CSCI 5451: Introduction to Parallel Computing

Lecture 12: MPI Collective Communications

computer science
& engineering

UNIVERSITY OF MINNESOTA
Driven to Discover®

Announcements (10/13)

- HW1
  - Updated due date to Oct 15
  - Logging is rather verbose. Make sure to scroll to the bottom.
  - Only resubmissions are graded when the autograder is run
- HWs 2-5 are pushed back one week
- Group Formation due Oct 19 (Canvas)

Lecture Overview

- ❑ Non-Blocking MPI Communications
- ❑ Review of Collective Communication Patterns
- ❑ MPI Collective Communications
- ❑ Groups + Communicators
- ❑ Questions

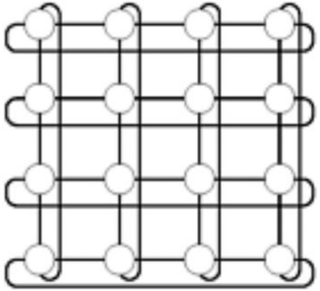
Lecture Overview

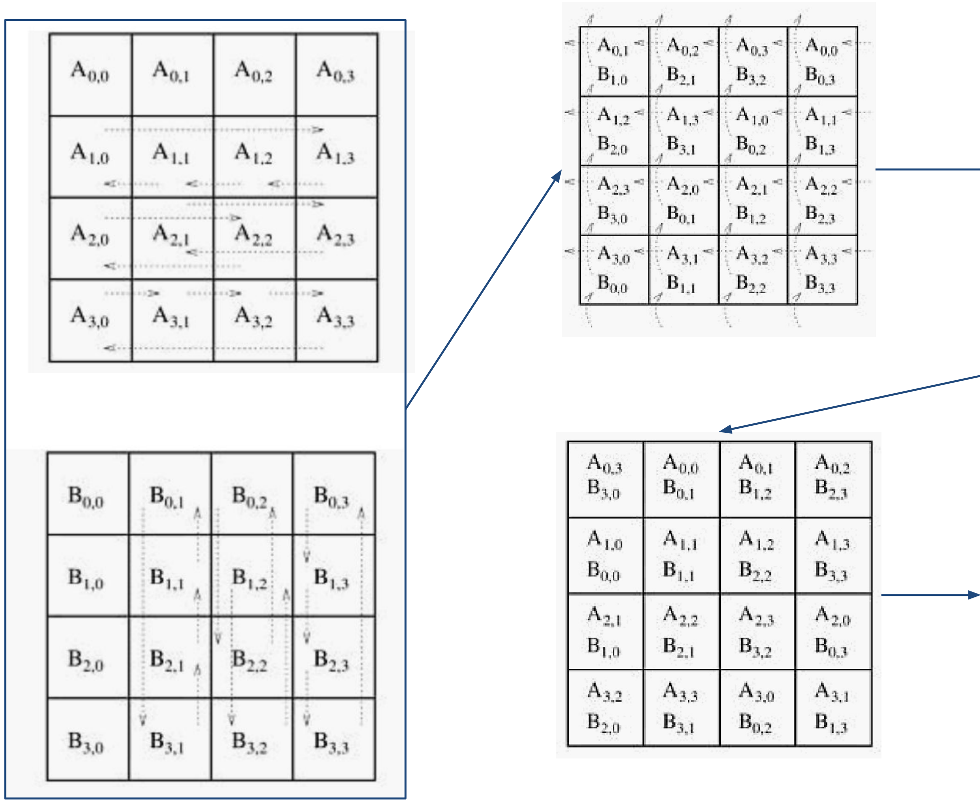
- **Non-Blocking MPI Communications**
- Review of Collective Communication Patterns
- MPI Collective Communications
- Groups + Communicators
- Questions

Cannon’s Algorithm

Recap

2-d mesh with wraparound





One more communication is performed to return to alignment before entering the loop

What is a downside of the current project to our implementation of Cannon's Algorithm?

What is a downside of the current project to our implementation of Cannon’s Algorithm?

The CPU will idle during communication steps.

What is a downside of the current project to our implementation of Cannon's Algorithm?

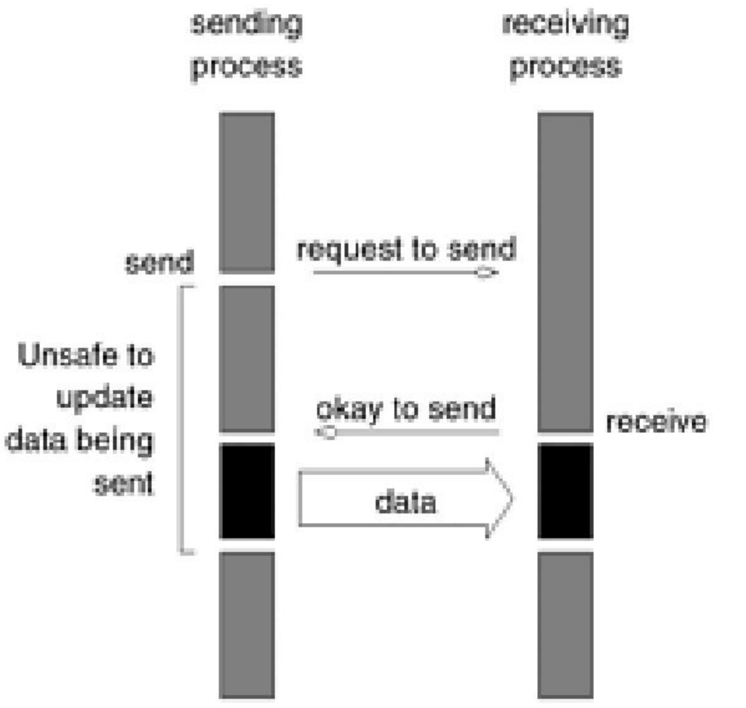
The CPU will idle during communication steps.

We can resolve this by overlapping communication + computation with non-blocking MPI calls.

Non-Blocking MPI Sends & Receives

Recap

- A program using non-blocking Sends & Receives will ***immediately*** continue after finishing the Send or Receive function
- We ***do not*** have any guarantees that our program will not overwrite the data being sent
- This is dependent on whether or not the data will be buffered, and how that buffering occurs



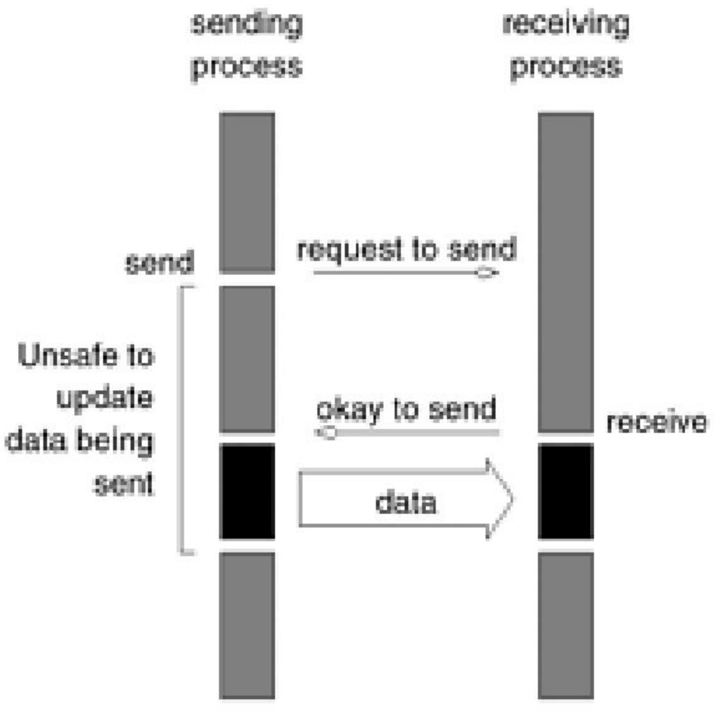
(a) Without hardware support

Non-Blocking MPI Sends & Receives

Recap

- A program using non-blocking Sends & Receives will ***immediately*** continue after finishing the Send or Receive function
- We ***do not*** have any guarantees that our program will not overwrite the data being sent
- This is dependent on whether or not the data will be buffered, and how that buffering occurs

Non-Buffered



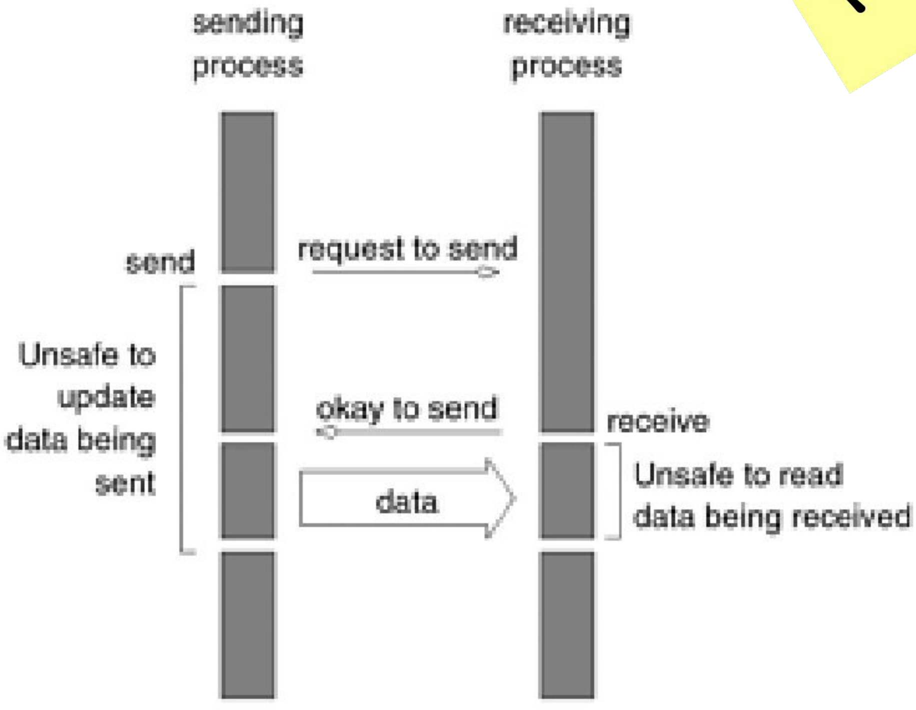
(a) Without hardware support

Non-Blocking MPI Sends & Receives

Recap

- A program using non-blocking Sends & Receives will ***immediately*** continue after finishing the Send or Receive function
- We ***do not*** have any guarantees that our program will not overwrite the data being sent
- This is dependent on whether or not the data will be buffered, and how that buffering occurs

Non-Buffered



(b) With hardware support

Non-Blocking MPI Sends & Receives

- ***MPI_Isend*** & ***MPI_Irecv*** enable non-blocking communication with MPI
- With hardware support, these operations enable us to perfectly overlap communication and computation
- Without hardware support, we remove any idling, but the CPU must still participate in the network communication
- MPI_Isend may be buffered, MPI_Irecv is not
- As such, we must be careful when structuring program execution to not overwrite the buffers

```
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 source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

Non-Blocking MPI Sends & Receives

- ***MPI_Isend*** & ***MPI_Irecv*** enable non-blocking communication with MPI
- With hardware support, these operations enable us to perfectly overlap communication and computation
- Without hardware support, we remove any idling, but the CPU must still participate in the network communication
- MPI_Isend may be buffered, MPI_Irecv is not
- As such, we must be careful when structuring program execution to not overwrite the buffers

```
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 source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

Same arguments as used in MPI_Send & MPI_Recv

Non-Blocking MPI Sends & Receives

- ***MPI_Isend*** & ***MPI_Irecv*** enable non-blocking communication with MPI
- With hardware support, these operations enable us to perfectly overlap communication and computation
- Without hardware support, we remove any idling, but the CPU must still participate in the network communication
- MPI_Isend may be buffered, MPI_Irecv is not
- As such, we must be careful when structuring program execution to not overwrite the buffers

```
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 source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

The ***request*** object allows us to monitor for when the corresponding send or receive has completed (we will explore this more in upcoming slides)

Non-Blocking MPI Sends & Receives

```
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Eliminates deadlocks

```
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
   MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
}
...
```

Non-Blocking MPI Sends & Receives

- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

Non-Blocking MPI Sends & Receives

- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

The ***request*** object - returned from the earlier ***MPI_ISend*** or ***MPI_Irecv*** calls

Non-Blocking MPI Sends & Receives

- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

A variable indicating whether or not the given communication operation has completed. This returns ‘1’ if it has, ‘0’ otherwise.

Non-Blocking MPI Sends & Receives

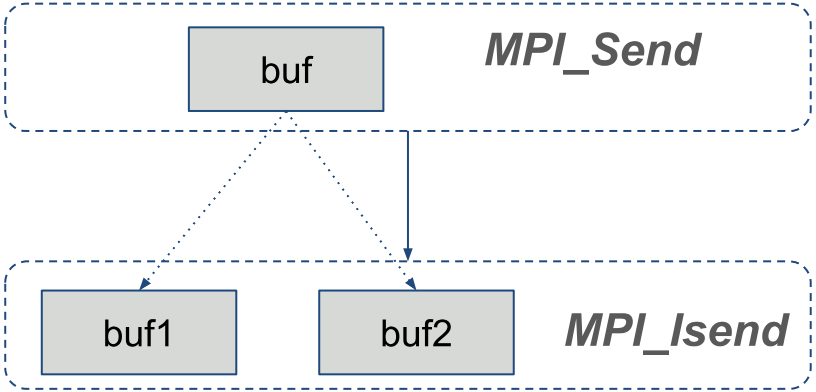
- ***MPI_Test*** & ***MPI_Wait*** are helpful utilities to determine the status of a non-blocking ***MPI_Isend*** or ***MPI_Irecv*** call
- ***MPI_Test*** is a non-blocking operation used to check whether the corresponding non-blocking communication has completed
- ***MPI_Wait*** is a *blocking* operation used to halt further execution until the corresponding non-blocking communication has completed

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

The ***status*** of the communication operation. This is the same object returned by ***MPI_Revc***.

Cannon's Algorithm in MPI

- We make the following changes to ensure that our MPI program works with non-blocking operations
  - Use ***MPI_Isend, MPI_Irecv, MPI_Wait***
  - Duplicate the local buffers ***a*** and ***b*** on each process
- In general, you will typically want to explicitly use duplicate buffers to ensure that one buffer is used for communication while another is used for computation

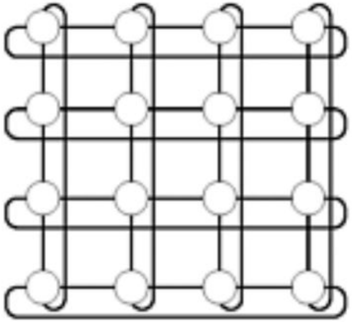


Non-Blocking Cannon's Algorithm in MPI

```
MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
                                  double *c, MPI_Comm comm)
{
  int i, j, nlocal;
  double *a_buffers[2], *b_buffers[2];
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;
  MPI_Request reqs[4];

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Non-Blocking Cannon's Algorithm in MPI

2-d mesh with wraparound

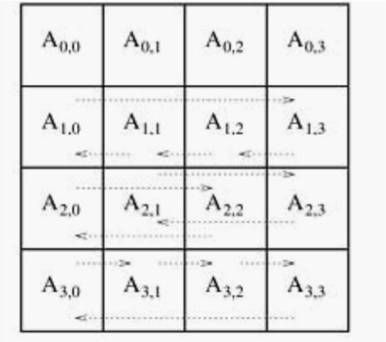


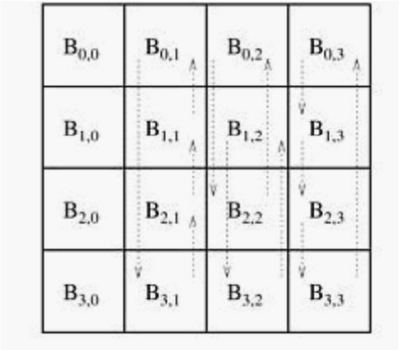
```
MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
                                  double *c, MPI_Comm comm)
{
  int i, j, nlocal;
  double *a_buffers[2], *b_buffers[2];
  int npes, dims[2], periods[2];
  int myrank, my2drank, mycoords[2];
  int uprank, downrank, leftrank, rightrank, coords[2];
  int shiftsource, shiftdest;
  MPI_Status status;
  MPI_Comm comm_2d;
  MPI_Request reqs[4];

  /* Get the communicator related information */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Set up the Cartesian topology */
  dims[0] = dims[1] = sqrt(npes);

  /* Set the periods for wraparound connections */
  periods[0] = periods[1] = 1;

  /* Create the Cartesian topology, with rank reordering */
  MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
```

Non-Blocking Cannon's Algorithm in MPI

```
/* Get the rank and coordinates with respect to the new topology */
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Compute ranks of the up and left shifts */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

/* Determine the dimension of the local matrix block */
nlocal = n/dims[0];

/* Setup the a_buffers and b_buffers arrays */
a_buffers[0] = a;
a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
b_buffers[0] = b;
b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));

/* Perform the initial matrix alignment. First for A and then for B */
MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

Non-Blocking Cannon's Algorithm in MPI



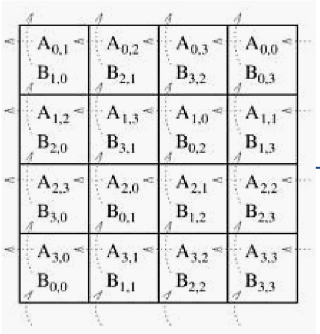


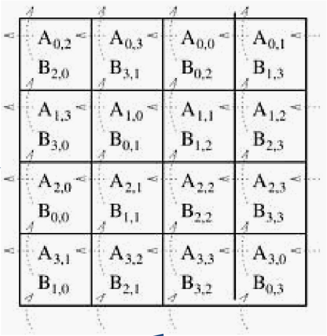

```
/* Get the rank and coordinates with respect to the new topology */
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Compute ranks of the up and left shifts */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

/* Determine the dimension of the local matrix block */
nlocal = n/dims[0];

/* Setup the a_buffers and b_buffers arrays */
a_buffers[0] = a;
 a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
b_buffers[0] = b;
 b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));

/* Perform the initial matrix alignment. First for A and then for B */
MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

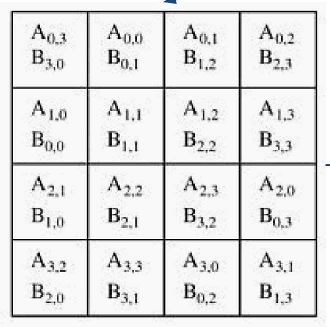
Non-Blocking Cannon's Algorithm in MPI

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
   MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
       leftrank, 1, comm_2d, &reqs[0]);
   MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
       uprank, 1, comm_2d, &reqs[1]);
   MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
       rightrank, 1, comm_2d, &reqs[2]);
   MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
       downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
   MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
 MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);

 MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
 MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
     shiftdest, 1, shiftsource, 1, comm_2d, &status);

 MPI_Comm_free(&comm_2d); /* Free up communicator */

 free(a_buffers[1]);
```

Non-Blocking Cannon's Algorithm in MPI



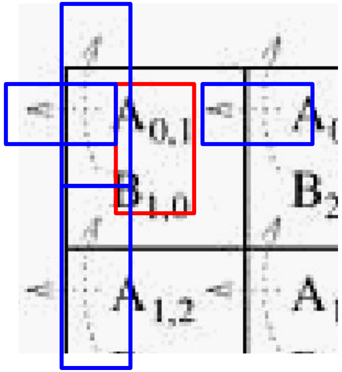




One more communication is performed to return to alignment before entering the loop

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Comm_free(&comm_2d); /* Free up communicator */

free(a_buffers[1]);
```

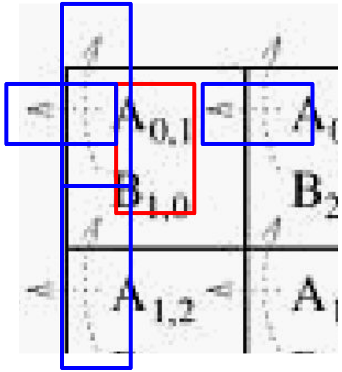
Non-Blocking Cannon's Algorithm in MPI



On each process, computations (red) + communications (blue) are done at the same time.

```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Comm_free(&comm_2d); /* Free up communicator */

free(a_buffers[1]);
```

Non-Blocking Cannon’s Algorithm in MPI



If there is no dedicated hardware, then the program will *poll* at regular intervals to see if it may communicate & the CPU will pause computation to carry out the necessary communication

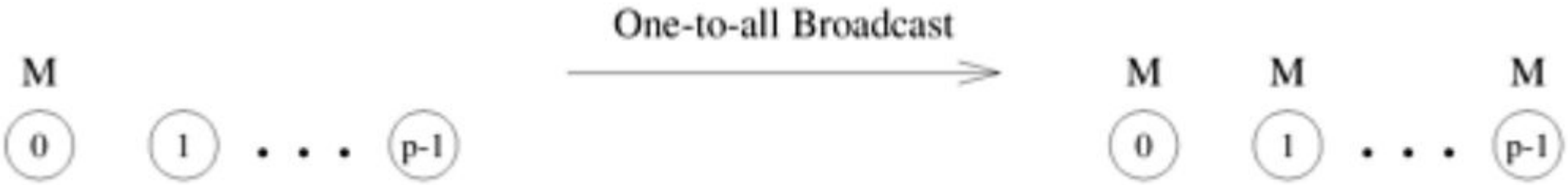
```
/* Get into the main computation loop */
for (i=0; i<dims[0]; i++) {
  MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      leftrank, 1, comm_2d, &reqs[0]);
  MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
      uprank, 1, comm_2d, &reqs[1]);
  MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      rightrank, 1, comm_2d, &reqs[2]);
  MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
      downrank, 1, comm_2d, &reqs[3]);

  /* c = c + a*b */
  MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);

  for (j=0; j<4; j++)
    MPI_Wait(&reqs[j], &status);
}

/* Restore the original distribution of a and b */
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
    shiftdest, 1, shiftsource, 1, comm_2d, &status);

MPI_Comm_free(&comm_2d); /* Free up communicator */

free(a_buffers[1]);
```

Lecture Overview

- Non-Blocking MPI Communications
- **Review of Collective Communication Patterns**
- MPI Collective Communications
- Groups + Communicators
- Questions

One-to All Broadcast

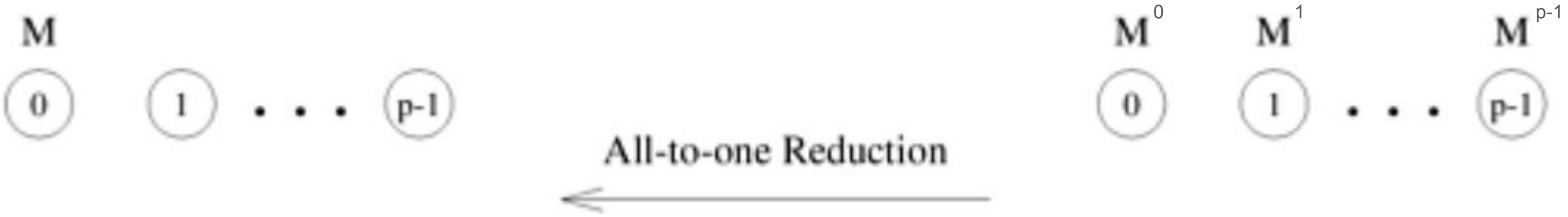
A message ***M*** exists on one processor which we want to send to all *p* other processors



All-to-One Reduction

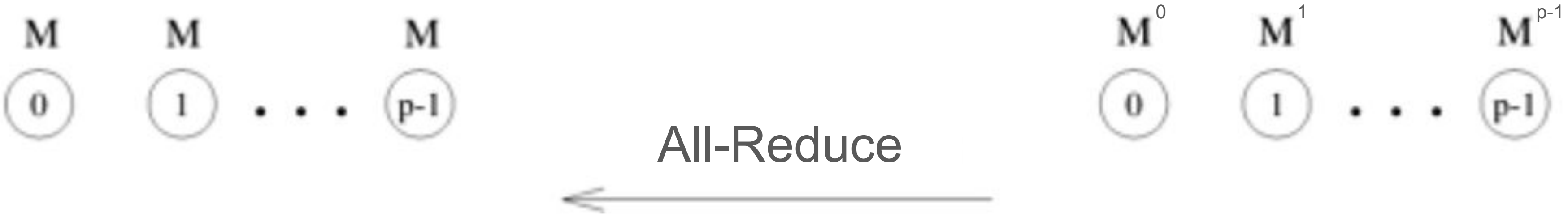
Recap

Messages M^i exist on each process i, and we want to combine these messages onto a single processor.



All-Reduce

Messages M^i exist on each process *i*, and we want to combine these messages in the same way on each processor.



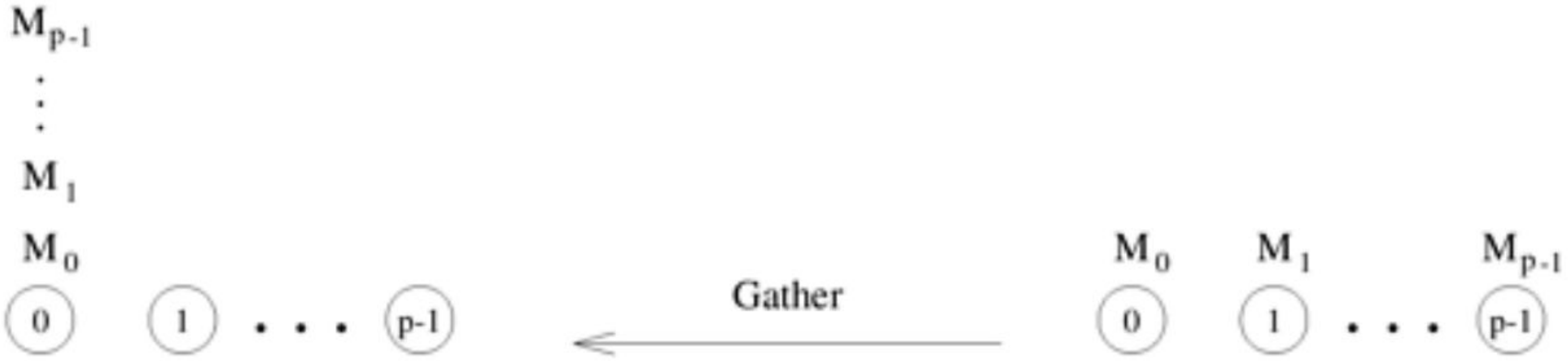
Prefix Sum

An array where each element stores the cumulative sum of all previous elements (useful for histograms, radix sort, etc.)

Input array [2, 4, 6, 8] → Prefix sums [2, 6, 12, 20]

Gather

Each processor has a message *m*, all of which must be collected onto a single processor



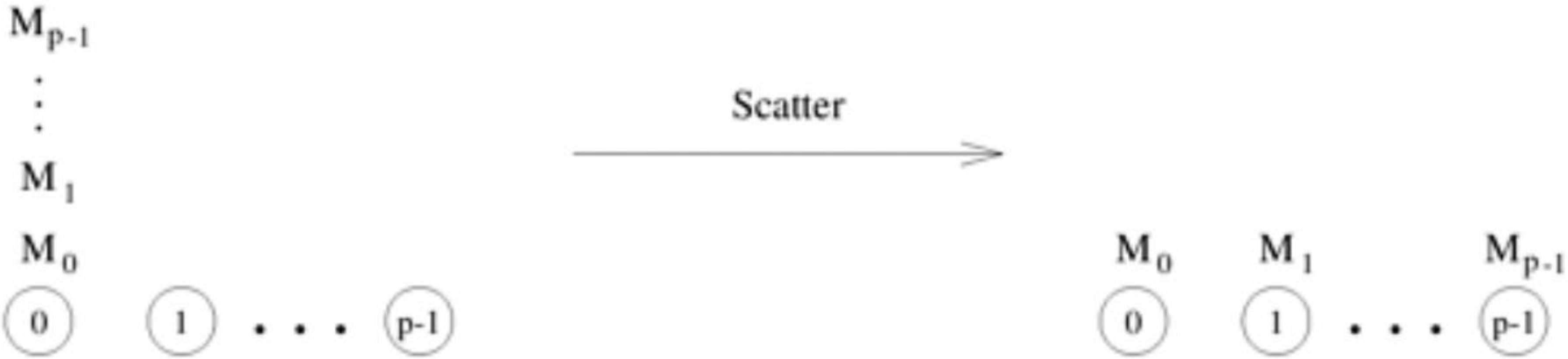
All-Gather

Each processor has a message *m*, all of which must be collected onto ***each*** processor

Scatter

Recap

A single processor has a separate message *m* for each other processor in the network



All-to-All Personalized Communication (AAPC)

All processors contain a unique message for each other processor.

Lecture Overview

- ❑ Non-Blocking MPI Communications
- ❑ Review of Collective Communication Patterns
- ❑ **MPI Collective Communications**
- ❑ Groups + Communicators
- ❑ Questions

MPI_Barrier

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

Synchronizes all processes within a communicator; no process exits until all have entered.

- comm: The communicator containing all participating processes.

MPI_Barrier

```
// Assume 4 processes
printf("Rank %d reached barrier\n", rank);
MPI_Barrier(MPI_COMM_WORLD);
printf("Rank %d passed barrier\n", rank);
```

int MPI_Barrier(MPI_Comm comm);

Synchronizes all processes within a communicator; no process exits until all have entered.

MPI_Barrier

```
// Assume 4 processes
printf("Rank %d reached barrier\n", rank);
MPI_Barrier(MPI_COMM_WORLD);
printf("Rank %d passed barrier\n", rank);
```

```
Rank 0 reached barrier
Rank 1 reached barrier
Rank 2 reached barrier
Rank 3 reached barrier
-- synchronization --
Rank 0 passed barrier
Rank 1 passed barrier
Rank 2 passed barrier
Rank 3 passed barrier
```

int MPI_Barrier(MPI_Comm comm);

Synchronizes all processes within a communicator; no process exits until all have entered.

MPI_Bcast

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

Broadcasts data from the root process to all other processes in the communicator.

- buffer: Starting address of data (send buffer for root, receive buffer for others).
- count: Number of elements in the buffer.
- datatype: Type of each buffer element (e.g., MPI_INT)
- root: Rank of the sending process.
- comm: Communicator handle

MPI_Bcast

```
// Assume 4 processes, rank 0 initializes the value
if (rank == 0) value = 42;
MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("Rank %d received value %d\n", rank, value);
```

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

Broadcasts data from the root process to all other processes in the communicator.

MPI_Bcast

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

Broadcasts data from the root process to all other processes in the communicator.

```
// Assume 4 processes, rank 0 initializes the value
if (rank == 0) value = 42;
MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("Rank %d received value %d\n", rank, value);
```

```
Rank 0 received value 42
Rank 1 received value 42
Rank 2 received value 42
Rank 3 received value 42
```

MPI_Reduce

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

Applies a reduction operation (e.g., sum, max) across all processes and delivers the result to the root.

- sendbuf: Starting address of send buffer.
- recvbuf: Starting address of receive buffer (used only by root)
- count: Number of elements in each buffer
- datatype: Data type of elements.
- op: Operation (e.g., MPI_SUM, MPI_MAX).
- root: Rank of root process.
  comm: Communicator handle.

MPI_Reduce

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

Applies a reduction operation (e.g., sum, max) across all processes and delivers the result to the root.

```
int local = rank + 1;  // {1, 2, 3, 4}
int result;
MPI_Reduce(&local, &result, 1, MPI_INT, MPI_SUM, 0,
                      MPI_COMM_WORLD);
if (rank == 0)
    printf("Sum = %d\n", result);
```

MPI_Reduce

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

Applies a reduction operation (e.g., sum, max) across all processes and delivers the result to the root.

```
int local = rank + 1;  // {1, 2, 3, 4}
int result;
MPI_Reduce(&local, &result, 1, MPI_INT, MPI_SUM, 0,
                    MPI_COMM_WORLD);
if (rank == 0)
    printf("Sum = %d\n", result);
```

Rank 0 prints: Sum = 10

MPI Operation	Meaning	Supported Datatypes
MPI_MAX	Returns the **maximum** value across all processes	C integers and floating-point types (MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc.)
MPI_MIN	Returns the **minimum** value across all processes	C integers and floating-point types
MPI_SUM	Computes the **sum** of all elements	C integers and floating-point types
MPI_PROD	Computes the **product** of all elements	C integers and floating-point types
MPI_LAND	Performs **logical AND** (element-wise)	C integer and logical types (MPI_INT, MPI_C_BOOL, etc.)
MPI_BAND	Performs **bitwise AND**	C integer types
MPI_LOR	Performs **logical OR** (element-wise)	C integer and logical types
MPI_BOR	Performs **bitwise OR**	C integer types
MPI_LXOR	Performs **logical XOR** (exclusive OR, element-wise)	C integer and logical types
MPI_BXOR	Performs **bitwise XOR**	C integer types
MPI_MAXLOC	Returns the **(value, index)** pair where value is maximum; index ties broken by smallest index	Pairs of (value, index) using special datatypes (MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, etc.)
MPI_MINLOC	Returns the **(value, index)** pair where value is minimum; index ties broken by smallest index	Same as MPI_MAXLOC (paired value-index types)

MPI_Allreduce

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

Performs a reduction operation across all processes and distributes the result to all.

- sendbuf: Address of input data.
- recvbuf: Address to store the result for each process.
- count: Number of elements in each buffer.
- datatype: Data type of elements.
- op: Reduction operation (sum, max, etc.).
- comm: Communicator handle.

MPI_Allreduce

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

Performs a reduction operation across all processes and distributes the result to all.

```
int local = rank + 1;
int result;
MPI_Allreduce(&local, &result, 1, MPI_INT, MPI_SUM,
                      MPI_COMM_WORLD);
printf("Rank %d: global sum = %d\n", rank, result);
```

MPI_Allreduce

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

Performs a reduction operation across all processes and distributes the result to all.

```
int local = rank + 1;
int result;
MPI_Allreduce(&local, &result, 1, MPI_INT, MPI_SUM,
                        MPI_COMM_WORLD);
printf("Rank %d: global sum = %d\n", rank, result);
```

```
Rank 0: global sum = 10
Rank 1: global sum = 10
Rank 2: global sum = 10
Rank 3: global sum = 10
```

MPI_Scan

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

Performs a prefix reduction (partial sum) such that process *i* receives the reduction of ranks ≤ *i*.

- sendbuf: Input buffer.
- recvbuf: Output buffer (prefix result).
- count: Number of elements in each buffer.
- datatype: Data type of elements.
- op: Reduction operation (sum, max, etc.).
- comm: Communicator handle.

MPI_Scan

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

Performs a prefix reduction (partial sum) such that process *i* receives the reduction of ranks ≤ *i*.

```
int local = rank + 1;  // {1,2,3,4}
int prefix;
MPI_Scan(&local, &prefix, 1, MPI_INT, MPI_SUM,
                    MPI_COMM_WORLD);
printf("Rank %d: prefix sum = %d\n", rank, prefix);
```

MPI_Scan

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

Performs a prefix reduction (partial sum) such that process *i* receives the reduction of ranks ≤ *i*.

```
int local = rank + 1;  // {1,2,3,4}
int prefix;
MPI_Scan(&local, &prefix, 1, MPI_INT, MPI_SUM,
                   MPI_COMM_WORLD);
printf("Rank %d: prefix sum = %d\n", rank, prefix);
```

```
Rank 0: prefix sum = 1
Rank 1: prefix sum = 3
Rank 2: prefix sum = 6
Rank 3: prefix sum = 10
```

MPI_Gather

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

Collects equal-sized data segments from all processes and concatenates them at the root.

- sendbuf: Buffer with data to send.
- sendcount: Number of elements sent from each process.
- sendtype: Type of each send element.
- recvbuf: Buffer to store gathered data (valid on root).
- recvcount: Number of elements received from each process.
- recvtype: Type of each received element.
- root: Rank of root process.
- comm: Communicator handle

MPI_Gather

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

Collects equal-sized data segments from all processes and concatenates them at the root.

```
int send = rank + 10;      // {10,11,12,13}
int recv[4];
MPI_Gather(&send, 1, MPI_INT, recv, 1, MPI_INT, 0,
                   MPI_COMM_WORLD);
if (rank == 0)
    printf("Gathered: [%d %d %d %d]\n", recv[0], recv[1],
              recv[2], recv[3]);
```

MPI_Gather

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

Collects equal-sized data segments from all processes and concatenates them at the root.

```
int send = rank + 10;      // {10,11,12,13}
int recv[4];
MPI_Gather(&send, 1, MPI_INT, recv, 1, MPI_INT, 0,
                    MPI_COMM_WORLD);
if (rank == 0)
    printf("Gathered: [%d %d %d %d]\n", recv[0], recv[1],
            recv[2], recv[3]);
```

```
Rank 0 prints: Gathered: [10 11 12 13]
```

MPI_Gatherv

```
int MPI_Gatherv(const void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int *recvcounts,
        const int *displs,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm);
```

Similar to MPI_Gather, but allows variable amounts of data to be received from each process.

- sendbuf: Buffer with data to send.
- sendcount: Number of elements sent by this process.
- sendtype: Type of send data.
- recvbuf: Buffer to store gathered data (root only).
- recvcounts: Array specifying number of elements received from each rank.
- displs: Array specifying offsets (displacements) in recvbuf.
- recvtype: Type of received data.
- root: Root process rank.
- comm: Communicator handle.

MPI_Gatherv

```
int MPI_Gatherv(const void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int *recvcounts,
        const int *displs,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm);
```

Similar to MPI_Gather, but allows variable amounts of data to be received from each process.

```
int sendcount = rank + 1;   // {1,2,3,4}
int sendbuf[4] = {rank, rank, rank, rank};
int recvbuf[10];
int counts[4] = {1,2,3,4};
int displs[4] = {0,1,3,6};

MPI_Gatherv(sendbuf, sendcount, MPI_INT,
            recvbuf, counts, displs, MPI_INT,
            0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Rank 0 gathered: ");
    for (int i = 0; i < displs[3] + counts[3]; i++)
        printf("%d ", recvbuf[i]);
    printf("\n");
}
```

MPI_Gatherv

```
int MPI_Gatherv(const void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int *recvcounts,
        const int *displs,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm);
```

Similar to MPI_Gather, but allows variable amounts of data to be received from each process.

```
int sendcount = rank + 1;   // {1,2,3,4}
int sendbuf[4] = {rank, rank, rank, rank};
int recvbuf[10];
int counts[4] = {1,2,3,4};
int displs[4] = {0,1,3,6};

MPI_Gatherv(sendbuf, sendcount, MPI_INT,
            recvbuf, counts, displs, MPI_INT,
            0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Rank 0 gathered: ");
    for (int i = 0; i < displs[3] + counts[3]; i++)
        printf("%d ", recvbuf[i]);
    printf("\n");
}
```

```
Rank 0 gathered: 0 1 1 2 2 2 3 3 3 3
```

MPI_Allgather

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

Collects equal-sized data from all processes and distributes the full concatenated result to everyone.

- sendbuf: Local data to send.
- sendcount: Number of elements sent per process.
- sendtype: Data type of send buffer.
- recvbuf: Buffer to store gathered results on all processes.
- recvcount: Number of elements received from each process.
- recvtype: Type of received data.
- comm: Communicator handle.

MPI_Allgather

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

Collects equal-sized data from all processes and distributes the full concatenated result to everyone.

```
int send = rank + 10;
int recv[4];
MPI_Allgather(&send, 1, MPI_INT, recv, 1, MPI_INT,
                    MPI_COMM_WORLD);
printf("Rank %d: received [%d %d %d %d]\n", rank,
recv[0], recv[1], recv[2], recv[3]);
```

MPI_Allgather

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

Collects equal-sized data from all processes and distributes the full concatenated result to everyone.

```
int send = rank + 10;
int recv[4];
MPI_Allgather(&send, 1, MPI_INT, recv, 1, MPI_INT,
                     MPI_COMM_WORLD);
printf("Rank %d: received [%d %d %d %d]\n", rank,
recv[0], recv[1], recv[2], recv[3]);
```

```
Rank 0: received [10 11 12 13]
Rank 1: received [10 11 12 13]
Rank 2: received [10 11 12 13]
Rank 3: received [10 11 12 13]
```

MPI_Allgatherv

```
int MPI_Allgatherv(const void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int *recvcounts,
        const int *displs,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Gathers variable-sized data from all processes and distributes the complete result to everyone.

- `sendbuf`: Local data to send.
- `sendcount`: Number of elements sent by this process.
- `sendtype`: Type of send data.
- `recvbuf`: Buffer to store gathered results.
- `recvcounts`: Array specifying number of elements received from each rank.
- `displs`: Array specifying displacement offsets in `recvbuf`.
- `recvtype`: Type of received data.
- `comm`: Communicator handle.

MPI_Allgatherv

```
int MPI_Allgatherv(const void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int *recvcounts,
        const int *displs,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Gathers variable-sized data from all processes and distributes the complete result to everyone.

```
int sendcount = rank + 1;   // {1,2,3,4}
int sendbuf[4] = {rank, rank, rank, rank};
int recvbuf[10];
int counts[4] = {1,2,3,4};
int displs[4] = {0,1,3,6};

MPI_Allgatherv(sendbuf, sendcount, MPI_INT,
               recvbuf, counts, displs, MPI_INT,
               MPI_COMM_WORLD);

printf("Rank %d received: ", rank);
for (int i = 0; i < displs[3] + counts[3]; i++)
    printf("%d ", recvbuf[i]);
printf("\n");
```

MPI_Allgatherv

```
int MPI_Allgatherv(const void *sendbuf,
        int sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int *recvcounts,
        const int *displs,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Gathers variable-sized data from all processes and distributes the complete result to everyone.

```
int sendcount = rank + 1;   // {1,2,3,4}
int sendbuf[4] = {rank, rank, rank, rank};
int recvbuf[10];
int counts[4] = {1,2,3,4};
int displs[4] = {0,1,3,6};

MPI_Allgatherv(sendbuf, sendcount, MPI_INT,
               recvbuf, counts, displs, MPI_INT,
               MPI_COMM_WORLD);

printf("Rank %d received: ", rank);
for (int i = 0; i < displs[3] + counts[3]; i++)
    printf("%d ", recvbuf[i]);
printf("\n");
```

```
Rank 0 received: 0 1 1 2 2 2 3 3 3 3
Rank 1 received: 0 1 1 2 2 2 3 3 3 3
Rank 2 received: 0 1 1 2 2 2 3 3 3 3
Rank 3 received: 0 1 1 2 2 2 3 3 3 3
```

MPI_Scatter

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

Distributes equal-sized blocks of data from the root to all processes.

- sendbuf: Root’s buffer containing the data to scatter.
- sendcount: Number of elements sent to each process.
- sendtype: Type of send data.
  recvbuf: Buffer to store received data.
- recvcount: Number of elements received per process.
- recvtype: Type of received data.
- root: Rank of root process.
- comm: Communicator handle.

MPI_Scatter

```
int send[4] = {10, 20, 30, 40};
int recv;
MPI_Scatter(send, 1, MPI_INT, &recv, 1, MPI_INT, 0,
MPI_COMM_WORLD);
printf("Rank %d received %d\n", rank, recv);
```

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

Distributes equal-sized blocks of data from the root to all processes.

MPI_Scatter

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

Distributes equal-sized blocks of data from the root to all processes.

```
int send[4] = {10, 20, 30, 40};
int recv;
MPI_Scatter(send, 1, MPI_INT, &recv, 1, MPI_INT, 0,
MPI_COMM_WORLD);
printf("Rank %d received %d\n", rank, recv);
```

```
Rank 0 received 10
Rank 1 received 20
Rank 2 received 30
Rank 3 received 40
```

MPI_Scatterv

```
int MPI_Scatterv(const void *sendbuf,
        const int *sendcounts, const int *displs,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm);
```

Distributes variable-sized blocks of data from the root to all processes.

- sendbuf: Root’s buffer containing data.
- sendcounts: Array specifying number of elements to send to each process.
- displs: Array specifying starting offsets of each block in sendbuf.
- sendtype: Type of send data.
- recvbuf: Buffer to receive data.
- recvcount: Number of elements received by this process.
- recvtype: Type of receive data.
- root: Rank of root process.
- comm: Communicator handle.

MPI_Scatterv

```c
int sendbuf[10] = {0, 1,1, 2,2,2, 3,3,3,3};
int counts[4] = {1,2,3,4};
int displs[4] = {0,1,3,6};
int recvbuf[4];
MPI_Scatterv(sendbuf, counts, displs, MPI_INT,
             recvbuf, counts[rank], MPI_INT,
             0, MPI_COMM_WORLD);
printf("Rank %d received:", rank);
for (int i = 0; i < counts[rank]; i++)
    printf(" %d", recvbuf[i]);
printf("\n");
```

```c
int MPI_Scatterv(const void *sendbuf,
        const int *sendcounts, const int *displs,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm);
```

Distributes variable-sized blocks of data from the root to all processes.

MPI_Scatterv

```
int MPI_Scatterv(const void *sendbuf,
        const int *sendcounts, const int *displs,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm);
```

Distributes variable-sized blocks of data from the root to all processes.

```
int sendbuf[10] = {0, 1,1, 2,2,2, 3,3,3,3};
int counts[4] = {1,2,3,4};
int displs[4] = {0,1,3,6};
int recvbuf[4];
MPI_Scatterv(sendbuf, counts, displs, MPI_INT,
             recvbuf, counts[rank], MPI_INT,
             0, MPI_COMM_WORLD);
printf("Rank %d received:", rank);
for (int i = 0; i < counts[rank]; i++)
   printf(" %d", recvbuf[i]);
printf("\n");
```

```
Rank 0 received: 0
Rank 1 received: 1 1
Rank 2 received: 2 2 2
Rank 3 received: 3 3 3 3
```

MPI_Alltoall

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

Each process sends equal-sized data to all others and receives equal-sized data from all others.

- sendbuf: Starting address of send buffer.
  sendcount: Number of elements sent to each process.
- sendtype: Type of send data.
  recvbuf: Starting address of receive buffer.
- recvcount: Number of elements received from each process.
  recvtype: Type of receive data.
- comm: Communicator handle.

MPI_Alltoall

```
int send[4] = {rank*10+0, rank*10+1, rank*10+2, rank*10+3};
int recv[4];
MPI_Alltoall(send, 1, MPI_INT, recv, 1, MPI_INT, MPI_COMM_WORLD);
printf("Rank %d received [%d %d %d %d]\n", rank,
       recv[0], recv[1], recv[2], recv[3]);
```

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

Each process sends equal-sized data to all others and receives equal-sized data from all others.

MPI_Alltoall

```
int send[4] = {rank*10+0, rank*10+1, rank*10+2, rank*10+3};
int recv[4];
MPI_Alltoall(send, 1, MPI_INT, recv, 1, MPI_INT, MPI_COMM_WORLD);
printf("Rank %d received [%d %d %d %d]\n", rank,
       recv[0], recv[1], recv[2], recv[3]);
```

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

```
Rank 0 received [0 10 20 30]
Rank 1 received [1 11 21 31]
Rank 2 received [2 12 22 32]
Rank 3 received [3 13 23 33]
```

Each process sends equal-sized data to all others and receives equal-sized data from all others.

MPI_Alltoallv

```
int MPI_Alltoallv(const void *sendbuf,
        const int *sendcounts, const int *sdispls,
        MPI_Datatype sendtype, void *recvbuf,
        const int *recvcounts, const int *rdispls,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Each process sends variable-sized data to all others and receives variable-sized data from all others.

- sendbuf: Starting address of send buffer.
- sendcounts: Array specifying number of elements sent to each rank.
- sdispls: Array of offsets in sendbuf.
- sendtype: Type of send data.
- recvbuf: Starting address of receive buffer.
- recvcounts: Array specifying number of elements received from each rank.
- rdispls: Array of offsets in recvbuf.
- recvtype: Type of receive data.
- comm: Communicator handle.

MPI_Alltoallv

```
int MPI_Alltoallv(const void *sendbuf,
        const int *sendcounts, const int *sdispls,
        MPI_Datatype sendtype, void *recvbuf,
        const int *recvcounts, const int *rdispls,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Each process sends variable-sized data to all others and receives variable-sized data from all others.

```
int sendbuf[16];
for (int i=0; i<16; i++) sendbuf[i] = rank*10 + i;
int sendcounts[4] = {1,2,3,4};
int sdispls[4]   = {0,1,3,6};

int recvcounts[4] = {1,2,3,4};
int rdispls[4]    = {0,1,3,6};
int recvbuf[10];

MPI_Alltoallv(sendbuf, sendcounts, sdispls, MPI_INT, recvbuf,
              recvcounts, rdispls, MPI_INT, MPI_COMM_WORLD);

printf("Rank %d received:", rank);
for (int i=0; i<recvcounts[rank]; i++)
    printf(" %d", recvbuf[i]);
printf("\n");
```

MPI_Alltoallv

```
int MPI_Alltoallv(const void *sendbuf,
        const int *sendcounts, const int *sdispls,
        MPI_Datatype sendtype, void *recvbuf,
        const int *recvcounts, const int *rdispls,
        MPI_Datatype recvtype,
        MPI_Comm comm);
```

Each process sends variable-sized data to all others and receives variable-sized data from all others.

```
int sendbuf[16];
for (int i=0; i<16; i++) sendbuf[i] = rank*10 + i;
int sendcounts[4] = {1,2,3,4};
int sdispls[4]    = {0,1,3,6};

int recvcounts[4] = {1,2,3,4};
int rdispls[4]     = {0,1,3,6};
int recvbuf[10];

MPI_Alltoallv(sendbuf, sendcounts, sdispls, MPI_INT, recvbuf,
                      recvcounts, rdispls, MPI_INT, MPI_COMM_WORLD);

printf("Rank %d received:", rank);
for (int i=0; i<recvcounts[rank]; i++)
    printf(" %d", recvbuf[i]);
printf("\n");
```

```
Rank 0 received: 0
Rank 1 received: 10 11
Rank 2 received: 20 21 22
Rank 3 received: 30 31 32 33
```

LECTURE ENDED HERE. THE REMAINDER OF THE LECTURE WILL BE COVERED ON 10/15.

Lecture Overview

- Non-Blocking MPI Communications
- Review of Collective Communication Patterns
- MPI Collective Communications
- **Groups + Communicators**
- Questions

MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm,
        int color, int key,
        MPI_Comm *newcomm);
```

Creates new communicators by splitting an existing communicator into subgroups based on color.

Processes with the same color are grouped together; within each group, ranks are ordered by key

MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm,
        int color, int key,
        MPI_Comm *newcomm);
```

Creates new communicators by splitting an existing communicator into subgroups based on color.

Processes with the same color are grouped together; within each group, ranks are ordered by key

- comm: Existing communicator to be split (e.g., MPI_COMM_WORLD).
  color: Integer identifier for subgroup membership.
  - All processes with the same color form a new communicator.
  - A process can set color = MPI_UNDEFINED to be excluded entirely.
- key: Determines ordering of ranks within the new communicator.
  - Lower keys get lower ranks.
  - Equal keys are ordered according to ordering in previous communicator
- newcomm: Output handle to the newly created communicator.

MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm,
        int color, int key,
        MPI_Comm *newcomm);
```

```
int key = 1;
if (rank <= 2)      color = 0;
else if (rank <= 6) color = 1;
else                color = 2;

MPI_Comm newcomm;
MPI_Comm_split(MPI_COMM_WORLD,
        color, key, &newcomm);
```

MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm,
        int color, int key,
        MPI_Comm *newcomm);
```

```
int key = 1;
if (rank <= 2)      color = 0;
else if (rank <= 6) color = 1;
else                color = 2;

MPI_Comm newcomm;
MPI_Comm_split(MPI_COMM_WORLD,
        color, key, &newcomm);
```

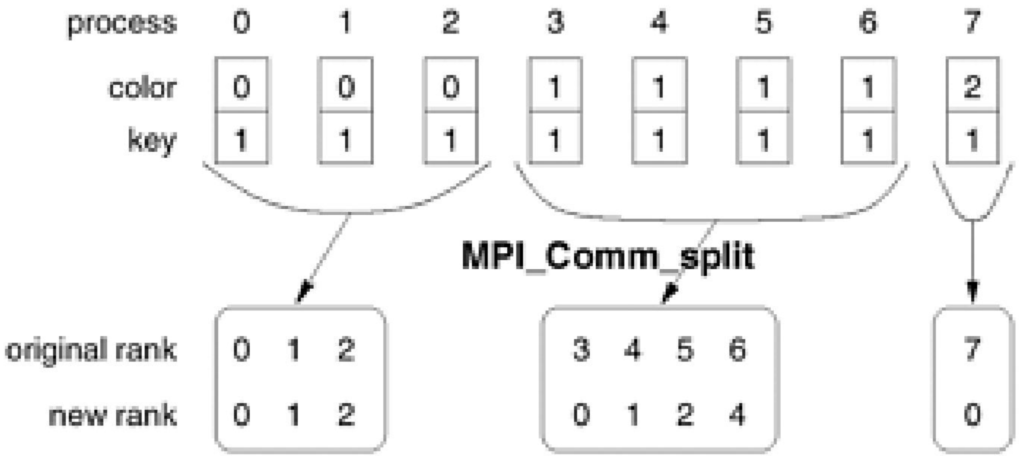
process	0	1	2	3	4	5	6	7
color	0	0	0	1	1	1	1	2
key	1	1	1	1	1	1	1	1

Which processes will be grouped together?

MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm,
        int color, int key,
        MPI_Comm *newcomm);
```

```
int key = 1;
if (rank <= 2)      color = 0;
else if (rank <= 6) color = 1;
else                color = 2;

MPI_Comm newcomm;
MPI_Comm_split(MPI_COMM_WORLD,
        color, key, &newcomm);
```



MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

Creates a lower-dimensional subcommunicator from an existing Cartesian topology communicator by selecting which dimensions to keep.

MPI_Cart_sub

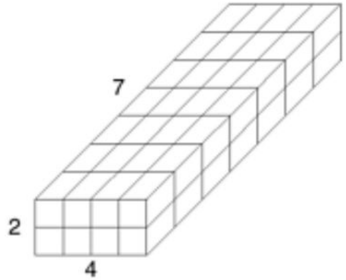
```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

Creates a lower-dimensional subcommunicator from an existing Cartesian topology communicator by selecting which dimensions to keep.

- comm: Input communicator with Cartesian topology (created via MPI_Cart_create).
- remain_dims: Logical array of length equal to the number of dimensions.
  - remain_dims[i] = 1 → keep this dimension.
  - remain_dims[i] = 0 → collapse (drop) this dimension.
- newcomm: Output communicator corresponding to the subgrid of the original topology.

MPI_Cart_sub

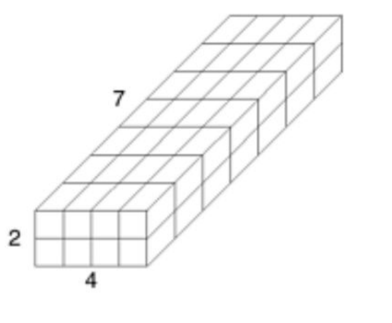
```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```




```
int dims[3]    = {2, 4, 7};        // grid dimensions
int periods[3] = {0, 0, 0};        // non-periodic in all dimensions
int reorder    = 0;                // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {1, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```

MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

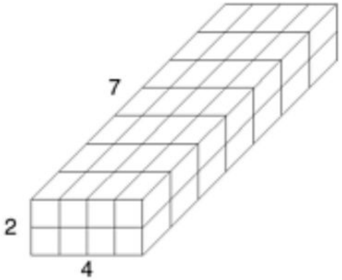


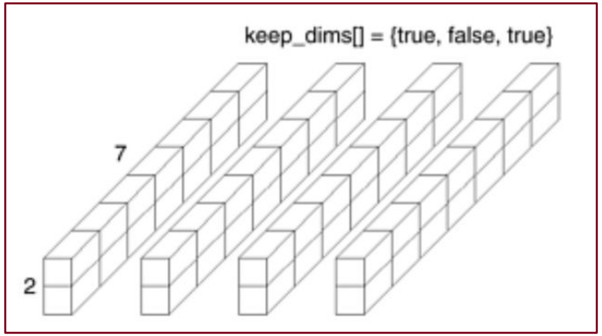

```
int dims[3]    = {2, 4, 7};        // grid dimensions
int periods[3] = {0, 0, 0};        // non-periodic in all dimensions
int reorder    = 0;                // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {1, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```

MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

```
int dims[3]    = {2, 4, 7};       // grid dimensions
int periods[3] = {0, 0, 0};       // non-periodic in all dimensions
int reorder    = 0;               // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {1, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```

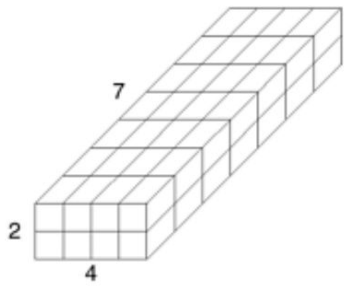




MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

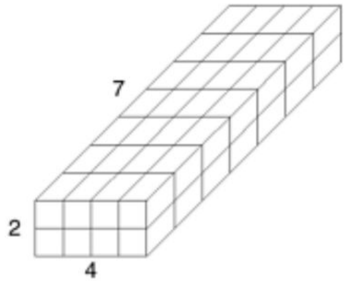
```
int dims[3]    = {2, 4, 7};       // grid dimensions
int periods[3] = {0, 0, 0};       // non-periodic in all dimensions
int reorder    = 0;               // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {0, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```

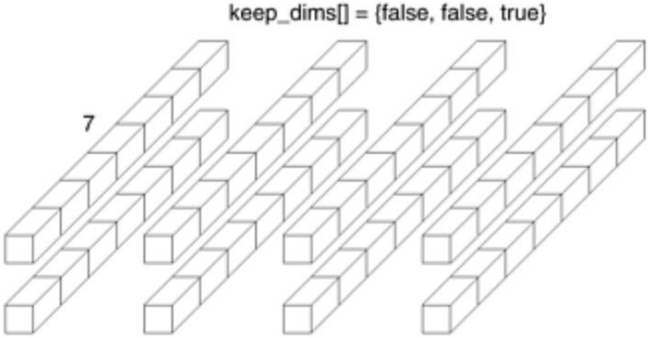


MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm,
        const int *remain_dims,
        MPI_Comm *newcomm);
```

```
int dims[3]    = {2, 4, 7};      // grid dimensions
int periods[3] = {0, 0, 0};      // non-periodic in all dimensions
int reorder    = 0;              // keep original ranks

MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods,
        reorder, &cart_comm);

int keep_dims[3] = {0, 0, 1};
MPI_Comm sub_comm;
MPI_Cart_sub(cart_comm, keep_dims, &sub_comm);
```





Lecture Overview

- Non-Blocking MPI Communications
- Review of Collective Communication Patterns
- MPI Collective Communications
- Groups + Communicators
- **Questions**