One-to-many UCT Transports

Alex Margolin and Morad Horany

UCF Annual Workshop, December 2020
Outline

1. Problem Statement

2. Collective operations on shared-memory

3. Using network multicast for collective operations
Collective operations in Open MPI

1. When Open MPI Starts – it chooses which (MCA) COLL components will be later used.

2. When user calls MPI_Bcast() – MPI passes the call to the chosen COLL component.

3. The chosen component can:
   a) Use P2P components (next slide)
   b) Call some external library (Part 3)
   c) Fail and fallback to another module...
Example: basic broadcast code (from: `coll_base_bcast.c`)

```c
ompi_coll_base_bcast_intra_generic( void* buffer, int original_count, struct ompi_datatype_t* datatype, ...
{
    rank = ompi_comm_rank(comm);

    /* Root code */
    if( rank == root ) {
        sendcount = count_by_segment;
        for( segindex = 0; segindex < num_segments; segindex++ ) {
            for( i = 0; i < tree->tree_nextsize; i++ ) {
                err = MCA_PML_CALL(isend(tmpbuf, sendcount, datatype,
                                      tree->tree_next[i],
                                      MCA_COLL_BASE_TAG_BCAST,
                                      MCA_PML_BASE_SEND_STANDARD, comm,
                                      &send_reqs[i]));
                if (err != MPI_SUCCESS) { line = __LINE__; goto error_hndl; }
            }
            /* complete the sends before starting the next sends */
            err = ompi_request_wait_all( tree->tree_nextsize, send_reqs,
                                         MPI_STATUSES_IGNORE );
            if (err != MPI_SUCCESS) { line = __LINE__; goto error_hndl; }
            tmpbuf += realsegsize;
        }
    }
}```
Our initial idea for UCG was to focus on consolidating calls:

**Batch (identical) send/receives!**

**UCG Reminder**

Current Usage of Open UCX

Consolidated Usage of Open UCX

```
Open MPI
  COLL
  PML
  MPI_Bcast()
  Open UCX
  UCP
  UCT
  Send #1
  Send #2

Open MPI
  COLL
  PML
  MPI_Bcast()
  Bcast
  UCP
  UCT
  Send #0
  Send #1
  Send #2
```
UCG Reminder

Our initial idea for UCG was to focus on consolidating calls:

**Batch (identical) send/receives!**

**Premature optimization too many abstraction layers are the root of all evil.**
UCG Reminder

Our initial idea for UCG was to focus on consolidating calls:

- **Batch (identical) send/receives!**
- **Premature optimization too many abstraction layers are the root of all evil.**

---

**Current Usage of Open UCX**

```
MPI_Bcast()
```

- **Open MPI**
  - **COLL**
  - **PML**
    - Send #1
    - Send #2

**Consolidated Usage of Open UCX**

```
MPI_Bcast()
```

- **Open MPI**
  - **COLL**
  - **PML**
  - **UCG**
    - Send #1
    - Send #2

---

![Time Cost Chart](image)

**The Reason I Am So Inefficient**
Anatomy of a Collective Operation

* Actual tree structure and radix may vary
Problem Statement

Reducing the latency of a “single level” of a one-to-many communication

Factors (not exhaustive nor prioritized):

• **Data pattern**: broadcast vs. scatter
• **Data size**: in UCX that’s short/bcopy/zcopy*
• **Process affinity** (w.r.t. memory hierarchy)
• **Typical process imbalance**
• **Non-MPI**: data availability (bcopy allows gradually providing chunks of it)
• **What we do with the buffer afterwards** (do we forward it?)

Another problem (mostly same factors): **many-to-one communication**.

*can we even consider zero-copy? Yes, we can! (just need a hint from MPI...)*
Initial thoughts

1. Looks like the best for small groups (2/3 ranks) is using P2P.

2. Re-use the mechanisms/API we already have for P2P. (not just send/recv – rcache and buffer pools are certainly useful here)

3. The tricky part is not how to place the data – it’s the sync. of N ranks. (we can choose to put this burden on the root or on the leaf)
Outline

1. Problem Statement

2. Collective operations on shared-memory

3. Using network multicast for collective operations
We have shared memory today

• Shared-memory component in Open-MPI (ompi/mca/coll/sm)
• Various closed-source collective libraries have shared-memory components

What’s missing?

1. Re-using the same buffer with RDMA (coll/sm to call ibv_reg_mr() ?)
2. Does one-size-fit-all? (how to use atomics, for example)
   *speaking of atomics - should we use atomic-add for reductions?
3. Avoiding memory copies like the plague pandemic!
4. Taking all the factors into account.
What we have today (P2P)

Send Message buffer (FIFO)

uct_mm_fifo_ctl_t

Head
Tail

uct_mm_fifo_element_t

flags
am_id
length
...

Recv Message buffer (FIFO)

uct_mm_fifo_ctl_t

Head
Tail
4 queues needed:
1+2. The existing P2P queues, for control messages (e.g. Rendezvous).
3. Fanin, for collectives like reduce or gather.
4. Fanout, for collectives like bcast and scatter.

**Fanin** Message buffer (FIFO)

**uct_mm_coll_fifo_element_t**
- pending
- am_id
- length
- ...

**uct_mm_fifo_ctl_t**
- Head
- Tail

**Fanout** Message buffer (FIFO)

**uct_mm_fifo_ctl_t**
- Head
- Tail

What was added
Multiple “modes” – Part 1

1. BATCHED mode, where buffers are written in separate cache-lines:

| element->pending = 0 | | | | |
| element->pending = 1 | | | 222p | |
| element->pending = 2 | | 111p | 222p | |
| element->pending = 3 | | 111p | 222p | 333p |

2. CENTRALIZED mode, like "batched" but with receive-side completion:

| element->pending = 0 | ???-0 | ???-0 | ???-0 |
| element->pending = 0 | ???-0 | 222-1 | ???-0 |
| element->pending = 2 | 111-1 | 222-1 | ???-0 | < rank#0 "triggers" checks
| element->pending = 3 | 111-1 | 222-1 | 333-1 |

^        ^        ^        ^
^        #1       #2       #3 -> the last byte is polled
^ by the receiver process.

The receiver process polls all these last bytes, and once all the bytes have been set - the receiver knows this operation is complete (none of the senders know).
Multiple “modes” – Part 2 (Reduction-specific)

3. LOCKED mode, where the reduction is done by the sender:

- element->pending = 0
- element->pending = 1
- element->pending = 2
- element->pending = 3

4. ATOMIC mode, same as LOCKED but using atomic operations to reduce:

- element->pending = 0
- element->pending = 1
- element->pending = 2
- element->pending = 3
<table>
<thead>
<tr>
<th>Burden is on the -</th>
<th>Mutual exclusion</th>
<th>Typically good for:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Batched</strong></td>
<td>Receiver</td>
<td>&quot;pending&quot; is atomic</td>
</tr>
<tr>
<td><strong>Centralized</strong></td>
<td>Receiver</td>
<td>not mutually excluding</td>
</tr>
<tr>
<td><strong>Locked</strong></td>
<td>Sender</td>
<td>element access uses lock</td>
</tr>
<tr>
<td><strong>Atomic</strong></td>
<td>Sender</td>
<td>element access is atomic</td>
</tr>
</tbody>
</table>
Where do these changes apply?

**UCS**
- Multi-process (pthread-)lock

**UCT**
- New endpoints + interfaces: mm_(sysv|posix)_bcast, mm_(sysv|posix)_incast

**UCP**
- The address of each process now contains these new UCT interfaces

**UCG**
- Make UCG aware of new UCT interface and use it accordingly
Some (*preliminary!) OSU results (*still work-in-progress...*)

1. x86 vs ARM

“flat” bcast/reduce, Intel Xeon 6240 (18 cores) vs. Huawei Kunpeng 920 (both at 2.6GHz).

2. P2P vs. one-to-many SM transport

Multi-level (tree-based) bcast and allreduce vs. simple P2P – both in shared memory (on a Huawei Kunpeng 920).

3. P2P vs. one-to-many SM transport

Bcast latency (PPN=64) as message size grows:

<table>
<thead>
<tr>
<th>8b</th>
<th>Xeon 6240 (x86)</th>
<th>Kunpeng 920 (ARM)</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>Bcast</td>
<td>Reduce</td>
<td>Bcast</td>
</tr>
<tr>
<td>3</td>
<td>0.6</td>
<td>0.81</td>
<td>0.18</td>
</tr>
<tr>
<td>10</td>
<td>0.73</td>
<td>0.83</td>
<td>0.37</td>
</tr>
<tr>
<td>18</td>
<td>0.86</td>
<td>0.89</td>
<td>0.37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8b</th>
<th>Bcast</th>
<th>Allreduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>P2P</td>
<td>SM</td>
</tr>
<tr>
<td>3</td>
<td>0.21</td>
<td>0.18</td>
</tr>
<tr>
<td>4</td>
<td>0.28</td>
<td>0.26</td>
</tr>
<tr>
<td>5</td>
<td>0.28</td>
<td>0.29</td>
</tr>
<tr>
<td>8</td>
<td>0.42</td>
<td>0.33</td>
</tr>
<tr>
<td>10</td>
<td>0.42</td>
<td>0.37</td>
</tr>
<tr>
<td>16</td>
<td>0.56</td>
<td>0.46</td>
</tr>
<tr>
<td>20</td>
<td>0.51</td>
<td>0.47</td>
</tr>
<tr>
<td>32</td>
<td>0.71</td>
<td>0.64</td>
</tr>
<tr>
<td>40</td>
<td>0.81</td>
<td>1.03</td>
</tr>
<tr>
<td>63</td>
<td>0.88</td>
<td>0.91</td>
</tr>
<tr>
<td>64</td>
<td>1.25</td>
<td>1.09</td>
</tr>
<tr>
<td>80</td>
<td>1.18</td>
<td>1.16</td>
</tr>
</tbody>
</table>
1. Problem Statement

2. Collective operations on shared-memory

3. Using network multicast for collective operations
Motivation

- **Multicast** is a mode of *communication* where one sender can send to multiple receivers by sending only one copy of the message.
- Higher bandwidth and utilization.
- Lower Latency on sender.
Multicast in Open-MPI

- MPI_Bcast
- MPI_Allgather
- MPI_scatter
- MPI_alltoall
Multicast Group Join

- Join a multicast group on the switch.
- Join a multicast group on the host.
join a multicast group on the switch

• IGMP snooping is a method that network switches use to identify multicast groups.
• IGMP enables switches to forward packets to the correct devices in their network.
• Any host who wish to listen to multicast group must notify the kernel and the switch.
  • Create and bind a socket to the desired Ethernet Interface.
  • Join a multicast group by sending a request via setsockopt (IP_ADD_MEMBERSHIP) to the IGMP routers.
Multicast Group Join

- Join a multicast group on the switch.
- Join a multicast group on the host.
Join a multicast group on the host.

• Multicast works only with UD QPs

• IP address ranges from 224.0.0.0 through 239.255.255.255 are considered IP Multicast addresses.

• we found an issue with Multicast over RoCEv1 and it’s being fixed by the Switch Vendor.

• Receiver QP must attach to multicast group using `ibv_attach_mcast` in order to receive packets on this group.

• **Challenge**: need to make sure that all Ranks are attached to the Multicast Group before the first Send of data, otherwise they won’t receive Connection Request/Response packets.
Multicast Interface

- ud_mcast_mlx5 inherits ud_mlx5 interface and overloads some of its operations
- ud_mcast_verbs inherits ud_verbs interface and overloads some of its operations
- Messages can be exchanged by Multicast or P2P (if we call ud_mlx5/ud_verbs)
- Root send Ctrl+Data messages via Multicast address.
- Receivers send back Ctrl messages via P2P.
One problem with UCX is that it was built for P2P connections
  - endpoint can be connected to only one endpoint

Since we have 1 message to send to all receivers – we need to allow one-to-many connection for an endpoint.
Multicast Reliability (Example)

- Message with PSN #5 didn’t arrive to destination C.
Multicast Reliability

- ACKs arrive from peer
- Resend will be triggered after timeout.
MPI_Bcast performance

The table below shows the performance comparison between Multicast and P2P communication using 32 bytes messages:

<table>
<thead>
<tr>
<th># of processes</th>
<th>Multicast (usec)</th>
<th>P2P (usec)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.48</td>
<td>0.54</td>
<td>11%</td>
</tr>
<tr>
<td>4</td>
<td>0.6</td>
<td>0.98</td>
<td>38%</td>
</tr>
<tr>
<td>6</td>
<td>0.6</td>
<td>1.3</td>
<td>53%</td>
</tr>
<tr>
<td>8</td>
<td>0.64</td>
<td>1.76</td>
<td>63%</td>
</tr>
</tbody>
</table>

The graph illustrates the latency (in usec) for different numbers of processes. The line for Multicast is consistently below that of P2P, indicating better performance.

- The more receivers we have, the more latency we save.