Discussion 3

Today’s Agenda

- Lab 2 (MPI) is out!
- Check MPI How-To Document on Class website!
- MPI review
- MPI Reduction
- Ring topology
- Tarry’s Algorithm and Histogram
- Some important pointers for the assignment.

MPI Basics

- MPI_Init —> Start
- MPI_Finalize —> End
- MPI_Comm_rank —> What is my id
- MPI_Comm_size —> How many nodes are there?
- MPI_Send —> Point-to-Point send
- MPI_Recv —> Point-to-Point receive
- MPI_Scatter —> Spread out different information from an array
- MPI_Gather —> Gather different information into an array

`int MPI_Init( int *argc, char ***argv )`

- Initializes MPI_Environment
- Always put it first, before ANY other code
- Undefined what happens with the code before it
- Avoid changing things that are external to the program state (for example open a file) before calling it

`int MPI_Finalize( void )`

- Terminates MPI Environment
- Undefined how many nodes continue to execute after it
- Must be last (before the return) of your program
- Must be called before the end of the program

MPI_COMM_WORLD

- In MPI you can define different communication groups
- We will use only one: MPI_COMM_WORLD
- Contains all of the available nodes

```
int MPI_Comm_size(
 MPI_Comm comm, int *size )
```

- Returns the size of the communication group
- If passed MPI_COMM_WORLD returns the number of nodes (should be the same as the number of processes you defined)

`int MPI_Comm_rank( MPI_Comm comm, int *rank )`

- Returns you rank in the communication group
- Goes from 0 to MPI_COMM_size(comm)-1
- Rank is unique inside of the communication group
- We call the process with rank 0 the Master node

MPI_Datatype

- Predefined MPI_Types (you can also define custom types, but we won't)
- Most of the basic types are defined there

- MPI_CHAR
- MPI_UNSIGNED_CHAR
- MPI_INT
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG
- MPI_COMPLEX
- lots more…

```
int MPI_Send(const void *buf, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

- Point-to-Point Send
- Must be paired with a receive on the other end
- Both are blocking
- You can send more than one element, of same datatype, at a time (for example send 3 ints, or 2 chars)

```
int MPI_Send(const void *buf, int count,
MPI_Datatype datatype, int dest, int tag,
                MPI_Comm comm)
```

- `*buf`: Address of the first element
- `count`: Number of elements to send (elements must be contiguous in memory)
- `datatype`: What type of data to send (int,char,double…)
- `dest`: Rank where to send it
- `tag`: Optional extra information about the message, must be an int. Just use 0 unless you need to specify an order
- `comm`: What communication group to use

Example Send

```
int a = 10;

char b[4] = {‘a’,’b’,’c’,’d’};

MPI_Send(&a, 1, MPI_INT, 5, 0, MPI_COMM_WORLD);

// Send {10} to node 5

MPI_Send(&b, 3, MPI_CHAR, 2, 0, MPI_COMM_WORLD);

// Send {‘a’,’b’,’c’} to node 2
```

```
int MPI_Recv(void *buf, int count,
  MPI_Datatype datatype, int source, int tag,
  MPI_Comm comm, MPI_Status *status)
```

- `*buf`: Address of where to store the first element
- `count`: Maximum number of elements to be received (may get less)
- `datatype`: What type of data to receive (int, char, double…)
- `source`: Rank of who sent it
- `tag`: Receives only messages with this tag. Just use 0
- `comm`: What communication group to use
- status: Status object useful for finding errors. Use MPI_STATUS_IGNORE if you don’t need it (which you probably don’t…)

Receive Wildcards

- If you don’t need a tag use:
  - MPI_ANY_TAG
- If you don’t need the source of the receive
  - MPI_ANY_SOURCE

Example Receive

```
int a;

char b[10];

MPI_Recv(&a, 1, MPI_INT, 5,

    MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Receive an int from node 5 and store it in

// variable a. No tag and ignore the status.

MPI_Recv(&b, 4, MPI_CHAR, MPI_ANY_SOURCE

   3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Receive at most 4 chars and store them in b. Can be from any

// source, must have tag 3. Ignore the status.
```

Send/Recv: Avoiding Deadlock

```
int a = 0, b;

if(processId == 0) {

  MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

  MPI_Recv(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

}

else if (processId == 1) {

  MPI_Send(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

  MPI_Recv(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

}
```

Non-Blocking Communication

- A nonblocking operation requests the MPI library to perform an operation (when it can).
- Nonblocking operations do not wait for any communication events to complete
- Nonblocking send and receive: return almost immediately.
- The user can modify a send [resp. receive] buffer only after send [resp. receive] is completed.
- There are “wait” routines to figure out when a nonblocking operation is done
- Nonblocking send can be posted whether a matching receive has been posted or not.
- Send is completed when data has been copied out of send buffer.
- Nonblocking send can be matched with blocking receive and vice versa.
- Communications are initiated by sender.
- A communication will generally have lower overhead if a receive buffer is already posted when a sender initiates a communication

Important non-blocking routines

- int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

- int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- int MPI_Wait(MPI_Request *request, MPI_Status *status)

Usage of nonblocking operations and MPI_Wait

```
Call MPI_COMM_RANK(comm, rank, ierr)

If (rank == 0):

    Call MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)

    **** do some computation ****

    Call MPI_WAIT(request, status, ierr)

else:

    call MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)

    **** do some computation ****

    call MPI_WAIT(request, status, ierr)
```

MPI_Bcast vs MPI_Scatter

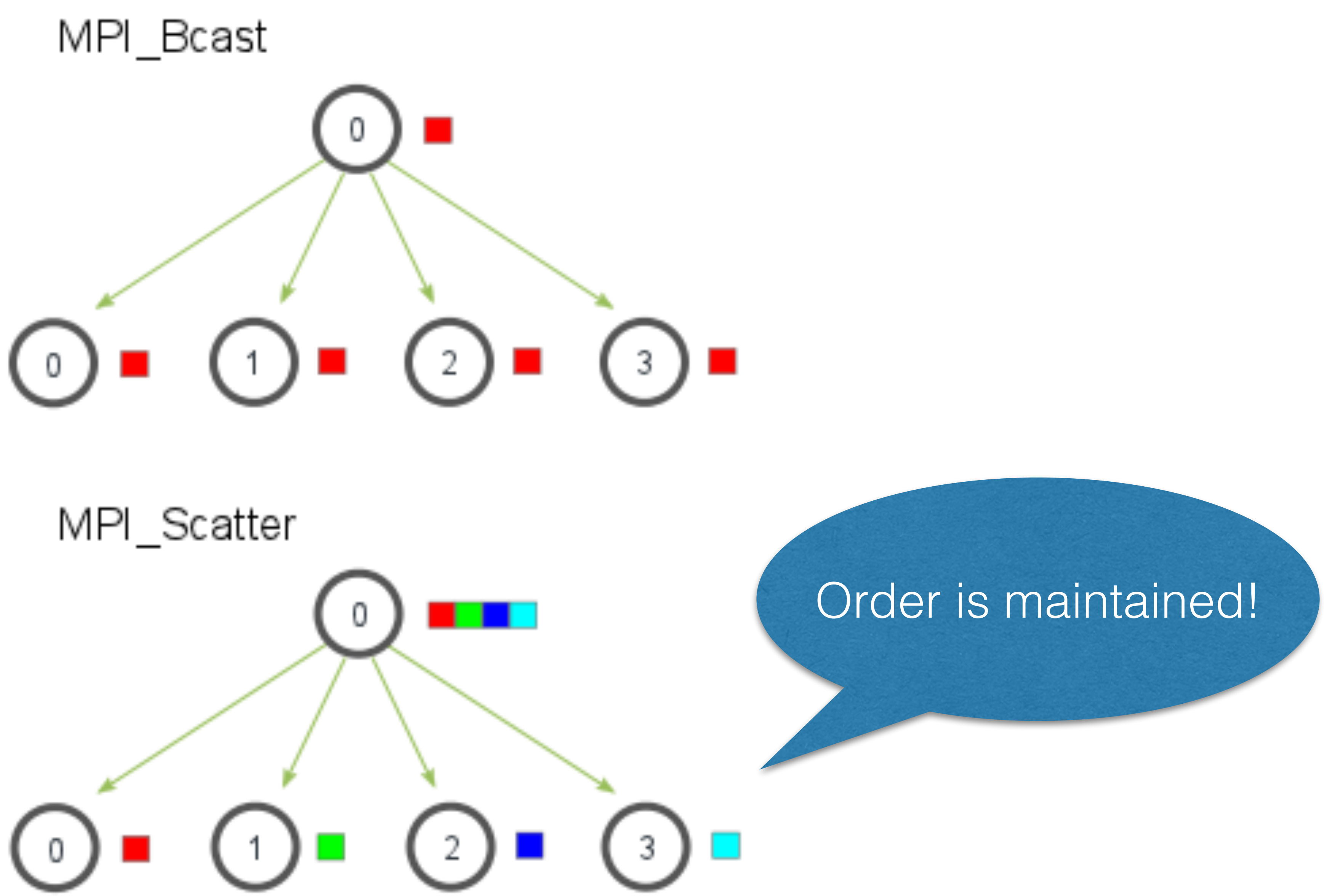


Image obtained from:
http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

```
int MPI_Bcast(
void *buffer, int count, MPI_Datatype datatype, int
root, MPI_Comm comm )
```

- `buffer`: address of send buffer (significant only in root)
- `count`: number of elements to be sent to all processes
- `datatype`: type of data to be sent
- `root`: Rank of the sender process
- `comm`: communication group

```
int MPI_Scatter(
  const void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

- `sendbuf`: address of send buffer (significant only in root)
- `sendcount`: number of elements sent to each process
- `sendtype`: type of data to be sent
- `recvbuf`: where to store the data that is received
- `recvcount`: number of elements to receive
- `recvtype`: type to receive
- `root`: Rank of the sender process
- `comm`: communication group

```
int MPI_Scatter(
 const void *sendbuf, int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

- Scatter the sendbuf of root to the other nodes
- All the nodes must execute the call
- The root node will distribute the information
- All the other node in the communication group will receive their piece of the array
- Notice the symmetry in the parameters, normally they should be the same

MPI_Gather

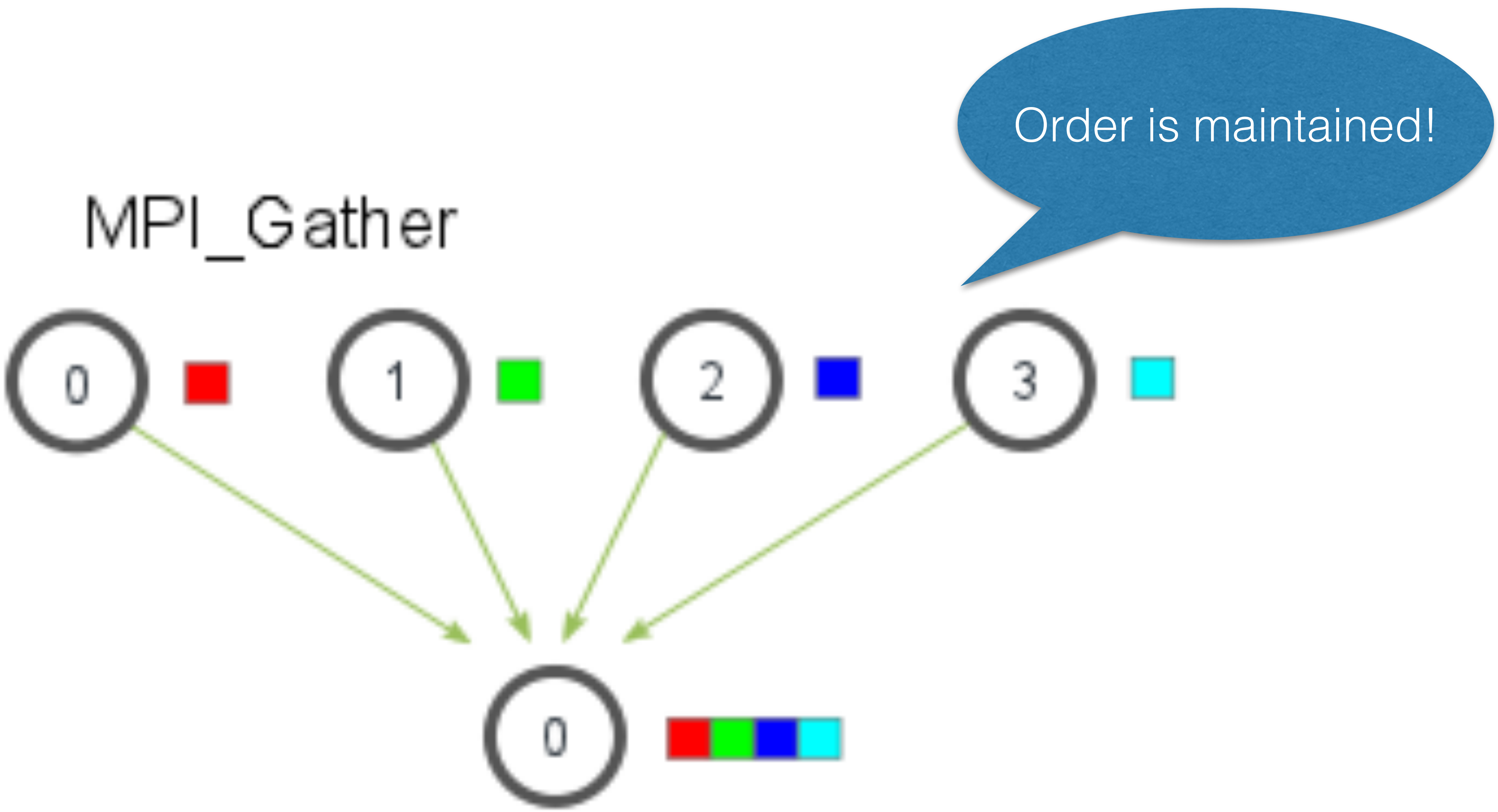


Image obtained from:
http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

```
int MPI_Gather(
 const void *sendbuf, int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

- `sendbuf`: address of send buffer
- `sendcount`: number of elements sent to root
- `sendtype`: type of data to be sent
- `recvbuf`: where to store the data that is received (significant only in the root)
- `recvcount`: number of elements to receive (significant only in the root)
- `recvtype`: type to receive (significant only to the root)
- `root`: Rank of the receiving process
- `comm`: communication group

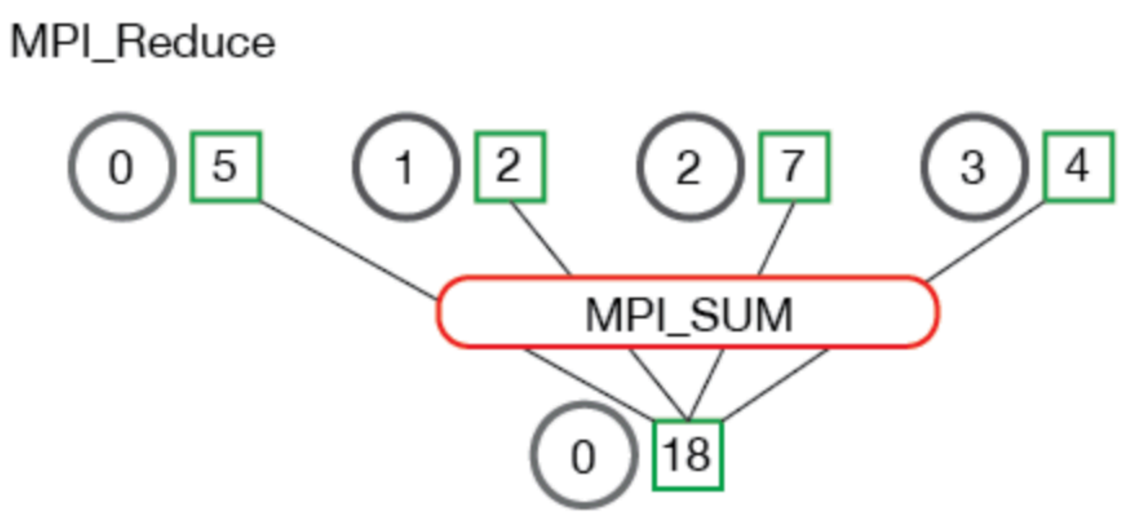
```
int MPI_Gather(
 const void *sendbuf, int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

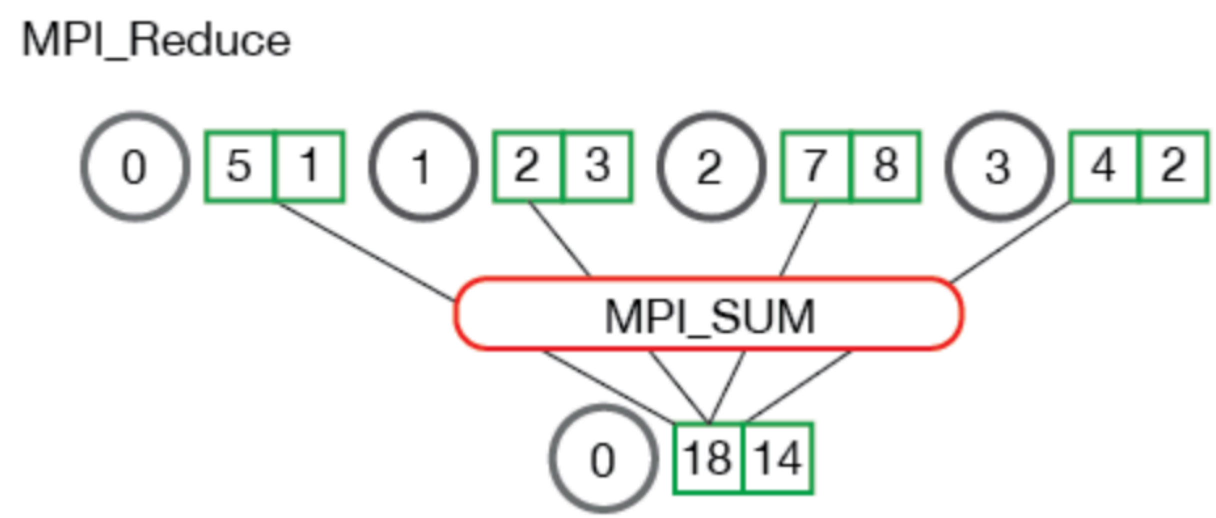
- Gather into the root information from the other nodes
- All the nodes must execute the call
- Every node in the communication group will send information to the root
- Notice the symmetry in the parameters, normally they should be the same

Example Scatter/Gather

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);

// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
}
```

MPI_Reduce





Images obtained from:
http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

```
int MPI_Reduce(
const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

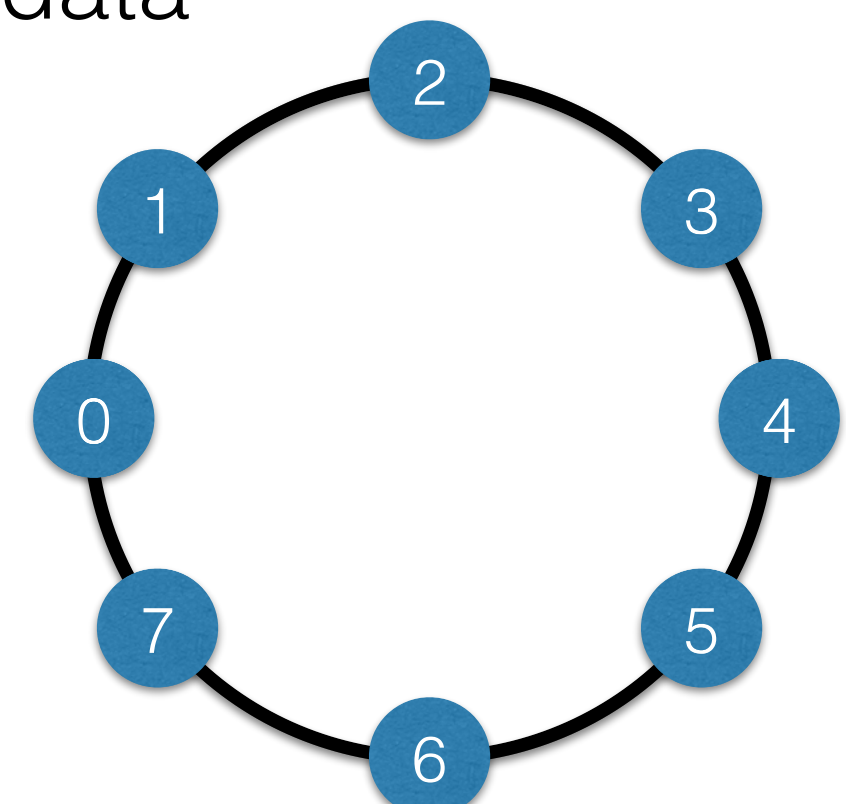
- `sendbuf`: address of send buffer
- `recvbuf`: where to store the result of reduction (significant only in the root)
- `count`: number of elements to receive (significant only in the root)
- `datatype`: type to receive (significant only to the root)
- `op`: Reduction operation to be performed
- `root`: Rank of the receiving process
- `comm`: communication group

```
int MPI_Reduce(
const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

- Available Reduction operations:
  - `MPI_MAX` - Returns the maximum element.
  - `MPI_MIN` - Returns the minimum element.
  - `MPI_SUM` - Sums the elements.
  - `MPI_PROD` - Multiplies all elements.
  - `MPI_LAND` - Performs a logical *and* across the elements.
  - `MPI_LOR` - Performs a logical *or* across the elements.
  - `MPI_BAND` - Performs a bitwise *and* across the bits of the elements.
  - `MPI_BOR` - Performs a bitwise *or* across the bits of the elements.
  - `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
  - `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

Ring Network Topology

- Each node connects to exactly two other nodes
- Can be (anti) clockwise or bi-directional
- Data goes through the network, each node "handles" the data



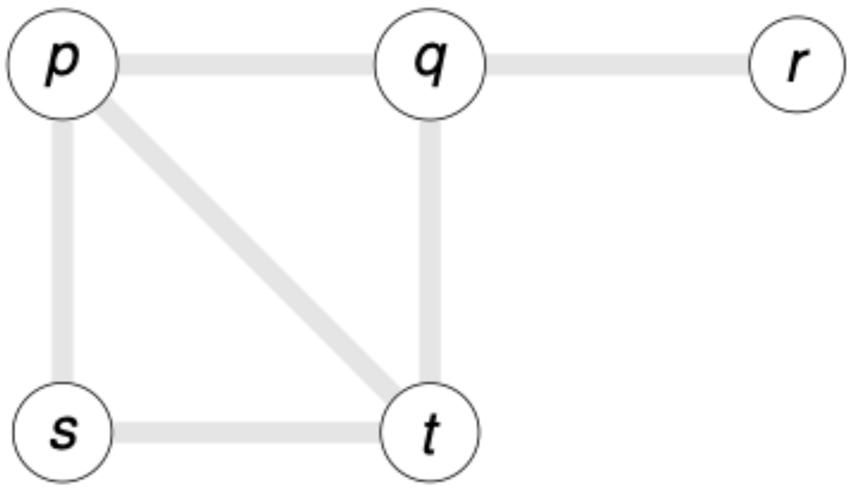
Advantages/Disadvantages

- Advantages
  - Very ordered
  - No central node
  - Simple to expand (Scalable)
- Disadvantages
  - Basic design is not fault tolerant, one bad node breaks the ring
  - Communication delay depends on size of the ring
  - Bandwidth is shared

Part A: Adjacency Matrix

- **Adjacency Matrix** is mirrored across main diagonal.
  - Bcoz an undirected network of processes.
- Use **MPI_Scatter** to send adjacency row to the correct MPI process.
  - No need to send entire matrix.
- If number of MPI processes is **N**, then Adjacency Matrix is **NxN**, and vice-versa.
- Each process MUST have at least one neighbor.

	p	q	r	s	t
p	0	1	0	1	1
q	1	0	1	0	1
r	0	1	0	0	0
s	1	0	0	0	1
t	1	1	0	1	0



Part A: Tarry’s Algorithm

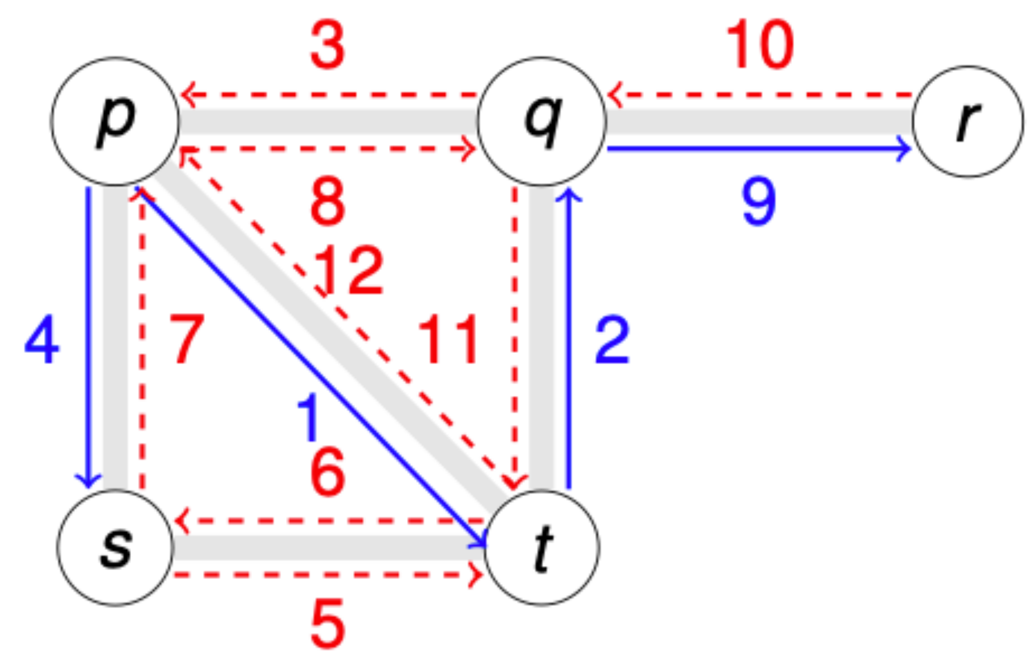
- **Tarry’s Algorithm Rules**

An initiator starts by sending out a token to its neighbors. A parent of a process is another process from which it first received a token.

- Rule 1: A process never forwards the token through the same channel twice.
- Rule 2: A process only forwards the token to its parent when there is no other option (i.e. no other link to send it on to)

- The token travels through each channel both ways, and finally ends up at the initiator.

	p	q	r	s	t
p	0	1	0	1	1
q	1	0	1	0	1
r	0	1	0	0	0
s	1	0	0	0	1
t	1	1	0	1	0



Solid arrows establish a parent-child relation (in the opposite direction).

Part A: Histogram

- Scatter Image to MPI processes.
- Calculate histogram per chunk on each MPI process.
- Using Tarry's Algorithm, add up/ collect the histogram values.
- Finally, write the 256 rows corresponding each bin in the output file. (256 rows, 1 column)

0	2	2	0
1	255	100	1
1	125	125	1
1	100	255	1
1	3	3	1

Pixel Intensity	Bins
0	2
1	8
2	2
3	2
.	
.	
100	2
.	
125	2
.	
.	
.	
.	
255	2

Some important pointers for the assignment!

- Do **<u>not</u>** use MPI_Bcast in the assignment.
- Part A use dynamic memory allocation for Image and Adjacency Matrices.
- Use **MPI_Scatter** to distribute required data.
    - Each process should only get the chunk of input data that it will work on.
        - No dynamic chunk allocation for this assignment
        - Part A: Image chunks, Adjacency Row
        - Part B: Book chunks
    - If some data needs to be used by two or more processes then use MPI_Send and MPI_Recv

Some important pointers for the assignment!

- Use **MPI_Gather** where it is possible to do so.
- Skeletons and input data is on Canvas. Timing mechanism is in there too.
- To get info about any MPI routine (on openlab):
    - ➡ module load openmpi
    - ➡ man <routine_name>
    - ➡ Example: man mpi_isend
- ICS Servers/Openlab Power Outage:
    - 5/4 8pm - 5/5 9am.