Introduction to Parallel Processing

Lecture 12 : MPI Communicators

Professor Amanda Bienz

MPI Communicators

- So far we have only talked about MPI_COMM_WORLD
- This is a communicator, or group, that contains all processes that were initialized during the ‘mpirun’ call
- A communicator provides a space in which processes can communicate with one another.

Custom Communicators

- Can create custom communicators : sub-group of processes, also known as a sub-communicator

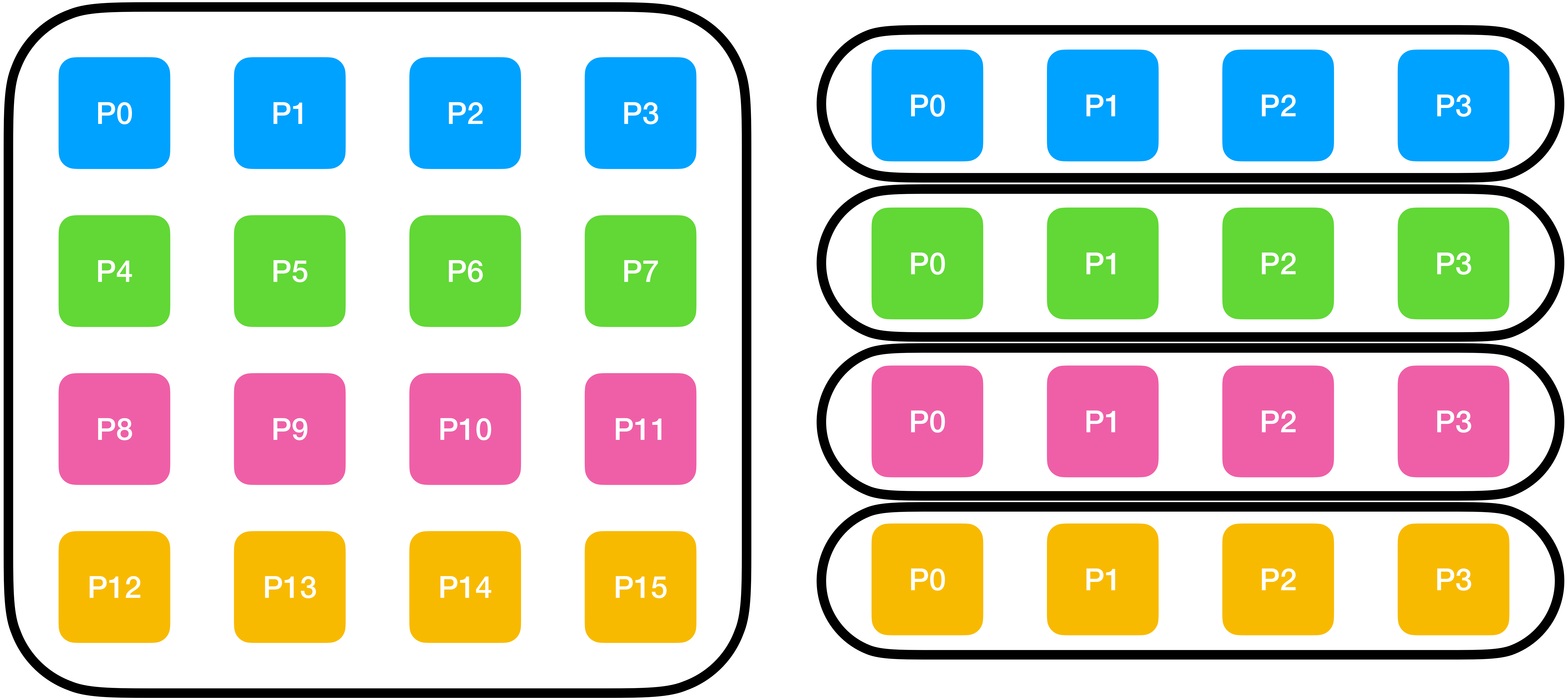
- ```
  MPI_Comm_split(MPI_Comm orig_comm,
          int color,
          int key,
          MPI_Comm* new_comm)
  ```

  Same color <—> same new communicator

  Key implies rank in new communicator;
  if tie, based off lowest original rank (in orig_comm)

- ```
  MPI_Comm_create(MPI_Comm orig_comm,
          MPI_Group group,
          MPI_Comm* new_comm)
  ```

Sub-Communicators



Topology-Aware Communicators

- As we know from performance models, relative locations of processes has a large impact on cost of inter-process communication
- Often, we would like to create custom communicators based on the topology
- You can do this through external parameters, such as setting mapping of ranks to nodes with variable such as MPICH_RANK_REORDER_METHOD
-
```
MPI_Comm_split_type(MPI_Comm orig_comm,
        int split_type,
        int color,
        int key,
        MPI_Comm* new_comm)
```

Most common split_type : MPI_COMM_TYPE_SHARED

- New splitting types available in MPI 4

OpenMPI Splitting Types

- OMPI_COMM_TYPE_NODE : same as MPI_COMM_TYPE_SHARED
- OMPI_COMM_TYPE_HWTHREAD : splits by hardware thread
- OMPI_COMM_TYPE_CORE : splits by core / processing unit
- OMPI_COMM_TYPE_L1CACHE : splits by L1 cache
- OMPI_COMM_TYPE_L2CACHE : splits by L2 cache
- OMPI_COMM_TYPE_L3CACHE : splits by L3 cache
- OMPI_COMM_TYPE_SOCKET : splits by socket
- OMPI_COMM_TYPE_NUMA : splits by NUMA-region
- OMPI_COMM_TYPE_BOARD : splits by board
- OMPI_COMM_TYPE_HOST : splits by host
- OMPI_COMM_TYPE_CU : splits by computational unit
- OMPI_COMM_TYPE_CLUSTER : splits by cluster

MPI_Info

- Stores an unordered set of (key, value) pairs
- Many routines take an info argument (can be MPI_INFO_NULL)
- Ignores keys that are not recognized
- Allows for hints to be passed to MPI implementation
- However, you need to know what keys to set…

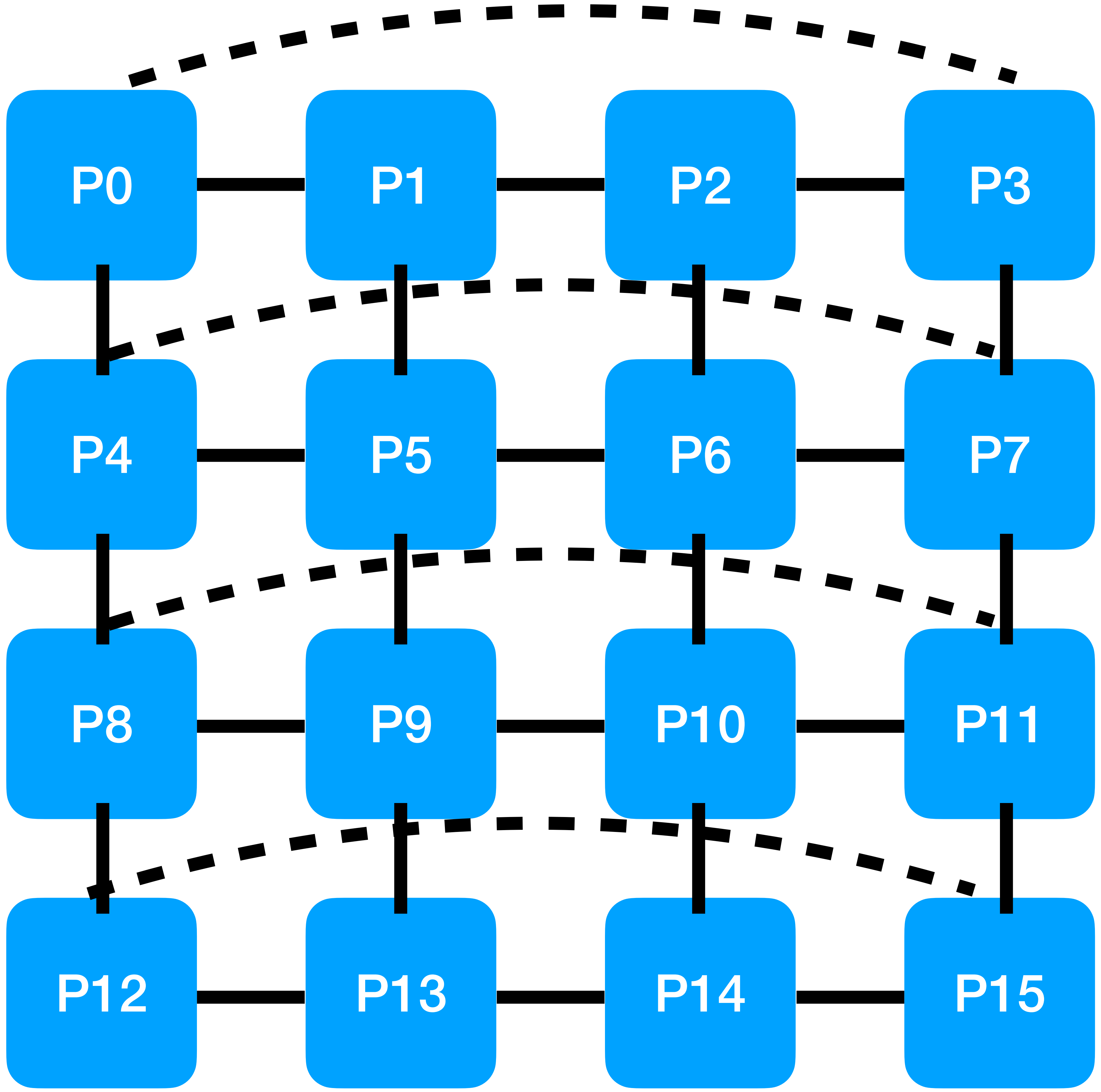
Types of Communicators

- There are two types of communicators :
  - **Intra-communicator:** the type that we have been talking about.
    - All processes in the communicator can send messages to each other and participate in collective communication
  - **Inter-communicator:**
    - Can send messages between processes belonging to disjoint intra-communicators

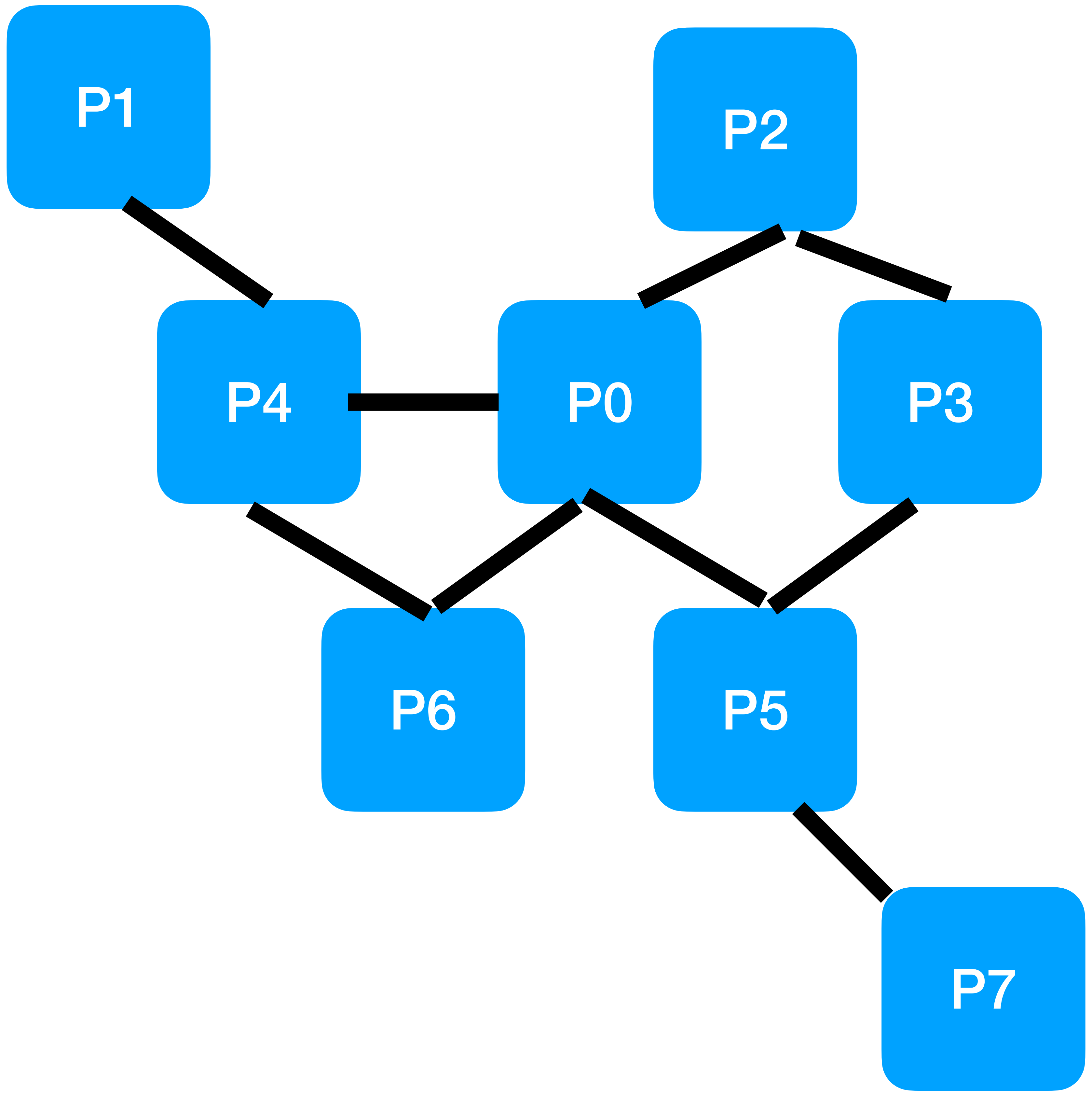
Inter-Communicator Example

- Communication Requirements:
  - Group 0 to communicate with Group 1
  - Group 1 to communicate with Groups 0 and 2
  - Group 2 to communicate with Group 1

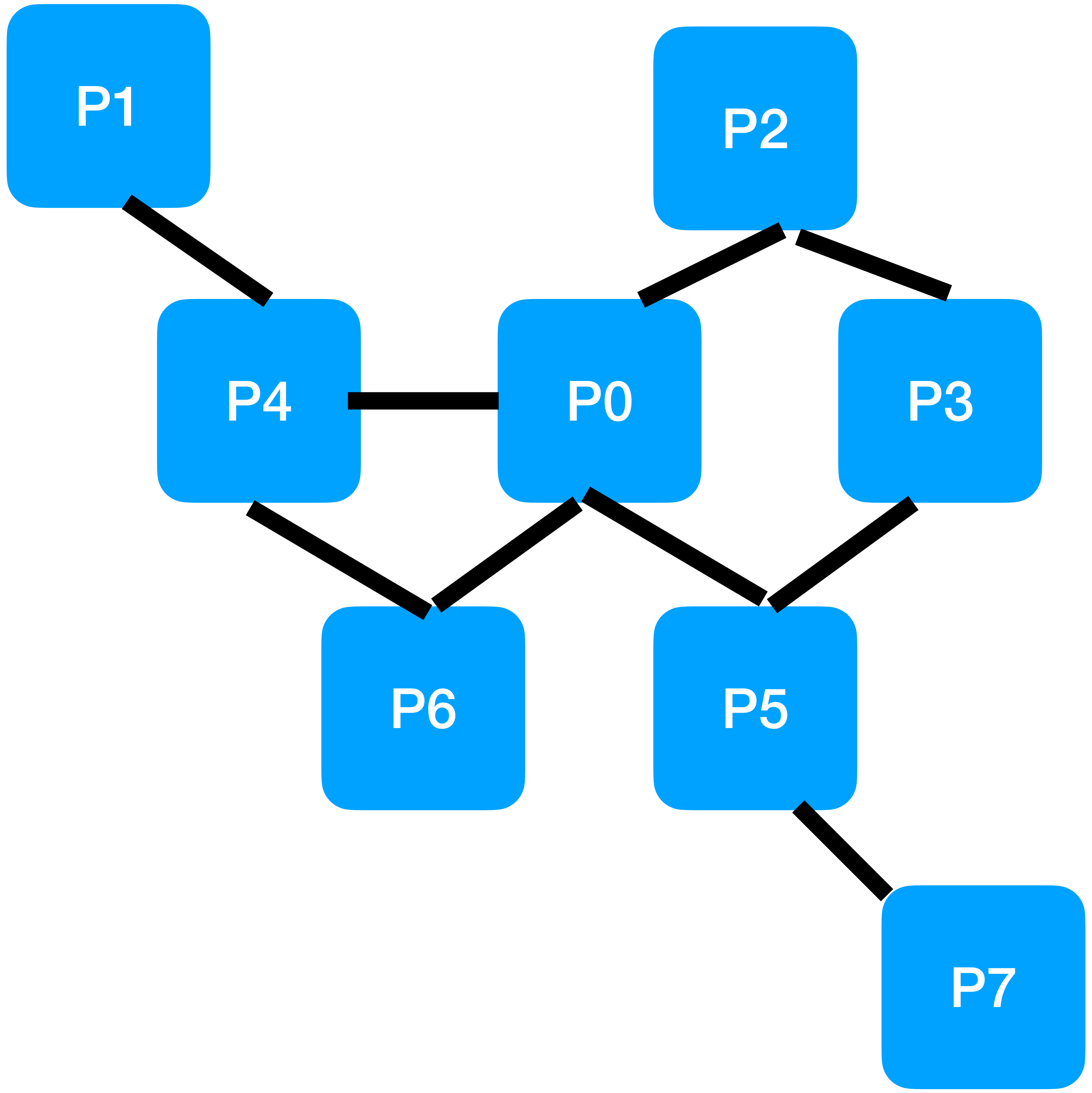
Inter-Communicator Example


- ```
  MPI_Intercomm_create(MPI_Comm local_comm,
                int local_leader,
                MPI_Comm peer_comm,
                int remote_leader,
                int tag,
                MPI_Comm& inter_comm)
  ```
  ```

Virtual Topology

- Communicator attributes include **topology**
- In MPI, we use virtual topologies (such as cartesian coordinates / grids and graphs) rather than mapping directly to the underlying hardware
- This is very useful! Many programs hold either a structured grid or some type of unstructured graph

Cartesian Topology



- What if all processes holding neighbors of a structured graph (or stencil code) need to communicate with one another
- We can tell MPI this is our virtual topology, and MPI will assume only neighboring nodes in this topology communicate
- ```
  MPI_Cart_create(MPI_COMM_WORLD,
                  int n_dims,
                  int* dims,
                  int* periods,
                  int reorder,
                  MPI_Comm* comm_cart)
  ```
  ```

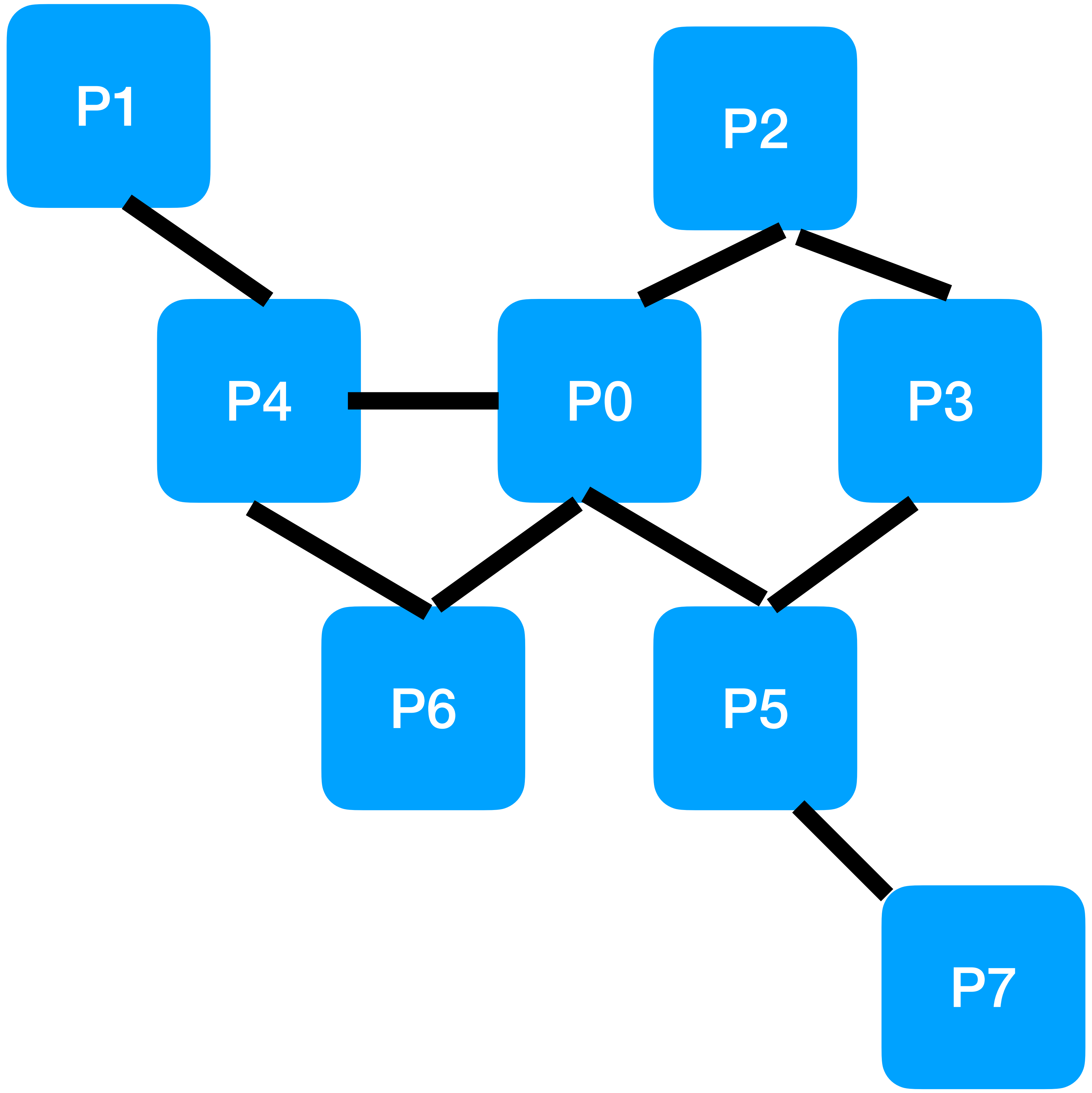
Unstructured Graphs



- What if we have an unstructured graph (or sparse matrix) partitioned across the processes, and a random subset of processes need to communicate with one another

- ```
  MPI_Graph_create(MPI_Comm old_comm,
                   int nodes,
                   const int indx[],
                   const int edges[],
                   int reorder,
                   MPI_Comm * comm_graph
  ```
  ```

Distributed Graph



- Typically, only know what is on my rank and which processes my rank is connected to

- ```
  MPI_Dist_graph_create(MPI_Comm comm_old,
                        int n,
                        const int sources[],
                        const int degrees[],
                        const int destinations[],
                        const int weights[],
                        MPI_Info info,
                        int reorder,
                        MPI_Comm* comm_dist_graph)
  ```
  ```

Distributed Graph



- Typically, only know what is on my rank and which processes my rank is connected to

- ```
  MPI_Dist_graph_create_adjacent(
          MPI_Comm comm_old,
          int indegree,
          const int sources[],
          const int sourceweights[],
          const int outdegree,
          const int destinations[],
          const int destweights[],
          MPI_Info info,
          int reorder,
          MPI_Comm* comm_dist_graph)
  ```
  ```

Neighborhood Collectives

- One of the reasons to use MPI_Cart_create and MPI_Dist_graph_create_adjacent is that you now know which processes should communicate with one another

- Instead of implementing that communication by hand, you can call a method such as MPI_Neighbor_alltoallv

MPI_Neighbor_allgather

- Each process sends the same message to all neighbors
- Each process receives a message from each of their neighbors (‘in degree’)
-
```
MPI_Neighbor_allgather(const void* sendbuf,
          int sendcount,
          MPI_Datatype sendtype,
          void* recvbuf,
          int recvcount,
          MPI_Datatype recvtype,
          MPI_Comm comm)
```

MPI_Neighbor_alltoallv

- Each process sends the a message to each of the destination neighbors ('out degree' messages)

- Each process receives a message from each of their source neighbors ('in degree' messages)

- ```
  MPI_Neighbor_alltoallv(const void* sendbuf,
          const int sendcounts[],
          const int sdispls[],
          MPI_Datatype sendtype,
          void* recvbuf,
          const int recvcounts[],
          Const int rdispls[],
          MPI_Datatype recvtype,
          MPI_Comm comm)
  ```
  ```

Neighborhood Collectives

- Unfortunately, the implementation behind neighborhood collectives in not currently optimized (this would be a great, longer term research project)
- This API allows for some really great optimizations under the hood (node-aware communication!)
- Ideally, these should be implemented to optimize communication under the hood (you know all information that is being sent / received, and so it is possible to aggregate that data on node and remove off-node communication)