Efficient MPI-based Communication for GPU-Accelerated Dask Applications

AU21 CSE 5449
miriyala.6
DASK

- Dask is a popular parallel and distributed computing framework
- Converts user application into a task-graph, and later executed lazily on distributed hardware
- Dask Distributed library supports distributed computation through scheduler, worker, and client
- Two communication devices: 1) TCP device 2) UCX backend using UCX-Py—a Cython wrapper to UCX
• Dask has mainly supported execution on hosts (CPUs) only,
• NVIDIA RAPIDS library, which aims to enable parallel and distributed computation on clusters of GPUs.
• RAPIDS support processing of distributed data stored in cuPy (numPy-like) and cuDF (Pandas-like) formats
• Effective communication device required to support execution of Dask programs on cluster of GPUs
• Dask distributed library is an asynchronous I/O application ie, it supports non-blocking and concurrent execution of its routines/functions
• So mandatory for any communication backend to be part of Dask Distributed library must implement coroutines that are non-blocking methods defined and awaited by using the async and await keywords respectively
Coroutines

Central idea is for the applications to not occupy the CPU unnecessarily while waiting for other tasks to finish, awaiting results, or while doing an I/O operation.

```
COROUTINES

Co-operative concurrent functions

Preempted when
- read/write from disk
- perform communication
- sleep, etc

Scheduler/event loop manages execution of all coroutines

Single thread utilization increases

def zzz():
    print("start", 1)
    time.sleep(2)
    print("finish", 1)

def main():
    zzz(1)
    zzz(2)

main()

Output:

start 1  # t = 0
finish 1  # t = 2
start 2  # t = 2 + \(\triangle\)
finish 2  # t = 4 + \(\triangle\)
```

```python
async def zzz():
    print("start", 1)
    await asyncio.sleep(2)
    print("finish", 1)

f = asyncio.create_task

async def main():
    task1 = f(zzz())
    task2 = f(zzz())
    await task1
    await task2

asyncio.run(main())

start 1  # t = 0
start 2  # t = 0 + \(\triangle\)
finish 1  # t = 2
finish 2  # t = 2 + \(\triangle\)
```
Communication requirements for Dask Distributed library

• 1) R1: Scalability—Provide scalability by exploiting low-latency and high-throughput RDMA networks.

• 2) R2: Coroutines—The communication backend needs to support point-to-point send and receive operations through asyncio coroutines defined using the async/await syntax.

• 3) R3: Elasticity—Dask computations are elastic in nature since worker processes may dynamically join or leave the cluster. The backend must support this dynamic behavior gracefully.

• 4) R4: Serialization/De-serialization—The communication backend should support communication to/from host based buffers including NumPy, Pandas DataFrames
Why new communication device?

• Dask traditionally supports 2 communication devices 1.TCP 2. UCX-Py— based on the UCX library
• The TCP backend clearly does not satisfy requirement R1: Scalability because it not designed for high-speed networks like InfiniBand.
• UCX-Py is designed for high-speed networks, and yet it fails to satisfy requirement R2: Coroutines,
• the UCX-Py library uses a separate coroutine that is responsible for making progress for the UCX communication engine so it delays progressing the communication engine
MPI4DASK – new communication device

• MPI is defacto standard for writing parallel applications
• MPI API defines a set of point-to-point and collective communication routines
• We use GPU-aware MPI library called MVAPICH2- GDR that provides optimized point-to-point and collective communication support for GPU devices
• Dask ecosystem is implemented in the Python programming language implies that the MPI communication backend must also be implemented in Python.
• we use the GPU-aware mpi4py library that provides Cython wrappers to native MPI library MVAPICH2-GDR
• Mpi4py supports efficient exchange of GPU data stored in cuPy and cuDF format.
Dask Layered Architecture with Communication Backends

Figure 3: Dask Layered Architecture with Communication Backends. Yellow boxes are designed and evaluated as a part of this paper.
Point-to-point Communication Coroutines (Dask Implementation)

- The challenge is to implement asynchronous communication coroutines using mpi4py over MVAPICH2-GDR.
- First effort to exploit MPI-based communication inside an asyncio application in Python.
- mpi4py provides two variants of point-to-point functions.
  1. Comm.send()/Comm.isend() to communicate data/to from Python objects.
  2. Comm.Send()/Comm.Isend() to communicate data to/from directly from user specified buffer.
- MPI4Dask makes use of Comm.Isend()/Comm.Irecv() methods.
- MPI4Dask calls the asyncio.sleep() method that allows other coroutines to make progress while waiting for communication to complete.

Listing 1: The Comm.Isend()-based Send Communication Coroutine Implemented by MPI4Dask.

```python
request = comm.Isend([buf, size], dest, tag)
status = request.Test()
while status is False:
    await asyncio.sleep(0)
    status = request.Test()
```

Listing 2: The Comm.Isend()-based Send Communication Coroutine Implemented by MPI4Dask.

```python
request = comm.Irecv([buf, size], src, tag)
status = request.Test()
while status is False:
    await asyncio.sleep(0)
    status = request.Test()
```

• Dask is a programming framework for writing data science applications so common to handle large amounts of data
• Comm.Isend() and Comm.Irecv() methods accept an argument int count used to specify the size of the message being sent or received.
• The Maximum value of count is $2^{31} - 1$ bytes, which corresponds a message size of 2 GB–1. But Dask Distributed library attempts to communicate messages larger than this value including upto 64 GB.
• So we divide the large message into several chunks of 1 GB for the actual communication
• Buffers specified to these communication functions are Python objects and hence subscriptable using the slice notation `array[start:end]`.
• This approach works for numPy and cuPy arrays and hence the buffer argument can be subscripted in a loop to implement chunking.
- After the initial bootstrapping, MPI4Dask has full connectivity between all processes as provided by default communicator MPI_COMM_WORLD.

- What if workers want to use the same tag for both data and control message exchanges?

- The solution is to rely on MPI sub-communicators.

- Creating a new MPI sub communicator is a costly operation and for this reason all of this is done at the startup.

- Our approach has negligible overhead because these sub communicators are initialized at the startup and re-used later during the program execution.

```python
for i in range(size):
    for j in range(i+1, size):
        incls = [i, j]
        new_group = MPI.Group.Incl(group, incls)
        new_comm = MPI.COMM_WORLD.Create(new_group)
        if rank == i:
            comm_table.update({j : new_comm})
        else if rank == j:
            comm_table.update({i : new_comm})
```
Handling Dynamic Connectivity

• The UCX and TCP communication devices maintain information for remote endpoints since this information is required for the actual communication.

• MPI4Dask, we have replaced the abstraction of endpoint with sub-communicator. This enables us to utilize the existing communication infrastructure of the Dask eco system.

• Apart from communications being setup at the startup, Dask allows dynamic connections between workers as well.

• MPI4Dask we handle dynamic connections by starting a server that listens for incoming connections—and invokes a connection handler callback function—using the asyncio.start_server() method.

• Dask scheduler and worker processes use this function since they act as listeners and other processes are allowed to connect with them.

• Later process used the asyncio.open_connection() for connection.

• New sub-communicator is built for every new dynamic connection.
Performance evaluation

Latency and Throughput Comparison

- Perform latency and throughput comparisons between MPI4Dask and other communication backends—in particular UCX-Py—using a Ping Pong benchmark.
Application BenchMarks

Evaluated MPI4Dask against UCX-Py for two application benchmarks:
1) sum of cuPy array and its transpose, and 2) cuDF merge

Figure 7: Sum of cuPy Array and its Transpose (cuPy Dims: 16K×16K, Chunk size: 4K, Partitions: 16): Performance Comparison between IPoIB, UCX, and MPI4Dask on the RI2 Cluster. This benchmark presents strong scaling results. 28 threads are started in a single Dask worker.
Scalability Results on the TACC Frontera (GPU) Cluster

- Scalability results for the two application benchmarks and this evaluation was done on the Frontera (GPU) system that is equipped with 360 NVIDIA Quadro RTX 5000 GPUs in 90 nodes.

![Graphs showing scalability results for different benchmarks.](image)

Figure 9: Execution Time Comparison for Sum of cuPy Array with its Transpose Benchmark, cuDF Merge Benchmark, and Average Throughput for Dask Workers for the cuDF Merge Benchmark. This evaluation was done on the Frontera (GPU) system at TACC. This comparison is between MPI4Dask, UCX, and TCP (using IPoIB) communication devices. There are 4 GPUs in a single node. Each Dask worker has 8 threads.
Towards Efficient RDMA-based Communication Coroutines for Parallel Python

AU21 CSE 5449
miriyala.6
There is need for new library to support all communication requirements of the Dask Distributed library over RDMA networks like InfiniBand.

To address these issues came up with a multilayered communication library called Blink that provides high performance support for communication coroutines for the Dask Distributed framework.

Blink is a new library designed to support all communication requirements of the Dask Distributed library over RDMA networks like InfiniBand.

Blink offers an alternate design approach instead of assigning the progress for the communication engine to a specific task, Blink ensures co-operative progression.

This ensures that all communication coroutine including send and receive operations probe the progress engine while waiting for its own completion.
Design of Blink library

Blink has the following four layers:

• 1) Blink-Dask: The communication backend provided by Blink that implements the communication API dictated by the Dask Distributed library.

• 2) Blink-Py: The Python layer that exports point-to-point send and receive operations.

• 3) Blink-Cy: The Cython wrapper-layer that implements communication functionality in conjunction with the native layer.

• 4) Blink-Native: The native C backend of Blink that provides communication support over RDMA networks using the InfiniBand verbs library.
Blink Architecture

• The top layer Blink-Dask implements the communication backend for Blink inside the Dask Distributed library
• This layer is followed by Blink-Py which is a thin wrapper Python API that exports coroutines implemented by the bottom layers for its higher layer e.g., Blink-Dask
• The next layer Blink-Cy is a Cython wrapper layer that implements the bulk of the communication logic at the Python end
• Blink-Cy provides support for communication coroutines like blink_send(), blink_recv(), and blink_probe().
• The bottom-most layer Blink-Native implements the communication logic for the InfiniBand verbs library. Blink-Native is based on the Unified Communication Runtime (UCR) library.

Figure 3: Layered Architecture of the Blink Library
Dynamic Connection Establishment

• It is very common for Dask end-users to scale up or scale down their Dask clusters as per the requirements of their applications and resource availability.

• So it is important that the Blink library is able to support this kind of dynamic behavior.

• Blink provides the notion of endpoints that are communication abstractions for sending data to/from remote processes.

Figure 4: An Example Scenario to Demonstrate the Dynamic Connection Establishment Provided by the Blink Library
A worker process primarily has two functions.

The first is to execute tasks for the user application that are part of the task graph.

The second responsibility of a worker process is to maintain communication with other co-workers and the scheduler process.

Blink implements communication in the bottom two layers of the stack namely: Blink-Cy and Blink-Native.
Eager vs Rendezvous Protocol

- The Blink-Native communication library uses eager protocol for messages smaller than 64KBytes.
- For messages larger than 64 KBytes, the protocol switches to rendezvous protocol.
Microbenchmark Evaluations

- Evaluated performance of ping-pong latency and bandwidth for native communication devices (e.g., UCX and Blink) as well as python wrappers (e.g., UCX-Py and Blink-Py). Both on Nowlab cluster and Comet cluster.
PERFORMANCE EVALUATION

• Blink is evaluated on three different cluster testbeds.
• Inhouse clusters RI2, NowLab and Comet, hosted at San Diego Supercomputing Center (SDSC)
• The specifications of different test beds are given

<table>
<thead>
<tr>
<th>Specification</th>
<th>Comet</th>
<th>RI2</th>
<th>NowLab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Family</td>
<td>Xeon Haswell</td>
<td>Xeon Broadwell</td>
<td>Xeon Haswell</td>
</tr>
<tr>
<td>Processor Model</td>
<td>E5-2680 v3</td>
<td>E5-2680 v4</td>
<td>E5-2687W v3</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.5 GHz</td>
<td>2.4 GHz</td>
<td>3.1 GHz</td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Cores Per socket</td>
<td>12</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>RAM (DDR4)</td>
<td>128 GB</td>
<td>128 GB</td>
<td>64 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>IB-FDR (56G)</td>
<td>IB-EDR (100G)</td>
<td>IB-IIDR (200G)</td>
</tr>
</tbody>
</table>
Application Evaluation

- Used three application kernels to evaluate the performance of Blink-Py against UCX-Py and TCP on the RI2 cluster
  1) Array of Sum and Transpose
  2) Pandas DataFrame Sum
  3) Truncated Singular Value Decomposition (SVD)