RDMA programming design and case studies – for better performance distributed applications

NTT Software Innovation Center
Yoshiro Yamabe
RDMA features, potential

- Remote Direct Memory Access (RDMA) is **low latency** and **low cpu overhead**, but **hard to implement**
- In simple micro-benchmark, RDMA’s latency is about one fifth of IPoIB’s latency (IPoIB: IP over Infiniband)
• To make use of RDMA potential, I tried to apply RDMA to MXNet, a distributed deep learning framework
  – MXNet adopts *Parameter Server mechanism* for distributed learning
• RDMA was seemed to be effective for this model
  – There are two times of communication between worker and server for each batch

**About case studies**

1. Calculate Parameter on each worker node
2. Push parameter to Server node
3. Aggregate parameters from all worker nodes
4. Pull parameter from server node
Data flow comparison – existing implementation and RDMA

- By applying RDMA, I can reduce **2 memcpyes per communication, total 4 copies per push and pull**

![Diagram](image)

**Existing (ZeroMQ)**
- worker: 1. calculation
- server: 2. push → 3. aggregation
- worker: 4. pull

**RDMA (my current RDMA)**
- worker: 1. calculation
- server: 2. push → 3. aggregation
- worker: 4. pull

Reduced overhead

- by applying RDMA, I can reduce 2 memcpies per communication, total 4 copies per push and pull
• By applying RDMA ideally, all 4 memcpy per communication will be reduced, but this is out of scope in this work due to very high implementation cost.

**Reduced overhead**
RDMA implementation Techniques

- **Optimal RDMA options**
  - RDMA operations (Operations: WRITE(put), READ(get), etc…)
  - Way to detect communication completions (Polling or MSI-X interrupt)

- **Importing Kernel layer mechanisms**
  - ring buffer for asynchronous communication (≒ socket buffer)
  - Separating detect completion thread

<table>
<thead>
<tr>
<th>Types</th>
<th>Techniques</th>
<th>STEP0</th>
<th>STEP1</th>
<th>STEP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDMA mechanisms</td>
<td>RDMA operations</td>
<td>RDMA WRITE</td>
<td>RDMA WRITE</td>
<td>RDMA WRITE</td>
</tr>
<tr>
<td>Detecting completion</td>
<td><strong>interrupt</strong></td>
<td><strong>interrupt</strong></td>
<td><strong>polling</strong></td>
<td></td>
</tr>
<tr>
<td>Kernel layer mechanisms</td>
<td>Ring buffer</td>
<td>×</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Separating detect completion thread</td>
<td>-</td>
<td>-</td>
<td>○</td>
<td></td>
</tr>
</tbody>
</table>

By implementing ring buffer, the performance increased over 10x !!
Result (Step1 vs Step2)

- Environment: 4 machines (1 server – 3 worker), one Tesla V100 GPU for each machine, cifar10
- **Step2 implementation is 1.5x faster than Step1 implementation! 😊**
Result (But…)

- **Existing implementation (zeromq) is fast!! 😞**
  - Step2 implementation is faster than it, but a bit…

  (There are still other tuning, so RDMA can be faster !!!)

![Graph showing time cost vs. number of workers for different implementations](image-url)
Conclusion

- **RDMA is efficient for reducing latency, cpu overhead, but there are many techniques to achieve high performance**
  - Selecting optimal options, flags, and redesign functions!
  - Porting a part of kernel layer tunings into user programs

- **As a matter of course, it is need to choose appropriate applications for RDMA**
  - CPU centric, network bottleneck, etc...
  - In this workload and environment, network load may be not adequate for confirming RDMA effect with lightweight change (Tesla V100 is very fast! & cifar10 is lightweight...)

- **My Conclusion**
  - To enjoy RDMA effect easily, libraries providing RDMA design patterns are needed
  - “true zero-copy RDMA implementation” is needed for higher performance
Appendix
The Features of RDMA

- RDMA can reduce overhead and accelerate applications by these two features
  - **zero-copy**
    - NIC can read/write data to user memory directly
  - **cpu-bypass**
    - When using one-sided RDMA, client can send data without involving server’s cpu

![RDMA Diagram](image-url)
Experimental environment

- Environment

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Ubuntu16.04</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz</td>
</tr>
<tr>
<td># of core, node</td>
<td>2 numa nodes and 14core 28thread for each numa node</td>
</tr>
<tr>
<td>Memory</td>
<td>256GB</td>
</tr>
<tr>
<td>Network</td>
<td>Mellanox ConnectX-5, 100Gbps cable</td>
</tr>
<tr>
<td>GPU</td>
<td>Tesla V100</td>
</tr>
</tbody>
</table>

- Using 1 server nodes and 1-3 worker nodes
- GPU and HCA are on the same numa node(e.g. node1) on all machines, and using numactl to avoid “remote access”
- Using Cifar-10 and ResNet50 to training
RDMA Programming Resources

- **Memory Region (MR)**
  - RDMA NIC can only read data on MR / write data to MR

- **Completion Queue (CQ)**
  - Communication’s success/failure information (completion) are pushed into CQ, and program can get it by pop from CQ by two ways
    - *Polling* or *completion channel (MSI-X interrupt)*

- **Queue Pair (QP)**
  - similar to socket
  - By posting request to QP, RDMA NIC execute data transfer referring to it.

- **RDMA Operations**
  - SEND, RDMA WRITE, RDMA READ, ATOMIC
    - SEND needs for receiver side to post receive request to QP (two-sided RDMA)
    - Others don’t need for receiver side to any operation (one-sided RDMA, cpu-bypass)
  - Service Types
    - RC, UC, UD, RD
    - RC can detect packet loss and re-send, but slower than other UC and UD
Ring buffer & separating polling thread

• **Ring Memory Region**
  – By using ring MR, sender can issue RDMA independent of receiver status
  – It’s similar to IP’s socket buffer in kernel space

• **Create additional thread for getting completion from CQ**
  – Getting completion in send function in application is inefficient because IP’s send function returns when send data pass to kernel
  – Using polling to get work completion because CPU resource waste can be avoided by pinning thread to core
Other implementation techniques

• Inline send
  – Can send short packet with low latency than normal send
  – Short : depend on RNIC’s kind. Mellanox ConnectX-5 is about 512 Bytes

• Immediate data
  – Can send 32bit data besides send data

• Serialize at Completion Queue
  – By sharing CQ with many QPs, can process completion serially
  – It can reduce performance, but it make implementation easier

• Parallelize at Completion Queue
  – Implementation becomes complicated, but performance will increase
  – Opposite to “Serialize at Completion Queue”