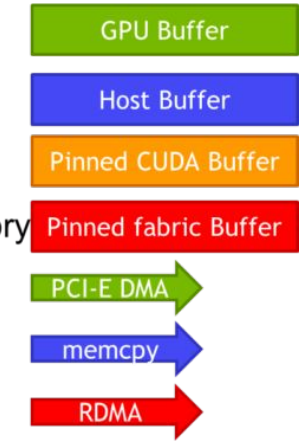
Example snippets using cupy python library:

```
# Example Set up for MPI
comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
number_of_ranks = comm.Get_size()

# Example arrays Created and shared to processes can also be
replaced for other MPI functions
if my_rank == 0:
    a = cupy.random.random(N * number_of_ranks)
else:
    a = cupy.empty(1)

comm.Scatter(a, a_partial, root=0)

# Example call for kernel function
partial_sum = cupy.zeros(1, dtype=cupy.float32)
parallel_sum[grid, block](a_partial, partial_sum)

# Example array reduction can be replaced for other MPI functions
total_sum = cupy.zeros(1, dtype=cupy.float32)
comm.Reduce(partial_sum, total_sum, op=MPI.SUM, root=0)
```



https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

- A green box is a GPU Buffer
- A blue box is a regular pageable host buffer
- A yellow box is a pinned CUDA buffer in host memory
- A red box is a pinned network Fabric buffer in host memory
- A green arrow is a DMA transfer over the PCI-E bus
- A blue arrow is a regular memcpy within host memory
- A red arrow is a RDMA network message.

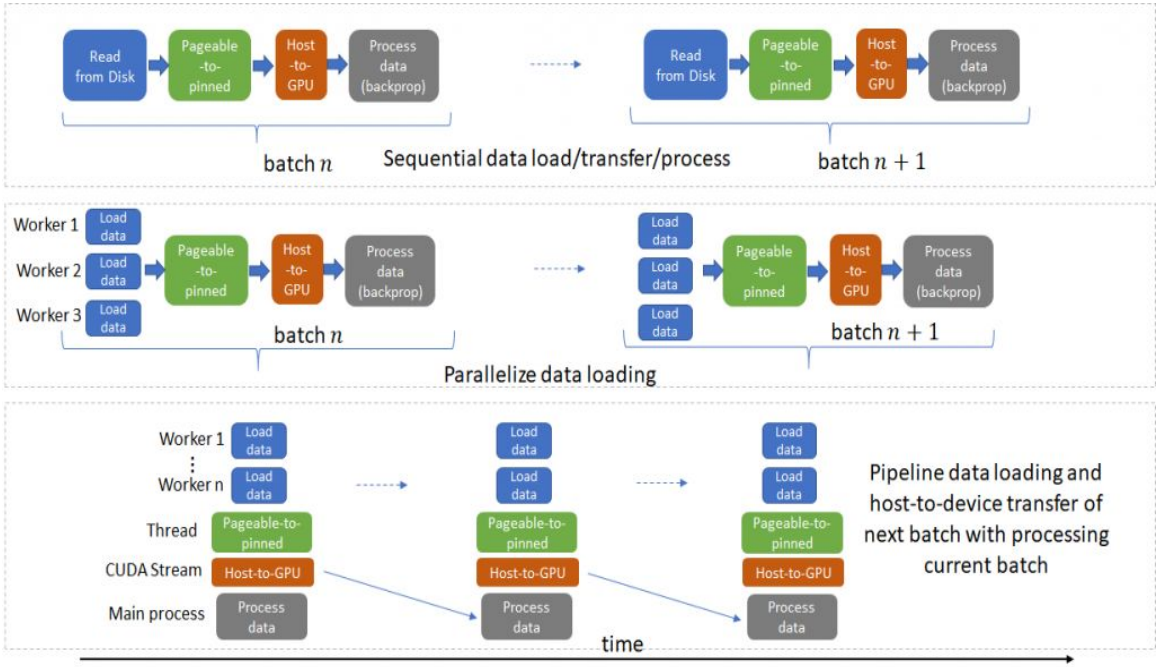


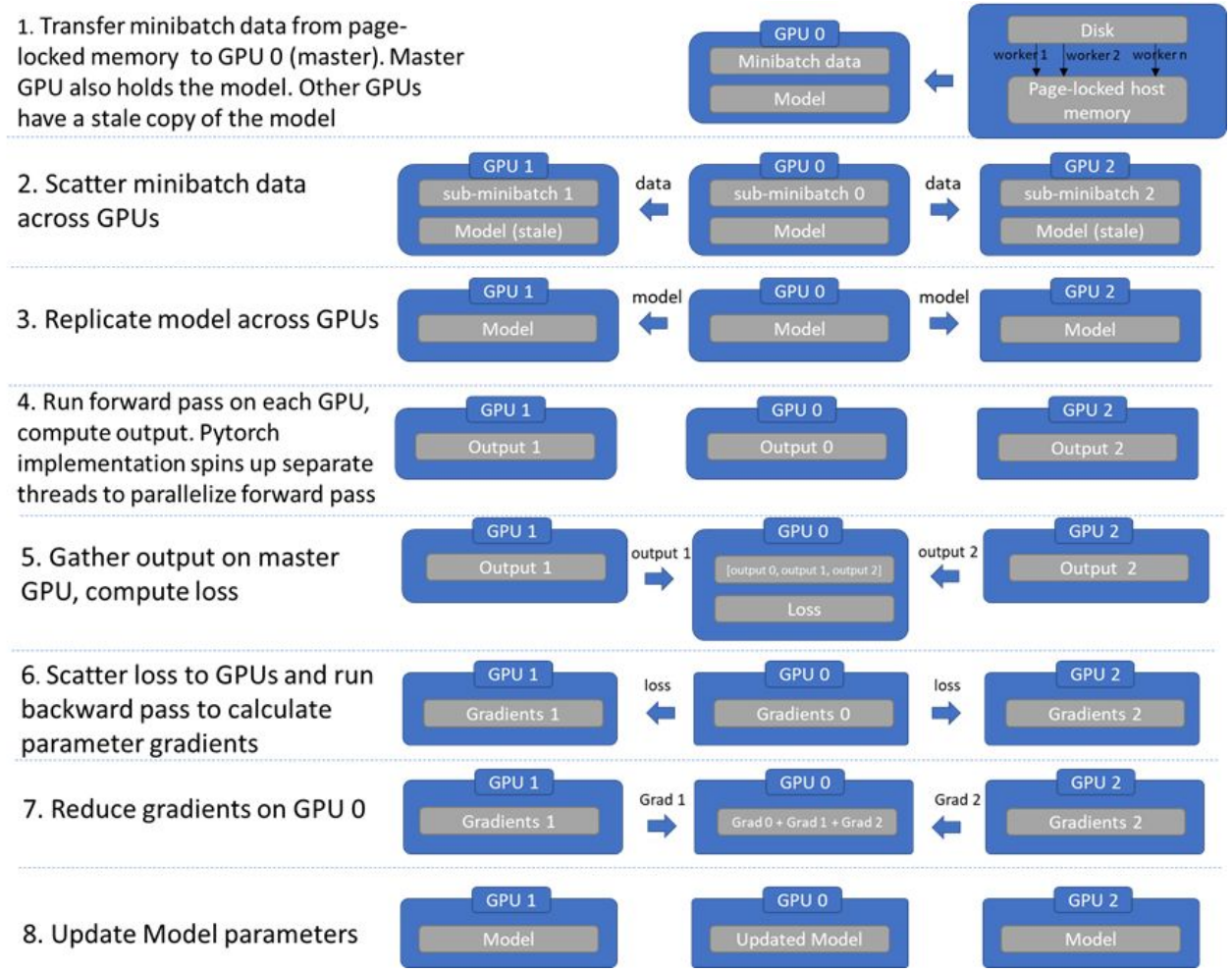
DP vs DDP

Data Parallel

One GPU (0) acts as the master GPU and coordinates data transfer.

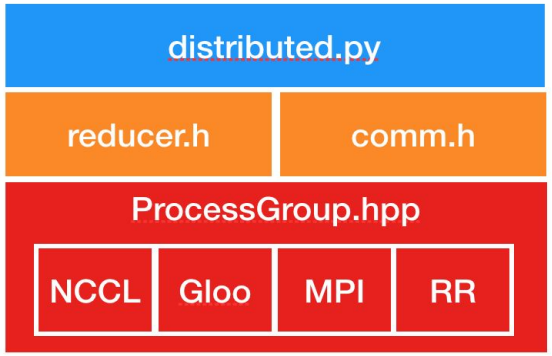
Implemented in PyTorch data_parallel module

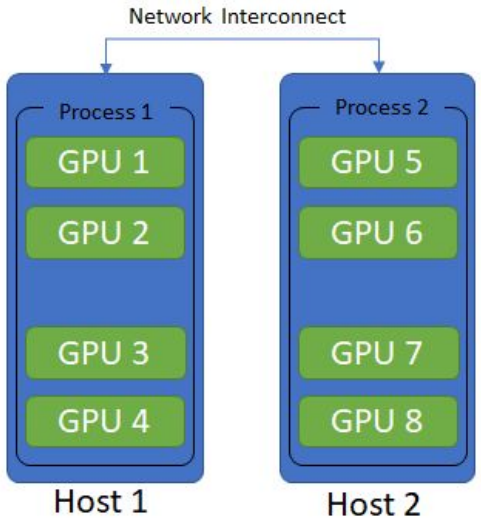




DP vs **DDP** (pytorch ddp in focus)

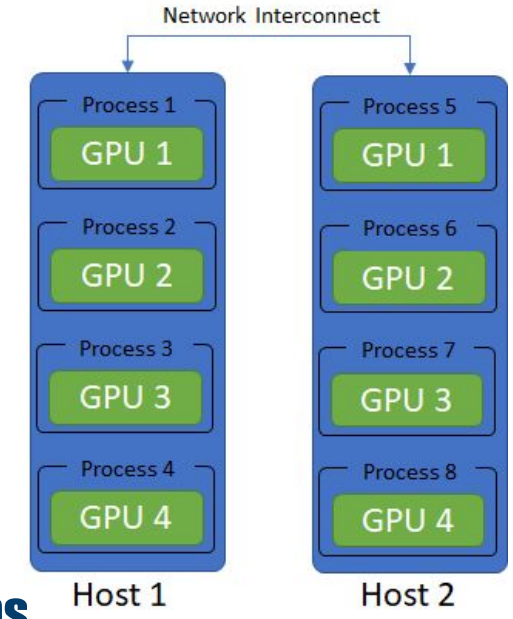
Torch DDP Architecture with backends



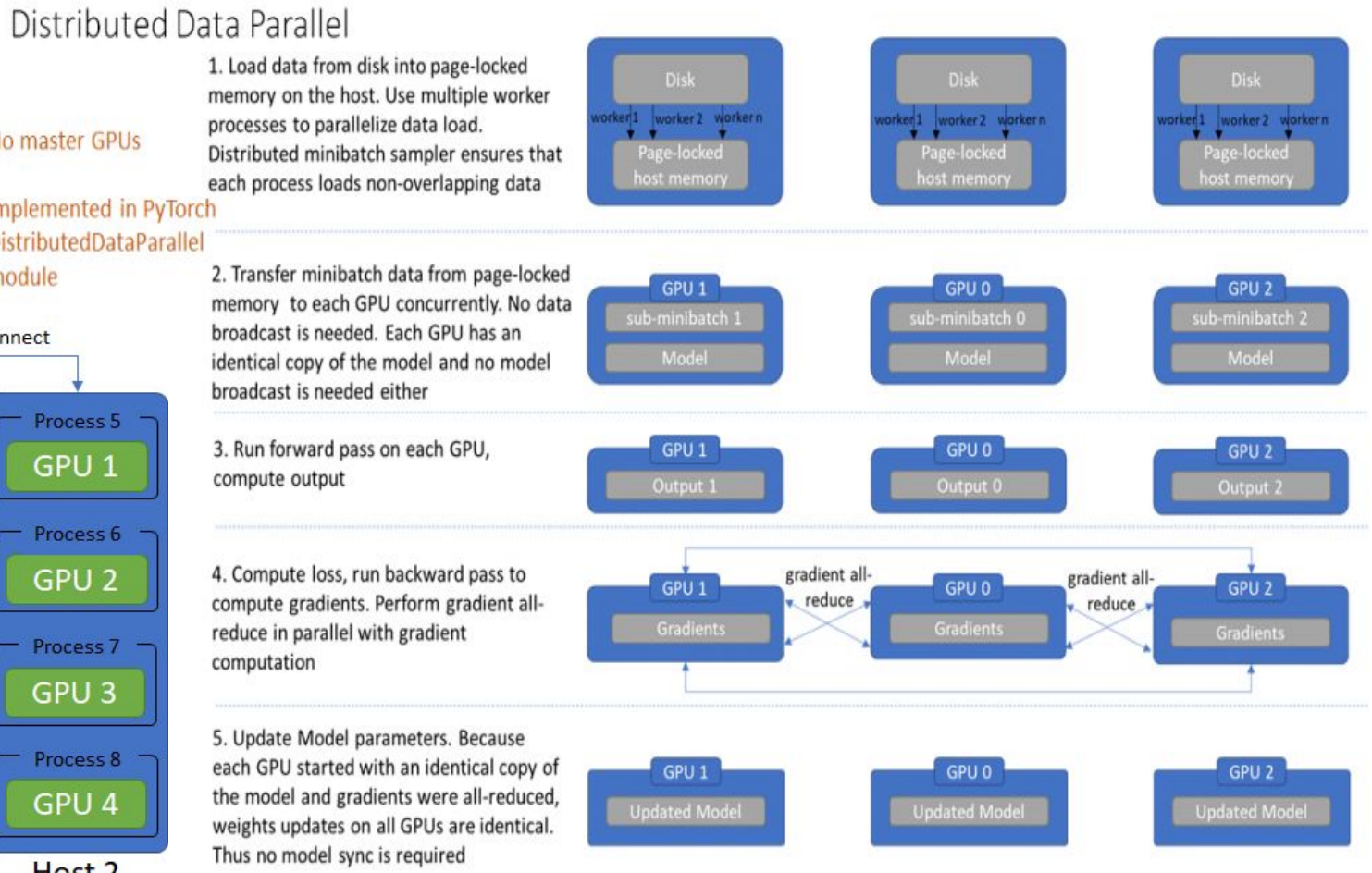


Configuration 1: Each process operates on all four GPUs in data parallel mode

DDP Options



Configuration 2: Each process operates on a single GPU



https://pytorch.org/docs/stable/notes/ddp.html

TREASURE HUNT

- **Task 1 - Parallel Processing (Multi-processing Single Node)**
- **Task 2 - Message Passing Interface (Multi-processing Multiple Nodes)**
- **Task 3 - CUDA (GPU Memory Computation)**
- **Task 4 - CUDA Aware MPI (GPU with Multi-processing)**
- **Task 5 - DDP Parallelization**

INSTRUCTIONS

- **1. Clone the Repository:**

```
git clone https://github.com/epavel1/ESDE-parallelism.git
cd ESDE-parallelism
```

- **2. Setting up the Environment:** Navigate to /env directory and source the environment.sh file.

```
cd env/
source environment.sh
```

- **3. Navigate to the Tasks:** Each task's Python script and corresponding job script are located in the /tasks directory.
- **4. Complete the Tasks:** Open each taskX.py file and fill in the missing code. Use the corresponding taskX.sh script to submit the job and test your solution. Hint: In the /solutions directory, you can find the solutions for each task and compare them with your own solutions if you get stuck.

```
sbatch taskX.sh
```

PYTHON PACKAGES

multiprocessing is a Python package that allows for the creation and management of separate processes, enabling parallel execution of tasks on multiple processors. It uses processes instead of threads to avoid the Global Interpreter Lock (GIL) limitations.

mpi4py is a Python package that provides bindings to the Message Passing Interface (MPI) library, allowing for efficient and scalable parallel computing across multiple nodes in HPC environments. It facilitates communication and coordination between processes running on different nodes.

numba is a Python package that uses Just-In-Time (JIT) compilation to optimize Python functions, translating them into machine code for significant performance improvements in numerical and scientific computing. It allows users to accelerate Python code without changing its syntax.

CuPy is a Python package that replicates the NumPy API but leverages NVIDIA GPUs to accelerate numerical computations. It allows users to perform NumPy-like operations on GPUs, providing an easy way to integrate GPU acceleration into existing NumPy-based code.

PyTorch is an open-source deep learning framework that provides flexible and efficient tools for building and training neural networks. It offers dynamic computation graphs, automatic differentiation, and GPU acceleration, making it a popular choice for both research and production.

torch.nn.parallel.DistributedDataParallel (DDP) is a module in PyTorch that facilitates multi-GPU training by distributing the training data and model parameters across multiple GPUs. It ensures synchronized updates to the model parameters, providing efficient and scalable parallel training.

CODE SNIPPETS

- **1. multiprocessing:** `starmap` in `multiprocessing.Pool` parallelizes the execution of a function across an iterable of argument tuples by unpacking them and passing them to the function. get stuck.

```
results = pool.starmap(func, iterable)
```

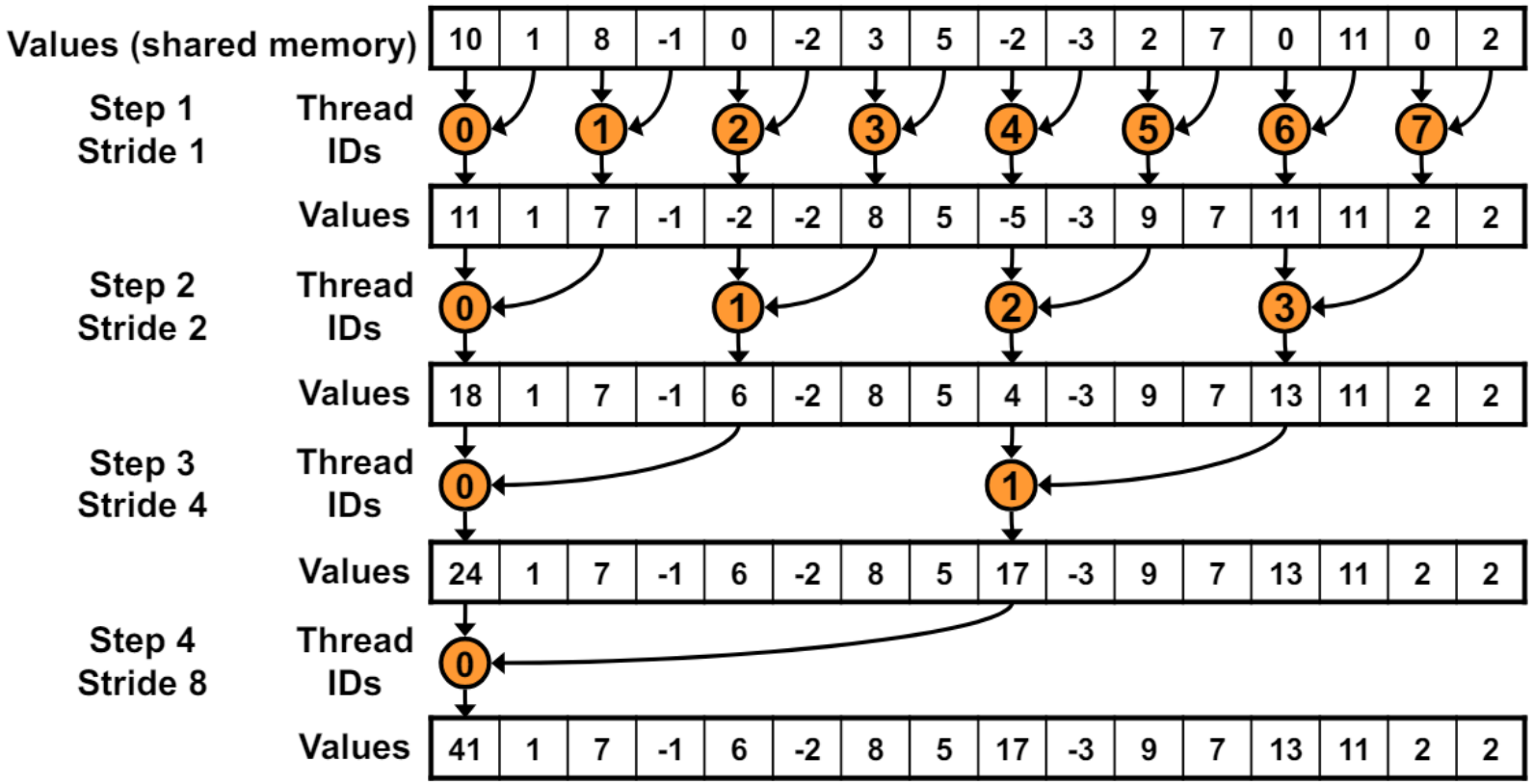
- **2. mpi4py:** `reduce` in mpi4py allows collective operations where each process contributes a value and the reduction operation combines these values to produce a result on a specified root process. `Scatter` in contrast distributes data from the root process (root=0) to all other processes in MPI.COMM_WORLD. It divides ` sendbuf ` into equal parts and sends each part to a different process, where it is received into ` recvbuf `.

```
result = MPI.COMM_WORLD.reduce(local_result, op=MPI.OPERATION, root=0)
MPI.COMM_WORLD. Scatter([sendbuf, sendtype], [recvbuf, recvtype], root=0)
```

- **3. numba:** the following syntax is used in numba to launch CUDA kernel functions on the GPU, enabling parallel execution of operations defined within the kernel function across multiple threads **and blocks.**

```
kernel_function_name[grid_dimensions, block_dimensions](arguments)
```

TASK 3 / 4


Values (shared memory)
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 1
Thread IDs
0 1 2 3 4 5 6 7
Values
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2
Stride 2
Thread IDs
0 1 2 3
Values
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3
Stride 4
Thread IDs
0 1
Values
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4
Stride 8
Thread IDs
0
Values
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

GOOD LUCK