CSC367 Parallel computing

Lecture 12: Distributed Memory Architectures and their Parallel Programming Model-Cont.

Building Blocks

- Interactions are carried out via passing messages between processes
- Building blocks: send and receive primitives – general form:
  - `send(void *sendbuf, int nelems, int dest)`
  - `receive(void *recvbuf, int nelems, int source)`
- Complexity lies in how the operations are carried out internally
- Example (pseudocode):

```
P0                          P1
---                         ---
msg = 5;                    recv(&msg, 1, 0);
send(&msg, 1, 1);           printf("%d", msg);
msg = 200;
```

- Key question: What will P1 receive?
  - Send operation may be implemented to return before the receipt is confirmed
  - Supporting this kind of send is not a bad idea

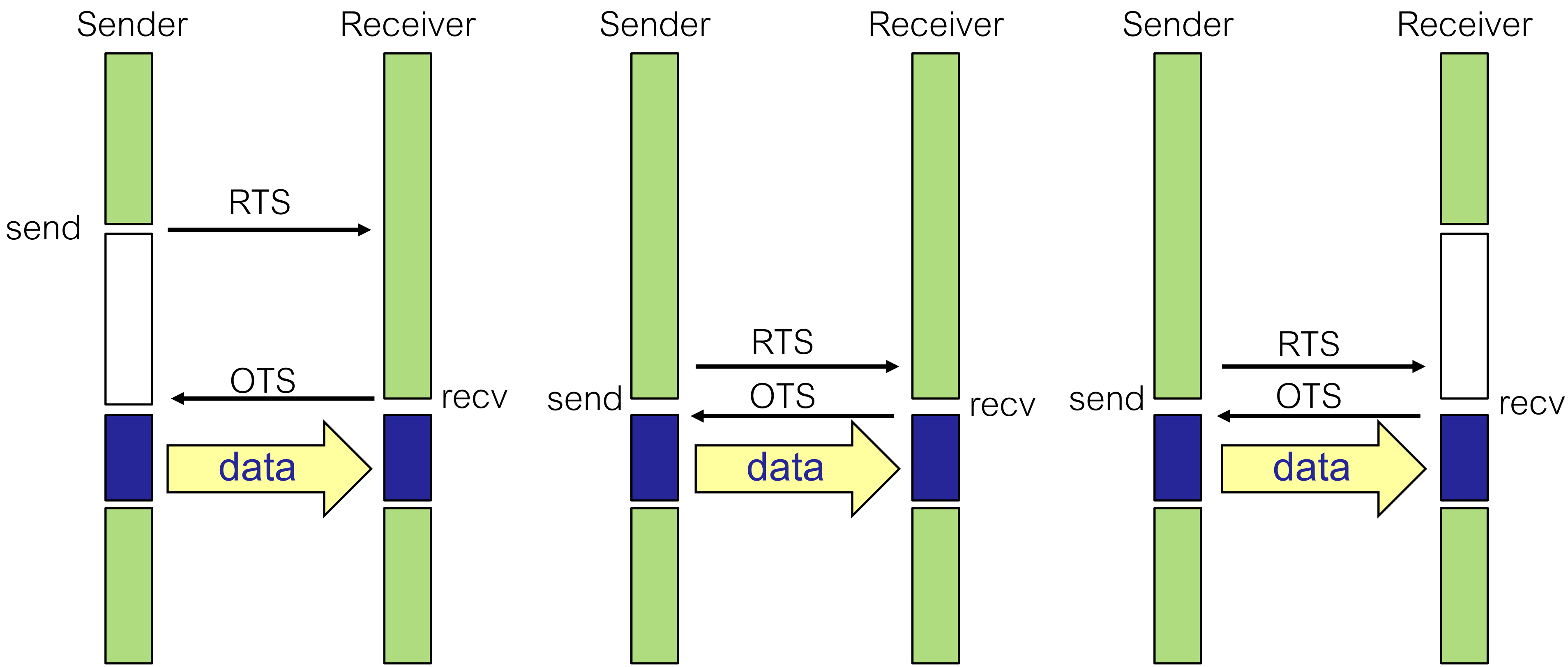
Blocking operations

- Only return from an operation once it's safe to do so
  - Not necessarily when the msg has been received, just guarantee semantics
- Two possibilities:
  - Blocking non-buffered send/receive
  - Blocking buffered send/receive

Blocking non-buffered send/recv

- Send operation **does not return** until matching receive is encountered at the receiver and communication operation is completed
- Non-buffered handshake protocol – idling overheads:

RTS = request to send
OTS = ok to send



1. Sender is first; idling at sender	2. Same time; idling minimized	3.Receiver is first; idling at recv

Deadlocks in blocking non-buffered comm

- Deadlocks can occur with certain orderings of operations, due to blocking
- Example – this deadlocks:

```
P0                              P1
---                             ---
send(&m1, 1, 1);                send(&m1, 1, 0);
recv(&m2, 1, 1);                recv(&m2, 1, 0);
```

- Solution: switch order in one of the processes
  - But, more difficult to write code this way, and could create bugs

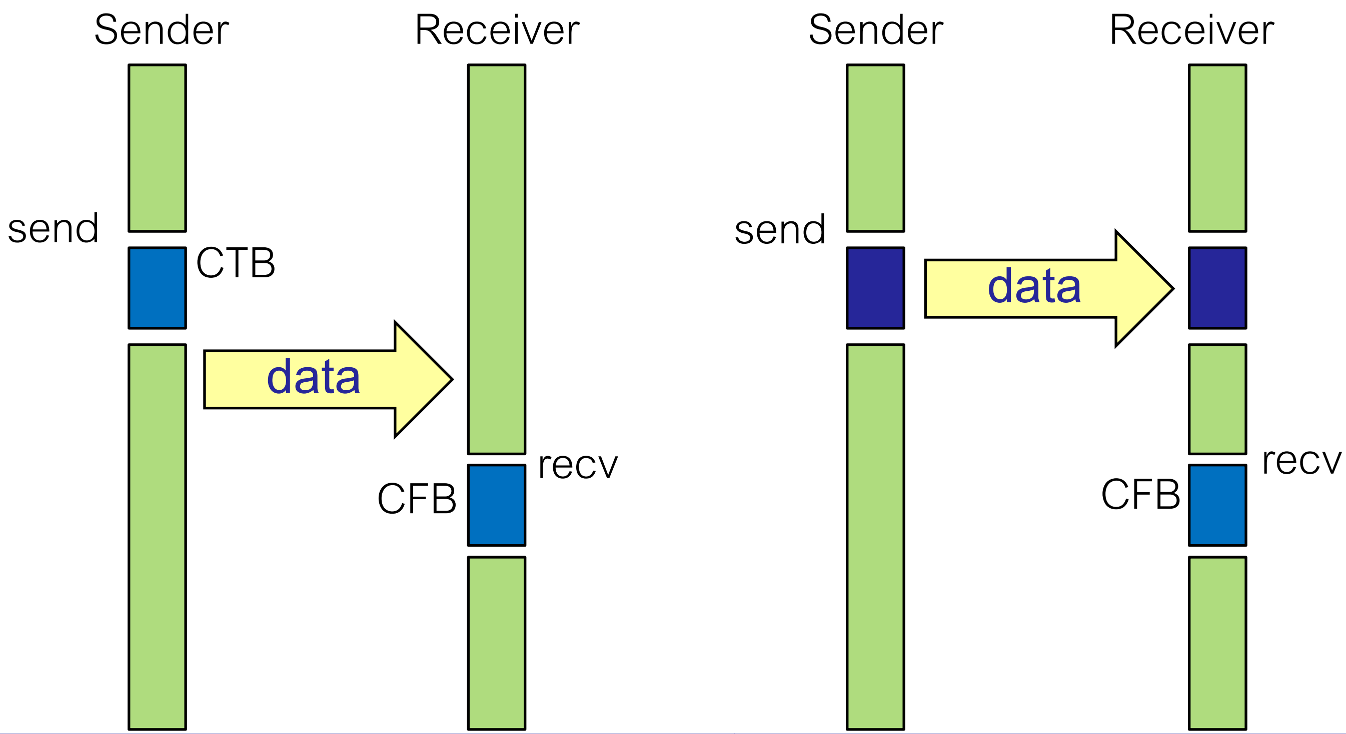
Hardware support for send/receives

- Most message passing platforms have additional hardware support for sending and receiving messages.

- They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware.
    - **Network interfaces** allow the transfer of messages from buffer memory to desired location without CPU intervention.
    - Similarly, **DMA** allows copying of data from one memory location to another (e.g., communication buffers) without CPU support

Blocking buffered send/recv

- Sender copies data into buffer and returns once the copy to buffer is completed
- Receiver must also store the data into a buffer until it reaches the matching recv
- Buffered transfer protocol – with or without hardware support:



CTB = Copy to buffer

CFB = Copy from buf

In both cases, no idling overheads! But, now buffer management overheads!

1. Use buffer at both sender and receiver Communication handled by H/W (network interface)

2. Buffer only on one side. E.g., sender interrupts receiver and deposits the data in a buffer (or vice-versa)

Problems with blocking buffered comm

- 1. Potential problems with finite buffers
  - Example:

```
P0 (producer)
---
for(i = 0; i < 1000000; i++){
   create_message(&m);
   send(&m, 1, 1);
}
```

```
P1 (consumer)
---
for(i = 0; i < 1000000; i++){
   recv(&m, 1, 0);
   digest_message(&m);
}
```

- 2. Deadlocks still possible
  - Example:

```
P0
---
recv(&m1, 1, 1);
send(&m2, 1, 1);
```

```
P1
---
recv(&m1, 1, 0);
send(&m2, 1, 0);
```

- Solution is similar: break circular waits
- Unlike previously, in this protocol, deadlocks can only be caused by waits on recv

Non-blocking operations

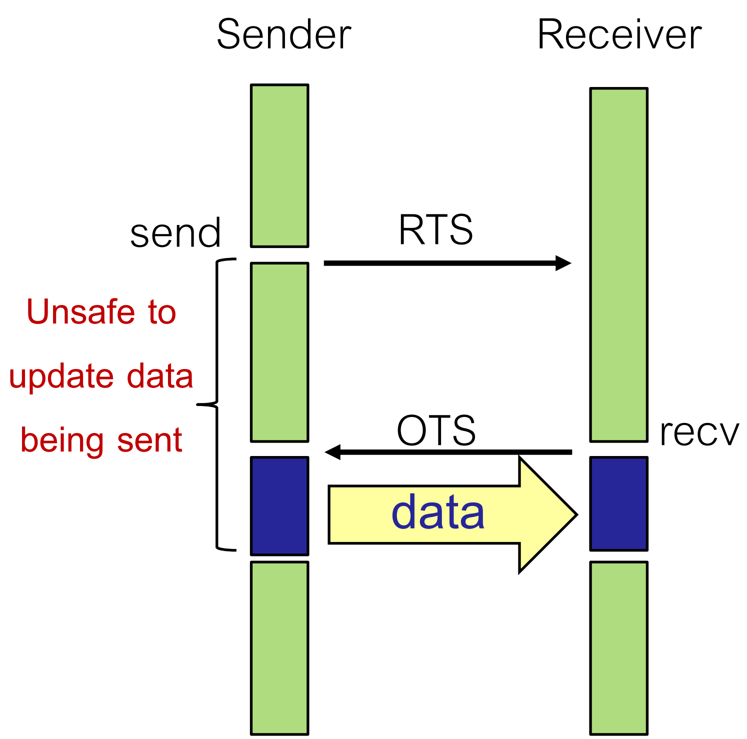
- Why non-blocking? Performance!
- User is responsible to ensure that data is not changed until it's safe
  - Typically a check-status operation indicates if correctness could be violated by a previous transfer which is still in flight
- Can also be buffered or non-buffered
  - Can be implemented with or without hardware support

Example: non-blocking non-buffered

- Sender issues a request to send and returns immediately
- When the receive is encountered, communication is initiated

RTS	= request to send
OTS	= ok to send

Without hardware support



1. When recv is encountered, transfer is handled by interrupting the sender

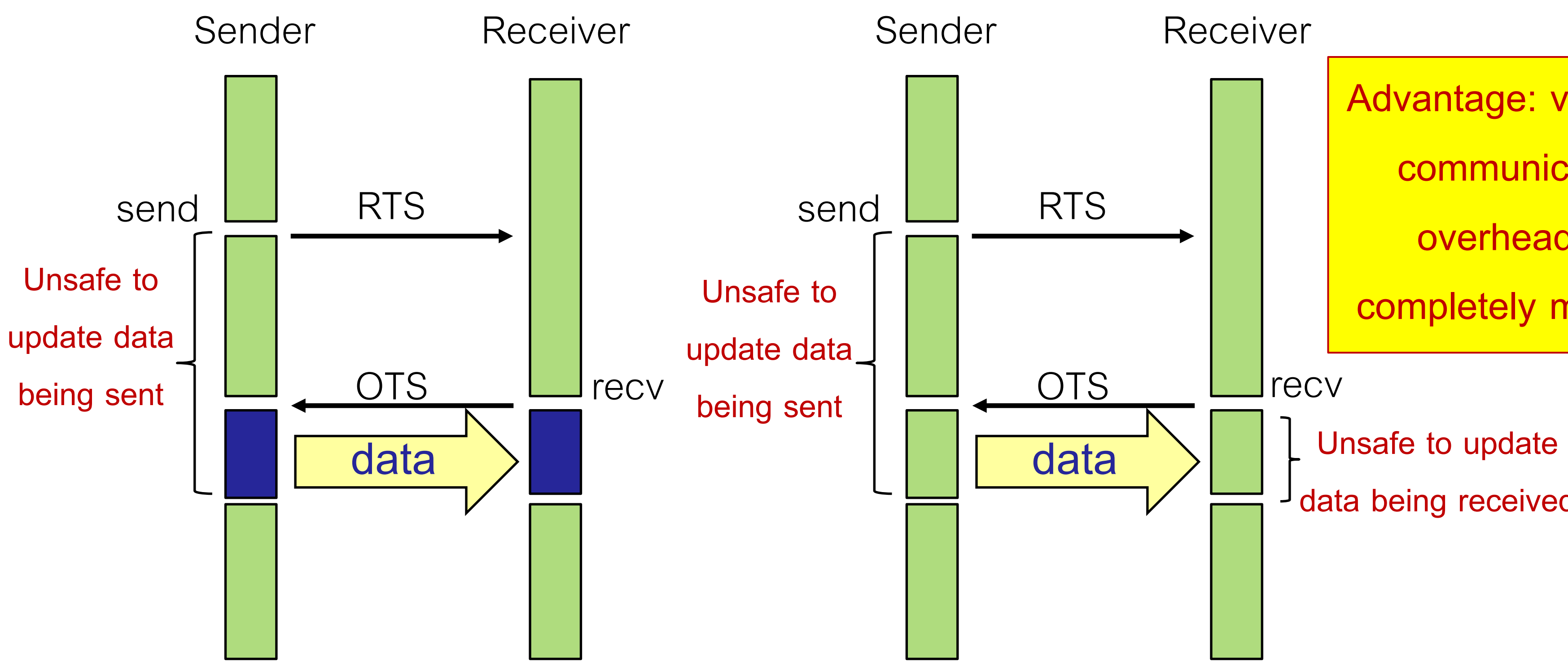
Example: non-blocking non-buffered

- Sender issues a request to send and returns immediately
- When the receive is encountered, communication is initiated

RTS = request to send

OTS = ok to send

Without hardware support | With hardware support



Advantage: very little communication overhead, or completely masked!

1. When recv is encountered, transfer is handled by interrupting the sender	2. When recv is found, comm. hardware handles the transfer and receiver can continue doing other work

Summery

	Buffered	Non-Buffered	
Blocking Operations	Sending process returns after data has been copied into communication buffer.	Sending process blocks until matching receive operation has been encountered.	send and recv semantics ensured by corresponding operation
Non-blocking Operations	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return.		Programmer must explicitly ensure semantics by polling to verify completion

Take-aways

- Carefully consider the implementation guarantees
  - Communication protocol and hardware support
  - Blocking vs. non-blocking, buffered vs. non-buffered
- Tradeoffs in terms of correctness and performance
  - Need automatic correctness guarantees => might not hide communication overhead that well
  - Need performance => user is responsible for correctness via polling

The Message Passing Interface (MPI)

MPI standard

- **Standard** library for message passing
  - Write portable message passing algorithms, mostly using C or Fortran
  - Rich API (over 100 routines, but only a handful are fundamental)
  - Must install OpenMPI or MPICH2, etc.
  - Include mpi.h header
- Example run command: `mpirun -np 8 ./myapp arg1 arg2`
- Basic routines:

MPI_Init: initialize MPI environment
MPI_Finalize: terminate the MPI environment
MPI_Comm_size: get number of processes
MPI_Comm_rank: get the process ID of the caller
MPI_Send: send message
MPI_Recv: receive message

MPI basics

- MPI_Init: only called once at start by one thread, to initialize the MPI environment
  - **`int MPI_init(int *argc, char ***argv);`**
  - Extracts and removes the MPI parts of the command line (e.g., mpirun –np 8) from argv
  - Process your application's command line arguments only after the MPI_Init
  - On success => MPI_SUCCESS, otherwise error code
- MPI_Finalize: called at the end, to do cleanup and terminate the MPI environment
  - **`int MPI_Finalize();`**
  - On success => MPI_SUCCESS, otherwise error code
  - No MPI calls allowed after this, not even a new MPI_init!
- These calls are made by all participating processes, otherwise results in undefined behaviour

MPI Communication domains

- MPI communication domain = set of processes which are allowed to communicate with each other
- Communicators (`MPI_Comm` variables) store info about communication domains
- Common case: all processes need to communicate to all other processes
  - Default communicator: `MPI_COMM_WORLD` includes all processes
- In special cases, we may want to perform tasks in separate (or overlapping) groups of processes => define custom communicators
  - No messages for a given group will be received by processes in other groups
- Communicator size and id of current process can be retrieved with:
  - `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - The process calling these routines must be in the communicator `comm`

Hello (C)

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Timing measurements

- Can use MPI_Wtime()
- Example:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf("Elapsed time: %f\n", t2 - t1);
```

MPI data types

- Equivalent to built-in C types, except for MPI_BYTE and MPI_PACKED

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A
MPI_PACKED	N/A

Flavors of communication in MPI

- Collective operations: All processes in the communicator or group have to participate!
    - Barrier, Broadcast, Reduction, Prefix sum, Scatter / Gather, All-to-all, etc.

- Point-to-point operations: A processor explicitly communicants with another processor with send and receive messages

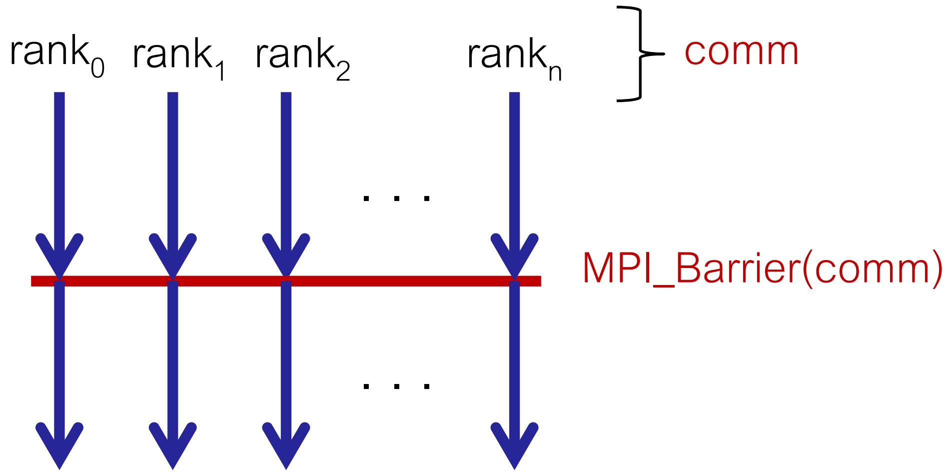
Collective communication / computation

- Common collective operations
  - Barrier
  - Broadcast
  - Reduction
  - Prefix sum
  - Scatter / Gather
  - All-to-all
- All processes in the communicator or group have to participate!

Barrier

- Blocks until all processes in the given communicator hit the barrier

```
int MPI_Barrier(MPI_Comm comm)
```



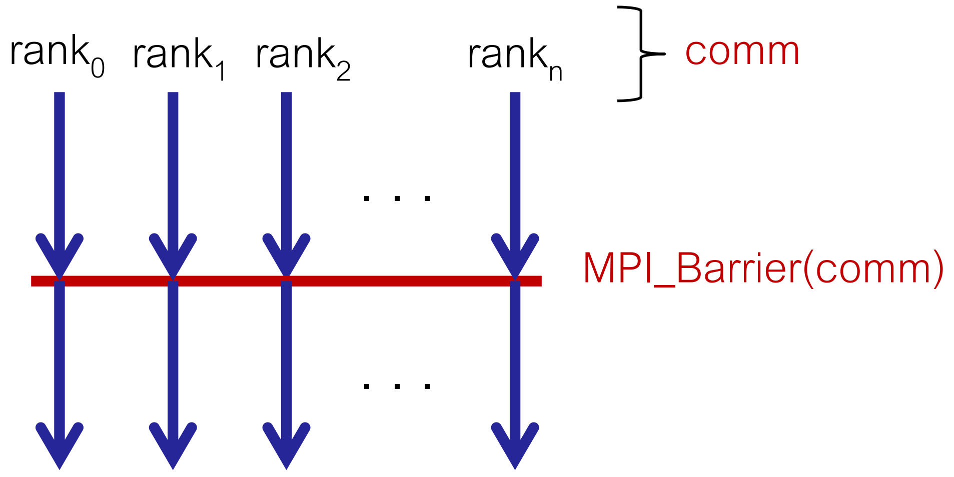
- Warning1: Careful with potential deadlocks!

```
if(my_rank % 2 == 0) {
    // do stuff
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Barrier

- Blocks until all processes in the given communicator hit the barrier

```
int MPI_Barrier(MPI_Comm comm)
```



- Warning2: Barrier does not magically wait for pending non-blocking operations!
  - If you are using nonblocking sends/receives and want the guarantee that the processes sent/received all data after the MPI_Barrier you should use MPI_Wait, more on this later!

Broadcast

- One-to-all: send buf of source to all other processes in the group (into their buf)

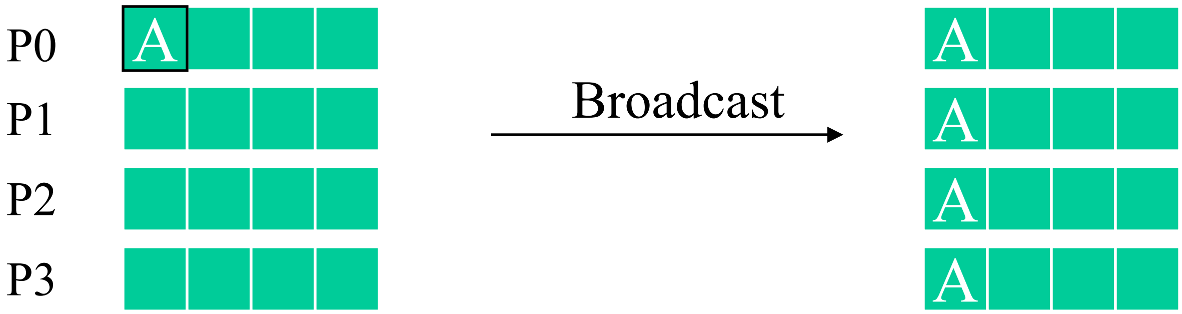
```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

dont!

- Common misconception: receiver processes have to do an MPI_Recv
- MPI_Bcast blocks until all processes make a matching MPI_Bcast call: It is not required to block on all processes until the operation fully completes though

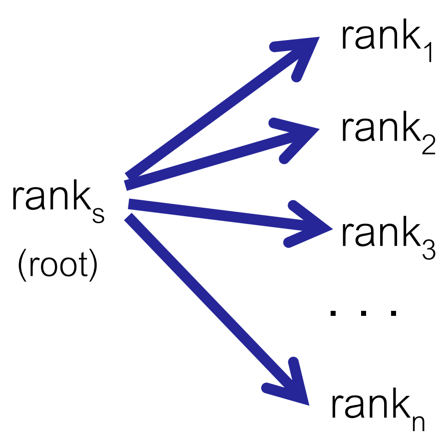
Simple example

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```

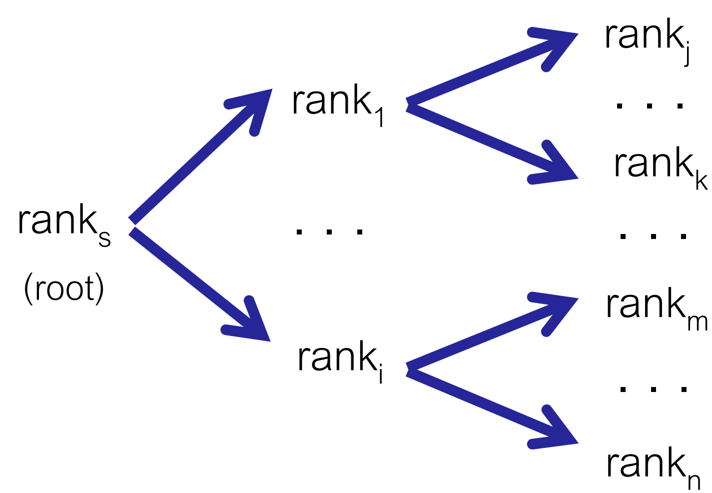


Broadcast vs. Send/Recv

- Why not implement this using Send/Recv pairs? Is MPI_Bcast just a fancy wrapper?
- Bcast communication pattern is optimized internally by the MPI library: In a Send/Recv implementation one processors send to all while Bcast uses a tree-based hierarchy to broadcast, removing contention from the root!



loop over all other ranks and issue Send/Recv



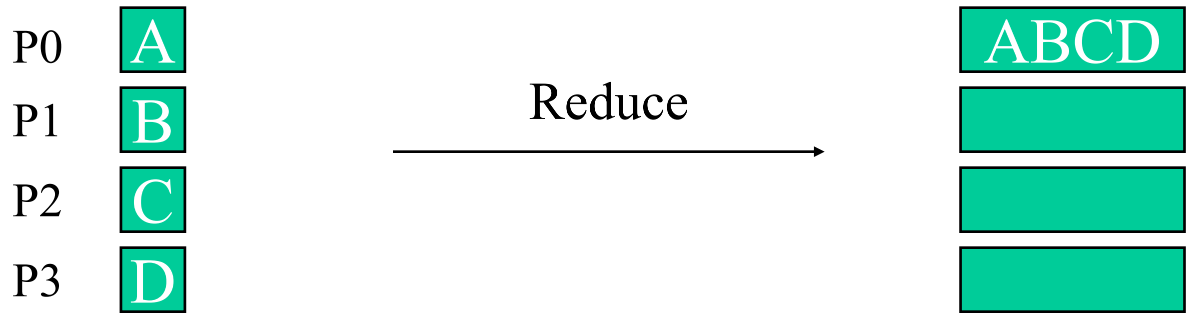
broadcast carried out hierarchically

Reduction

- Reduce: combines the elements from buffer of each process in the group and stores result in recvbuf at the target receiver

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

- All processes must provide send and recv buffers of the same size and data type
- Built-in operations: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, etc. (see docs!)
  - Allows user-defined operations as well (see MPI documentation)



Reduction to all ranks

- If all processes need the reduction result: MPI_Allreduce

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

- No target necessary, all processes get the result in their own recvbuf
- Example: calculate standard deviation
  - calculate average
  - calculate sums of all the squared differences from the mean
  - square root the average of the sums to get the standard deviation
- Use MPI_Reduce and MPI_Allreduce to implement this

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

```
rand_nums = create_rand_nums(num_elements_per_proc);
// Sum the numbers locally
 float local_sum = 0; int i;
for (i = 0; i < num_elements_per_proc; i++) { local_sum += rand_nums[i]; }

// Reduce all of the local sums into the global sum in order to // calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++)
{ local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean); }

// Reduce the global sum of the squared differences to the root process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the // squared differences.
if (world_rank == 0)
       { float stddev = sqrt(global_sq_diff / (num_elements_per_proc * world_size));
printf("Mean - %f, Standard deviation = %f\n", mean, stddev); }
```

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

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

```
rand_nums = create_rand_nums(num_elements_per_proc);
// Sum the numbers locally
 float local_sum = 0; int i;
for (i = 0; i < num_elements_per_proc; i++) { local_sum += rand_nums[i]; }

// Reduce all of the local sums into the global sum in order to // calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++)
{ local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean); }

// Reduce the global sum of the squared differences to the root process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the // squared differences.
if (world_rank == 0)
       { float stddev = sqrt(global_sq_diff / (num_elements_per_proc * world_size));
printf("Mean - %f, Standard deviation = %f\n", mean, stddev); }
```

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

```
rand_nums = create_rand_nums(num_elements_per_proc);
// Sum the numbers locally
 float local_sum = 0; int i;
for (i = 0; i < num_elements_per_proc; i++) { local_sum += rand_nums[i]; }

// Reduce all of the local sums into the global sum in order to // calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++)
{ local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean); }

// Reduce the global sum of the squared differences to the root process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the // squared differences.
if (world_rank == 0)
      { float stddev = sqrt(global_sq_diff / (num_elements_per_proc * world_size));
printf("Mean - %f, Standard deviation = %f\n", mean, stddev); }
```

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

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

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

```
rand_nums = create_rand_nums(num_elements_per_proc);
// Sum the numbers locally
 float local_sum = 0; int i;
for (i = 0; i < num_elements_per_proc; i++) { local_sum += rand_nums[i]; }

// Reduce all of the local sums into the global sum in order to // calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++)
{ local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean); }

// Reduce the global sum of the squared differences to the root process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the // squared differences.
if (world_rank == 0)
      { float stddev = sqrt(global_sq_diff / (num_elements_per_proc * world_size));
printf("Mean - %f, Standard deviation = %f\n", mean, stddev); }
```

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

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

```
rand_nums = create_rand_nums(num_elements_per_proc);
// Sum the numbers locally
 float local_sum = 0; int i;
for (i = 0; i < num_elements_per_proc; i++) { local_sum += rand_nums[i]; }

// Reduce all of the local sums into the global sum in order to // calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++)
{ local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean); }

// Reduce the global sum of the squared differences to the root process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the // squared differences.
if (world_rank == 0)
      { float stddev = sqrt(global_sq_diff / (num_elements_per_proc * world_size));
printf("Mean - %f, Standard deviation = %f\n", mean, stddev); }
```

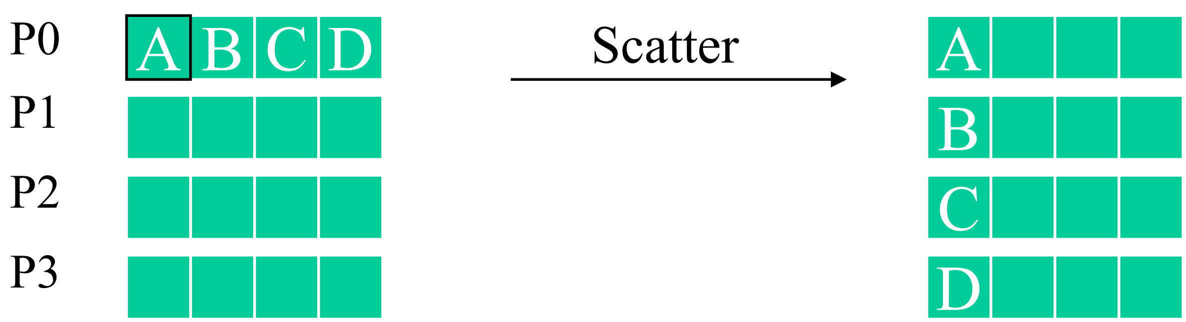
http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Scatter

- The source process sends a different part of sendbuf to all others (including itself)

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                int source, MPI_Comm comm)
```

  - Send recvcount contiguous elements to each process (all of them receive the same amount)
  - sendcount is the number of elements sent to each individual process
  - Send-related args are only applicable to the source, and are ignored for all others
  - MPI_Scatterv variant – allows a *different* number of items to be sent to each of the receivers

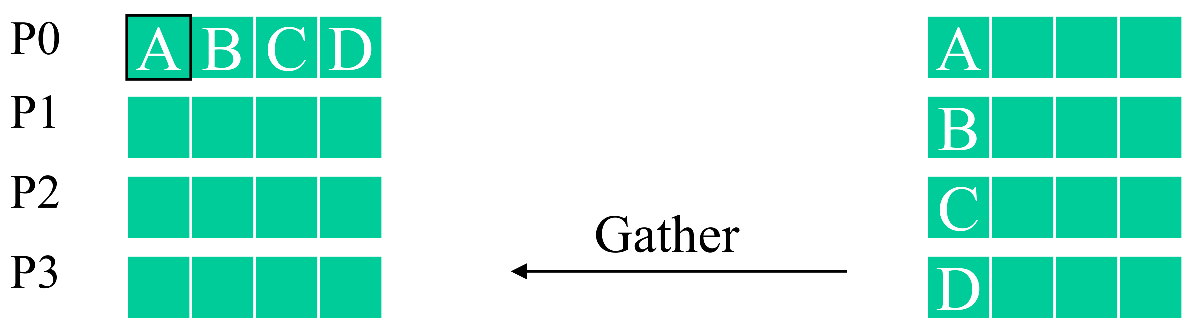


Gather

- Each process (including target) send their sendbuf data to the target process

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
               void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
```

  - The sent data must be of the same size and type at all processes
  - Recv-related args are only applicable to the target, and are ignored for all others
  - recvcount is the count received per process, not the total sum of counts from all processes
  - See also: MPI_Gatherv variant

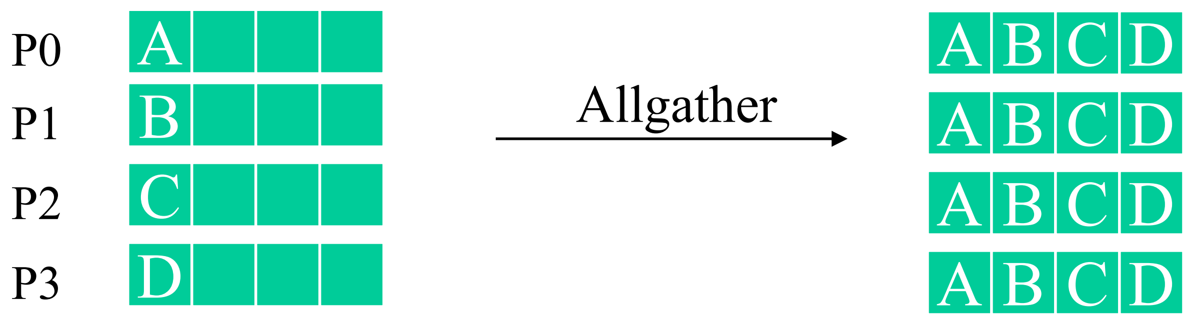


All-Gather

- Same as Gather, but data is gathered to all the processes, not just one target

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

  - Unlike Gather, all processes must provide a valid recvbuf to store incoming data



All-to-all

- Each process sends a different portion of sendbuf (sendcount contiguous items) to each other process (in order or rank), including itself

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype,
                 void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
                 MPI_Comm comm)
```

- Also, vector variant MPI_Alltoallv (see documentation for further details)