Impatient with mpirun? Me too :-/

MVAPICH User Group Meeting

Adam Moody

August 16, 2016

LLNL-PRES-700424

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

What'chu talkin' 'bout?!

```
time srun -n 73728 -N 2048 osu_init
```

6:03	3:21	1:50
Intel MPI 5.1	Open MPI 2.0.0	MVAPICH 2.2

Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
Dual-socket, 18 cores per chip
128 GB memory / node
Intel Omni-Path (PSM2)

Benchmark: Init, Barrier, Finalize

```
int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

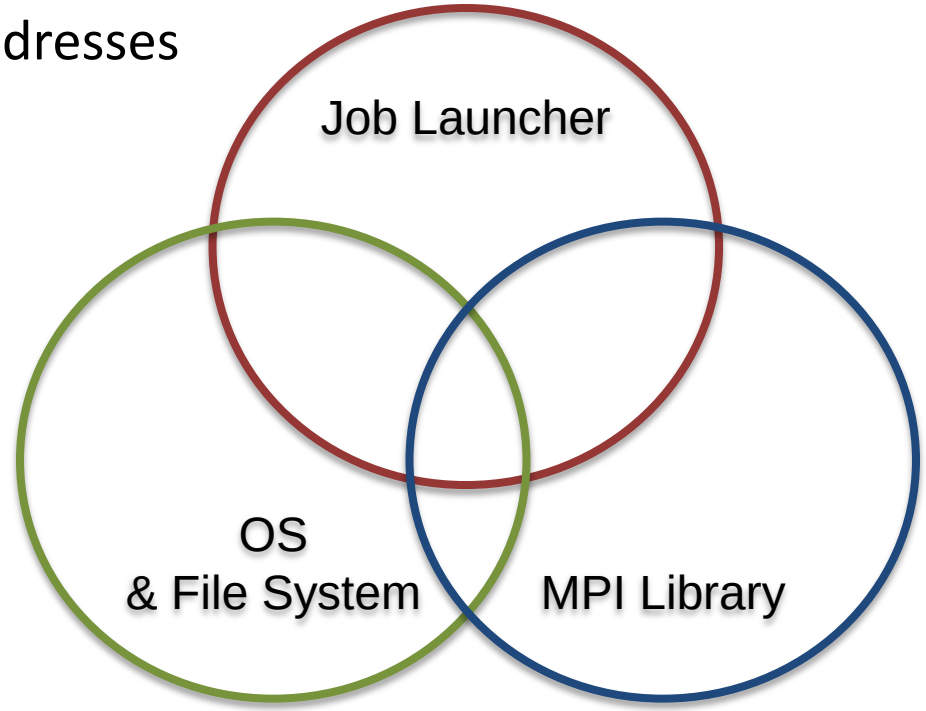
Upon closer inspection, MPI_Init dominates

```
time srun -n 73728 -N 2048 osu_initbarfin
```

	Intel MPI 5.1	Open MPI 2.0.0	MVAPICH2 2.2
MPI_Init	94% 338.8	91% 181.0	88% 96.4
MPI_Barrier	5.8	2.7	0.0
MPI_Finalize	5.7	1.6	2.1
Other	11.7	14.7	11.5
Total (secs)	362.0	200.0	110.0

Five major tasks to start an MPI job, distributed among different components

1. Broadcast commands and fork processes on target nodes
2. Read application binary from file system
3. Discover global environment across processes
4. Exchange process network addresses
5. Connect processes



Baseline profile of MPI_Init in MV2-2.2

```
    /-- proc_sync_1: 27.919508
    /-- populate_vcs: 21.167241
    /-- coll_arch_get: 0.000000
    /-- coll_arch_init: 1.224993
        /-- shm_coll_init: 6.170572
        /-- shm_coll_mmap: 0.000050
        /-- shm_coll_unlink: 0.000082
    /-- shm_coll: 6.170727
    /-- psm_bcast_uuid: 0.442145
    /-- psm_init: 0.000139
    /-- psm_ep_open: 0.068458
    /-- psm_mq_init: 0.000003
    /-- psm_ep_allgather: 29.125557
    /-- psm_ep_connect: 10.285549
    /-- psm_other_init: 0.008648
psm_doinit: 96.413171 secs
```

Focus on the initial barrier

```
    /-- proc_sync_1: 27.919508
    /-- populate_vcs: 21.167241
    /-- coll_arch_get: 0.000000
    /-- coll_arch_init: 1.224993
        /-- shm_coll_init: 6.170572
        /-- shm_coll_mmap: 0.000050
        /-- shm_coll_unlink: 0.000082
    /-- shm_coll: 6.170727
    /-- psm_bcast_uuid: 0.442145
    /-- psm_init: 0.000139
    /-- psm_ep_open: 0.068458
    /-- psm_mq_init: 0.000003
    /-- psm_ep_allgather: 29.125557
    /-- psm_ep_connect: 10.285549
    /-- psm_other_init: 0.008648
psm_doinit: 96.413171 secs
```

Focus on the initial barrier

- First run in a new allocation always slow

```
/-- proc_sync_1: 27.919508
```

- Immediate, subsequent runs are faster but variable

```
/-- proc_sync_1: 8.599889
/-- proc_sync_1: 3.639224
/-- proc_sync_1: 3.611729
```

- If we wait long enough, subsequent runs are always slow
- If we flush page cache between runs, even immediate subsequent runs are slow

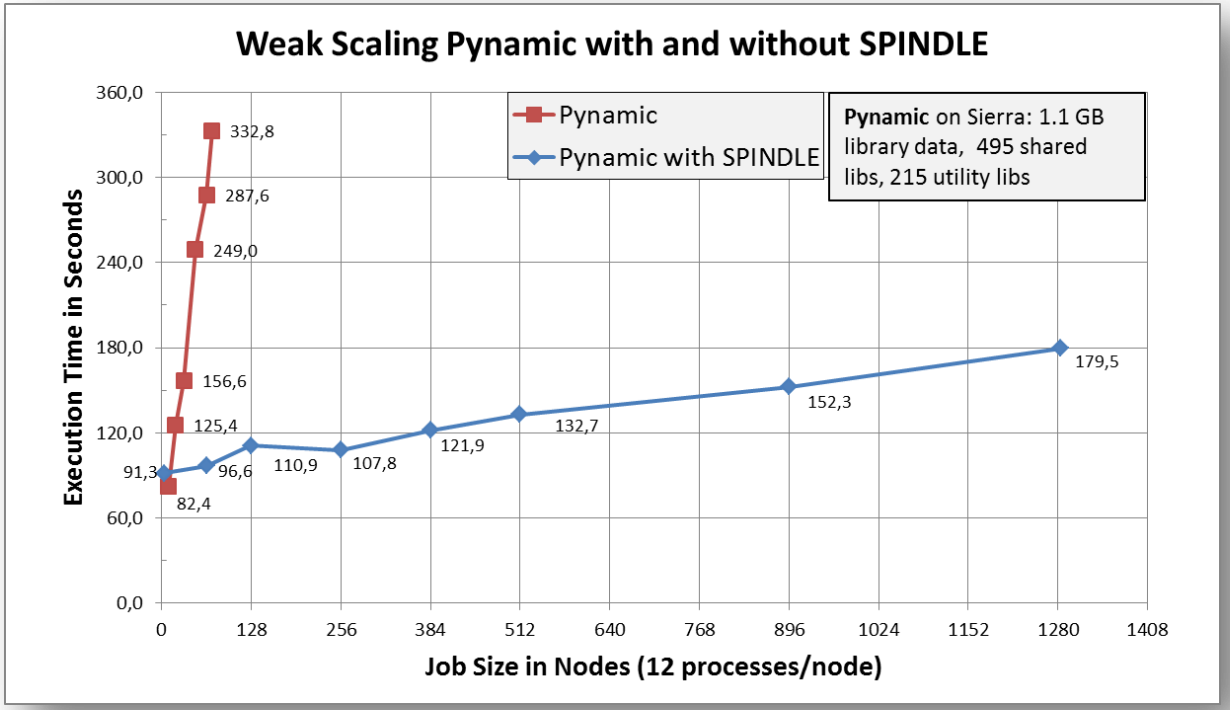
Remove RPATH from executable and all libraries, copy everything to local storage on compute nodes

```
    /-- proc_sync_1: 0.377670
    /-- populate_vcs: 20.792887
    /-- coll_arch_get: 0.000000
    /-- coll_arch_init: 1.228093
        /-- shm_coll_init: 6.010748
        /-- shm_coll_mmap: 0.000040
        /-- shm_coll_unlink: 0.000043
    /-- shm_coll: 6.010848
    /-- psm_bcast_uuid: 0.390050
    /-- psm_init: 0.000158
    /-- psm_ep_open: 0.069987
    /-- psm_mq_init: 0.000003
    /-- psm_ep_allgather: 29.084990
    /-- psm_ep_connect: 10.261098
    /-- psm_other_init: 0.008617
psm_doinit: 68.224546
```

27.5 secs faster

Variable cost eliminated by copying binary and libs to ramdisk

- Solutions:
  - Spindle (http://computation.llnl.gov/projects/spindle)
  - "Spindle-lite" built into mpirun
  - Prestaging of executables / libs via resource manager



To understand the other costs, look at PMI2

```
    /-- proc_sync_1: 0.377670
    /-- populate_vcs: 20.792887
    /-- coll_arch_get: 0.000000
    /-- coll_arch_init: 1.228093
        /-- shm_coll_init: 6.010748
        /-- shm_coll_mmap: 0.000040
        /-- shm_coll_unlink: 0.000043
    /-- shm_coll: 6.010848
    /-- psm_bcast_uuid: 0.390050
    /-- psm_init: 0.000158
    /-- psm_ep_open: 0.069987
    /-- psm_mq_init: 0.000003
    /-- psm_ep_allgather: 29.084990
    /-- psm_ep_connect: 10.261098
    /-- psm_other_init: 0.008617
psm_doinit: 68.224546
```

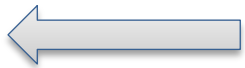
Intro to PMI2

- Standard programming interface, called by MPI library
- Informs each MPI proc of its rank and size
  - PMI2_Init(&spawned, &size, &rank, &appnum)
- Implements a key/value store across MPI procs
  - PMI2_Job_GetId(&jobid, sizeof(jobid))
  - PMI2_KVS_Put(key, value)
  - PMI2_KVS_Fence()
  - PMI2_KVS_Get(jobid, src_id, key, &value)
- Often implemented by job launcher (e.g., SLURM or mpirun)

Example allgather with PMI2 in MPI_Init

```
int spawned, size, rank, appnum;
PMI2_Init(&spawned, &size, &rank, &appnum);

char jobid[1024];
PMI2_Job_GetId(&jobid, sizeof(jobid));

char key[1024], value[1024];
sprintf(key, “%d”, rank);
sprintf(value, “%s”, address)
PMI2_KVS_Put(key, value);

PMI2_KVS_Fence();

for (int i = 0; i < size; i++) {
   sprintf(key, “%d”, i);
   PMI2_KVS_Get(jobid, i, key, value);
    // process value for MPI rank i
}
```

Initialize and get our rank and the job size



Need this to call Get

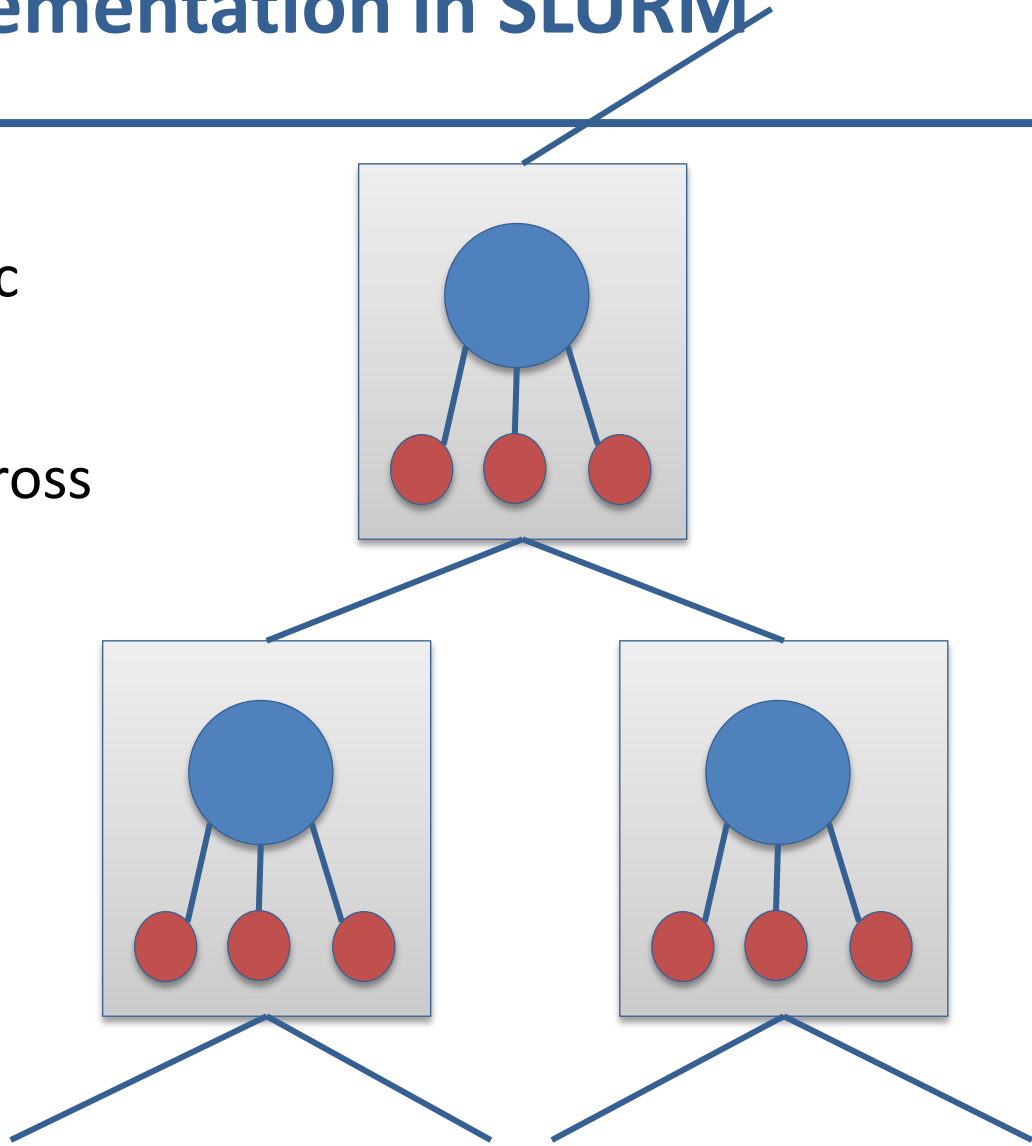
Put our value, indexed by our rank

Global exchange

Get value put by every rank in the job

Tree-based PMI2 implementation in SLURM

- Put:
  cache key/values in MPI proc
- Fence:
  allgather (gather + bcast) across stepd tree
- Get:
  ask local stepd for key/value

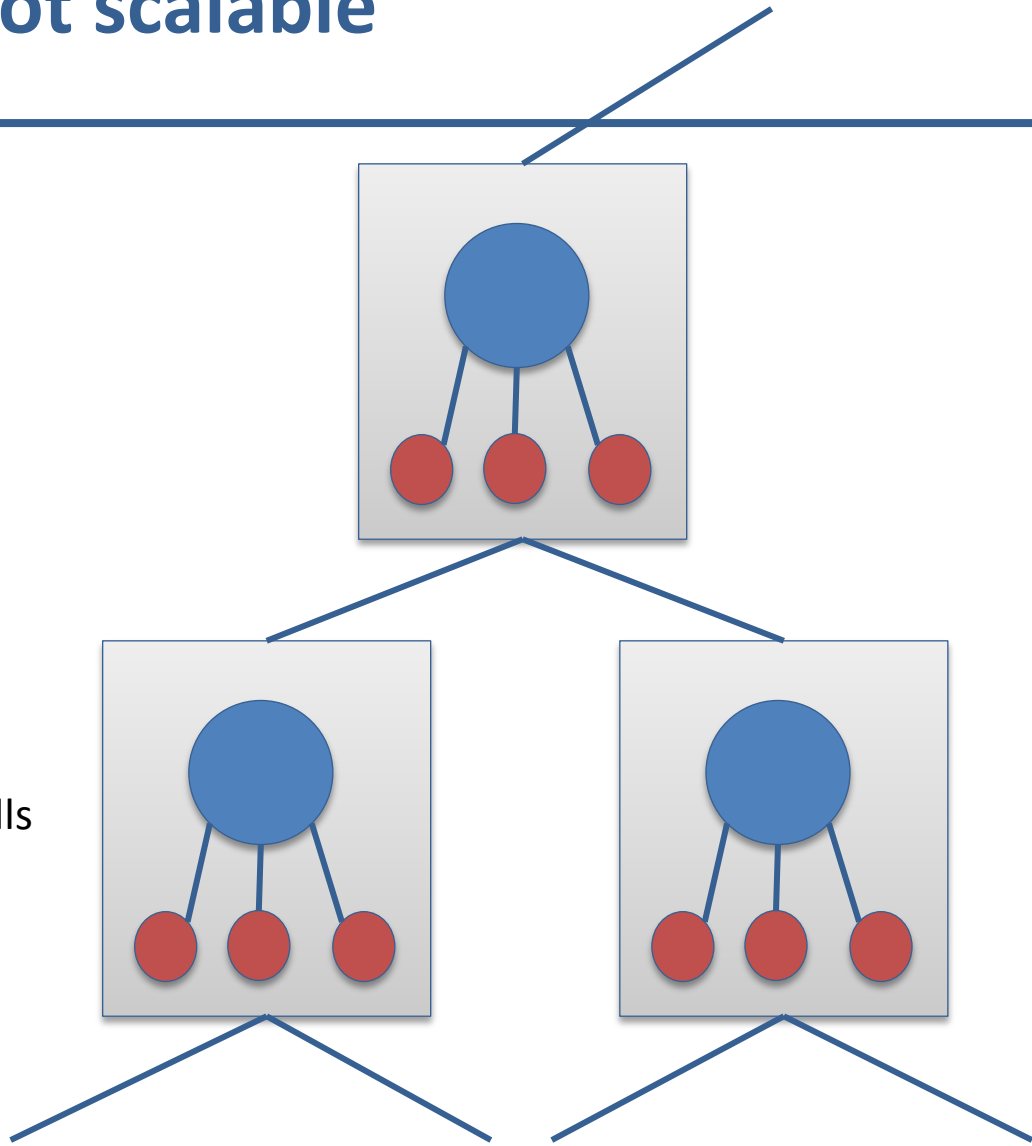


Allgather via PMI2 is not scalable

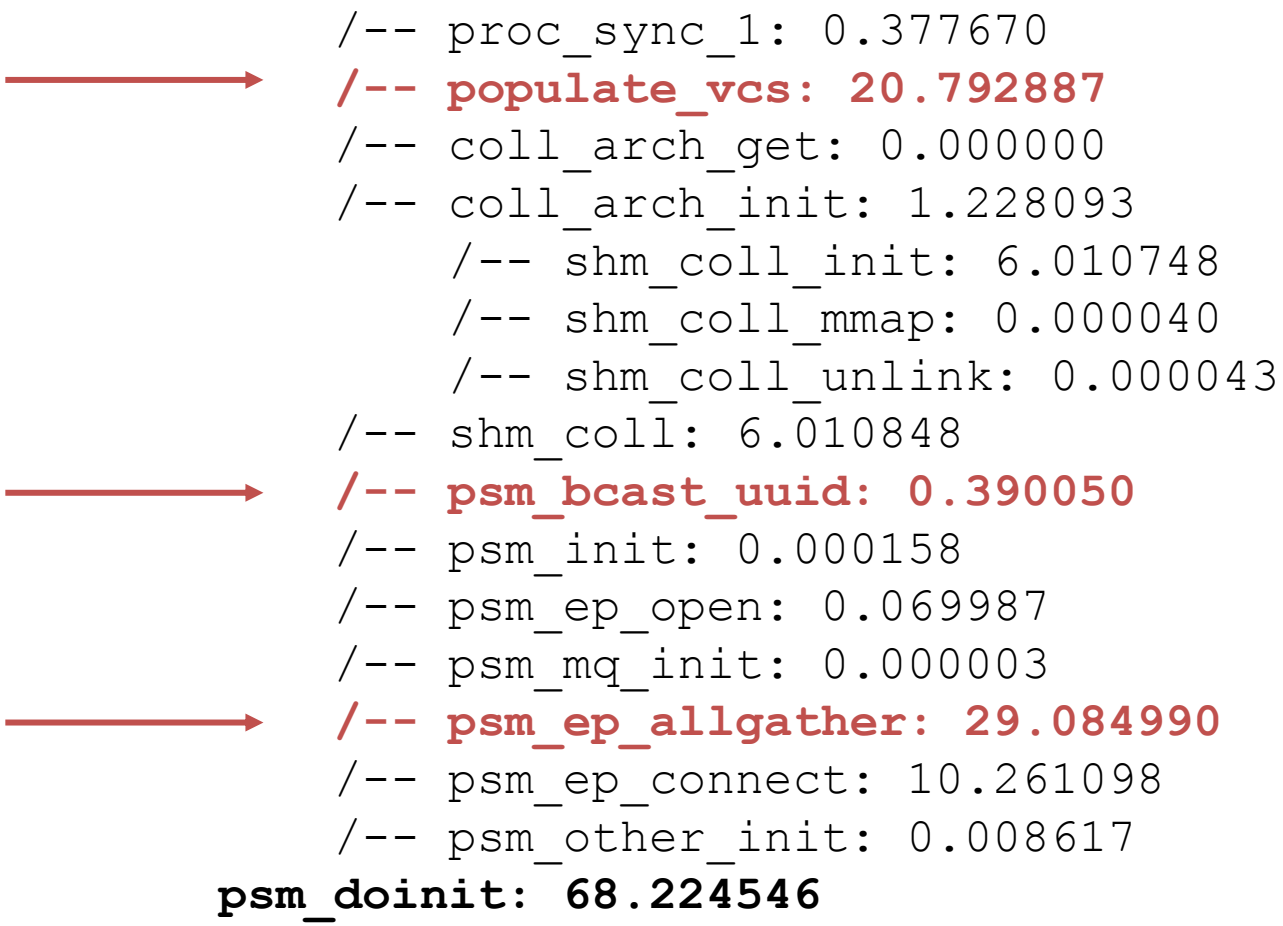
- Assume
  - P MPI procs in job
  - N nodes
  - p = P / N procs per node
- Then a for loop like this

```
for (int i=0; i < P; i++)
{
  sprintf(key, "%d", i);
  PMI2_Get(key, value);
}
```

- Scales as O(p * P)
  - Each proc on a node issues P Get calls
- But p=36 and P=73728
- p * P = 2.65 million Gets / stepd
- Yikes!



PMI2 exchanges account for 50 of 68 seconds

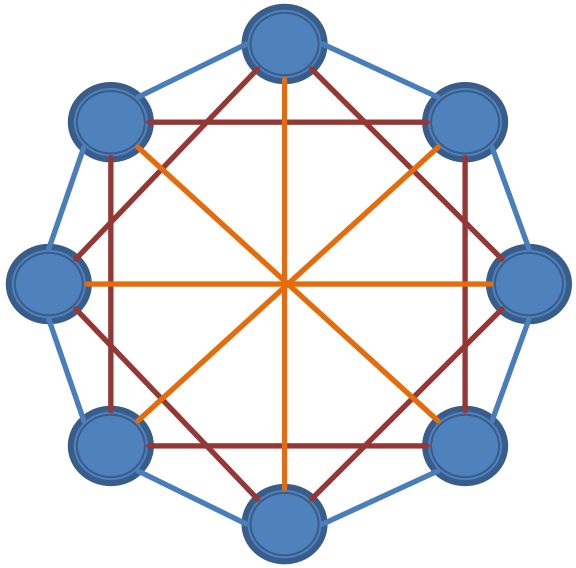


MPI needs collectives to create its connections, but how to do collectives without connections?

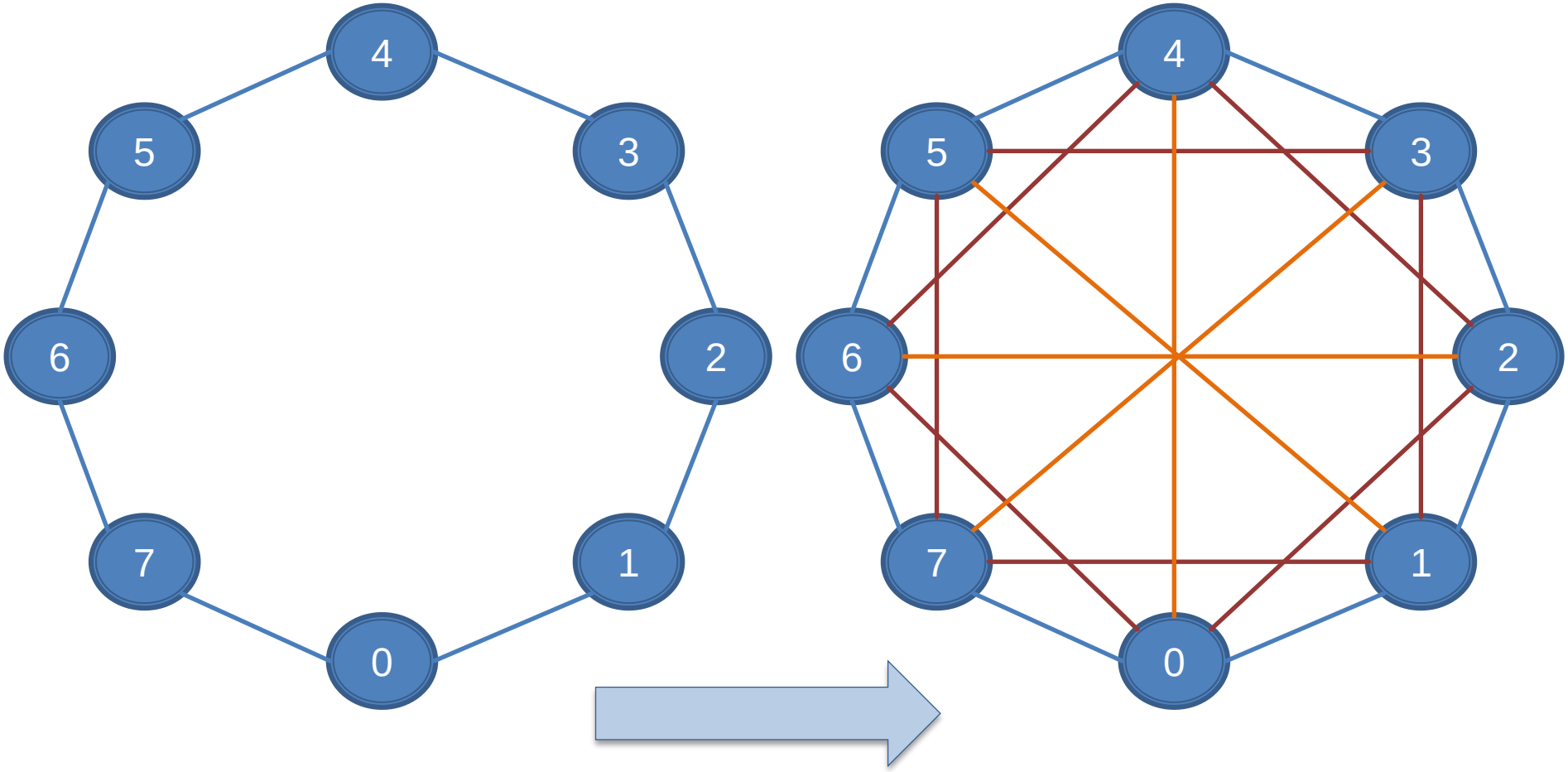
- Catch 22
- Answer
  - Use a temporary communication network to bootstrap MPI connections
- spawnnet – a fast, temporary network communication interface to start MPI jobs

spawnnet: https://github.com/llnl/spawnnet

- Low-latency communication
  - TCP sockets w/ TCP_NODELAY
  - IBUD
- Light-weight groups stored as binomial graphs
  - Each proc connects to 2*log(P) others
  - O(log P) time and space
- Split into subgroups using bitonic sort (like MPI_Comm_split)
  - O(log P * log P) time
- Collectives over binomial graphs
  - Barrier
  - Broadcast
  - Allreduce
  - Allgather
  - Ring-based shift
- Bootstrap initial group from a ring



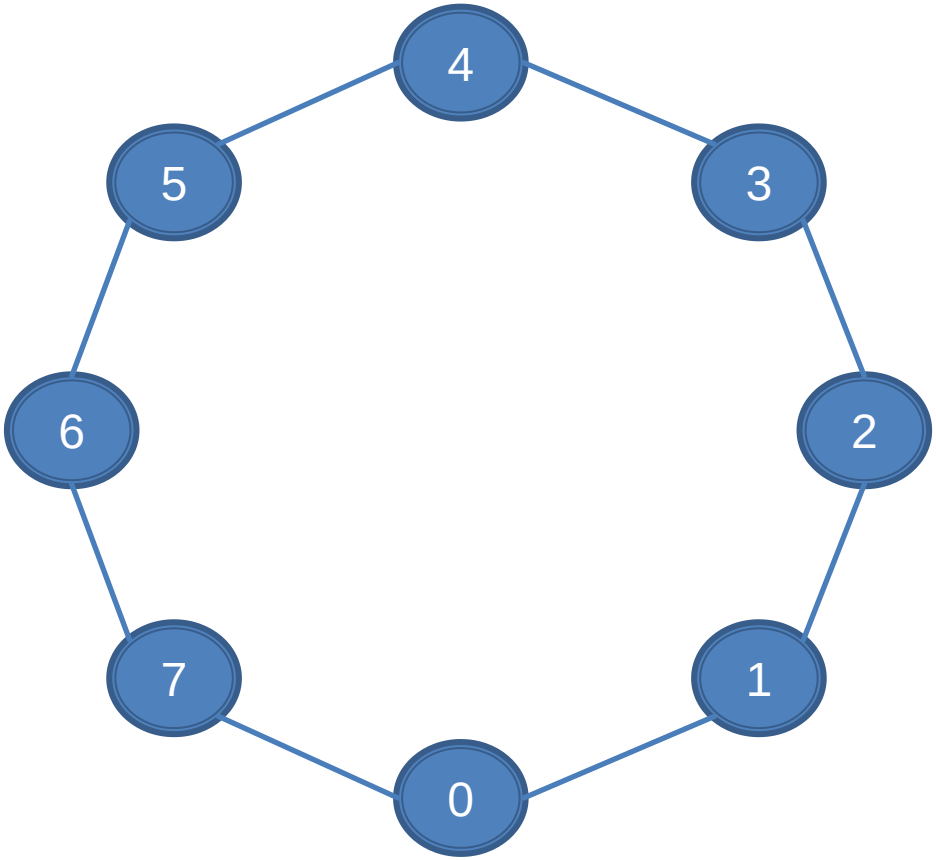
We can build a binomial graph from a ring in O(log P) time



How to build a binomial graph from a ring in O(log P) time

Step 1:

Given addresses of left and right neighbors, connect to form a ring



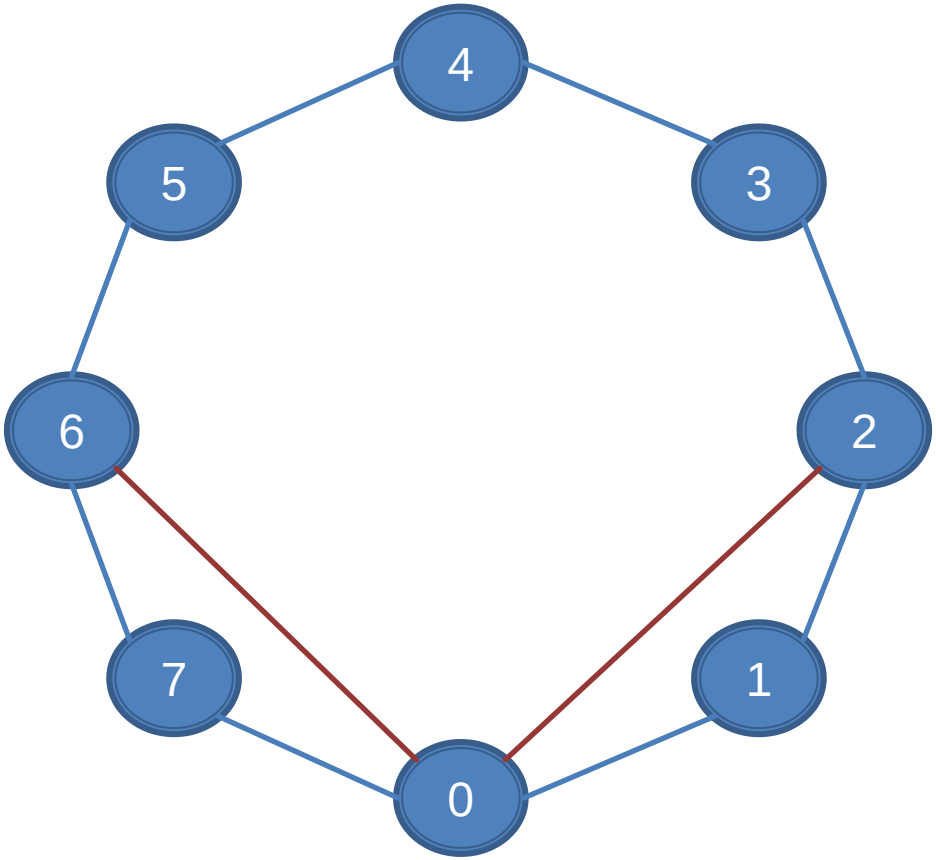
How to build a binomial graph from a ring in O(log P) time

Step 2 (rank 0):

After connecting, rank 0 receives address of rank 2 from rank 1.

Gets address of rank 6 from rank 7.

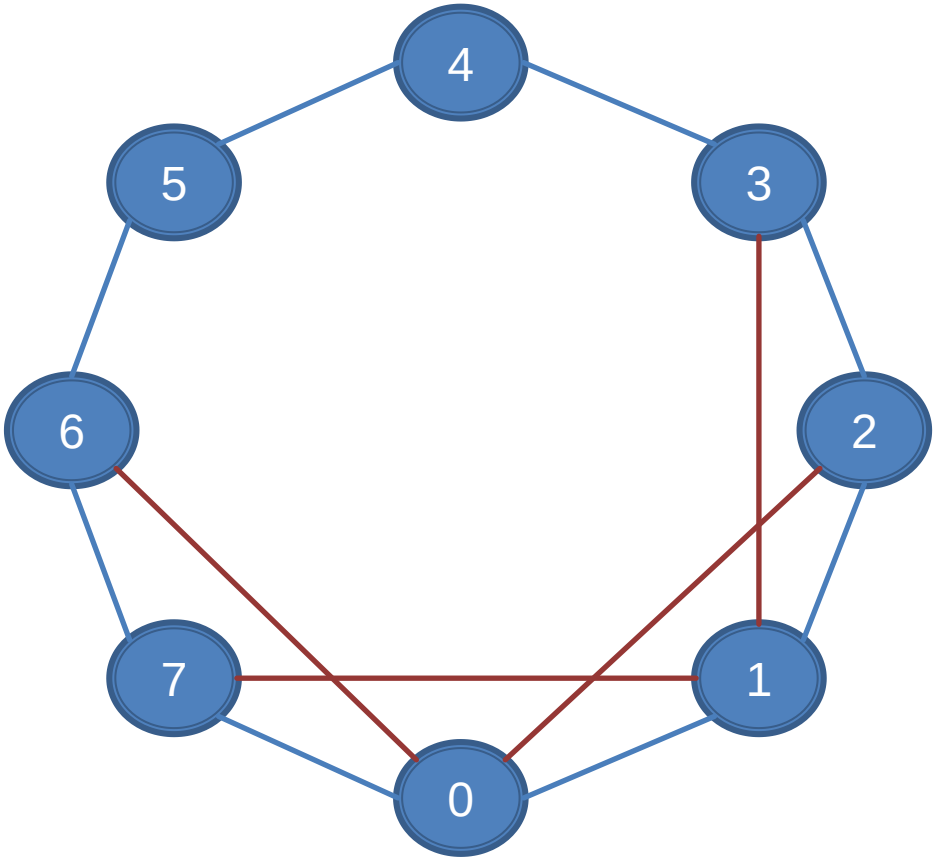
Connects to ranks 2 and 6.



How to build a binomial graph from a ring in O(log P) time

Step 2 (rank 1):

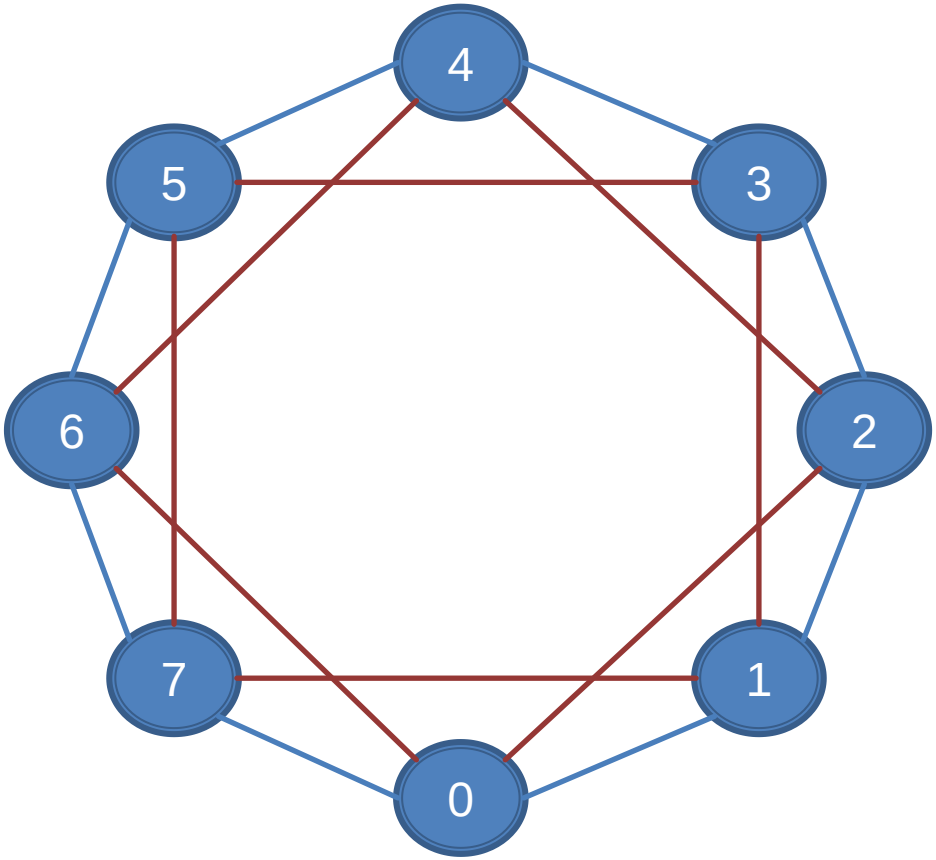
In parallel, rank 1 connects to ranks 3 and 7.



How to build a binomial graph from a ring in O(log P) time

Step 2 (all):

In fact, all ranks create similar connections in parallel.

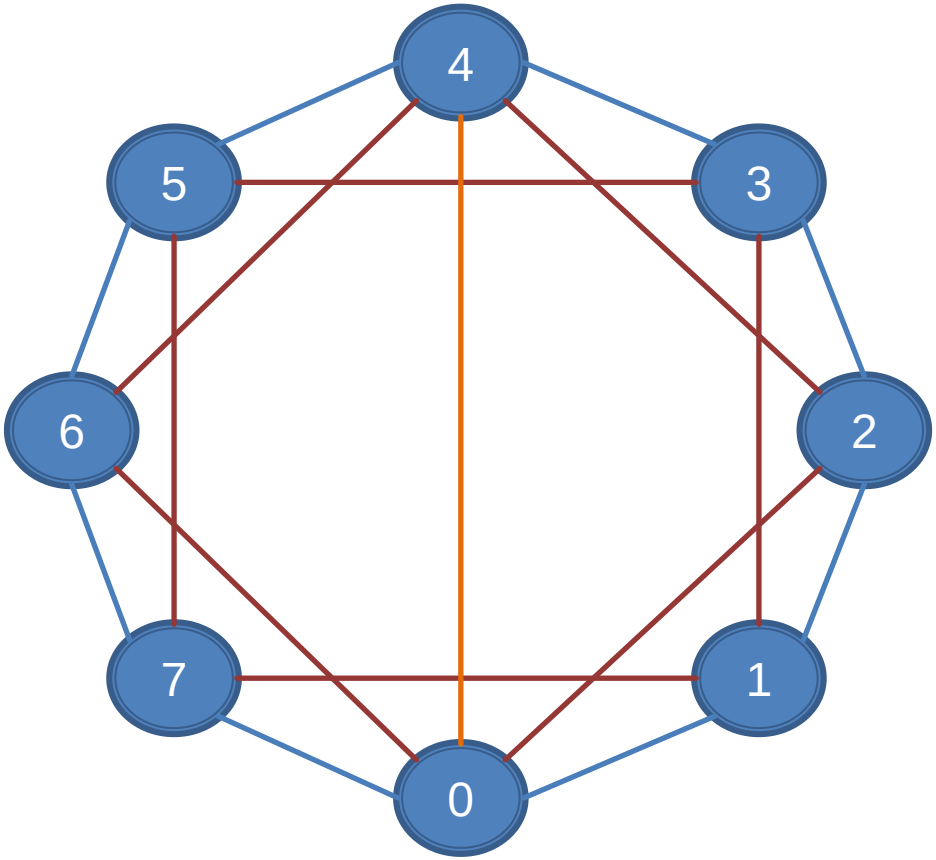


How to build a binomial graph from a ring in O(log P) time

Step 3 (rank 0):

After connecting, rank 0 receives address of rank 4 from rank 2.

Connects to rank 4.

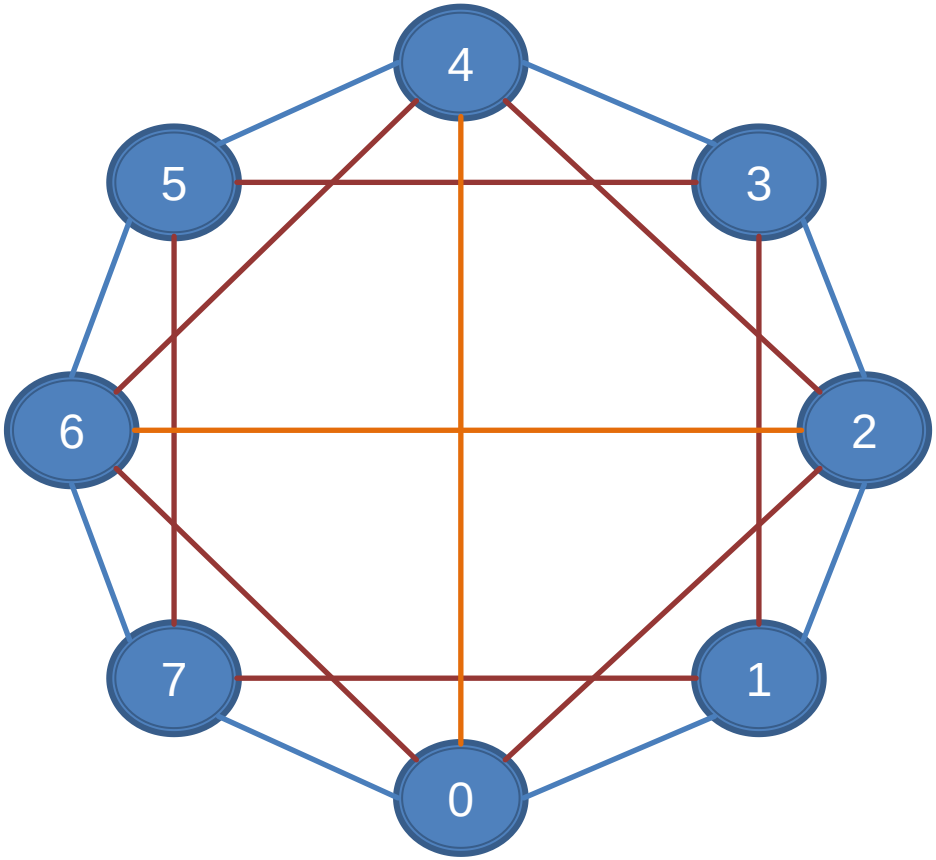


How to build a binomial graph from a ring in O(log P) time

Step 3 (rank 2):

Rank 2 gets address of rank 6 from rank 0.

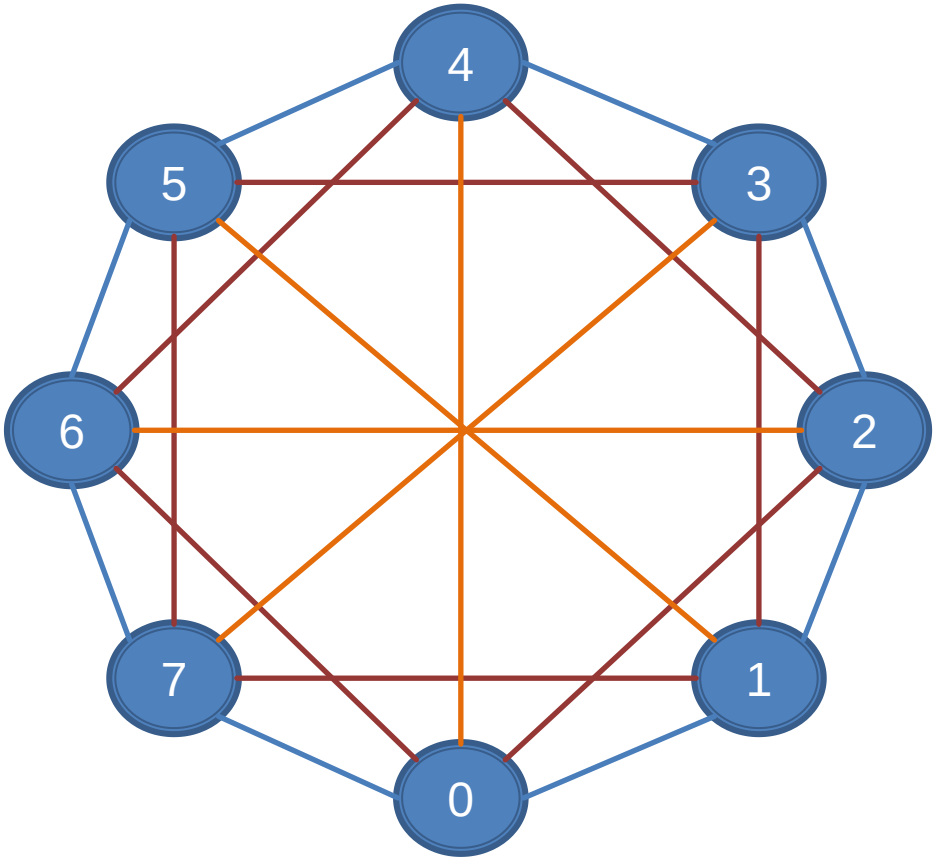
Connects to rank 6.



How to build a binomial graph from a ring in O(log P) time

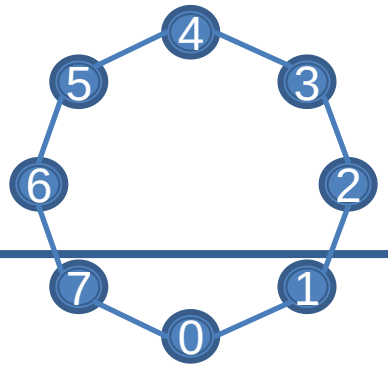
Step 3 (all):

Each rank makes a similar connection.



Given P processes, in step D, rank X receives addresses of and connects to ranks (X +/- 2^{D-1}) mod P

And to get the ring, we use PMI2




```
char key[1024], value[1024];
sprintf(key, “%d”, rank);
sprintf(value, “%s”, address)
PMI2_KVS_Put(key, value);
```

Put our address, indexed by our rank

```
PMI2_KVS_Fence();
```

Exchange key/values

```
char left_key[1024], left_val[1024];
char right_key[1024], right_val[1024];

int left_rank = (rank + size – 1) % size;
int right_rank = (rank + size + 1) % size;

sprintf(left_key, “%d”, left_rank);
PMI2_KVS_Get(jobid, left_rank, left_key, left_val);
```

Get address of left rank

```
sprintf(right_key, “%d”, right_rank);
PMI2_KVS_Get(jobid, right_rank, right_key, right_val);
```

Get address of right rank

Reduces complexity from O(p * P) to O(p + P)

gather P addresses to each node, then each proc issues 2 Get calls

Replacing PMI2 with spawnnet saves 50 seconds

```
    /-- proc_sync_1: 0.394532
        /-- spawn_endpoint_open_ibud: 0.019523
        /-- proc_sync_2: 0.022202
        /-- ring_exchange_pmiputget: 0.381654
        /-- create_ring: 4.286503
        /-- create_world: 0.000000
        /-- create_node: 0.102327
        /-- create_leader: 0.075720
    /-- create_groups: 4.868432
    /-- populate_vcs: 0.506299
    /-- coll_arch_get: 0.661773
    /-- coll_arch_init: 0.000351
        /-- shm_coll_init: 0.041892
        /-- shm_coll_mmap: 0.000041
        /-- shm_coll_unlink: 0.000296
    /-- shm_coll: 0.042238
    /-- psm_bcast_uuid: 0.265881
    /-- psm_init: 0.000112
    /-- psm_ep_open: 0.073720
    /-- psm_mq_init: 0.000002
    /-- psm_ep_allgather: 1.173816
    /-- psm_ep_connect: 10.470668
    /-- psm_other_init: 0.008533
    /-- teardown_lwgrp: 0.089755
psm_doinit: 18.575852
```

+4.9 seconds to setup light-weight groups

-19.5 secs for faster populate_vcs

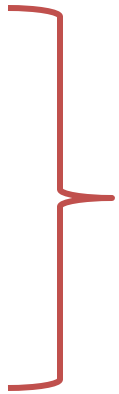
-6.0 secs for faster tuned collectives init

-28.0 secs for faster psm epid allgather

+0.1 seconds to teardown light-weight groups

But it's a shame about that +5 sec to create the groups!

```
    /-- proc_sync_1: 0.394532
        /-- spawn_endpoint_open_ibud: 0.019523
        /-- proc_sync_2: 0.022202
        /-- ring_exchange_pmiputget: 0.381654
        /-- create_ring: 4.286503
        /-- create_world: 0.000000
        /-- create_node: 0.102327
        /-- create_leader: 0.075720
    /-- create_groups: 4.868432
    /-- populate_vcs: 0.506299
    .
    .
    .
psm_doinit: 18.575852
```



+4.9 seconds to setup light-weight groups

Switch from PMI2 Put/Get to PMIX_Ring

```
char jobid[1024];
PMI2_Job_GetId(&jobid, sizeof(jobid));

char key[1024], value[1024];
sprintf(key, “%d”, rank);
sprintf(value, “%s”, address)
PMI2_KVS_Put(key, value);

PMI2_KVS_Fence();

char left_key[1024], left_val[1024];
char right_key[1024], right_val[1024];

int left_rank = (rank + size – 1) % size;
int right_rank = (rank + size + 1) % size;

sprintf(left_key, “%d”, left_rank);
PMI2_KVS_Get(jobid, left_rank, left_key, left_val);

sprintf(right_key, “%d”, right_rank);
PMI2_KVS_Get(jobid, right_rank, right_key, right_val);
```

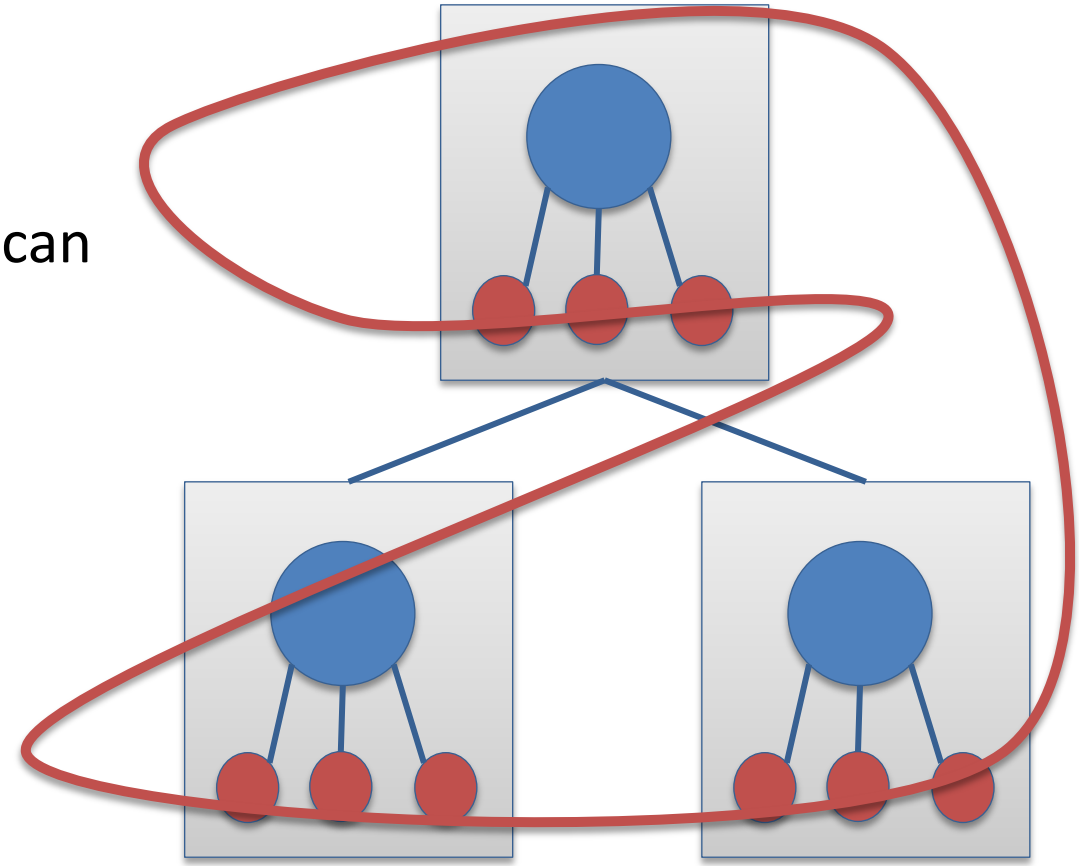
```
int rank, size;
char value[1024], left[1024], right[1024];
sprintf(value, “%s”, address);
PMIX_Ring(value, &rank, &size, left, right);
```

“PMI Extensions for Scalable MPI Startup”,
S. Chakraborty et al, EuroMPI’14

PMIX_Ring in SLURM since v15.08

- Executes a ring exchange among MPI processes
- Replaces Allgather with Scan
- Assume
  - P MPI procs in job
  - N nodes
  - p = P / N procs per node
- Reduces complexity

  O(p + P) -> O(p + log(N))



PMIX_Ring saves another 4.5 secs

```
    /-- proc_sync_1: 0.349794
        /-- spawn_endpoint_open_ibud: 0.020745
        /-- proc_sync_2: 0.022623
        /-- ring_exchange_pmixring: 0.060534
        /-- create_ring: 0.145208
        /-- create_world: 0.125836
        /-- create_node: 0.044211
        /-- create_leader: 0.055238
    /-- create_groups: 0.453673
    /-- populate_vcs: 0.546944
    .
    .
    .
psm_doinit: 13.840950
```

What's left?

```
    /-- proc_sync_1: 0.349794
        /-- spawn_endpoint_open_ibud: 0.020745
        /-- proc_sync_2: 0.022623
        /-- ring_exchange_pmixring: 0.060534
        /-- create_ring: 0.145208
        /-- create_world: 0.125836
        /-- create_node: 0.044211
        /-- create_leader: 0.055238
    /-- create_groups: 0.453673
    /-- populate_vcs: 0.546944
    /-- coll_arch_get: 0.678521
    /-- coll_arch_init: 0.000307
      /-- shm_coll_init: 0.042388
      /-- shm_coll_mmap: 0.000085
      /-- shm_coll_unlink: 0.000344
    /-- shm_coll: 0.043064
    /-- psm_bcast_uuid: 0.155392
    /-- psm_init: 0.000192
    /-- psm_ep_open: 0.069808
    /-- psm_mq_init: 0.000003
    /-- psm_ep_allgather: 1.160175
    /-- psm_ep_connect: 10.265013
    /-- psm_other_init: 0.008511
    /-- teardown_lwgrp: 0.088599
psm_doinit: 13.840950
```

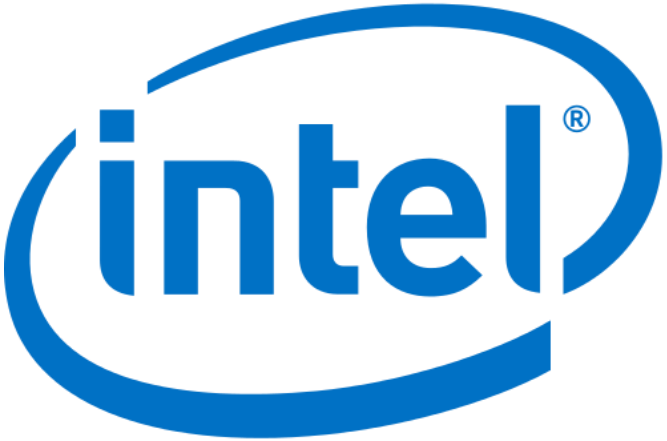
11.4 of 13.8 seconds

What's left?

```
    /-- psm_ep_open: 0.069808
    /-- psm_mq_init: 0.000003
    /-- psm_ep_allgather: 1.160175
    /-- psm_ep_connect: 10.265013
    /-- psm_other_init: 0.008511
    /-- teardown_lwgrp: 0.088599
psm_doinit: 13.840950
```

Eliminate with either…

1) Reimplement psm2_ep_connect() with efficient alltoall, e.g., Bruck's index algorithm



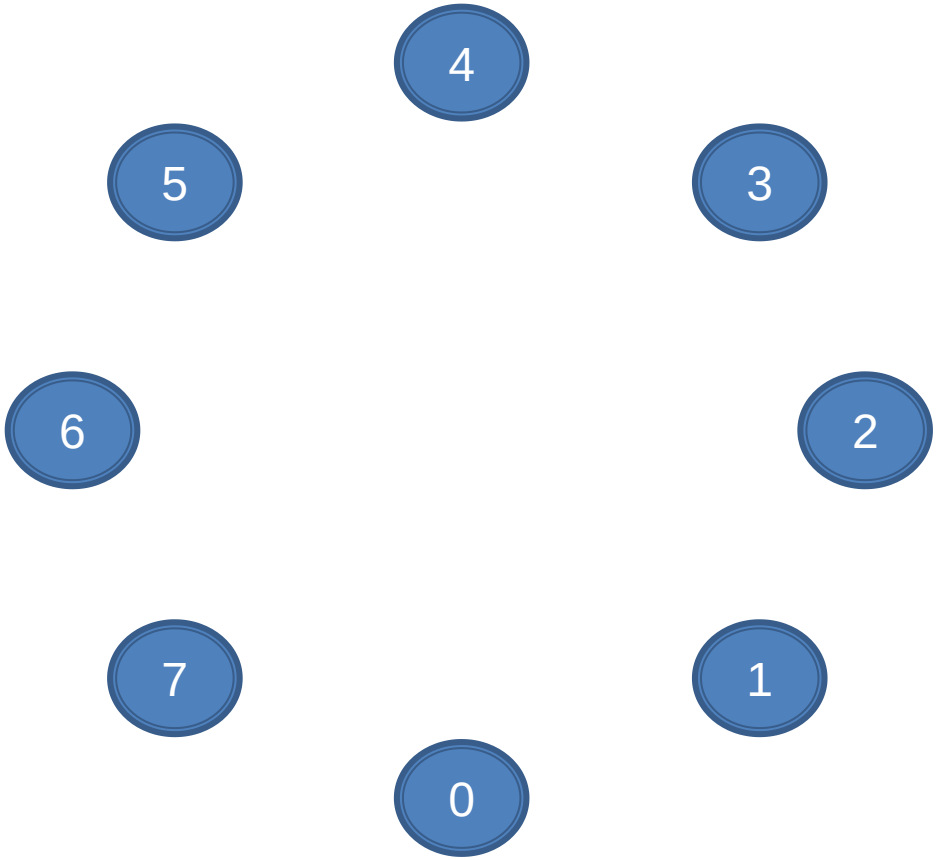
vs.

2) Add connection manager to PSM channel in MVAPICH

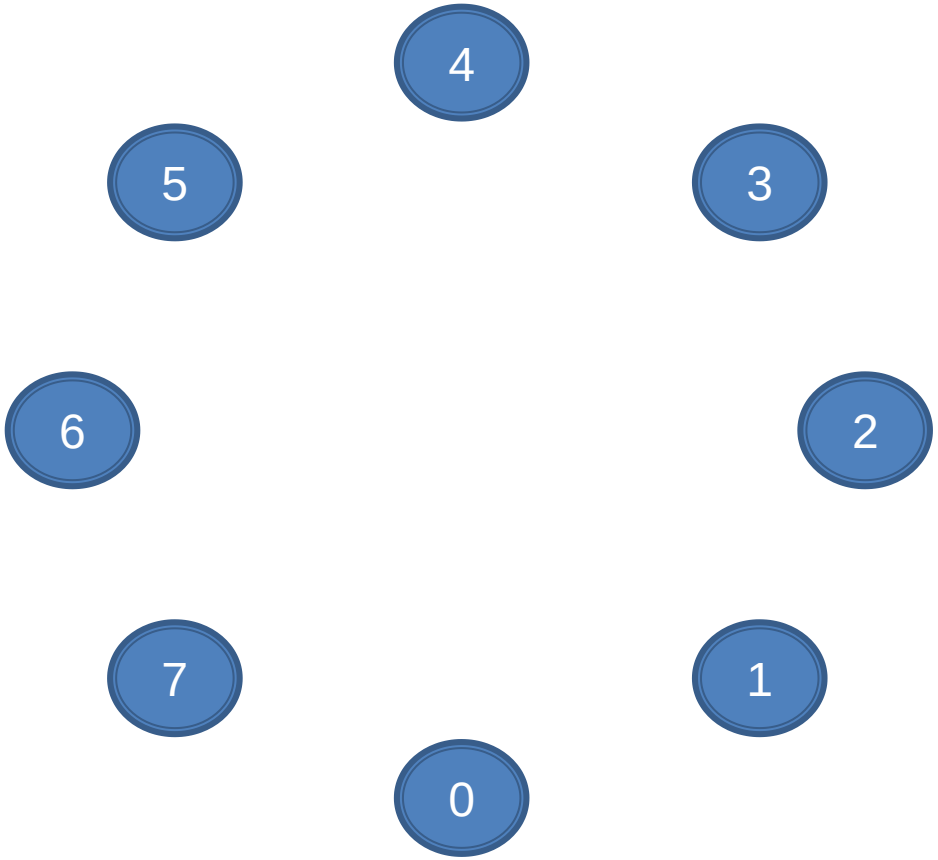
Intro to the MVAPICH2 connection manager

- Problem: Infiniband Reliable Connections often offer faster communication, but setup is expensive
- So MVAPICH2 creates RC connections on demand (the first time a process sends to another process)
- How?
  - Each process opens an IBUD queue pair
  - Global exchange of IBUD QP addresses via PMI
  - On the first send:
    - Source looks up IBUD QP address for target using PMI_Get
    - Send “connection request” packet to destination
  - Destination creates Reliable Connection Queue Pair for this connection, replies with its address
  - Source completes connection of RC QP

Ex: Rank 0 requesting RCQP connection with rank 5 using MVAPICH2 connection manager



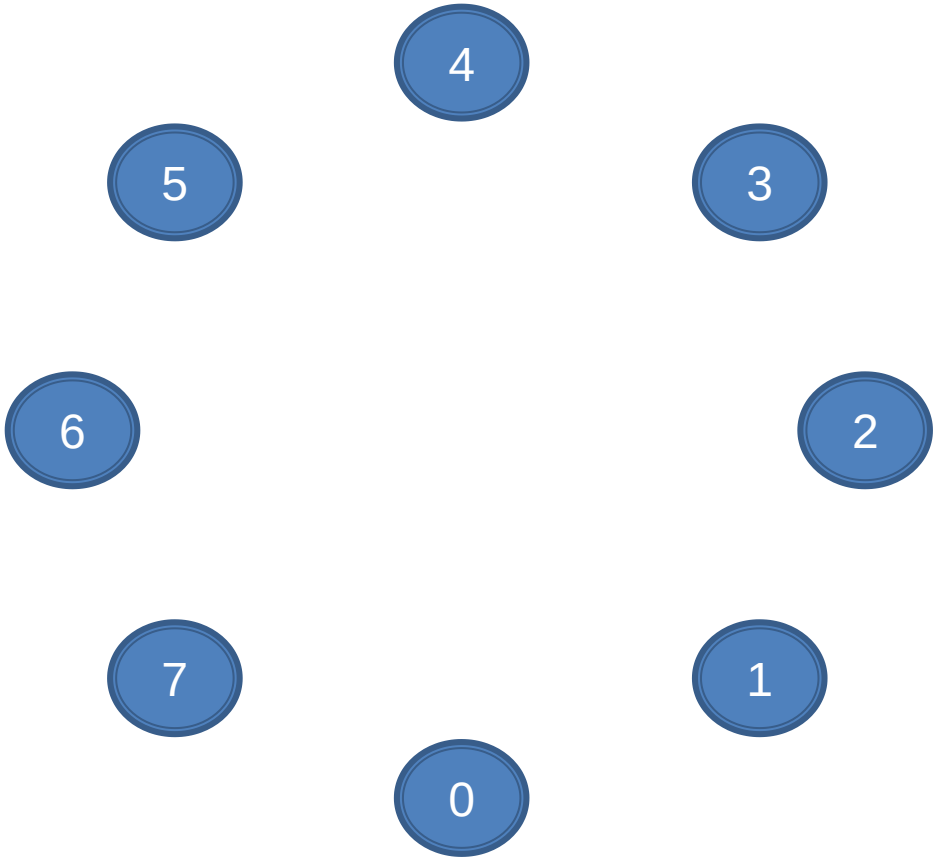
Gather UDQP address of every rank



P

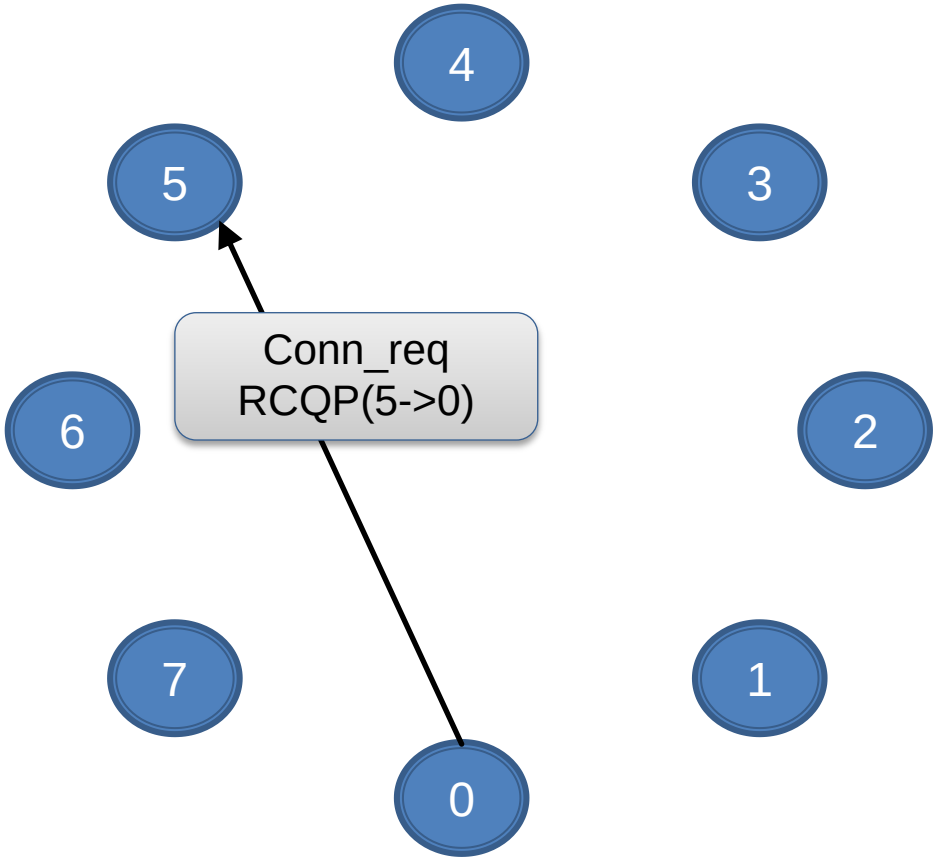
Key	Value
0	UDQP(0)
1	UDQP(1)
2	UDQP(2)
3	UDQP(3)
4	UDQP(4)
5	UDQP(5)
6	UDQP(6)
7	UDQP(7)

Lookup UDQP address of rank we want to connect to (rank 5)



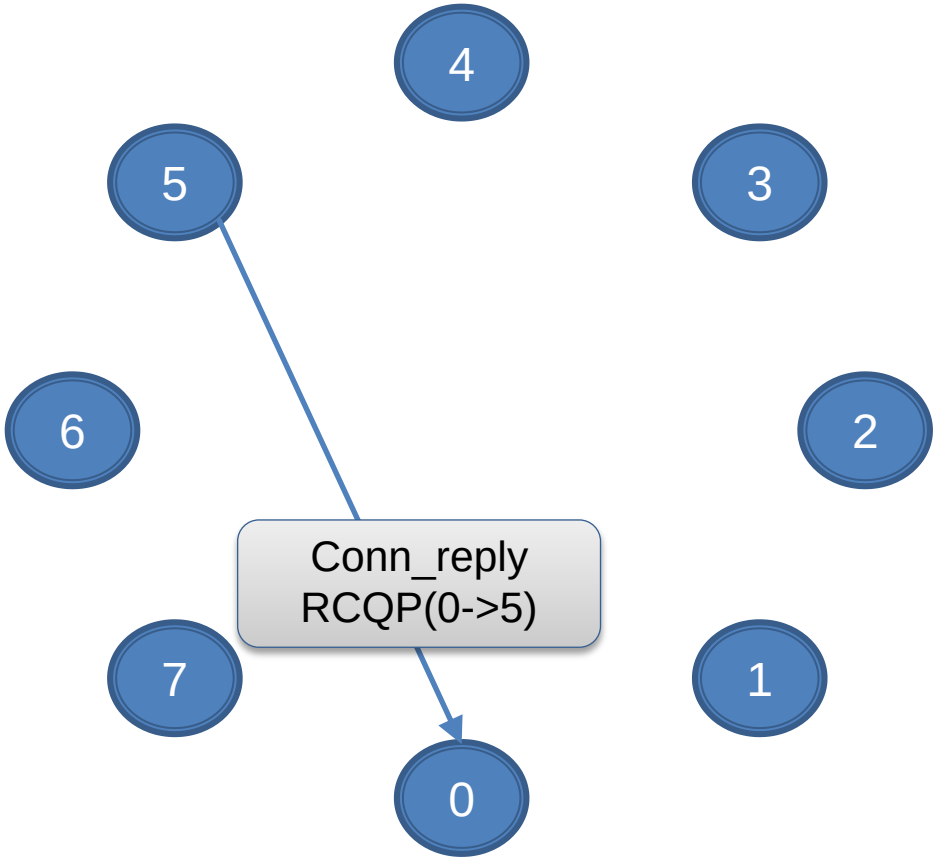
Key	Value
0	UDQP(0)
1	UDQP(1)
2	UDQP(2)
3	UDQP(3)
4	UDQP(4)
5	UDQP(5)
6	UDQP(6)
7	UDQP(7)

Send “connection request” packet over UDQP, include address of RCQP for connection



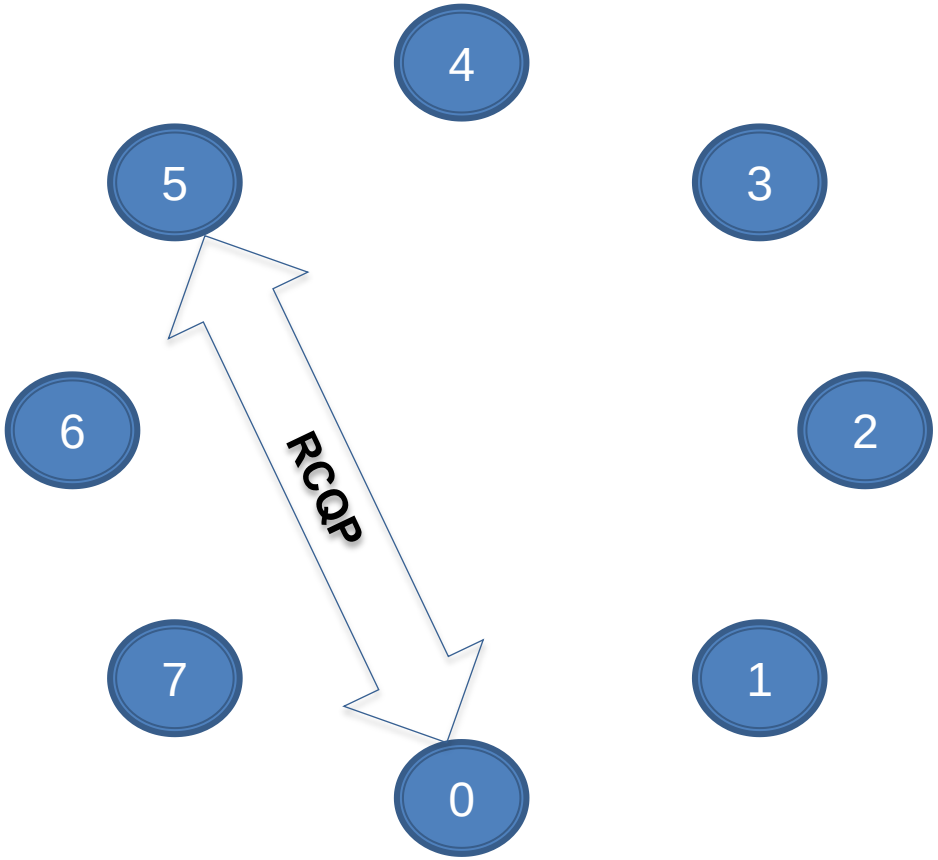
Key	Value
0	UDQP(0)
1	UDQP(1)
2	UDQP(2)
3	UDQP(3)
4	UDQP(4)
5	UDQP(5)
6	UDQP(6)
7	UDQP(7)

Reply with “connection accept”, including RCQP address to use at rank 5



Key	Value
0	UDQP(0)
1	UDQP(1)
2	UDQP(2)
3	UDQP(3)
4	UDQP(4)
5	UDQP(5)
6	UDQP(6)
7	UDQP(7)

Now both sides have both RCQP endpoint addresses, create the RCQP connection

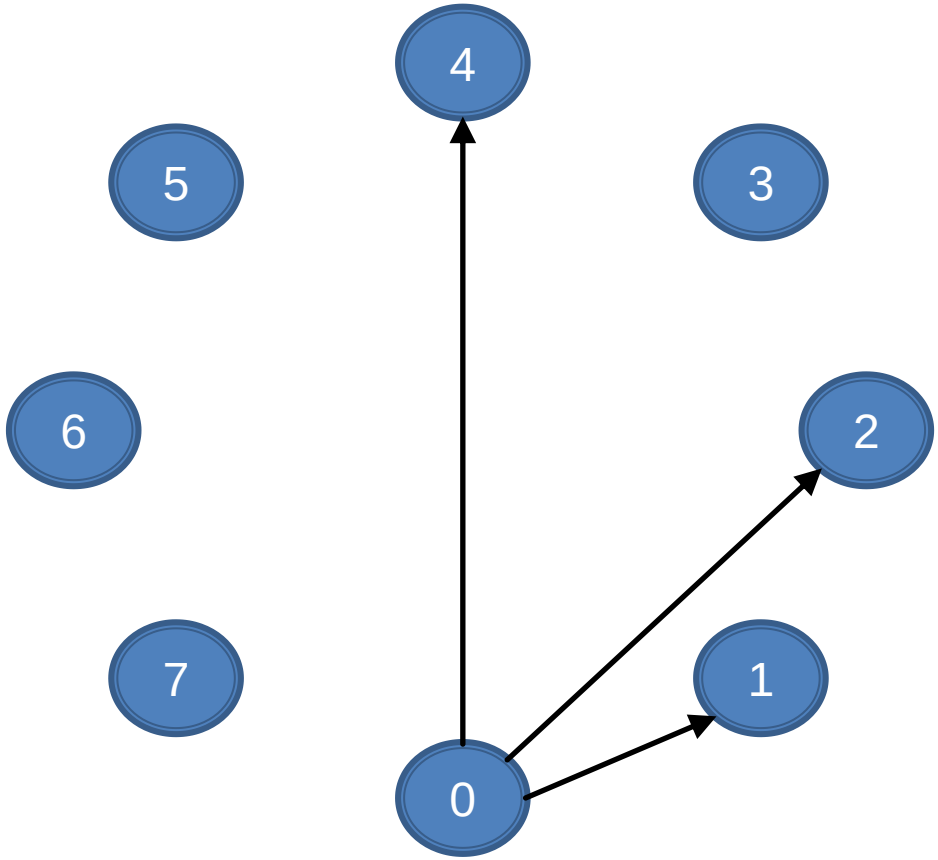


Key	Value
0	UDQP(0)
1	UDQP(1)
2	UDQP(2)
3	UDQP(3)
4	UDQP(4)
5	UDQP(5)
6	UDQP(6)
7	UDQP(7)

New address request packets

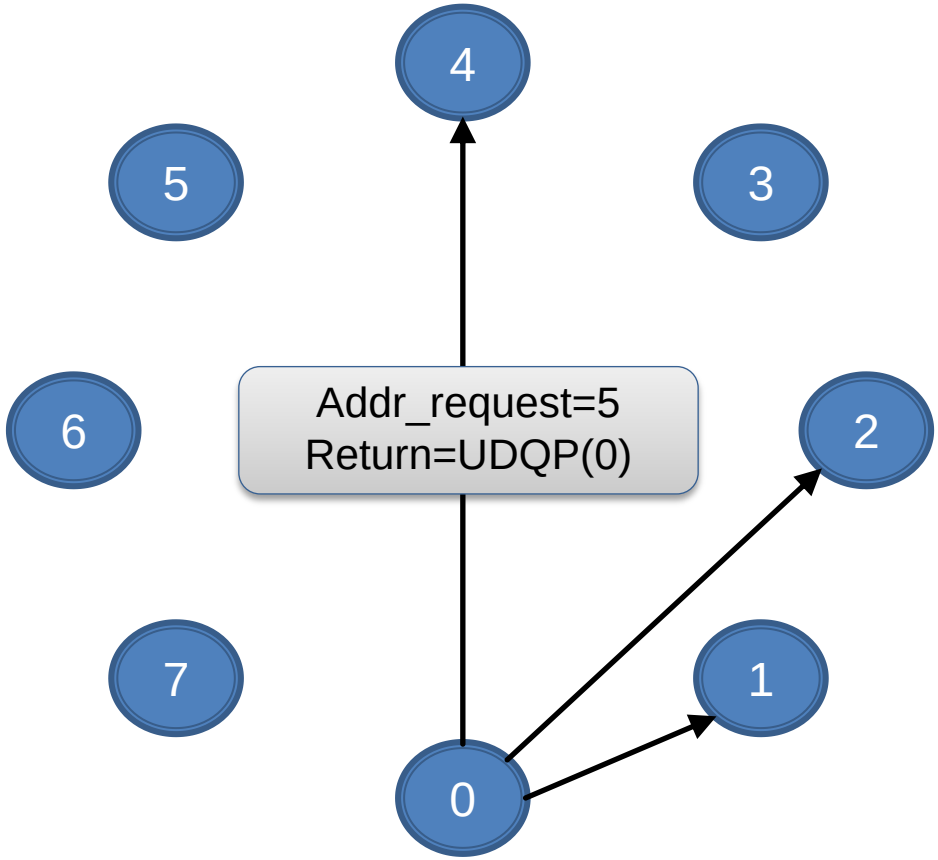
- Another problem: Global exchange of IBUD addresses requires an exchange of O(P) data in PMI2
- Solution: Define a new “address request” packet for existing connection manager protocol
- How?
  - Bootstrap each process with a small number of addresses to other ranks, e.g., create a binomial graph (yep, that again)
  - If a source process does not have the target address, lookup address for a rank which is closest to target
  - Send “address request” packet to this rank
  - That rank replies with the destination address if it has it
  - Otherwise, it replies with the address of a rank that is even closer to the target

Initial addresses known by rank 0, it sends “address request” packets for address of others



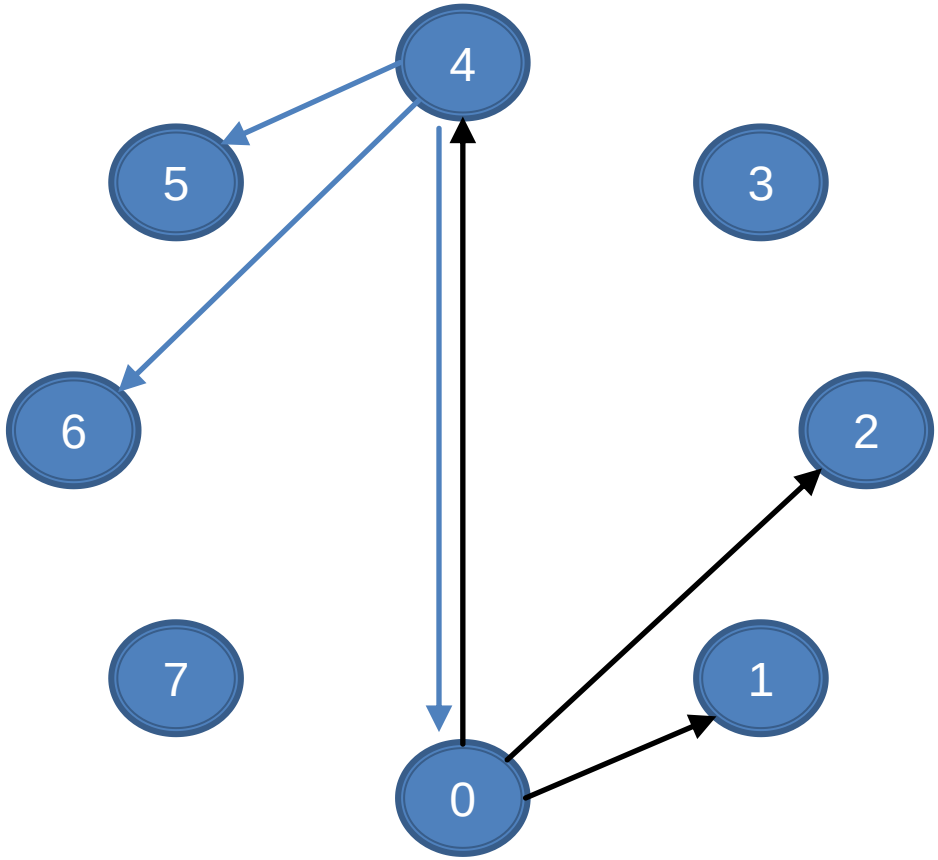
Key	Value
1	UDQP(1)
2	UDQP(2)
4	UDQP(4)

Rank 0 sends “address request” over IBUD to closest rank it knows (rank 4)



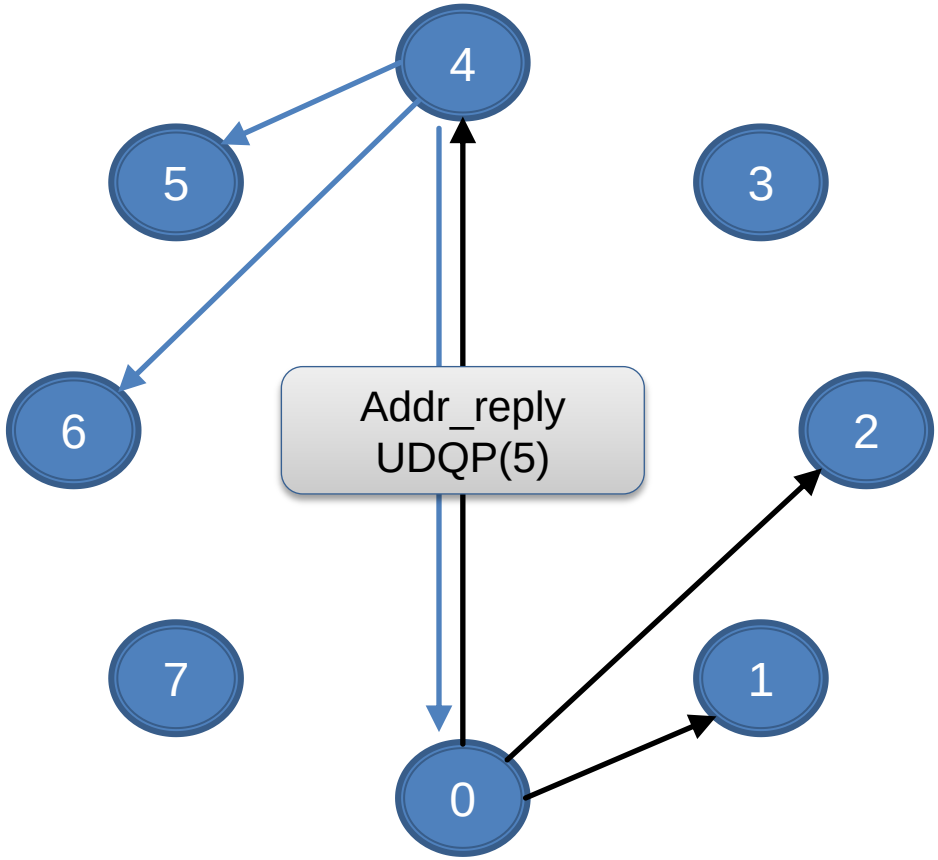
Key	Value
1	UDQP(1)
2	UDQP(2)
4	UDQP(4)

Initial addresses known by rank 4, rank 4 happens to know the address of rank 5



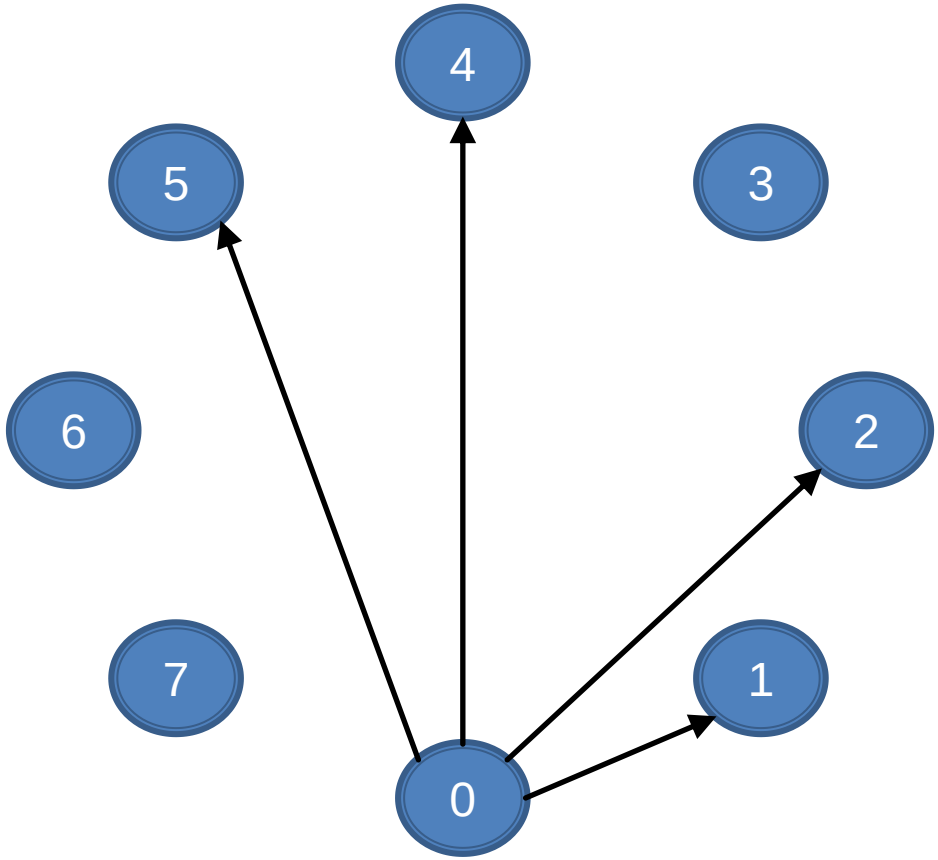
Key	Value
1	UDQP(1)
2	UDQP(2)
4	UDQP(4)

Rank 4 sends address of rank 5 back to rank 0 in “address reply”



Key	Value
1	UDQP(1)
2	UDQP(2)
4	UDQP(4)

Rank 0 caches address of rank 5, so it can now reply if other ranks ask it about rank 5



Key	Value
1	UDQP(1)
2	UDQP(2)
4	UDQP(4)
5	UDQP(5)

Number of known UDQP addresses per node is reduced from P to p * (log(P) + k), where k is the number of RCQP connections.

Address request lookup

- Adds overhead to RCQP connection establishment
- Latency to get address is at most log(P) hops
- Each process will record addresses of initial set + those it has creates IB RC connections with
- Number of UDQP per node
  - PMI: P
  - Address request packets: p * (log(P) + k), where k is number of RCQP connections per process
- Saves memory and time for applications that are not densely connected
- Adds cost for densely connected apps

How have we done?

```
time srun -n 73728 -N 2048 osu_initbarfin
```

	MV2-2.2 Baseline	Stage executable and libs	Replace PMI2 with spawnnet	PMIX_Ring	Connection Manager (est)
MPI_Init	96.4	68.2	18.4	13.9	< 3
MPI_Barrier	0.0	0.0	0.1	0.1	0.1
MPI_Finalize	2.1	2.0	2.0	2.0	2.0
Other	11.5	11.8	11.7	7.0	7.0
Total (secs)	110.0	82.0	32.2	23.0	< 10 ?

Summary of major concepts for scalable startup

- Need to prestage or broadcast app and libs
  - Status: still needed in mpirun
- Swap PMI2 for collectives
  - Status: done, use spawnnet
- Use PMIX_Ring at scale. It exists and it helps!
  - Status: done
- Connection manager
  - Status: still needed for PSM channel
- Address request packets in connection manager
  - Status: done
  - Procs on a node could save further memory / time by storing known addresses in shared data structure