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

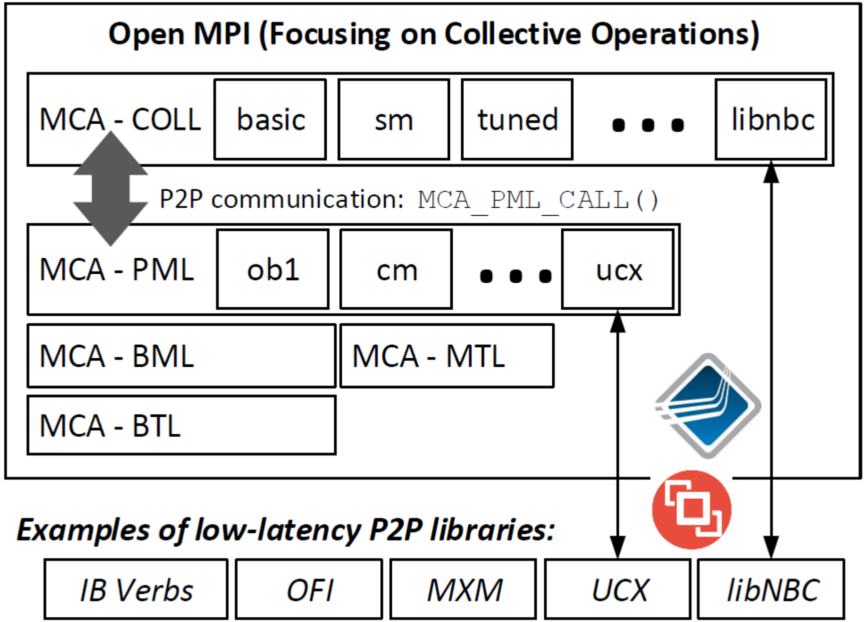
Collective operations in Open MPI

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...



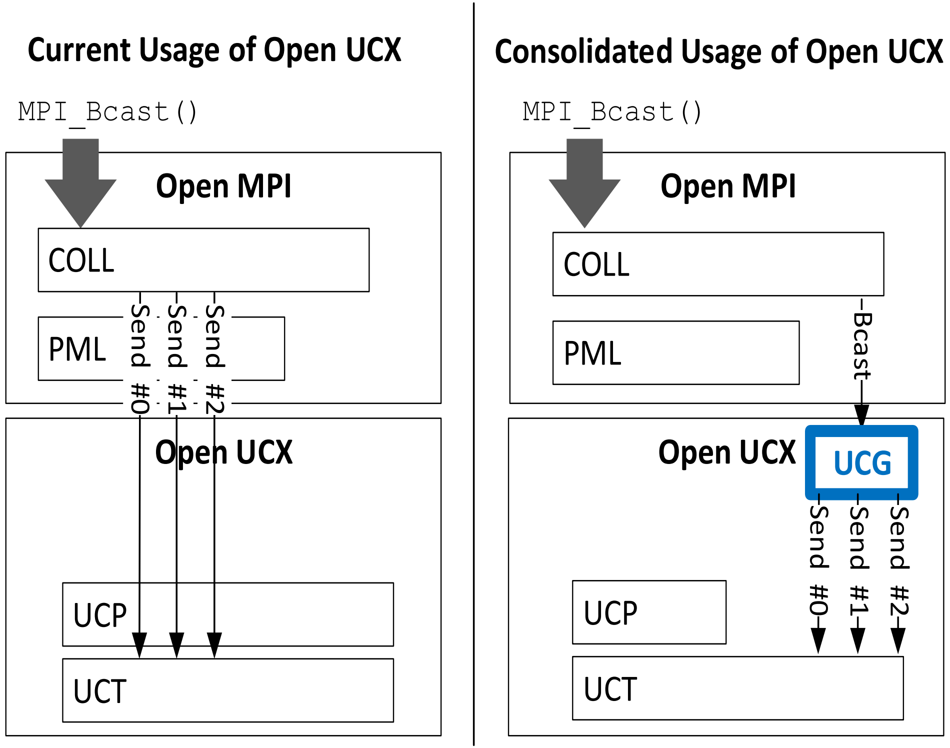
Example: basic broadcast code (from: *coll_base_bcast.c*)

```
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;
        }
    }
```

UCG Reminder

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

<u>Batch (identical) send/receives!</u>

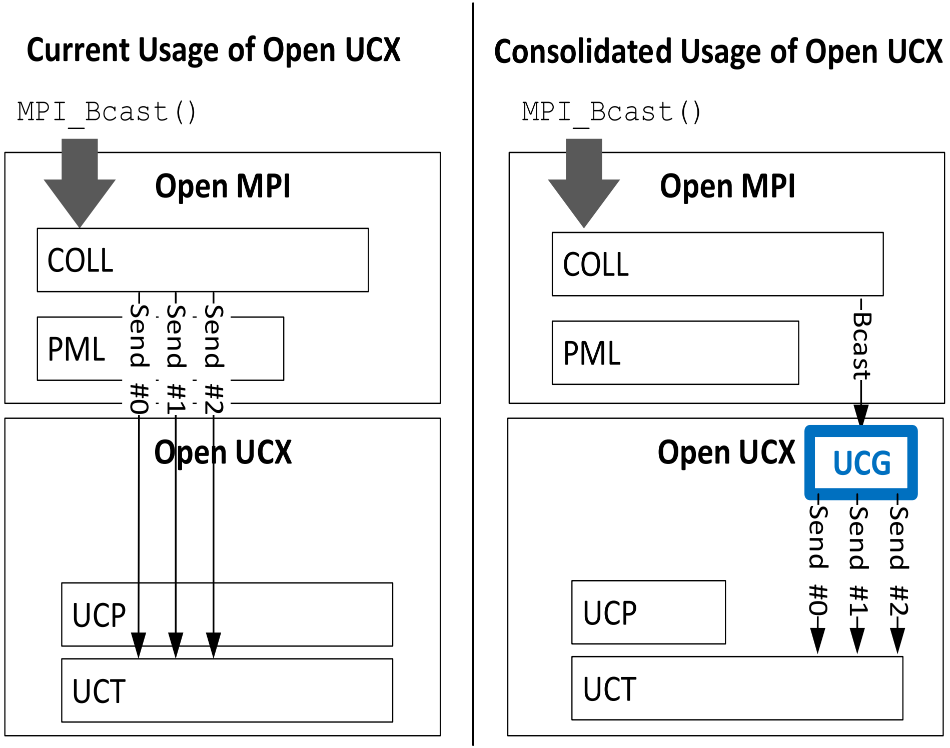


UCG Reminder

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

<u>Batch (identical) send/receives!</u>

~~Premature optimization~~ too many abstraction layers are the root of all evil.

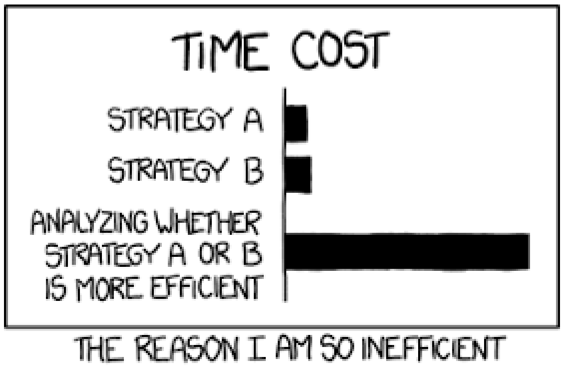


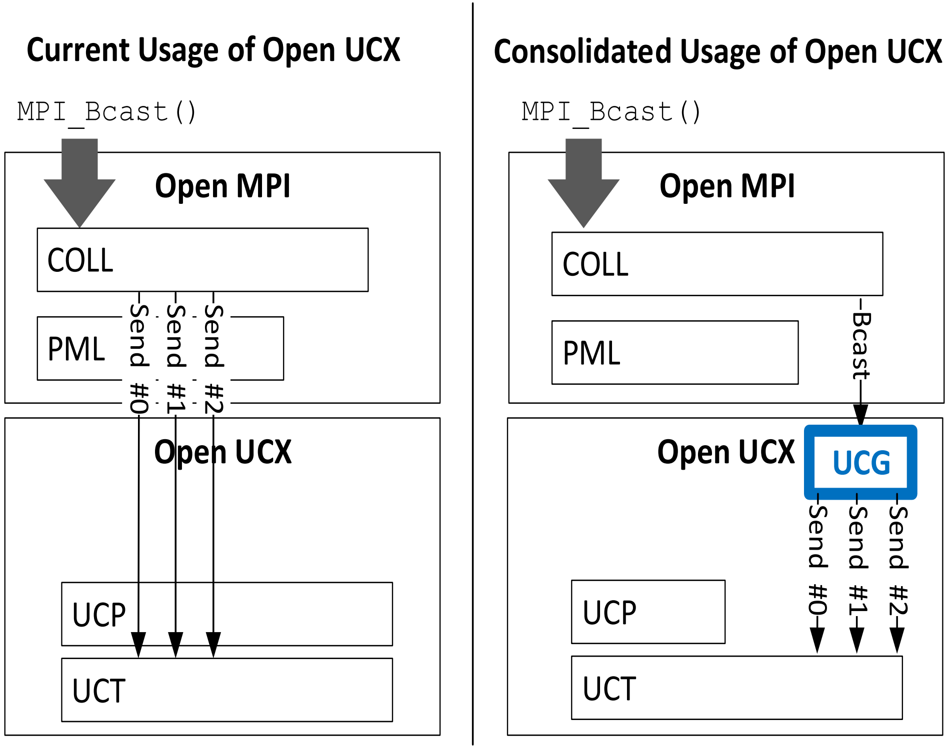
UCG Reminder

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

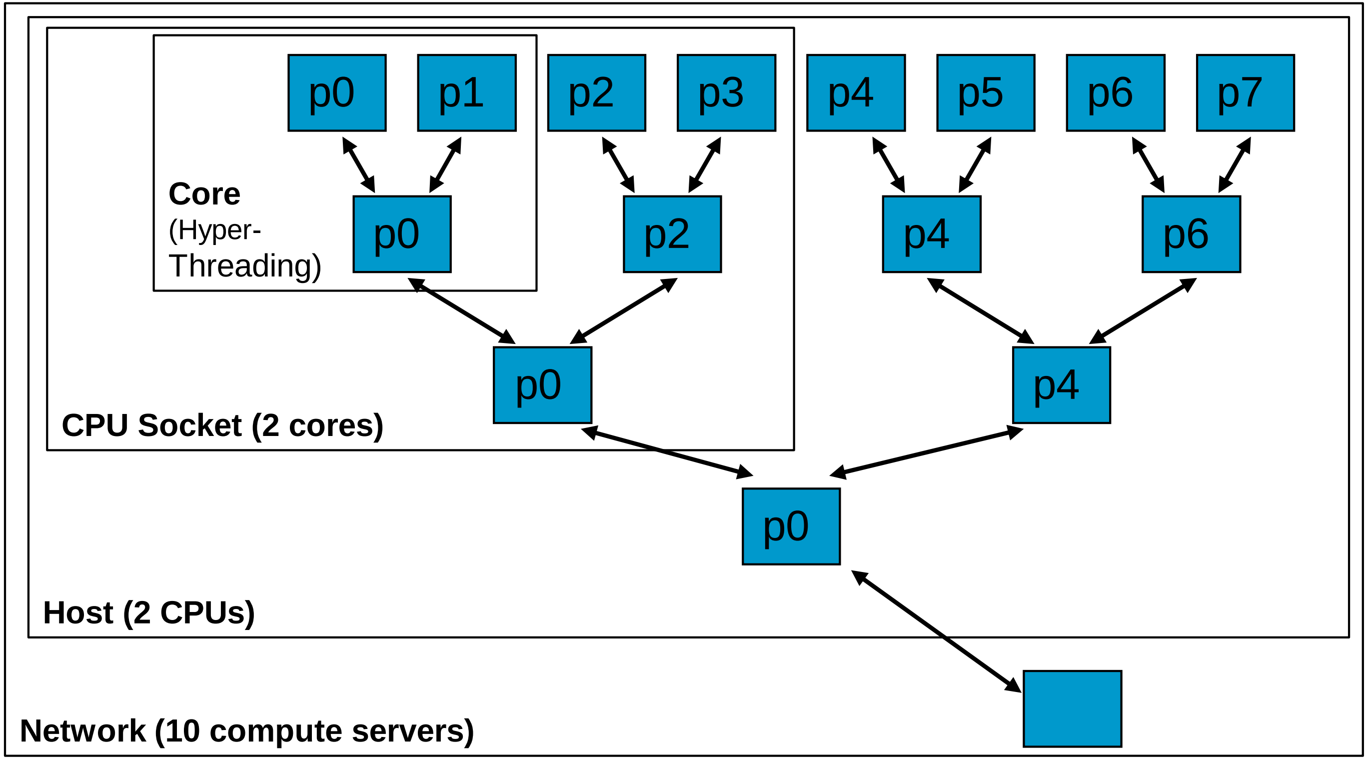
Batch (identical) send/receives!

~~Premature optimization~~ too many abstraction layers are the root of all evil.





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

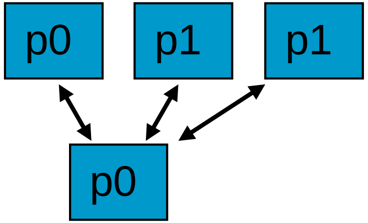
<u>Factors (not exhaustive nor prioritized):</u>

- *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 chucks of it)
- <u>What we do with the buffer afterwards</u> (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

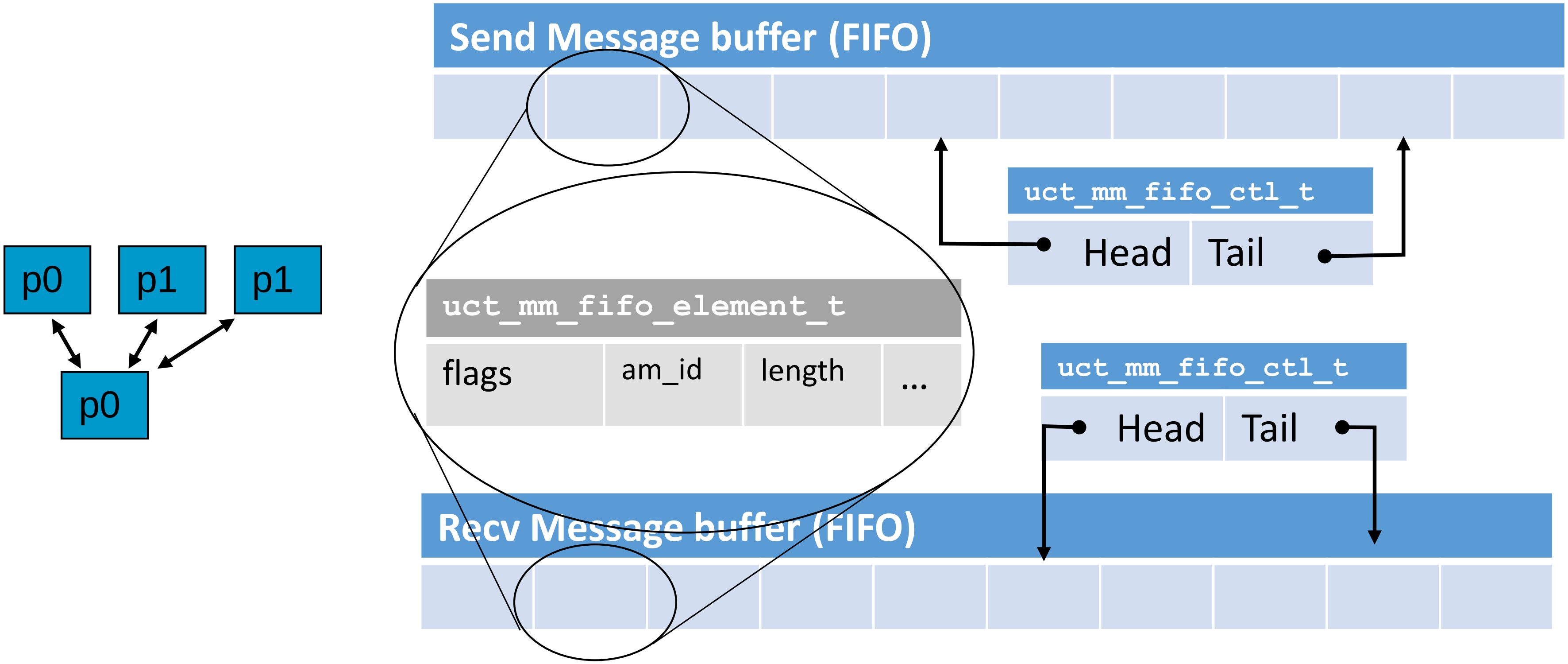
- Shared-memory component in Open-MPI (`ompi/mca/coll/sm`)
- Various closed-source collective libraries have shared-memory components

<u>What's missing?</u>

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. <u>Taking all the factors into account.</u>

What we have today (P2P)



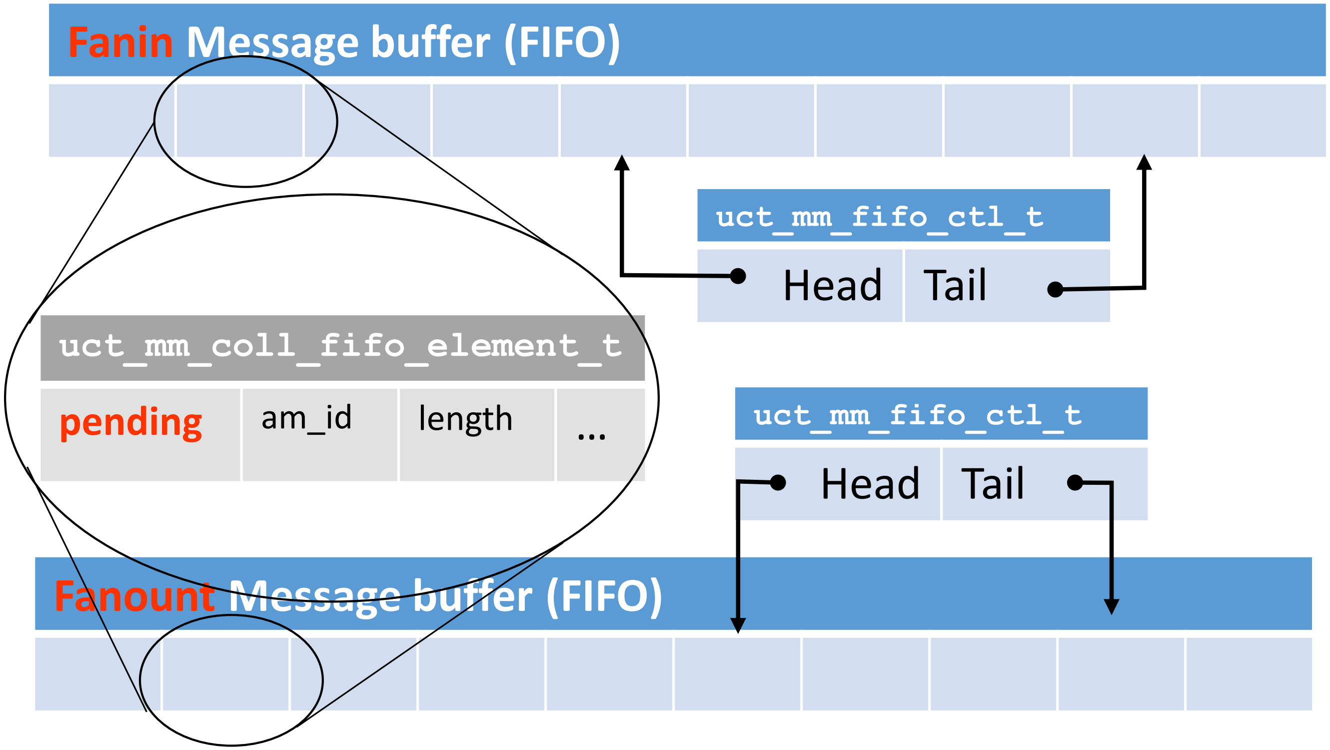
What was added

4 queues needed:

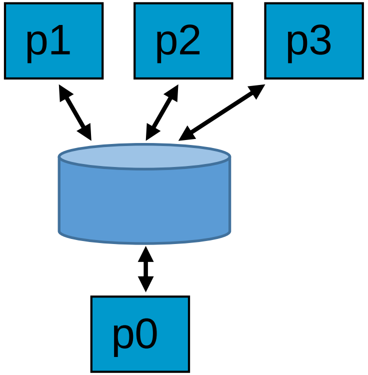
1+2. The existing P2P queues, for control messages (e.g. Rendezvous).

3. Fanin, for collectives like reduce or gather.

4. Fanout, for collectives like bcast and scatter.



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).
```

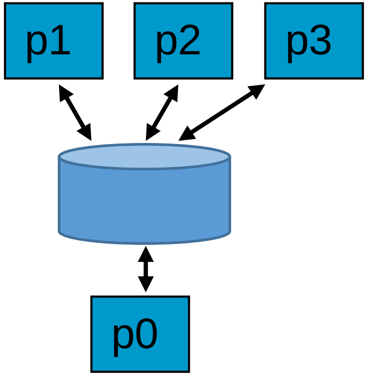
Multiple "modes" – Part 2 (Reduction-specific)

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

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

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

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



Some Comparison

	Burden is on the -	Mutual exclusion	Typically good for:
Batched	Receiver	"pending" is atomic	small size, low PPN
Centralized	Receiver	not mutually excluding	small size, high PPN
Locked	Sender	element access uses lock	large size
Atomic	Sender	element access is atomic	imbalance + some ops

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).

8b	Xeon 6240 (x86)		Kunpeng 920 (ARM)		Improvement (%)	
PPN	Bcast	Reduce	Bcast	Reduce	Bcast	Reduce
3	0.6	0.81	0.18	0.25	70.0%	69.1%
10	0.73	0.83	0.37	0.32	49.3%	61.4%
18	0.86	0.89	0.75	0.37	12.8%	58.4%

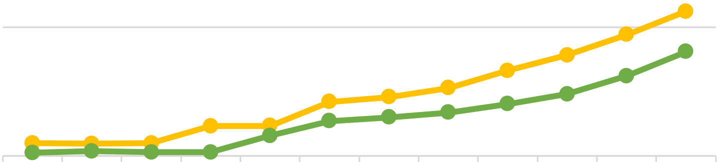
2. P2P vs. one-to-many SM transport

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

8b	Bcast			Allreduce		
PPN	P2P	SM	Improvement	P2P	SM	Improvement
3	0.21	0.18	14%	0.57	0.62	-9%
4	0.28	0.26	7%	0.72	0.70	3%
5	0.28	0.29	-4%	0.83	0.74	11%
8	0.42	0.33	21%	1.04	0.88	15%
10	0.42	0.37	12%	1.12	0.98	13%
16	0.56	0.46	18%	1.38	1.23	11%
20	0.51	0.47	8%	1.55	1.34	14%
32	0.71	0.64	10%	2.59	2.63	-2%
40	0.81	1.03	-27%	2.95	3.08	-4%
63	0.88	0.91	-3%	3.9	3.94	-1%
64	1.25	1.09	13%	3.75	3.98	-6%
80	1.18	1.16	2%	3.97	4.30	-8%

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

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



	4	8	16	32	64	128	256	512	1024	2048	4096	8192
P2P	1.26	1.25	1.26	1.71	1.72	2.64	2.87	3.38	4.61	6.08	8.80	13.33
SM	1.06	1.09	1.07	1.07	1.44	1.87	2.00	2.18	2.54	3.03	4.20	6.49

HUAWEI TEL AVIV RESEARCH CENTER

Outline

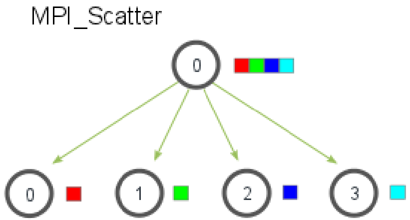
1. Problem Statement
2. Collective operations on shared-memory
3. <u>Using network multicast for collective operations</u>

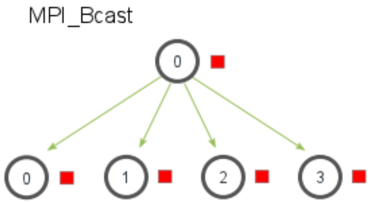
Motivation

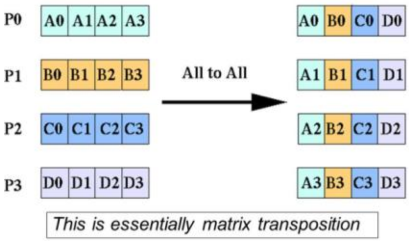
- **Multicast** is a mode of **communication** where one sender can send to multiple receivers by sending only one copy of the message
- Higher bandwidth and utilization
- Lower Latency on sender

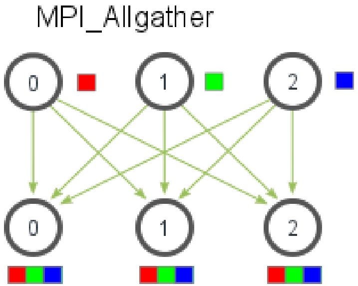
Multicast in Open-MPI

- 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.

```
Heron [standalone: master] (config) # show ip igmp snooping groups

-----------------------------------------------------
Vlan ID      Group           St/Dyn      Ports
-----------------------------------------------------
1            239.255.255.240 Dyn         Eth1/11,  Eth1/5

Total Num of Dynamic Group Addresses: 1
Total Num of Static Group Addresses : 0
```

Multicast Group Join

- Join a multicast group on the switch.
- <u>Join a multicast group on the host.</u>

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

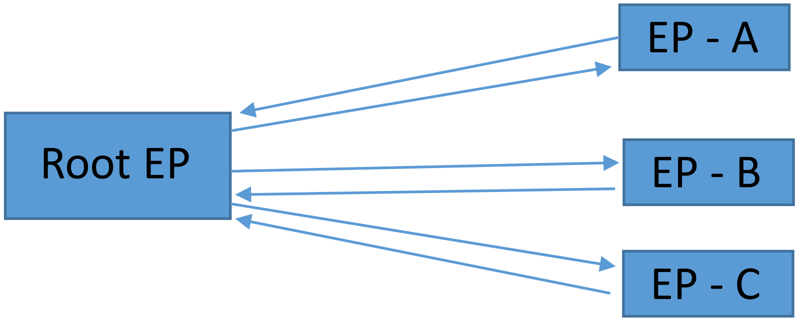
- **ud_mlx5**
  - send_ctrl
  - create_qp
  - .....
- unpack_addr
- Iface_get_addr
- ep_get_addr
- ep_create
- iface_query

ud_mcast_verbs

- **ud_verbs**
  - send_ctrl
  - create_qp
  - .....
- unpack_addr
- Iface_get_addr
- ep_get_addr
- ep_create
- iface_query

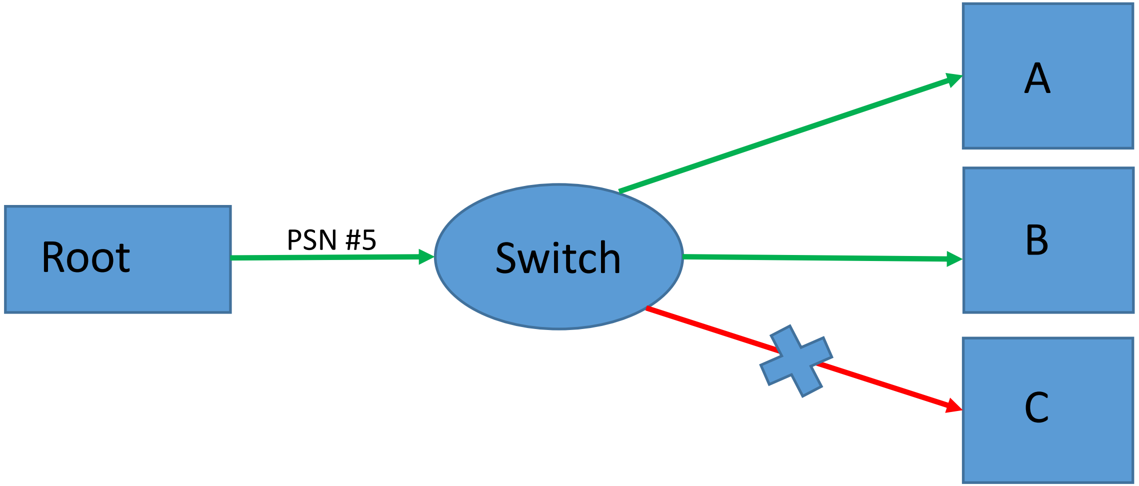
- ud_mcast_mlx5 inherits ud_mlx5 interface and overloads some of its operations
- ud_mcast_verbs inherits ud_verbs interface and overloads some of its operations
- Messages can be exchanged by Multicast or P2P (if we call ud_mlx5/ud_verbs)
- Root send Ctrl+Data messages via Multicast address.
- Receivers send back Ctrl messages via P2P.

Multicast Endpoint



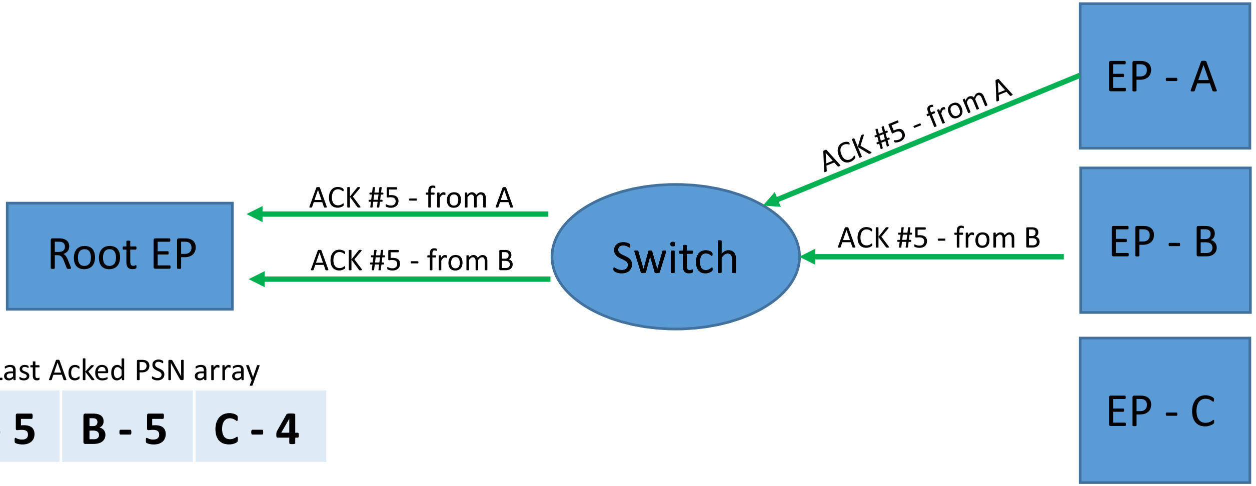
- One problem with UCX is that it was built for P2P connections
  - endpoint can be connected to only one endpoint
- Since we have 1 message to send to all receivers – we need to allow one-to-many connection for an endpoint.

Multicast Reliability (Example)



- Message with PSN #5 didn't arrive to destination C.

Multicast Reliability



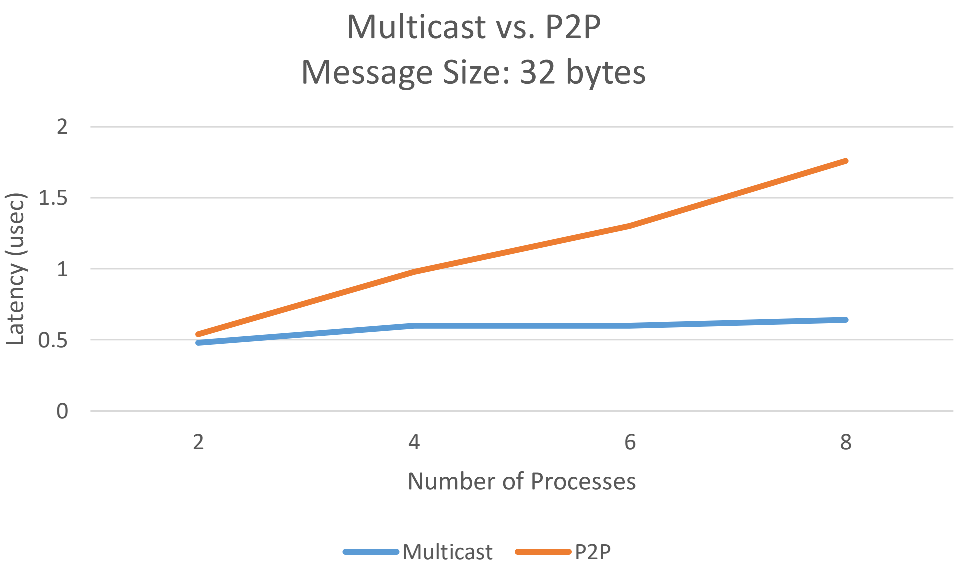
Last Acked PSN array

A - 5	**B - 5**	**C - 4**

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

MPI_Bcast performance

# of processes	Multicast	P2P	Improvement
2	0.48	0.54	11%
4	0.6	0.98	38%
6	0.6	1.3	53%
8	0.64	1.76	63%



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

www.huawei.com