

Lecture 20: Basic communications in MPI

1. Point to point communications
 1. MPI_Send
 2. MPI_Receive
2. Collective communications
 1. MPI_Bcast
 2. MPI_Reduce
 3. MPI_Allreduce
 4. MPI_Scatter
 5. MPI_Gather
 6. MPI_Allgather

Send

CALL MPI_SEND(buff, count, MPI_TYPE, dest, tag, comm, ierr)

FORTRAN_TYPE:: buff

INTEGER:: count, dest, ierr

- *buff* : The variable you want to send.
- *count* : The number of variables you're passing. If you're passing only a single value, this should be 1. If you're passing an array, it's the overall size of the array. For example, if you wanted to send a 4 by 5 array, count would be $4*5=20$.
- *MPI_TYPE* : The kind of variable you're passing so the MPI routine knows what to expect.
 - MPI_INTEGER,
 - MPI_REAL
 - MPI_DOUBLE_PRECISION,
 - MPI_COMPLEX,
 - MPI_LOGICAL,
 - MPI_CHARACTER
- *dest* : The ID number of the task you're sending the message to.
- *tag* : a message tag. This is a way for the receiver to verify that it's getting the message it expects. The message tag is an integer number that you can assign any value. Can be MPI_ANY_TAG
- *comm* : This is the group ID of tasks that your message is going to. In large, complex programs tasks may be divided into groups to speed connections and transfers. In small programs, this will more than likely be MPI_COMM_WORLD.
- *ierr* : an integer error code.

Receive

CALL MPI_RECV(rbuf, count, MPI_TYPE, source, tag, comm, status, ierr)

FORTTRAN_TYPE:: rbuf

INTEGER:: count, source, tag, status(MPI_STATUS_SIZE), ierr

- The arguments that are different from those in MPI_SEND are
- *rbuf* : This is the name of the variable where you'll be storing the received data.
- *source* : This replaces the destination in the send command. This is the return ID of the sender. Can be MPI_ANY_SOURCE
- *status* : You can check this variable to see if the receive was completed.

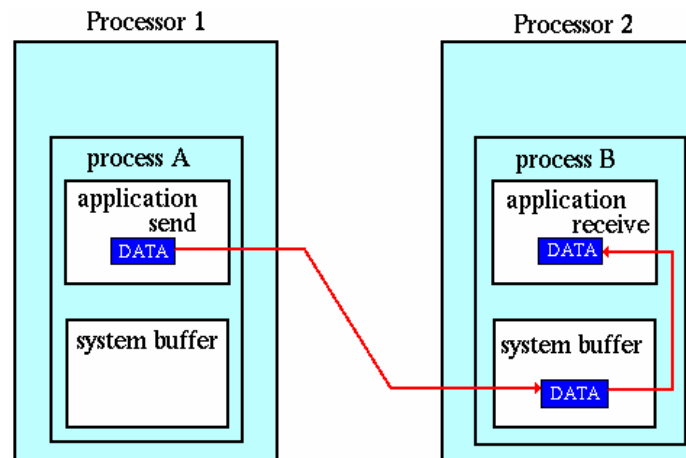
Point to point communications

- Allow two processes (or tasks) to communicate (send or receive data)
- There are different ways in MPI to send / receive messages. They differ in the way, the message (data + envelope) is handled.
- Buffering: In practice, a send operation is often not perfectly synchronized with its matching receive. The MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Example:

A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?

Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?

The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a **system buffer** area is reserved to hold data in transit. For example:



Path of a message buffered at the receiving process

- **Blocking vs. Non-blocking:**
- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- Blocking: [subject of this course]
 - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
 - A blocking receive only "returns" after the data has arrived and is ready for use by the program.
- Non-blocking:
 - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
 - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
- **Order and Fairness:**
 - Order:
 - MPI guarantees that messages will not overtake each other.
 - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
 - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
 - Fairness:
 - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
 - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.

```

! program sums all numbers from 1 to 1000
  program main
!
    implicit none
    include 'mpif.h'
!
    integer, parameter:: n=1000
    integer:: i, sum, ibuf
!
! MPI variables
    integer :: comm, myid, nprocs, ierr, tag
    integer :: status(MPI_STATUS_SIZE)
!
    call MPI_INIT(ierr)
    comm=MPI_COMM_WORLD
    call MPI_COMM_RANK(comm, myid, ierr)
    call MPI_COMM_SIZE(comm, nprocs, ierr)
!
    tag=myid*10
!
    sum=0
    do i=1+myid, n, nprocs
        sum = sum + i
    enddo
!
    if(myid.eq.0) then
        do i=1, nprocs-1
            call MPI_RECV(ibuf,1,MPI_INTEGER,MPI_ANY_SOURCE,MPI_ANY_TAG,&
                comm,status,ierr) !receive in any order
            sum = sum + ibuf
            write(*,*) 'received from task ', status(MPI_SOURCE), &
                'with tag', status(MPI_TAG)
        enddo
        write(*,*) 'the sum is ', sum, 'compare with', (n*(n+1))/2
    else
        call MPI_SEND(sum,1,MPI_INTEGER,0,tag,comm,ierr)
    endif
!
    call MPI_FINALIZE(ierr)
end program main

```

```

[papenbro@max phys573]$ mpirun -np 4 ./a.out
received from task      1 with tag      10
received from task      2 with tag      20
received from task      3 with tag      30
the sum is      500500 compare with      500500

```

Collective communications

Broadcast from task root to everyone:

CALL MPI_BCAST(buff, count, MPI_TYPE, root, comm, ierr)

FORTRAN_TYPE:: buff

INTEGER:: count, root, ierr

- *buff* : the variable name of the data being broadcast. When the call returns, all tasks will have the same value placed in *buff*
- *root* : This is the task which has the value to be shared with all the others.

Reduction and collection of results in root:

CALL MPI_REDUCE(sendbuf, recbuf, count, MPI_TYPE, MPI_OP, root, comm, ierr)

FORTRAN_TYPE:: sendbuf, recbuf

INTEGER:: count, root, ierr

- *sendbuf* : the variable to be collected from all tasks.
- *recbuf* : the variable where the result of operating MPI_OP on all those values will be placed
- *MPI_OP* : one of a number of functions to be performed on the values being collected. E.g. MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD
- *root* : the only task that will collect the result of the reduce command in *recbuf*.

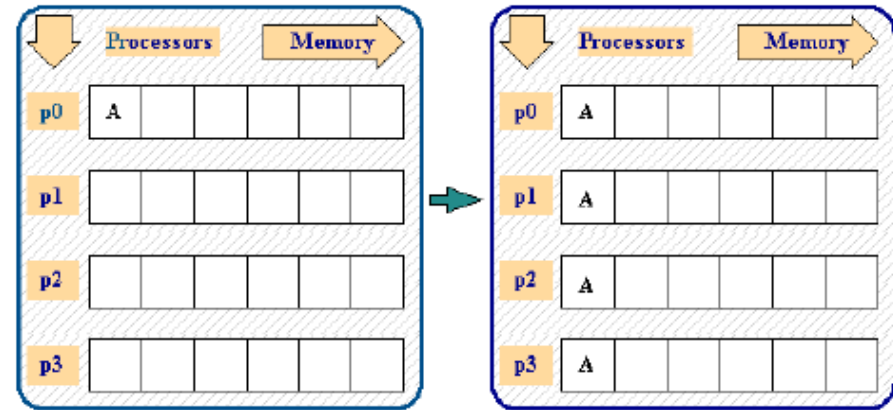
Reduction and collection of result by everyone

CALL MPI_ALLREDUCE(sendbuf, recbuf, count, MPI_TYPE, MPI_OP, comm, ierr)

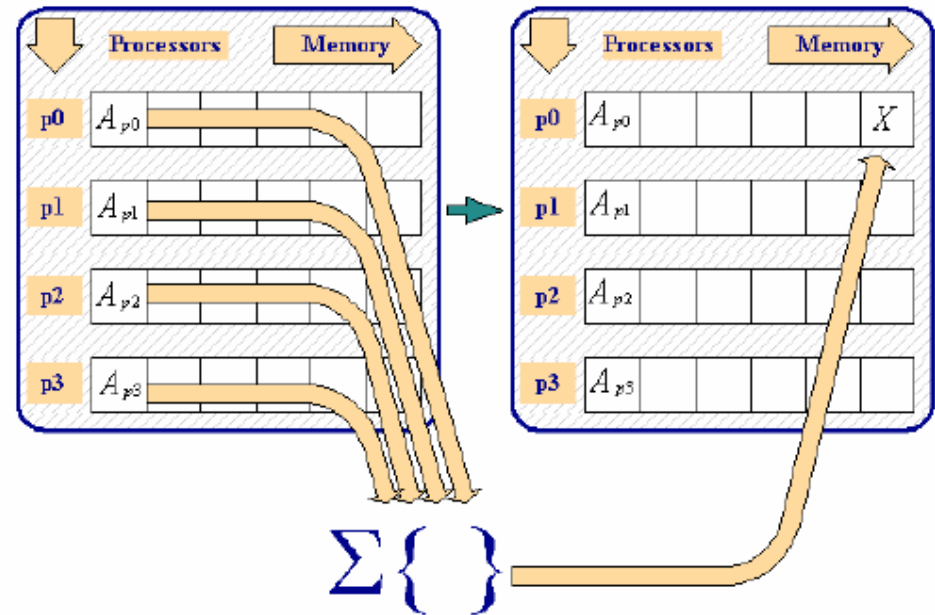
Note: Collective communications faster than nprocs send/receive

Broadcast and Reduce

MPI_BCast



MPI_Reduce



```

! program sums all numbers from 1 to 1000
  program main
!
!   implicit none
!   include 'mpif.h'
!   include '/home/papenbro/lam/include/mpif.h'
!
!   integer, parameter:: n=1000
!   integer:: i, sum, total
!
!   integer :: comm, myid, nprocs, ierr
!
!   call MPI_INIT(ierr)
!   comm=MPI_COMM_WORLD
!   call MPI_COMM_RANK(comm, myid, ierr)
!   call MPI_COMM_SIZE(comm, nprocs, ierr)
!
!   sum=0
!   do i=1+myid, n, nprocs
!     sum = sum + i
!   enddo
!
!   write(*,*) myid, sum
!   total=0
!
!   call MPI_REDUCE(sum,total,1,MPI_INTEGER,MPI_SUM,0,comm,ierr)
!
!   if(myid.eq.0) write(*,*) 'the sum is ', total, 'compare with', (n*(n+1))/2
!
!   call MPI_FINALIZE(ierr)
end program main

```

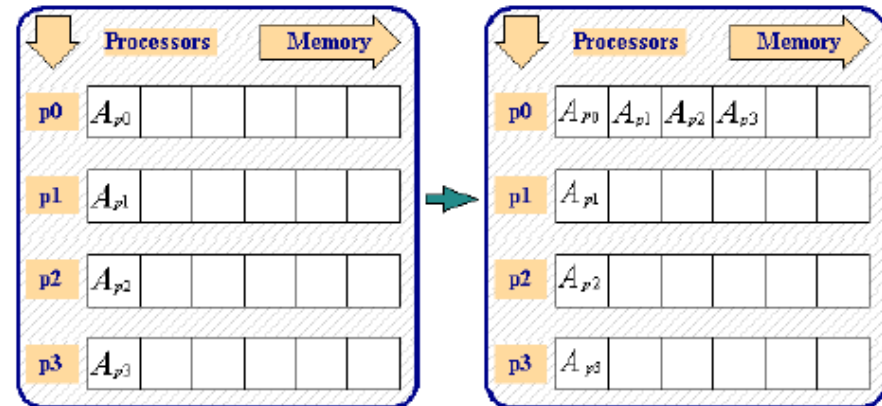
```

[papenbro@max phys573]$ mpirun -np 4 ./a.out
      0      124750
      1      125000
the sum is      500500 compare with      500500
      2      125250
      3      125500

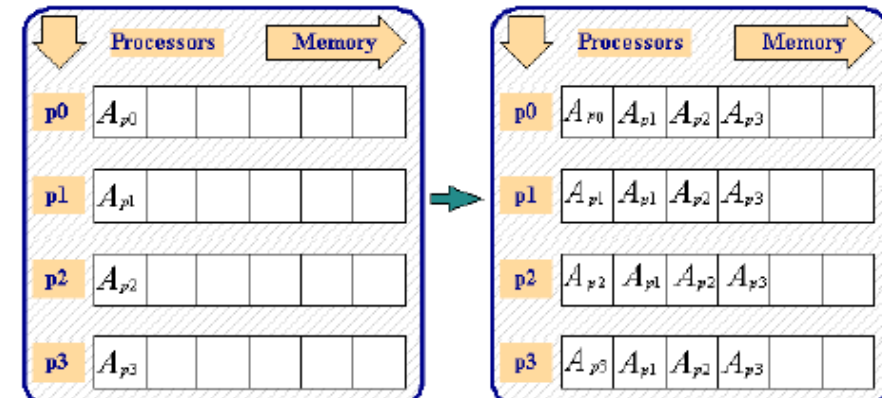
```

Further collective communication subroutines

MPI_Gather



MPI_Allgather



MPI_Scatter

