Accelerating Spark with UCX

Dec 2019
**Unified Communication X (UCX) - high performance communication layer library (1/2)**

<table>
<thead>
<tr>
<th>Unified API</th>
<th>Focus on performance</th>
<th>Production quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications driven, simple, extendable, HW-agnostic</td>
<td>Fast, scalable, highly optimized low latency high bandwidth messaging framework</td>
<td>Multi-tier testing, used by top Mellanox customers in production</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Open source</th>
<th>Innovation</th>
<th>Multi arch/transports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collaboration between industry, laboratories, and academia</td>
<td>Concepts and ideas from research in academia and industry</td>
<td>RoCE, InfiniBand, Cray, TCP, shared memory, GPUs, x86, ARM, POWER</td>
</tr>
</tbody>
</table>

**Co-design of Network APIs**
Unified Communication X (UCX) - high performance communication layer library (2/2)

Applications

- HPC (MPI, SHMEM, …)
- Storage, RPC, AI
- Web 2.0 (Spark, Hadoop)

UCP – High Level API (Protocols)
Transport selection, multi-rail, fragmentation

- HPC API: tag matching, active messages
- I/O API: Stream, RPC, remote memory access, atomics
- Connection establishment: client/server, external

UCT – Low Level API (Transports)

- RDMA: RC, DCT, UD, iWarp
- GPU / Accelerators: CUDA, AMD/ROCM
- Others: Shared memory, TCP, OmniPath, Cray

Hardware

- OFA Verbs Driver
- Cuda
- ROCM
JUCX – java bindings for UCX

- Transport abstraction - implemented on top of UCP layer
  - Can run over different types of transports (Shared memory, Infiniband/RoCE, Cuda,...)
- Ease of use API wrapper over high level UCP layer
- Supported operations: non blocking send/recv/put/get
JUCX API example

1. Instantiate ucp context:

   ```java
   UcpContext context = new UcpContext(new UcpParams().requestRmaFeature());
   ```

2. Instantiate ucp worker:

   ```java
   UcpWorker worker = context.newWorker(new UcpWorkerParams());
   ```

3. Instantiate ucp endpoint:

   ```java
   EndpointParams epp = new UcpEndpointParams().setSocketAddress(InetSocketAddress("1.2.3.4:1234")
   UcpEndpoint endpoint = worker.newEndpoint(epp);
   ```

4. Perform get/put/send/recv operation on endpoint:

   ```java
   UcxRequest request = endpoint.getNonBlocking(remoteAddress, remoteKey, localBuffer);
   ```

5. Progress request until it's completed:

   ```java
   while(!request.isCompleted()) {
       worker.progress();
   }
   ```
Spark’s Shuffle Basics

Map

Input

Map output

File

Spark Master

Reduce

Reduce task

Fetch blocks

Reduce task

Fetch blocks

Reduce task

Fetch blocks

Reduce task

Fetch blocks

Reduce task

Fetch blocks
The Cost of Shuffling

- Shuffling is very expensive in terms of CPU, RAM, disk and network I/Os.
- Spark users try to avoid shuffles as much as they can.
- Speedy shuffles can relieve developers of such concerns, and simplify applications.
SparkUCX Shuffle Plugin

https://github.com/openucx/sparkucx
ShuffleManager Plugin

- Spark allows for external implementations of ShuffleManagers to be plugged in
  - Configurable per-job using: “spark.shuffle.manager”
  - Interface allows proprietary implementations of Shuffle Writers and Readers, and essentially defers the entire Shuffle process to the new component

- SparkUCX utilizes this interface to introduce RDMA in the Shuffle process

SortShuffleManager

UcxShuffleManager
SparkUCX memory layout object model

- **Driver global metadata buffer**

  - Mapper0 Metadata:
    - Index {address, rkey}
    - Data {address, rkey}

- **Mapper Index file**

- **Mapper data file**
SparkUCX operation flow

- **Initialization:**
  Spark driver allocates global metadata buffer per shuffle stage, to hold addresses and memory keys of data and index files on mappers.

- **Mapper phase:**
  - mmap() and register index and data files
  - Publish {address, rkey} to driver metadata buffer (ucp_put).

- **Reduce phase:**
  - Fetch metadata from driver (ucp_get)
  - For each block:
    - Fetch offset in data file, from index file (ucp_get).
    - Fetch block contents from data file (ucp_get).
Benchmarking eco-system

- Benchmarks: Terasort + Pagerank
  - https://github.com/zrlio/crail-spark-terasort
  - https://github.com/Intel-bigdata/HiBench

- Terasort:
  - 1.2 TB input, 10K mappers, 15k reducers

- Pagerank:
  - Bigdata Hibench workload (600 Gb), 5K mappers, 15K reducers

- 15 nodes: Broadwell @ 2.60GHz, 250GB RAM, 500GB HDD

- ConnectX-5: Infiniband: 100G EDR. TCP device: IPoIB 100G

- Red Hat Enterprise Linux Server release 7.5 (Maipo) (kernel: 3.10.0-862.el7.x86_64)

- MLNX_OFED_LINUX-4.6-1.0.1.1.

- Spark-2.4.3, Hadoop-2.9.2, UCX v1.7.0
TCP vs UCX Terasort scalability (2/3)

Scalability on number of executors

- UCX total time
- TCP total time
- Reduce time UCX
- Reduce time TCP

Scalability on number of executors

- UCX
- TCP

54-92%
TCP vs UCX Terasort scalability (3/3)

Scalability on number of reducers

- UCX total time
- TCP total time
- Reduce time UCX
- Reduce time TCP

25-92%
## SparkRDMA vs. SparkUCX

<table>
<thead>
<tr>
<th>SparkRDMA</th>
<th>SparkUCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on abandoned IBM DiSNi verbs package</td>
<td>Based on UCX high-level API which has dedicated R&amp;D and wide community. Production grade.</td>
</tr>
<tr>
<td>Supports IB/ROCE with RC only</td>
<td>Supports IB, ROCE with RC/DC/Shared memory, and TCP as fallback</td>
</tr>
<tr>
<td>Not scalable, CQ and progress thread per connection</td>
<td>Scalable: CQ per executor</td>
</tr>
<tr>
<td>Communications progress on dedicated thread which consumes CPU %</td>
<td>Communications are initiated from application threads and progressed asynchronously by hardware</td>
</tr>
<tr>
<td>RDMA protocols are implemented in Java</td>
<td>Based on standard UCX API and protocols hiding complexity of RDMA</td>
</tr>
<tr>
<td>Registering each data block with different key</td>
<td>Registering all data as single chunk</td>
</tr>
<tr>
<td>Showed improved vs. <strong>worst</strong> TCP numbers</td>
<td>Showed improved vs. <strong>best</strong> TCP numbers</td>
</tr>
</tbody>
</table>
Future work

- Optimizations on multiple benchmarks (TPC-DS, TPC-H, etc.)
- Support shuffle data larger than memory
- GPU memory support
- HDFS optimization with UCX
Thank You