MPI

- Library intended for distributed, high-performance computing applications
  - The de-facto standard for high-performance computing (HPC)
  - Some say MPI is essentially “high-level sockets” (to be clear, that’s an insult)
  - More importantly: MPI provides high-level operations (in addition to typical point-to-point operations) that appear in many HPC apps
    - barrier, all-to-all, etc.

MPI, continued

- Library intended for distributed, high-performance computing applications
  - Programming model is SPMD (single program multiple data), where each process:
    - runs identical code image but operates on different data
    - occasionally executes global sync operations
  - Remember that the term SPMD is ill-defined (each node could run the same code *image*, but call a unique function)
  - MPI programmer writes one program
    - executed on N hosts, according to a user host file
      - if no host file, all MPI processes execute on the same machine

MPI programs

- Must have four functions
  - MPI_Init (note: implicit barrier)
  - MPI_Finalize (we're done; also implicit barrier)
- Practically, must have the following functions
  - MPI_Comm_size (returns total number of MPI processes)
  - MPI_Comm_rank (returns caller's process id)

- The actual computation is placed in between MPI_Comm_rank/MPI_Comm_size and MPI_Finalize

Sending and Receiving in MPI

- All four combinations of blocking/nonblocking send/receive are possible
  - MPI_Ssend (blocking send)
  - MPI_Isend (nonblocking send)
  - MPI_Recv (blocking receive)
  - MPI_Irecv (nonblocking receive)
  - MPI_Wait (paired with Isend or Irecv to make sure operation has completed; i.e., it is safe to overwrite [Send] or use [Recv] the data)

MPI_Send: nonblocking if data is small

Sending and Receiving in MPI

- MPI_Send (MPI_Recv) takes as parameters:
  - buffer, which is the data being sent or received
  - number of elements in the buffer
  - type of elements in the buffer
  - destination (source)
  - tag---must match other end for send/receive to match
  - “communicator”; always MPI_COMM_WORLD in 422
    - communicators can in general be used for non-global barriers

One extra parameter in MPI_Recv, which is the status (we will never rely on this; use MPI_STATUS_IGNORE)

Example use of MPI_Isend (with 2 MPI processes)

```
int A[10] = {…}, B[10];
MPI_Status status; MPI_Request request;

// initialization here (MPI_Init, MPI_Comm_rank, MPI_Comm_size)
if (myId == 0) {
  MPI_Isend(A, 10, MPI_INT, 1, 17, MPI_COMM_WORLD, &request);
  // do anything that doesn't involve writing to A (if A is written here, it's a race condition)
  MPI_Wait(&request, &status);
  // now can safely overwrite A
}
else {
   MPI_Recv(B, 10, MPI_INT, 0, 17, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
```

Example use of MPI_Irecv (with 2 MPI processes)

```
int A[10] = {…}, B[10];
MPI_Status status; MPI_Request request;

// initialization here (MPI_Init, MPI_Comm_rank, MPI_Comm_size)
if (myId == 0) {
  MPI_Irecv(B, 10, MPI_INT, 1, 17, MPI_COMM_WORLD, &request);
  // do anything that doesn't involve using B (if B is used here, it's a race condition)
  MPI_Wait(&request, &status);
  // now can safely use B
}
else {
   MPI_Send(A, 10, MPI_INT, 0, 17, MPI_COMM_WORLD);
 }
```

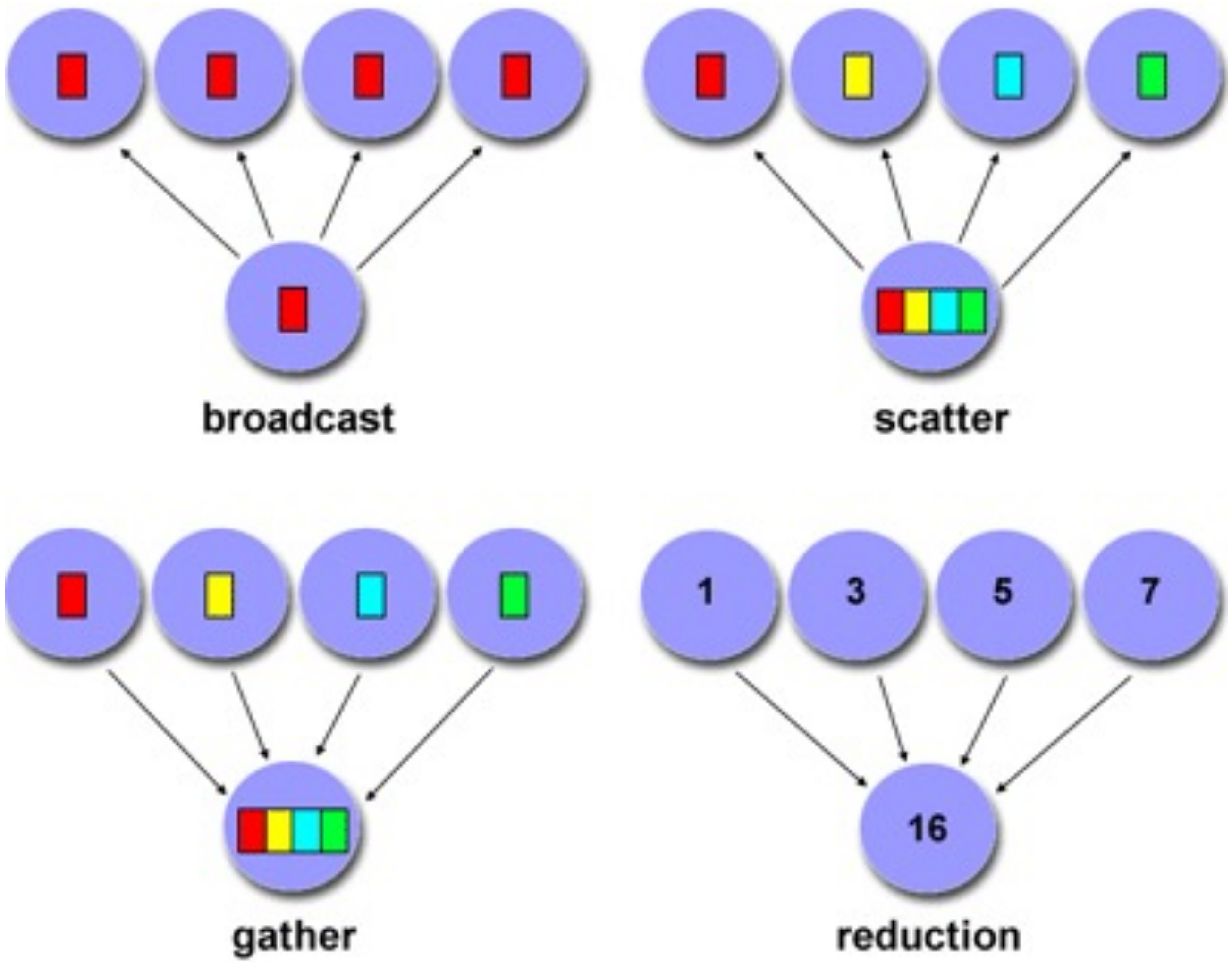
Collective Communication in MPI

- Collective calls are ones that involve all processes in a communicator (for this class, this means all processes)
  - MPI_Bcast (usual definition; one sends to many)
  - MPI_Scatter (given array on root; send equal-size subarray to each non-root process; awkward because must each specify sendbuf/recvbuf)
  - MPI_Gather (reverse of MPI_Scatter)
  - MPI_Reduce (reduced value ends up at root)
  - MPI_Allreduce (MPI_Reduce + dissemination to all)

Collective Communication in MPI

- Collectives aren't strictly needed
  - I.e., every collective can be implemented with some sequence of sends and receives
- Collectives have advantages, though:
  - Easier for the programmer
  - MPI runtime knows about the operation ahead of time, so it can implement it efficiently
    - Example: MPI_Allreduce can use a tree of log(P) levels or a tree of 1 level

Collective Communication in MPI



Running an MPI Program

- To compile, use `mpicc`

  Example: `mpicc -o mm mpi-mm.c`

- To run, **you must use** `mpirun`. Use the `-n` option to specify number of MPI processes
- Also, mpirun arguments come first, then executable, then program command line args

  Example: `mpirun -n 4 ./mm 100`

Args to mpirun

Args to mm program

Executable

Collective Communication in MPI

- MPI_Alltoall (every process sends a unique part of its buffer to each other process)
- MPI_Alltoallv (generalized MPI_Alltoall; parts of buffer can have variable size)

Collective Communication in MPI