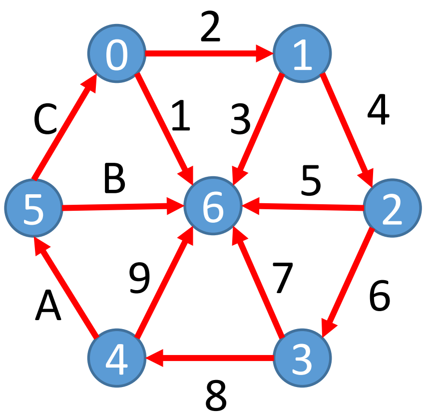
More advanced MPI and mixed programming topics

Extracting messages from MPI

- MPI_Recv delivers each message from a peer in the order in which these messages were send
  - No coordination between peers is possible



- Take a scenario where we have a ring of processors with (P-1) participants, and a lone process that centralize messages from all peers.
- Each processor (except 0) waits for a message from its predecessor in the ring before sending a message to the coordinator
- In which order the messages are received at the coordinator ?
- How we can implement this if each ring participant send a message of a different length ?
- What if we assume a large number of processes?

- Missing functionality: the capability to peek (but not alter) into the network to extract what message will be the next to be locally received
  - Functionality that behaves as MPI_Recv but without altering the matching queue

MPI Probe

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,    MPI_Status *status);
Int MPI_Get_count( MPI_Status* status, MPI_Datatype datatype, int* count);
MPI_Status a structure containing the fields MPI_SOURCE, MPI_TAG and MPI_ERROR
```

- MPI_ANY_SOURCE and MPI_ANY_TAG can be used as markers for unnamed receives
- The usual usage scenario is probe, memory allocation and then receive
  - How can we use this functionality in a thread safe application when all threads work on the same communicator ?
  - Assume 2 threads (X,Y) doing the probe (P), alloc (A) and receive (R) operation each one on its own context
    - $X_P \longrightarrow X_A \longrightarrow X_R \longrightarrow Y_P \longrightarrow Y_A \longrightarrow Y_R$
  - What happens if the order of the operations is $X_P \longrightarrow X_A \longrightarrow Y_P \longrightarrow Y_A \longrightarrow Y_R \longrightarrow X_R$
- The access to the matching queue need to be protected for concurrent accesses

Message Probe

- Functionality that extracts the message from the matching queue but without receiving it
  - Supported by functionality to extract the content of the message into a user provided buffer
  - Any partial ordering between our threads X and Y is now correct: $X_{P'} \longrightarrow X_A \longrightarrow Y_{P'} \longrightarrow Y_A \longrightarrow Y_{R'} \longrightarrow X_{R'}$

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm,    MPI_Message *message,
                         MPI_Status *status);
int MPI_Improbe(int source, int tag, MPI_Comm comm,    int *flag, MPI_Message *message,
                         MPI_Status *status)
int MPI_Mrecv(void *buf, int count, MPI_Datatype type,    MPI_Message *message,
                       MPI_Status *status);
int MPI_Imrecv(void *buf, int count, MPI_Datatype type,    MPI_Message *message,
                       MPI_Request *request);
```

Collective Communication with threads

- What is happening if multiple threads issue in the same communicator in same time
  - Multiple blocking collectives ?
  - Multiple non-blocking collective with the same datatype and count ?
  - Multiple non-blocking collective with the different datatype and count ?

Shared Memory

- Potential for memory reduction as initialization data can be shared between processes
  - Avoid recomputing the same initial state by multiple applications (on the same node)
  - POSIX provides shared memory regions but (1) not all Oses have support for them and (2) it does not integrate with MPI functionality
- Need functionality to split a communicator in disjoint groups with shared capabilities
  - Similar to MPI_Comm_split with architecture aware color (key will then be the rank in the original communicator)
  - Single info key standardized: MPI_COMM_TYPE_SHARED
  - Some MPI implementations provide support for different granularities of sharing (Open MPI)

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,    MPI_Info info,
                        MPI_Comm *newcomm);
```

Shared Memory Window

- Allocates shared memory regions in win
  - Collective call resulting in a fully capable RMA window
  - Constraint: all processes in the communicator must be capable of physically sharing memory (usually same node)
  - The call returns a pointer to the local part
  - The info key define how the global shared memory region is defined:
    - Contiguous: process i memory starts right after the end of process i-1
    - Non contiguous (key alloc_shared_noncontig): allow the MPI to provide NUMA-aware optimizations.
  - One way to create the communicator needed is to use MPI_Comm_split_type

```
int MPI_Win_allocate_shared (MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
                             void *baseptr, MPI_Win *win);
```

Shared Memory Window

- In non contiguous cases we need to extract the remote address in order to complete RMA operations
  - As the memory region might be mapped at different addresses in different processes each process local address has no meaning
    - Unlike in Open SHMEM where the RMA operations applied on symmetric memory (!)
  - Only works for windows of type MPI_WIN_FLAVOR_SHARED (aka. created via MPI_Win_allocate_shared)

```
int MPI_Win_shared_query (MPI_Win win, int rank, MPI_Aint *size, int *disp_unit,
                          void *baseptr);
```

RMA and pt2pt puzzle ?

- Assuming a correctly initialized window what is the outcome of the following code ?

```
for(i = 0; i < len; a[i] = (double)(10*me+i), i++);
if (me == 0) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Send(NULL, 0, MPI_BYTE, 2, 1001, MPI_COMM_WORLD);
    MPI_Get(a,len,MPI_DOUBLE,1,0,len,MPI_DOUBLE,win);
    MPI_Win_unlock(1, win);
    for(i = 0; i < len; i++) printf("a[%d] = %d\n", a[i]);
} else if (me == 2) {     /* this should block till 0 releases the lock. */
    MPI_Recv(NULL, 0, MPI_BYTE, 0, 1001, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Put(a,len,MPI_DOUBLE,1,0,len,MPI_DOUBLE,win);
    MPI_Win_unlock(1, win);
}
```