CSC367 Parallel computing

Lecture 14: Distributed Memory Architectures and their Parallel Programming Model-Cont.

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)

```
#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

Point-to-point communication

Blocking sends and receives

Non-blocking sends and receives

Sending and receiving messages

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

- Each message has a tag associated to distinguish it from other messages
- The source and tag can be **MPI_ANY_SOURCE/MPI_ANY_TAG**
- The status can be used to get info about the **MPI_recv** operation:

```
typedef struct MPI_Status {
   int MPI_SOURCE; // source of the received message
   int MPI_TAG;    // tag of the received message
   int MPI_ERROR;  // a potential error code
};
```

- The length of the received message can be retrieved using:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

Implementations of MPI_Recv and MPI_Send

- **MPI_Recv** is blocking
  - It returns only after message is received and copied into the buffer!
  - Buffer can be safely reused right after MPI_Recv
- **MPI_Send** can be implemented with two options:
  - Option 1: returns only after the matching MPI_Recv is executed and the message was sent
  - Option 2: copy msg into buf and returns, without waiting for MPI_Recv
  - In both, the buffer can be safely reused right after MPI_Send

Deadlock avoidance

- Restrictions on MPI_Send/MPI_Recv, in order to avoid deadlocks
- Example: behaviour is MPI_Send implementation-dependent

```
int a[20], b[20], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 20, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 20, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 20, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
  MPI_Recv(a, 20, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
...
```

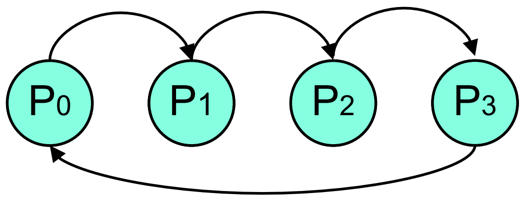
- If MPI-send is option 1 then deadlock, option 2 leads to no deadlock!
- Fix by matching the order of the send and recv operations
- We want to write "safe" programs which are not implementation dependent!

Deadlock avoidance

- Another example: Circular chain of send/recv operation:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
```

- Works fine if send proceeds after copying to data

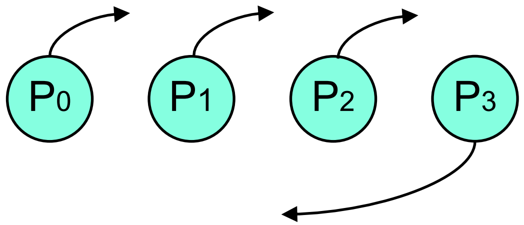


Deadlock avoidance

- Another example: Circular chain of send/recv operation:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
```

- Works fine if send proceeds after copying to data, but deadlocks if send has to wait for the receive.



- Must rewrite the code to make it safe:

Sends are waiting for receive: deadlock!

Deadlock avoidance

- Another example: Circular chain of send/recv operation:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
```

- Must rewrite the code to make it safe:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank % 2 == 1) {
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
}
else if (myrank % 2 == 0) {
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
}
```

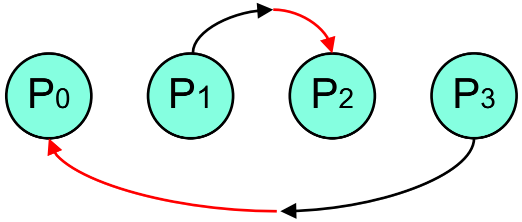
Deadlock avoidance

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank % 2 == 1) {
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
}
else if (myrank % 2 == 0) {
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
}
```

- The above code resolves the deadlock problem and is safe independent of the send implementation.

Deadlock avoidance

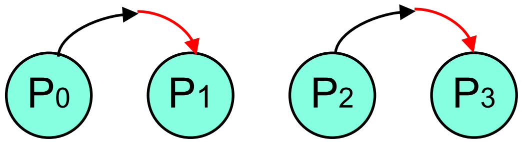
```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank % 2 == 1) {
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
}
else if (myrank % 2 == 0) {
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
}
```



Odd-numbered processors send while even-numbers processors receive

Deadlock avoidance

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank % 2 == 1) {
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
}
else if (myrank % 2 == 0) {
  MPI_Recv(b,20, MPI_INT, (myrank-1+npes)%np, 1, MPI_COMM_WORLD,&status);
  MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
}
```



Now even-numbered processors send while odd-numbers processors receive

Send/recv simultaneously

- Previous example is a common pattern

  => Combine the MPI_Send and MPI_Recv primitives into MPI_Sendrecv

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);
```

- Previous program becomes easier to write:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Sendrecv(a, 20, MPI_INT, (myrank+1)%np, 1,
             b, 20, MPI_INT, (myrank-1+npes)%np, 1,
             MPI_COMM_WORLD, &status);
```

- Restriction: send and receive buffers must be disjoint, otherwise use:

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
                         int dest, int sendtag,
                         int source, int recvtag,
                         MPI_Comm comm, MPI_Status *status);
```

Overlap communication with computation

- MPI provides non-blocking primitives
    - MPI_Isend starts a send operation, and returns before the data is copied out of the buffer
    - MPI_Irecv starts a recv operation, and returns before the data is received into the buffer

```
int MPI_Isend(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 src, int tag, MPI_Comm comm, MPI_Request *request)
```

- Allocate a request object and return a pointer to it in the request argument (details later)

- Non-blocking operation can be matched with a corresponding blocking operation

Overlap communication with computation

- Problems with using these non-blocking primitives
  - At some point, we need the data to be guaranteed to have been sent/received, otherwise violates correctness

- Use MPI_Test and/or MPI_Wait to determine whether a non-blocking operation has finished, and/or to wait (block) until the non-blocking operation is completed
  - Use the request object provided by MPI_Isend/ MPI_Irecv to test/wait on the completion

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

  - Returns non-zero if completed (if so, request is deallocated, set to MPI_REQUEST_NULL)

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

  - Blocks until request completes, then deallocates request and sets it to MPI_REQUEST_NULL
  - Status is similar to the one for the blocking Send/Recv

Avoiding deadlocks

- Recall deadlock example for blocking operations
  - Implementation dependent – may cause deadlocks

```
int a[20], b[20], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 20, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 20, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 20, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
  MPI_Recv(a, 20, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
```

Avoiding deadlocks

- Recall deadlock example for blocking operations
  - Implementation dependent – may cause deadlocks

```
int a[20], b[20], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 20, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 20, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Irecv(b, 20, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
  MPI_Irecv(a, 20, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
}
```

- If we replace either MPI_Send or MPI_Recv operations with non-blocking versions, the code will be safe, regardless of MPI implementation.

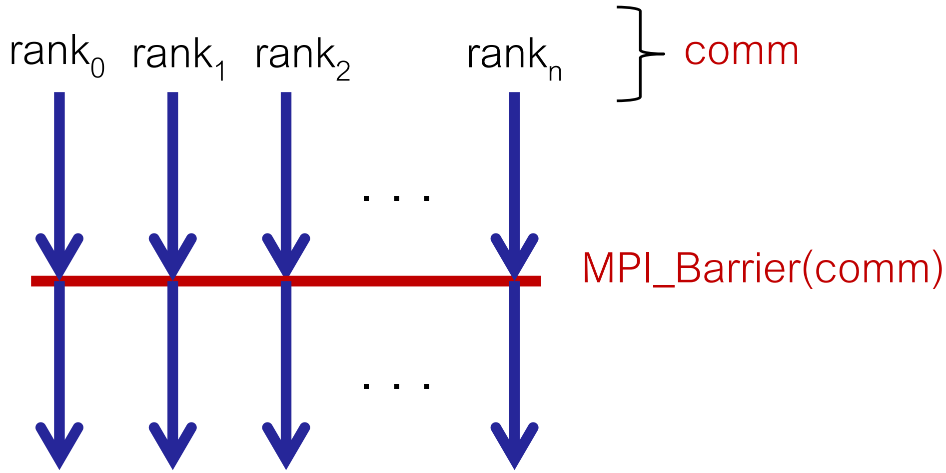
Collective communication / computation

- 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)
```



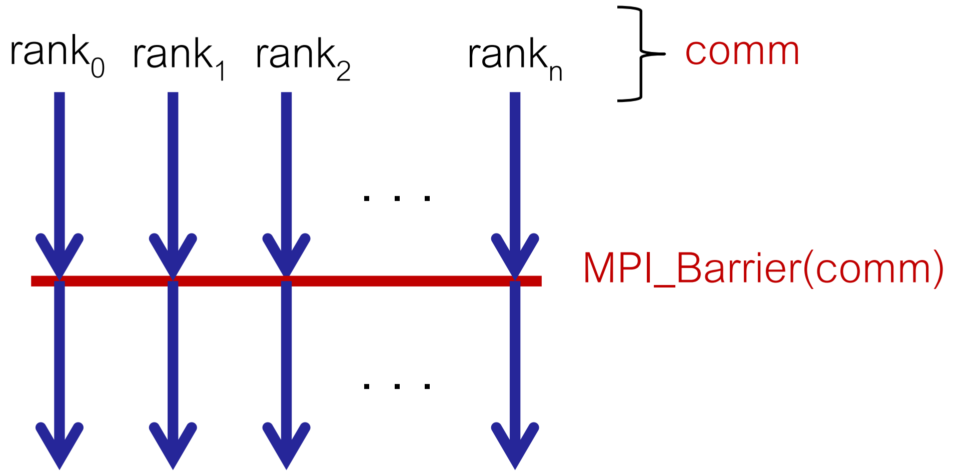
- Warning1: Careful with potential deadlocks!

```
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.

Broadcast

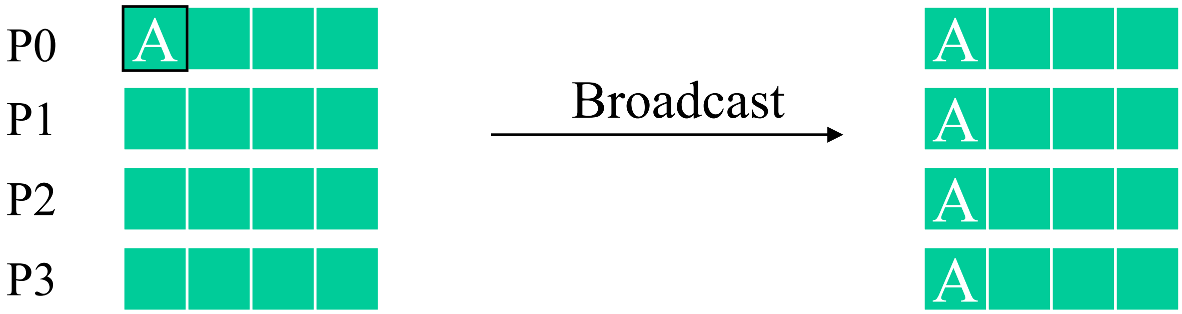
- One-to-all: send buf of source to all other processes in the group (into their buf)

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

- 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

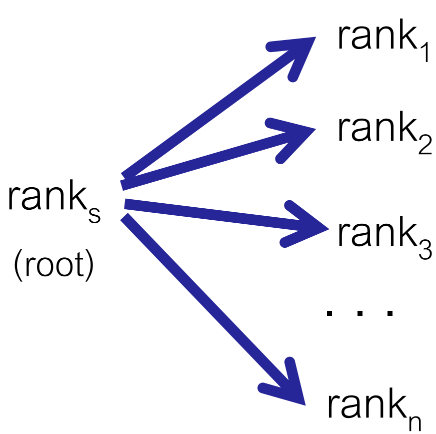
Simple example

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

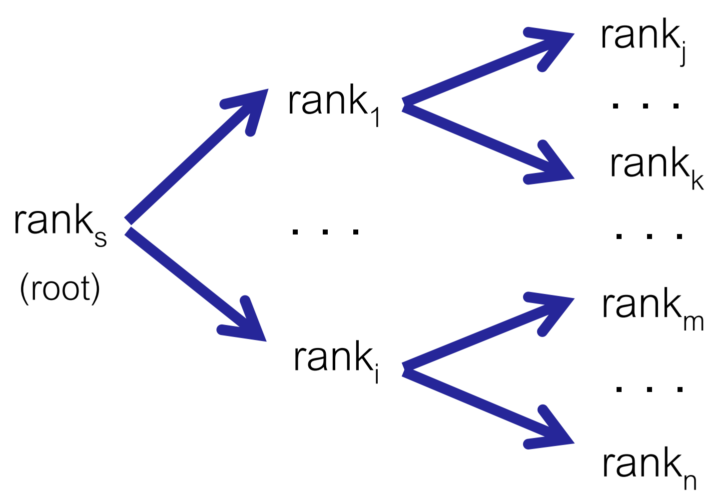


Broadcast vs. Send/Recv

- 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



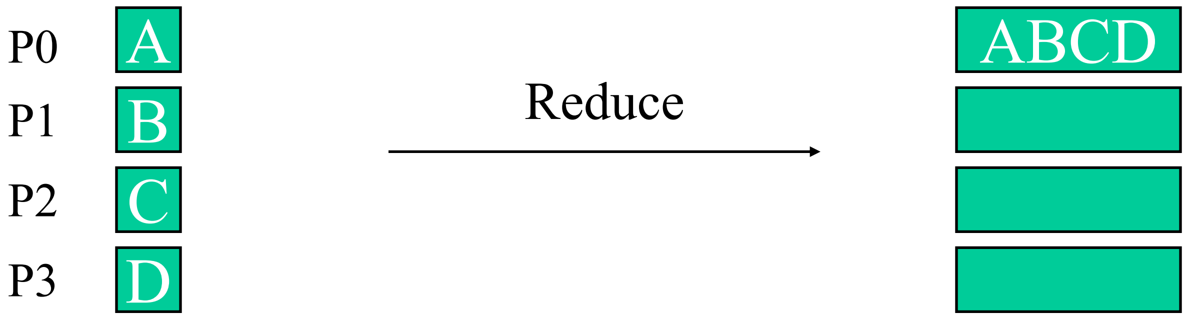
broadcast carried out hierarchically

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/

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

```
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/

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

```
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); }
```

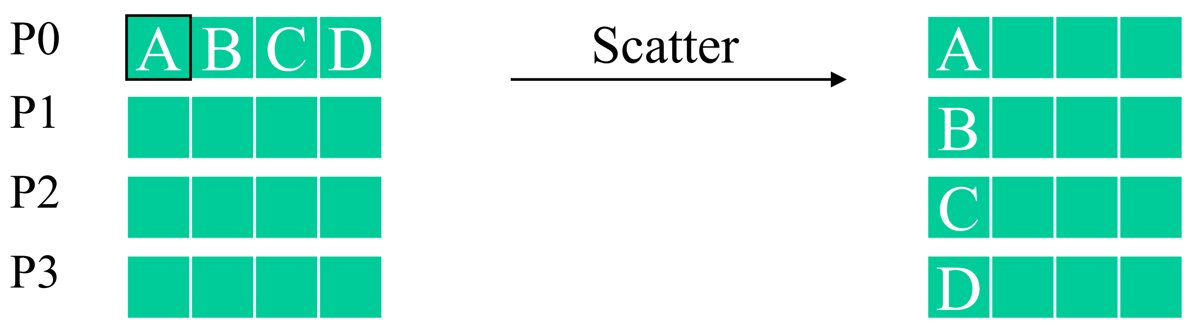
http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

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, see MPI manual (might help for project!)

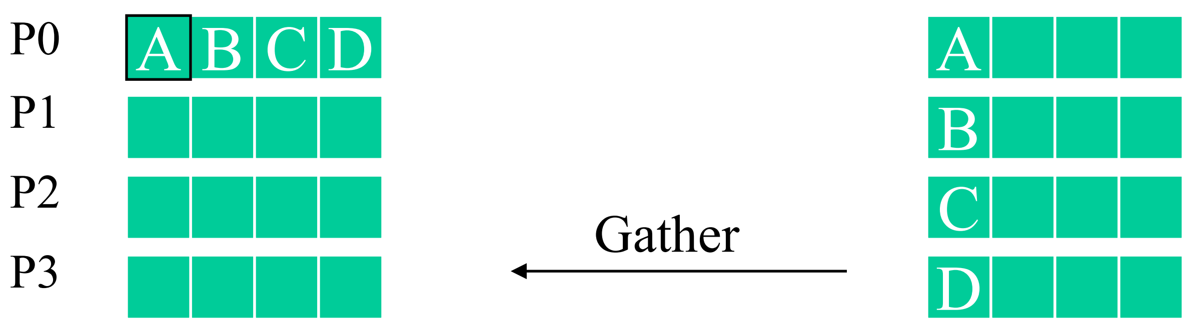


Gather

- 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, , see MPI manual (might help for project!)

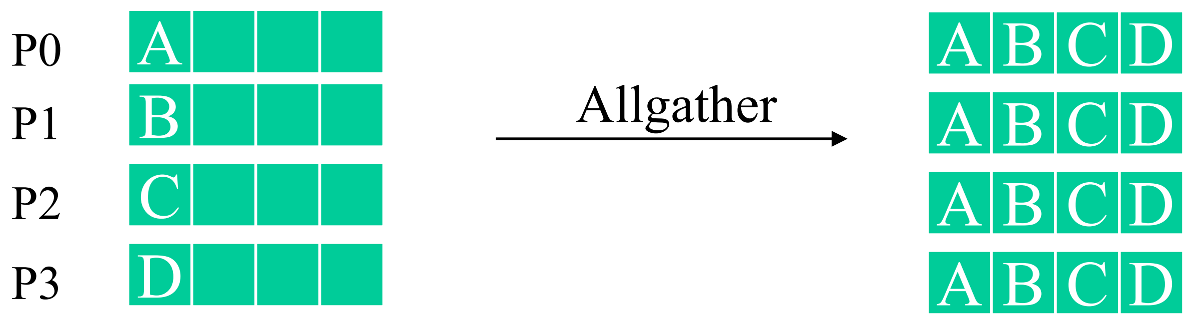


All-Gather

- 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 MPI documentation for further details)