Point to Point (P2P) Communication

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Consider: Sum of the first N Integers

```
// serial solution
int main() {
  int N = 100000000, sum = 0;
  for(int i = 1; i<=N ; i++) {
    sum += i ;
  }
  cout << "The sum from 1 to << N << " is " << sum << endl;
}
```

- How to parallelize?

Develop your logic for Sum of the first N Integers in Parallel

- Each process (rank) do “partial sum” with
  - start_val = rank * (N / nprocs) + 1
  - end_val = (rank+1) * (N / nprocs)
  - E.g., if nprocs = 4, then
    - rank 0 sums from 1 to 25,000,000
    - rank 1 sums from 25,000,001 to 50,000,000
    - rank 2 sums from 50,000,001 to 75,000,000
    - rank 3 sums from 75,000,001 to 100,000,000
- Then, master (rank 0) receives each partial sum and add them.

Develop your logic for Sum of the first N Integers in Parallel

- Assume N is large and divisible by nproces without a remainder.
- Each process (rank) do “partial sum” with
  - start_val = rank * (N / nprocs) + 1
  - end_val = (rank+1) * (N / nprocs)
  - E.g., if nprocs = 4, then
    - rank 0 sums from 1 to 25,000,000
    - rank 1 sums from 25,000,001 to 50,000,000
    - rank 2 sums from 50,000,001 to 75,000,000
    - rank 3 sums from 75,000,001 to 100,000,000
- Then, master (rank 0) receives each partial sum and add them to print out total sum.
- Let me give you 15 mins to implement this program and let’s check together.

MPI_Send & MPI_Recv

- MPI_Send and MPI_Recv are blocking ! Operation must be completed before jump to next instruction.
- **Asynchronous** communication: possible delay between Send and Receive, sent data could be buffered. Even if Send is completed, it doesn't mean that message has already been received.
- Be cautious with **Deadlocks**: two processes waiting for a message that never come

Drawbacks of Blocking non-buffered send/recv

- Idling overheads
  - Be posted at roughly simultaneously
  - Asynchronous environment, hard to predict
- Deadlocks
  - Break the cycle waits
  - Inverse the sequence, but more buggy

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

MPI_Send & MPI_Recv : Deadlocks

- This code hangs.

```
if( rank == 0 ) {
  MPI_Recv( &value, 100, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, status );
  MPI_Send( &value, 100, MPI_FLOAT, 1, tag, MPI_COMM_WORLD );
}
else if (rank == 1) {
  MPI_Recv( &value, 100, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, status );
  MPI_Send( &value, 100, MPI_FLOAT, 0, tag, MPI_COMM_WORLD );
}
```

MPI_Send & MPI_Recv : Deadlocks

- How about this case?

```
1 #include <iostream>
2 #include <mpi.h>        // MPI header file
3
4 using namespace std;
5
6 int main(int argc, char **argv) {
7   int nprocs, rank;
8
9   // initialize for MPI
10  MPI_Init(&argc, &argv);
11  // get number of processes
12  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
13  // get the rank = this process's number
14  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15  // MPI status
16  MPI_Status status;
```

```
18  double d = 100.0;
19  int tag = 1;
20
21  // master send value to workers
22  if (rank == 0) {
23    // synchronous send: return when the destination has started
24    // to receive the message
25    MPI_Ssend(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
26    MPI_Recv(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
27  }
28  // workers get the message
29  else {
30    MPI_Ssend(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
31    MPI_Recv(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
32  }
33
34  // clean up for MPI
35  MPI_Finalize();   // C style
36
37  return 0;
38 }
```

MPI_Send & MPI_Recv : Deadlocks

- Both ranks wait for the other one to receive the message!

```cpp
#include <iostream>
#include <mpi.h>        // MPI header file

using namespace std;

int main(int argc, char **argv) {
  int nprocs, rank;

  // initialize for MPI
  MPI_Init(&argc, &argv);
  // get number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  // get the rank = this process's number
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // MPI status
  MPI_Status status;
```

```cpp
  double d = 100.0;
  int tag = 1;

  // master send value to workers
  if (rank == 0) {
    // synchronous send: return when the destination has started
    // to receive the message
    MPI_Ssend(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    MPI_Recv(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
  }
  // workers get the message
  else {
    MPI_Ssend(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    MPI_Recv(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
  }

  // clean up for MPI
  MPI_Finalize();   // C style

  return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/MPI$ mpic++ mpi_p2p_block_deadlock.cpp -o mpi_p2p_block_deadlock
ai@ubuntu-20-04:~/Lab/MPI$ mpirun -np 2 ./mpi_p2p_block_deadlock
```

MPI_Send & MPI_Recv : Deadlocks Remedy

- A remedy: use standard send & receive

```
22   if (rank == 0) {
23     // synchronous send: return when the destination has started
24     // to receive the message
25     //MPI_Ssend(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
26     MPI_Send(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
27     MPI_Recv(&d, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
28   }
29   // workers get the message
30   else {
31     //MPI_Ssend(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
32     MPI_Send(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
33     MPI_Recv(&d, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
34   }
```

- For such a small message MPI will always buffer it when using a standard send.
  However, the deadlock may still occur depending on the size of the buffer.

Non-blocking Communication

- For the moment, we have only seen **blocking** point-to-point communication. That means that when a process sends or receive information, it has to wait for the transmission to end to get back to what it was doing.
- In some applications, this can be terribly limiting.
- Let's take a first example to figure out why.

Non-blocking Communication Example

- Example:
  - In this case, process 0 has some information to send to process 1.
  - But both are working on very different things, so take different time to finish their computations.
  - Process 0 is ready to send its data first, but since process 1 has not finished its own computations, process 0 has to wait for process 1 to be ready before getting back to its own work.
  - Process 1 finishes treating the data really quickly and now waits for process 0 to finish for getting new data.
  - Blocking case:

  - Non-blocking case:

Non-blocking Communication Example

- Example:
  - Blocking case:

  - Non-blocking case:

- Blocking Case: In terms of pseudo-time and pseudo-code

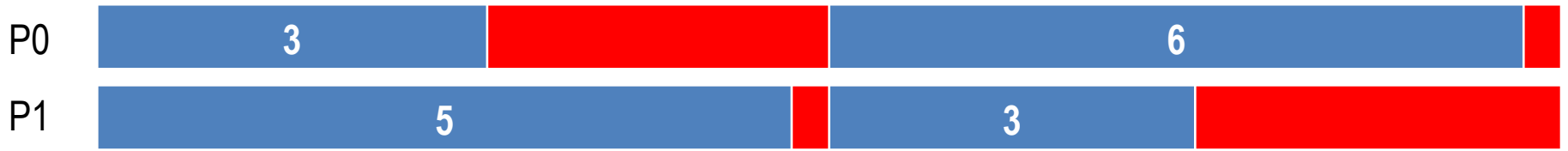
P0
- Work for 3 seconds
- Blocking-send data to P1
- Work for 6 seconds
- Blocking-send data to P1

P1
- Work for 5 seconds
- Blocking-receive data from P1
- Work for 3 seconds
- Blocking-receive data to P1

```
if (rank == 0) {
    sleep(3);
    MPI_Send(buffer, buffer_count, MPI_INT, 1, 0, MPI_COMM_WORLD);
    sleep(6);
    MPI_Send(buffer, buffer_count, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else {
    sleep(5);
    MPI_Recv(buffer, buffer_count, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    sleep(3);
    MPI_Recv(buffer, buffer_count, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Non-blocking Communication Example

- Example:
  - Blocking case:



  - Non-blocking case:

- Non-Blocking Case: In terms of pseudo-time

P0

Work for 3 seconds
Initialize the send to P1
Work for 6 seconds
    Every 1ms, probe P1 and communicate if necessary
Initialize the second send to P1
Wait for P1 to receive the data

P1

Work for 5 seconds
Initialize receive from P0
Wait for a communication from P0
Work for 3 seconds
Initialize receive from P0
Wait for a communication from process 0

It makes the code a little bit more complex. Also, note that we are sort of cheating. Since we already know the time taken by both processes in the work phases, we make things a bit easier by forcing P1 to wait for the data. The important part is in P0 where the probing is done every millisecond.

Non-blocking Communication Example

- Non-Blocking Case: In terms of pseudo-time and pseudo-code

P0

Work for 3 seconds
Initialize the send to P1
Work for 6 seconds
  Every 1ms, probe P1 and communicate if necessary
Initialize the second send to P1
Wait for P1 to receive the data

P1

Work for 5 seconds
Initialize receive from P0
Wait for a communication from P0
Work for 3 seconds
Initialize receive from P0
Wait for a communication from process 0

```
if rank == 0 then
    Work for 3 seconds
    Initialize the send to P1
    Work for 6 seconds
        Every milli-second, test if P1 is ready to communicate
    Send the second batch of data to P1
    Wait for process 1 to receive the data
else if rank == 1 then
    Work for 5 seconds
    Initialize receive from P0
    Wait for a communication from P0
    Work for 3 seconds
    Initialize receive from P0
    Wait for a communication from P0
```

Non-blocking Communication

MPI_Isend

Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISEND (buf,count,datatype,dest,tag,comm,request,ierr)
```

MPI_Irecv

Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

```
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
MPI_IRECV (buf,count,datatype,source,tag,comm,request,ierr)
```

Non-blocking Send/Recv Details

- The "request" is used to query the status of the communicator or to wait for its completion.
  - Something we haven't seen before : MPI_Request.
  - Unlike MPI_Status though, MPI_Request is a complex object.
  - See it that way : MPI_Isend is preparing a request. This request is going to be executed when both processes are ready to synchronize.
  - This command only sets up the send, but actually does not transfer anything to the destination process, only prepares it.
  - As for its sending equivalent, MPI_Irecv returns a MPI_Request object.
  - The request must then be completed by either waiting or testing.

MPI_Wait()

Name

MPI_Wait - Waits for an MPI send or receive to complete.

Syntax

C Syntax

```
#include <mpi.h>
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Fortran Syntax

```
INCLUDE 'mpif.h'
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER    REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

C++ Syntax

```
#include <mpi.h>
void Request::Wait(Status& status)
void Request::Wait()
```

- Waiting forces the process to go in "blocking mode".
- The sending process will simply wait for the request to finish.
- If your process waits right after MPI_Isend, the send is the same as calling MPI_Send.
- MPI_Wait just waits for the completion of the given request.
- As soon as the request is complete an instance of MPI_Status is returned in status.

MPI_Test()

Name

MPI_Test - Tests for the completion of a specific send or receive.

Syntax

C Syntax

```
#include <mpi.h>
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Fortran Syntax

```
USE MPI
! or the older form: INCLUDE 'mpif.h'
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL    FLAG
    INTEGER    REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

- Testing is a little bit different.
- Waiting blocks the process until the request (or a request) is fulfilled, but testing checks if the request can be completed.
- If it can, the request is automatically completed and the data transferred.
- Remember that testing is non-blocking, so in any case the process continues execution after the call.
- The variable flag is there to tell you if the request was completed during the test or not. If flag != 0 that means the request has been completed.

Non-blocking Communication Example

- Non-Blocking Case: In terms of pseudo-time and pseudo-code

```
if rank == 0 then
    Work for 3 seconds
    Initialize the send to P1
    Work for 6 seconds
        Every milli-second, test if P1 is ready to
communicate
    Send the second batch of data to P1
    Wait for process 1 to receive the data
else if rank == 1 then
    Work for 5 seconds
    Initialize receive from P0
    Wait for a communication from P0
    Work for 3 seconds
    Initialize receive from P0
    Wait for a communication from P0
```

```
MPI_Request request;
MPI_Status  status;
int         request_complete = 0;

// Rank 0 sends, rank 1 receives
if (rank == 0) {
  sleep(3);
  MPI_Isend(buffer, buffer_count, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

  // Here we do some work while waiting for process 1 to be ready
  while (has_work) {
    do_work();

    // We only test if the request is not already fulfilled
    if (!request_complete)
       MPI_Test(&request, &request_complete, &status);
  }

  // No more work, we wait for the request to be complete if it's not the case
  if (!request_complete)
    MPI_Wait(&request, &status);
}
else {
  sleep(5);
  MPI_Irecv(buffer, buffer_count, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
  // Here we just wait for the message to come
  MPI_Wait(&request, &status);

  …
}
```

Lab (by yourself): Numerical Integration with MPI P2P

- Numerical integration is chosen as the instructional example for its trivially simplistic algorithm to parallelize; the task is narrowly confined yet computationally relevant.
- For this lab, the integrand is the cosine function and the range of integration, from a to b, has length b-a.
- The work share is evenly divided by p, the number of processors, so that each processor is responsible for integration of a partial range, (b-a)/p.
- Upon completion of the local integration on each processor, processor 0 is designated to collect the local integral sums of all processors to form the total sum.
- Copy the partial code located at apex.slu.edu:/tmp/mpi_p2p_block_integ.cpp to your working directory, prepare a shell, and submit a job!

```cpp
/**********************************************************************
 * FILE: mpi_p2p_block_integ.cpp
 * DESCRIPTION:
 *   Integration with MPI Nonblocking Communications
 * AUTHOR: Tae-Hyuk (Ted) Ahn
 * LAST REVISED: 02/20/2020
 **********************************************************************/
#include <iostream>
#include <math.h>
#include <mpi.h>
#define MASTER 0

void other_work(int rank);
float integral(float ai, float h, int n);

using namespace std;

int main(int argc, char **argv)
{
  int nprocs, rank, tag, n;
  float pi, a, b, h, ai, my_integral, integral_sum;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  pi = acos(-1.0);  /* = 3.14159... */
  a = 0.;           /* lower limit of integration */
  b = pi*1./2.;     /* upper limit of integration */
  n = 500;          /* number of increment within each process */
  tag = 123;        /* set the tag to identify this particular job */
  h = (b-a)/n/nprocs;    /* length of increment */

  ai = a + rank*n*h;  /* lower limit of integration for partition rank */
  my_integral = integral(ai, h, n);    /* 0<=rank<=p-1 */

  cout << "Process " << rank << " has the partial result of " << my_integral << endl;

  if (rank == MASTER) {
    integral_sum = my_integral;
    for (int i=1; i<nprocs; i++) {
      MPI_Recv(&my_integral, 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      integral_sum += my_integral;
    }
    cout << "Total Integral = " << integral_sum << endl;
  }
  else {
    MPI_Send(&my_integral, 1, MPI_FLOAT, MASTER, tag, MPI_COMM_WORLD);
  }

  MPI_Finalize();
}

void other_work(int rank)
{
  cout << "More work on process " << rank << endl;
}

float integral(float ai, float h, int n)
{
  int j;
  float aij, integ;

  integ = 0.0;              /* initialize */
  for (j=0; j<n; j++) {     /* sum integrals */
    aij = ai + (j+0.5)*h;   /* mid-point */
    integ += cos(aij)*h;
  }
  return integ;
}
```

```cpp
/*************************************************************
* FILE: mpi_p2p_nonblock_integ.cpp
* DESCRIPTION:
*   Integration with MPI Nonblocking Communications
* AUTHOR: Tae-Hyuk (Ted) Ahn
* LAST REVISED: 02/20/2020
*************************************************************/
#include <iostream>
#include <math.h>
#include <mpi.h>
#define MASTER 0

void other_work(int rank);
float integral(float ai, float h, int n);

using namespace std;

int main(int argc, char **argv)
{
  int nprocs, rank, tag, n;
  float pi, a, b, h, ai, my_integral, integral_sum;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  pi = acos(-1.0);  /* = 3.14159... */
  a = 0.;           /* lower limit of integration */
  b = pi*1./2.;     /* upper limit of integration */
  n = 500;          /* number of increment within each process
  tag = 123;        /* set the tag to identify this particular
  h = (b-a)/n/nprocs;    /* length of increment */

  ai = a + rank*n*h;  /* lower limit of integration for partition rank */
  my_integral = integral(ai, h, n);    /* 0<=rank<=p-1 */

  cout << "Process " << rank << " has the partial result of " << my_integral << endl;

  if (rank == MASTER) {
    integral_sum = my_integral;
    for (int i=1; i<nprocs; i++) {
      MPI_Recv(&my_integral, 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      integral_sum += my_integral;
    }
    cout << "Total Integral = " << integral_sum << endl;
  }
  else {
    MPI_Isend(&my_integral, 1, MPI_FLOAT, MASTER, tag, MPI_COMM_WORLD, &request);
    other_work(rank);
    MPI_Wait(&request, &status);
  }

  MPI_Finalize();
}

void other_work(int rank)
{
  cout << "More work on process " << rank << endl;
}

float integral(float ai, float h, int n)
{
  int j;
  float aij, integ;

  integ = 0.0;              /* initialize */
  for (j=0; j<n; j++) {     /* sum integrals */
    aij = ai + (j+0.5)*h;   /* mid-point */
    integ += cos(aij)*h;
  }
  return integ;
}
```

Discussion

- A nonblocking MPI_Isend call returns immediately to the next statement without waiting for the task to complete. This enables other_work to proceed right away. This usage of nonblocking send (or receive) to avoid processor idling has the effect of “latency hiding,” where latency is the elapse time for an operation, such as MPI_Isend, to complete.
- Another performance enhancement parameter applied to this example is the use of MPI_ANY_SOURCE to specify message source. The wildcard nature of MPI_ANY_SOURCE enables the messages to be summed in the order of their arrival rather than any imposed sequence (such as the loop-index order used in the preceeding examples). It is important to note that summation is a mathematical operation that satisfies the associative and commutative rules and hence the order in which the integral sums from processors are added is not pertinent to the outcome.
- You can check the source and tag using status.

Collective Communication

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

MPI Collective Communications

- **Point-to-Point (P2P) Communication** sends data from one point to another, one task sends while another receives.
  - **Blocking** (MPI_Send() and MPI_MPI_Recv()) send routing will only return after it is safe to modify the buffer.
  - **Non-blocking** (MPI_Isend() and MPI_Irecv()) send/receive routines return immediately.
  - Improper use of blocking receive/send will result in **deadlock**, where two processors can't progress because each of them is waiting on the other.
- **Collective communication** is defined as communication between more than two (usually many) processors. A few forms:
  - One-to-many
  - Many-to-one
  - Many-to-many

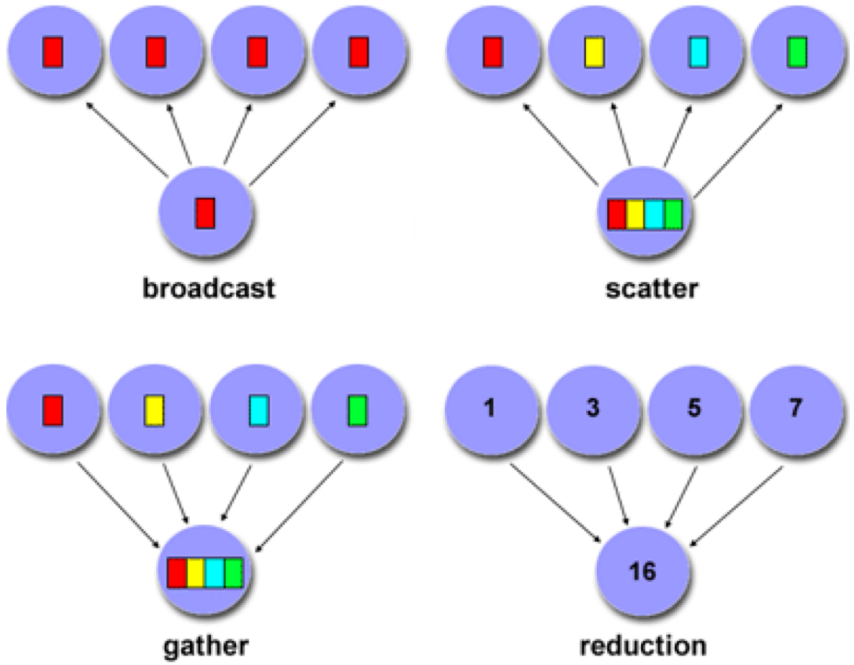
Scope

- Collective communication routines must involve **all** processes within the scope of a communicator.
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
  - Additional communicators can be defined by the programmer.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Types of Collective Operations

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.

- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



Barrier for Synchronization

- MPI has a special function that is dedicated to synchronizing processes:

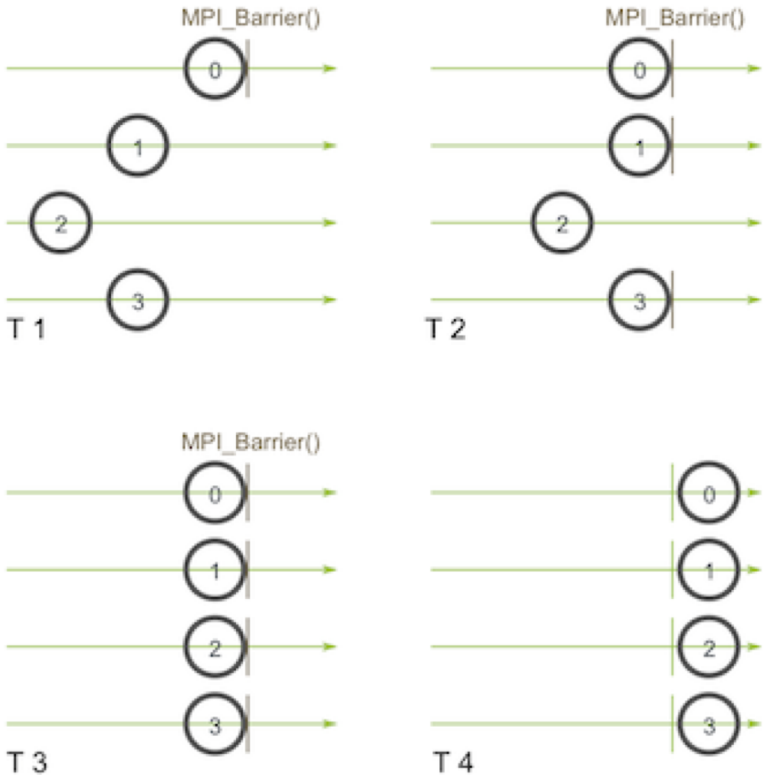
  ```
  MPI_Barrier( MPI_COMM comm )
  ```

  comm: the group of processes

- Blocks until all processes in the group call it.
- The name of the function is quite descriptive - the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function.

Barrier for Synchronization

- Process zero first calls MPI_Barrier at the first time snapshot (T 1).
- While process zero is hung up at the barrier, process one and three eventually make it (T 2).
- When process two finally makes it to the barrier (T 3), all of the processes then begin execution again (T 4).

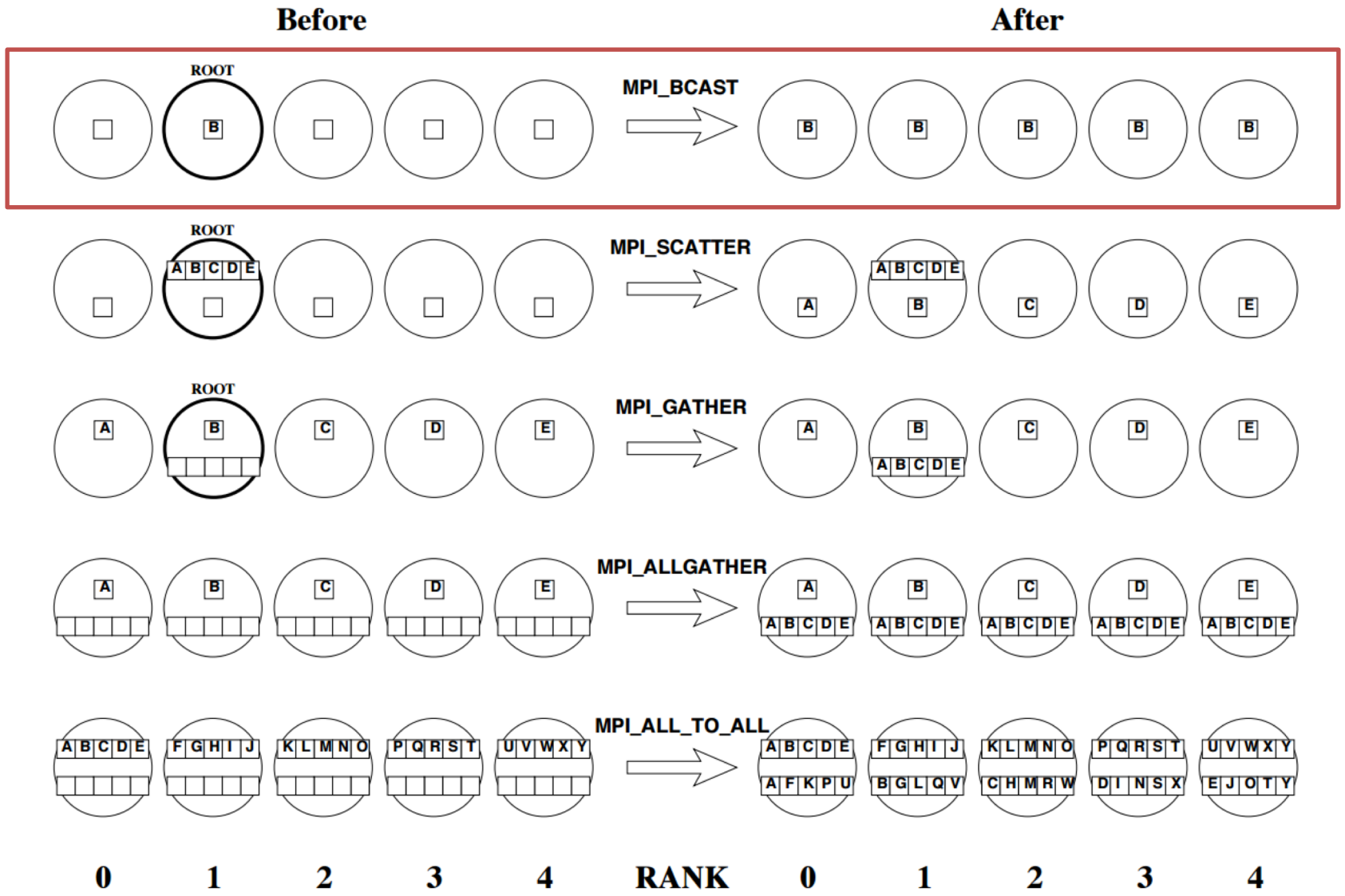


- MPI_Barrier can be useful for many things. One of the primary uses of MPI_Barrier is to synchronize a program so that portions of the parallel code can be timed accurately.

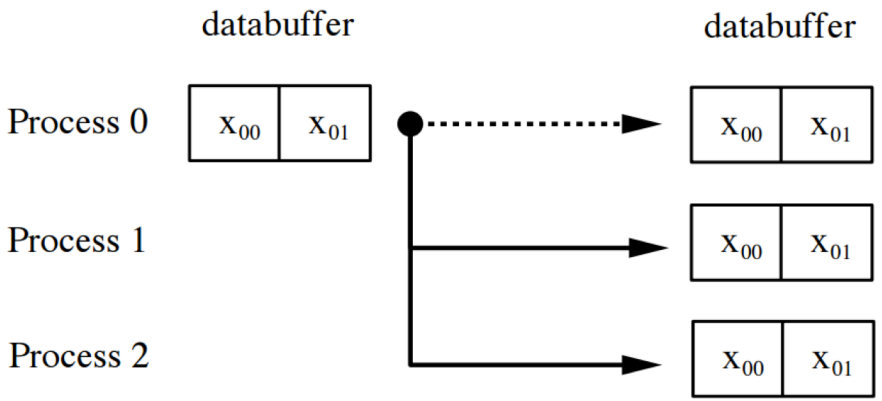
Barrier for Synchronization

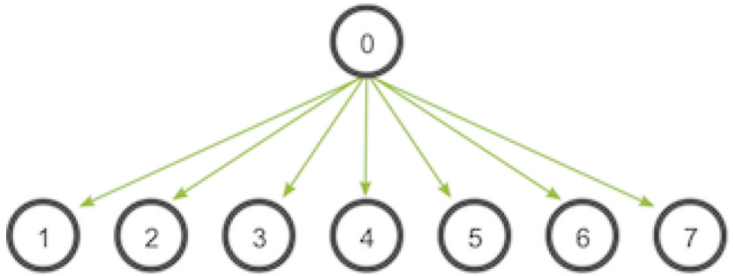
- All collective operations in MPI are blocking, which means that it is safe to use all buffers passed to them after they return.
  - In particular, this means that all data was received when one of these functions returns. (However, it does not imply that all data was sent!) So MPI_Barrier is not necessary (or very helpful) before/after collective operations, if all buffers are valid already.
- Please also note, that MPI_Barrier does not magically wait for non-blocking calls.
  - If you use a non-blocking send/recv and both processes wait at an MPI_Barrier after the send/recv pair, it is not guaranteed that the processes sent/received all data after the MPI_Barrier. Use MPI_Wait (and friends) instead.

Data Movement



Broadcast





- A **broadcast** is one of the standard collective communication techniques.
- During a broadcast, one process sends the same data to all processes in a communicator.
- One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

Broadcast: MPI_Bcast

- C Syntax: MPI_Bcast(data, count, datatype, root, comm)
- C++ Syntax: MPI::Comm.Bcast(data, count, datatype, root)
- No tag!!
- A broadcast has a specified root process and every process receives one copy of the message from the root.
- All processes must specify the same root (and communicator).
- The root argument is the rank of the root process.
- The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

Lab: mpi_collective_bcast.cpp

```
# include <iostream>
# include <cstdlib> // has exit(), etc.
# include <ctime>
# include "mpi.h"   // MPI header file

using namespace std;

//*************************************************************************80
int main (int argc, char **argv)
//*************************************************************************80
{
  int nprocs, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int data = 0;
  cout << "Before Bcast, data = " << data << " in rank = " << rank << endl;

  if (rank == 0) {  // root
    data = 100;
  }

  MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);     // MPI_Bcast
  cout << "After Bcast, data = " << data << " in rank = " << rank << endl;

  MPI_Finalize();
  return 0;
}
```