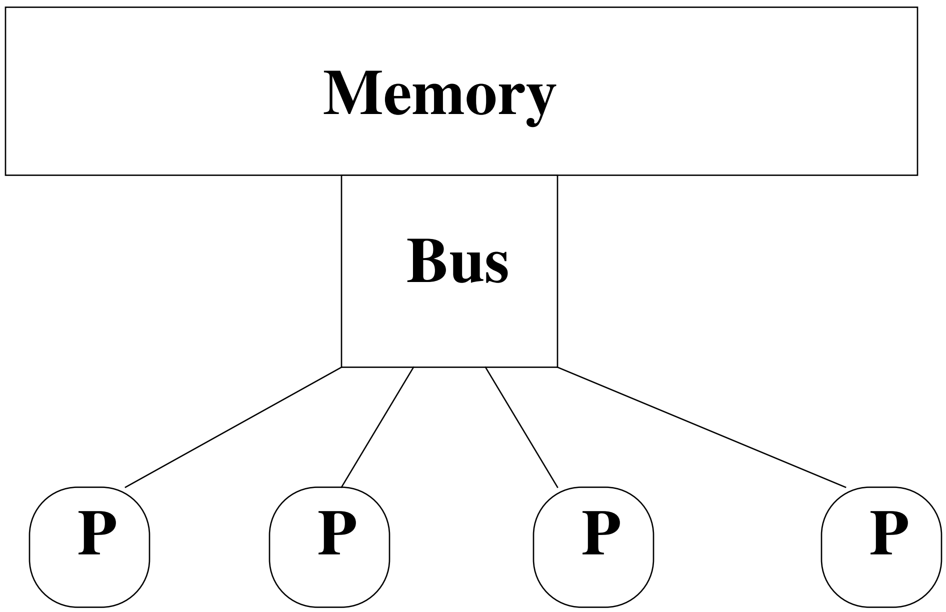
Parallel Programming with MPI

Chris Ding
NERSC/LBNL
cding@nersc.gov

- Distributed Architectures
- Parallelisms
- Why MPI
- Getting started
- Point-to-point communication
- Message Buffering
- Collective Communications
- Process Groups
- Summary
- Coding Examples

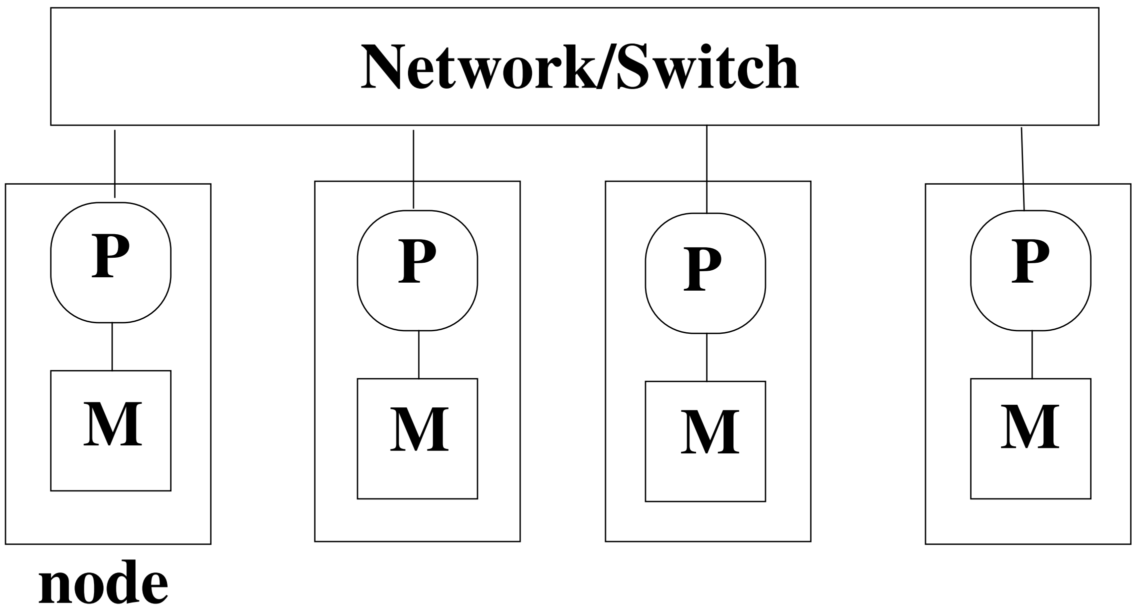
A complete references on MPI including tutorials, books is on the web:
http://www.mcs.anl.gov/mpi/

Shared-Memory Architectures (SMP)
(Cray YMP, DEC Alpha)



- Easy to program (parallel regions, locks)
- Difficult to scale to large number of processors

Distributed-Memory Architectures
(T3D/T3E, SP2, Paragon)

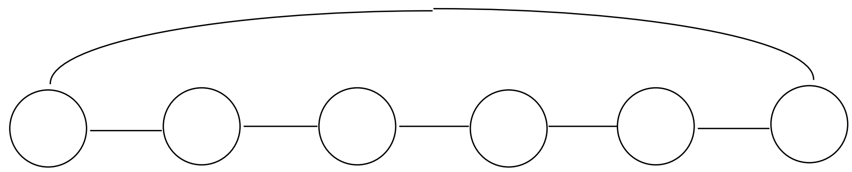


- Scalable to larger number of nodes
- Use message passing at hardware level
- Difficult to program
- Global system level OS not yet matured

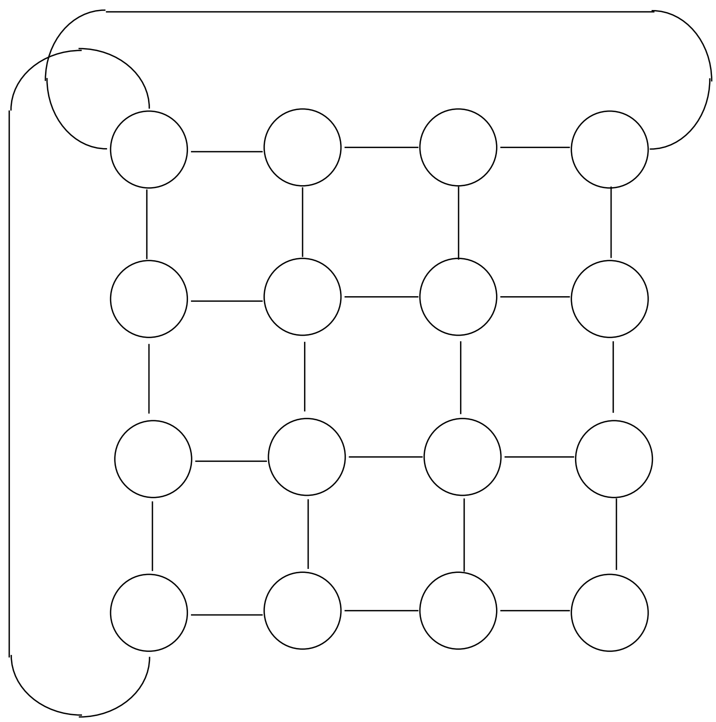
Cluster of SMP nodes
- Single node becomes very powerful
- Commodity products --- economics

Distributed-Memory Architectures

Ring



2D Torus



Point-to-point Communication:

- Latency
- bandwidth

Network congestion:

- bisection bandwidth

Tightly coupled : Massively Parallel Processors (MPP)

Loosely coupled: cluster of workstations (NOW)

Distributed-Memory Architectures

SIMD - single instruction, multiple data

MIMD - multiple instructions, multiple data

- Same program on different nodes
- Different programs on different nodes

SPMD - mode : Single program, multiple data.

- Identical program (binary executable) run on all processes.
- Different processor differ only a processor id.
- Independent execution threads
- Synchronized by barrier or messages

SPMD Programming Models

Explicit Message Passing

- Sequential codes + message passing library
- Multi-threaded programming
- Native msg passing lib: NX, MPL, SHMEM
- PVM
- MPI

HPF (High Performance Fortran)

- Data- parallelism
- Global memory space
- Single-thread programming

Languages supporting global memory space

- Using language constructs to replace msg passing library
- Split-C
- F - -

Why MPI?

- A unified standard for portability
- More features/functionalities

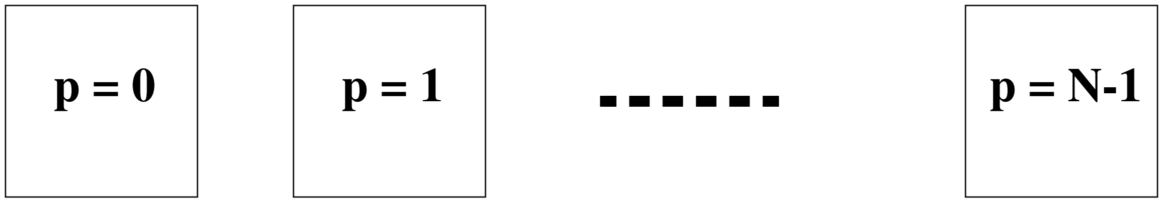
What is MPI

- A message passing interface standard, a set of library specifications

What is in MPI

- Point-to-point communication
- Collective communication
- Process groups/Communication context
- Application process topologies
- Profiling
- System environment

Message Passing Model



- Processes (processors, processing element, etc) are identical, except each has a different process id, the rank in MPI.
- Information exchange between the processes are through explicit message passing only.
- SPMD (Single program, multiple data) mode, i.e., same identical program (binary executable) run on all processes.

Multi-threaded Programming:

```
my_id = processor_id()
if(my_id  .EQ. 0) then
        !  do fft,  then pass data to P1
else if(my_id  .EQ.  1) then
        !  solve an equation, then pass data to P0
end if
```

Getting Started with MPI

```
      implicit none
      include 'mpif.h'

      integer myPE, totPEs, ierr

      call MPI_INIT( ierr )

      call MPI_COMM_RANK( MPI_COMM_WORLD,
     &                            myPE, ierr)

      call MPI_COMM_SIZE( MPI_COMM_WORLD,
     &                            totPEs, ierr )

      call MPI_FINALIZE(ierr)

      print *, "myPE is ", myPE, " of total ", totPEs, "
     &        PEs"

      end
```

To Compile on T3E:

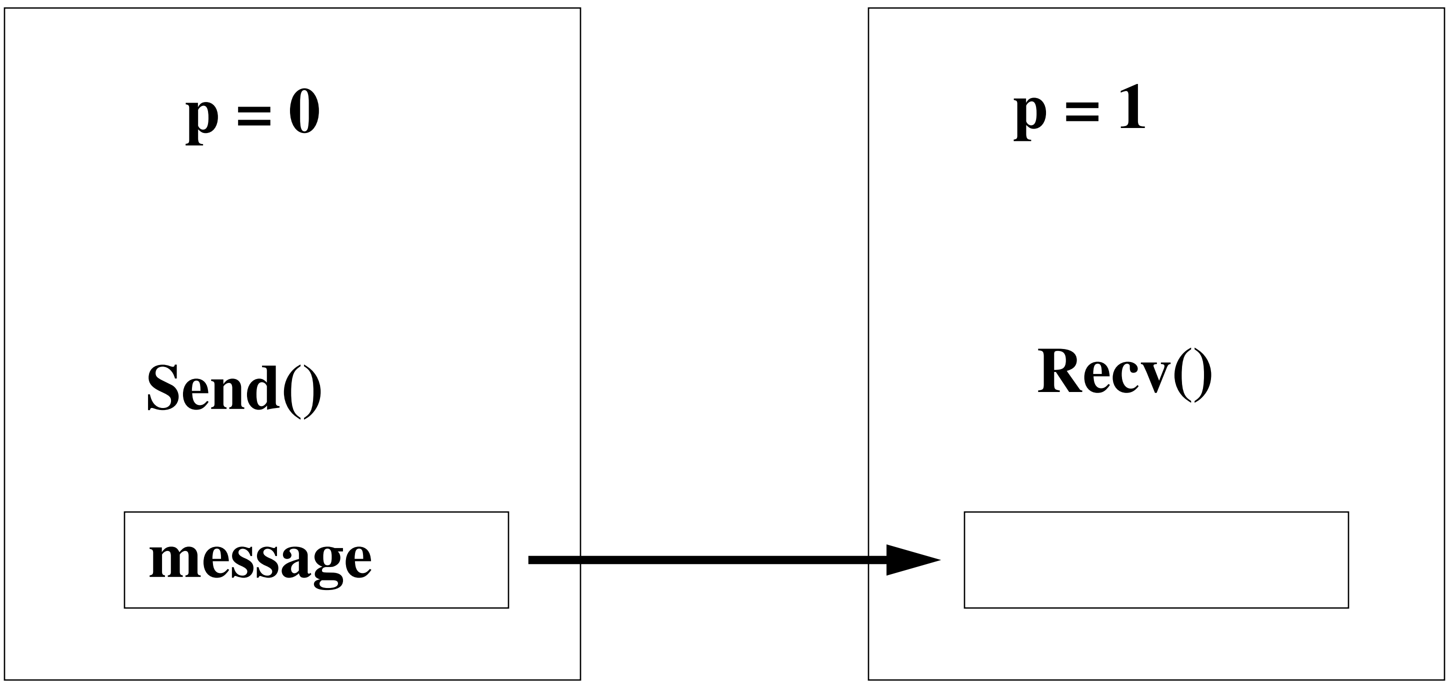
```
f90 hello.f -o hello
```

To run on T3E

```
mpprun -n4 ./hello
```

Send/Receive

(Point-to-Point Communication)



- What is being sent out (message content)
- Sent to whom? (destination process)
- How to uniquely identify this message (message tag)

Basic MPI Send/Recv

subroutine **MPI_Send**(array, count, datatype,
dest_pro, tag, comm, ierror)

subroutine **MPI_Recv**(array, count, datatype,
src_pro, tag, comm, status, ierror)

integer array(*) ! data to be sent out (MPI_Send)
! data received (MPI_Recv)
real array(*)
byte array(*)
integer count ! # of data elements in array
integer datatype ! **MPI_INTEGER**, **MPI_REAL**, **MPI_BYTE**.

Send/Recv are matched by the message envelop:
integer src_pro ! source processor
integer dest_pro ! destination processor
integer tag ! message tag, non-negative
integer comm ! communicator, **MPI_COMM_WORLD**

integer ierror ! returned error message
integer status(**MPI_STATUS_SIZE**) ! information on
actually received message

Wild Cards in Recv()

subroutine **MPI_Recv**(array, count, datatype,
MPI_ANY_SOURCE, **MPI_ANY_TAG**,
comm, status, ierror)

To get the count of the actually received message (with a given datatype), use

subroutine **MPI_Get_Count**(status, datatype,
recved_count, ierror)

To get source_proc, use

src_pro = status(**MPI_SOURCE**)

To get message tag, use

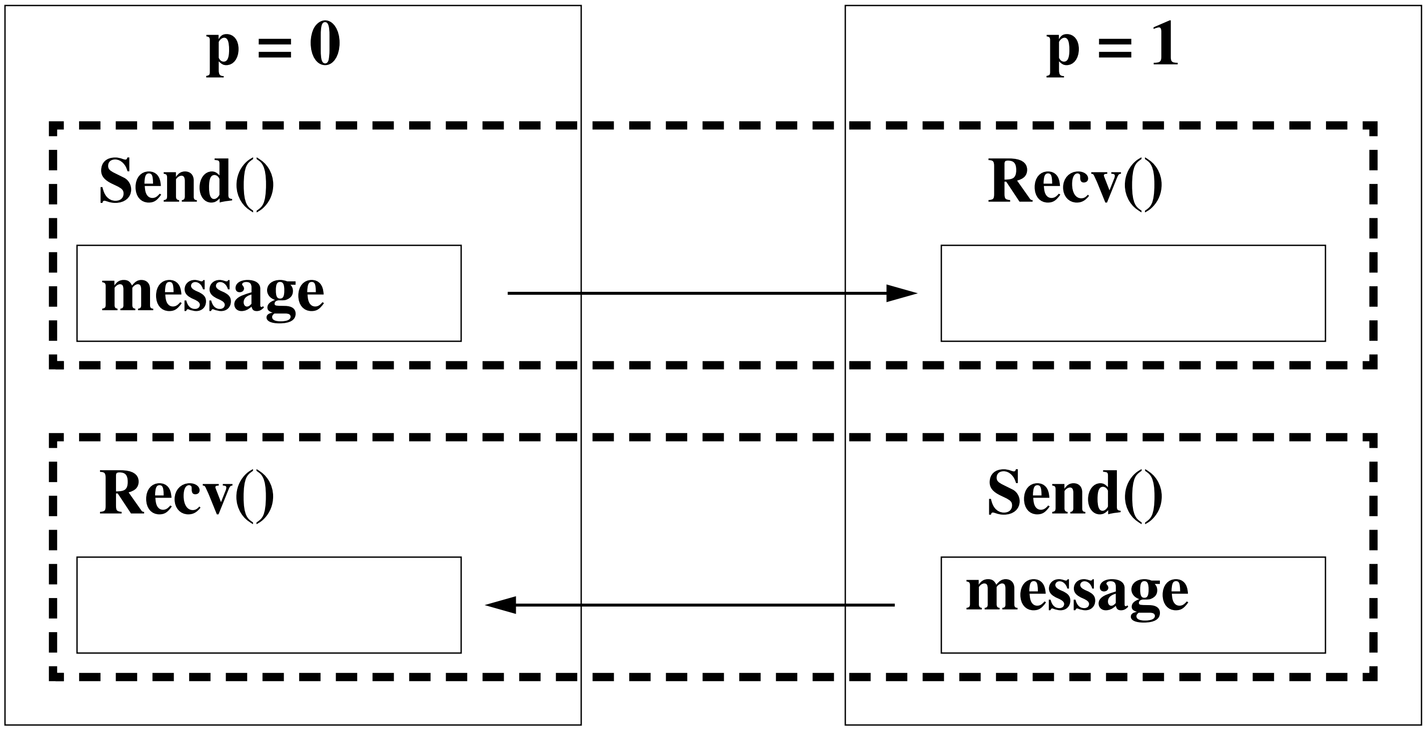
msg_tag = status(**MPI_TAG**)

Send/recv are non-symmetric:

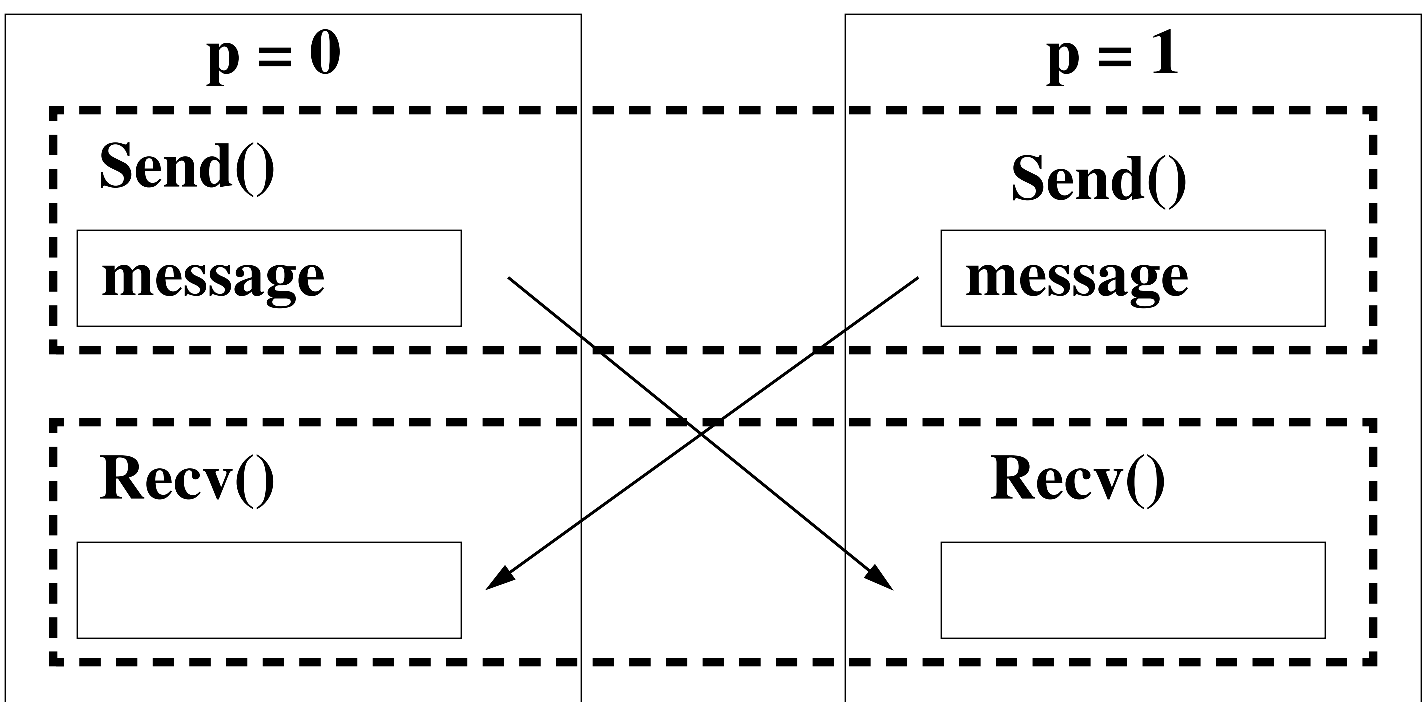
- Actually received msg may differ from specified count, datatype (check status)
- wild cards on recv() side.

Message Buffering in Send/Recv

Normal pattern



Is the following wrong? No, but needs buffering.



Blocking Send/Recv

Send will not return until the message data is sent out or buffered in system buffer and the array containing the message is ready for re-use.

Standard mode: MPI_Send()

Completion **does not** indicate whether the message has been sent out from this processor, or it has been received by the destination processor.

Buffered mode: MPI_Bsend()

Guarantees the message being buffered.

Ready mode: MPI_Rsend()

Returns only after a matching recv() is posted

Synchronous mode: MPI_Ssend()

Return only after a matching recv() is posted and the receive operation is started.

Standard mode: MPI_Recv()

Returns only after a matching message is received and the massage data is copied into the message array.

Non-Blocking Send/Recv

- Initiate a communication request with proper arguments
- Return immediately and the array containing the message is not ready for re-use after returning.
- Must call **MPI_Test**() or **MPI_Wait**() to make sure the operation is completed.

Standard mode: MPI_Isend()

Completion **does not** indicate the message has been buffered, sent out, or received.

Buffered mode: MPI_Ibsend()

Ready mode: MPI_Irsend()

Synchronous mode: MPI_Issend()

Standard mode: MPI_Irecv()

Completion **does not** indicate a matching message is received.

Why Non-Blocking Send/Recv ?

- Overlap communication with computations
- Write “safe” message passing codes

Example for overlaping communication with computation
On two processors: myPE=0, 1.

```
  real a_array(10), b_array(10)

  if(myPE .EQ. 0) then
     call MPI_Isend(a_array, count, MPI_REAL,
&                          1, tag, comm, request, ierror)

C ----   doing some useful work ------
    call MPI_Wait(request, status, ierror)

 else
    call MPI_Irecv(b_array, count, MPI_REAL,
&                          0,tag, comm, request, ierror)

C ----   doing some useful work ------
    call MPI_Wait(request, status, ierror)
```

Non-blocking Send/Recv

Example for “safe” (no buffering necessary) messages.
On N processors: myPE=0, 1, ...,N-1.
All N-1 slave processors send to master a msg of “myPE” real numbers

```
  real a_array(10), b_array(10,N)
  integer  request(N)

  if(myPE .NE. 0) then
     call MPI_Send(a_array, myPE, MPI_REAL,
  &                          0, myPE+1000, comm, ierror)
 else   !  on PE=0, the master processor
    do PE = 1, N-1
    call MPI_Irecv(b_array(1,PE), PE, MPI_REAL,
  &                PE,PE+1000, comm, request(PE), ierror)
    end do

   do PE = 1, N-1
     call MPI_Wait(request(PE), status, ierror)
C  --- put the received msg in proper place, etc.
   end do
```

Message ordering and matching in MPI

- Order is determined with same “dest/src, tag”

```
if(myPE .EQ. 0) then
   call MPI_Send(a_array, count1, MPI_REAL, 1, tag, comm,  ierror)
   call MPI_Send(b_array, count2, MPI_BYTE, 1, tag, comm,  ierror)
else
  call MPI_Recv(c_array, count3, MPI_XXXX, 0, tag, comm, status, ierror)
  call MPI_Recv(d_array, count4, MPI_XXXX, 0, tag, comm, status, ierror)
end if
```

  ~ c_array get the 1st msg; d_array get the 2nd msg

  ~ “c_array,count3, MPI_XXXX” are irrelevant in matching send/recv

- msg order ovetake for differeing “dest/src, tag”

```
if(myPE .EQ. 0) then
   call MPI_Send(a_array, count1, MPI_REAL, 1, tag1, comm,  ierror)
   call MPI_Send(b_array, count2, MPI_BYTE, 1, tag2, comm,  ierror)
else
  call MPI_Recv(c_array, count3, MPI_XXXX, 0, tag2, comm, status, ierror)
  call MPI_Recv(d_array, count4, MPI_XXXX, 0, tag1, comm, status, ierror)
end if
```

  ~ c_array get the 2nd msg; d_array get the 1st msg

Message ordering and matching in MPI

Relie on msg buffering

```
if(myPE .EQ. 0) then
   call MPI_Send(a_array, count1, MPI_REAL, 1, tag, comm,  ierror)
   call MPI_Recv(d_array, count4, MPI_XXXX, 1, tag, comm, status, ierror)
 else
  call MPI_Send(b_array, count2, MPI_BYTE, 0, tag, comm,  ierror)
  call MPI_Recv(c_array, count3, MPI_XXXX, 0, tag, comm, status, ierror)
end if
```

Re-order send/recv to match the sequence

```
if(myPE .EQ. 0) then
   call MPI_Send(a_array, count1, MPI_REAL, 1, tag, comm,  ierror)
   call MPI_Recv(d_array, count4, MPI_XXXX, 1, tag, comm, status, ierror)
 else
  call MPI_Recv(c_array, count3, MPI_XXXX, 0, tag, comm, status, ierror)
  call MPI_Send(b_array, count2, MPI_BYTE, 0, tag, comm,  ierror)
end if
```

Deadlock

```
if(myPE .EQ. 0) then
    call MPI_Recv(d_array, count4, MPI_XXXX, 0, tag, comm, status, ierror)
    call MPI_Send(a_array, count1, MPI_REAL, 1, tag, comm,  ierror)
 else
    call MPI_Recv(c_array, count3, MPI_XXXX, 0, tag, comm, status, ierror)
    call MPI_Send(b_array, count2, MPI_BYTE, 1, tag, comm,  ierror)
 end if
```

Remove deadlock using Irecv

```
if(myPE .EQ. 0) then
    call MPI_Irecv(d_array, count4, MPI_XXXX, 0, tag, comm, request, ierror)
    call MPI_Send(a_array, count1, MPI_REAL, 1, tag, comm,  ierror)
    call MPI_Wait(request,status,ierror)
 else
    call MPI_Irecv(c_array, count3, MPI_XXXX, 0, tag, comm, request, ierror)
    call MPI_Send(b_array, count2, MPI_BYTE, 1, tag, comm,  ierror)
    call MPI_Wait(request,status,ierror)
 end if
```

A Summary of Send/Recv

- Blocking

  MPI_ Send / MPI_Recv

  MPI_Bsend

  MPI_Ssend

  MPI_Rsend
- Nonblocking

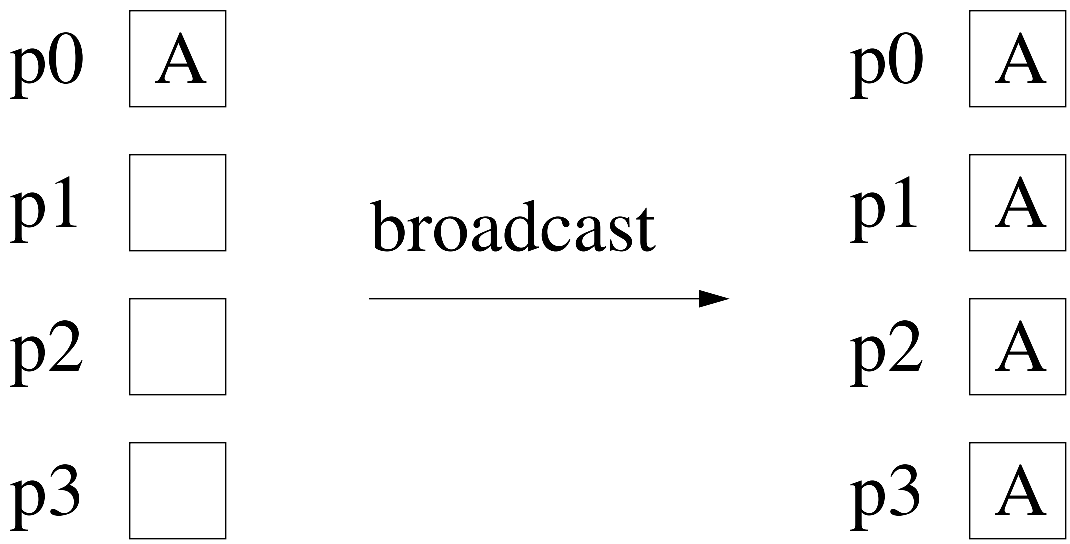
  MPI_Isend / MPI_Irecv

  MPI_Ibsend

  MPI_Issend

  MPI_Issend
- Send/recv are non-symmetric
- Order send/recv to minimize message buffering, but not to worry about it too much.
- Use **MPI_ANY_SOURCE** to ease coding
- Use **MPI_Irecv** and proper message tags to ensure “safe”, “correctness”, and to improve efficiency.
- Use communicator to ensure modularity

Collective Operations

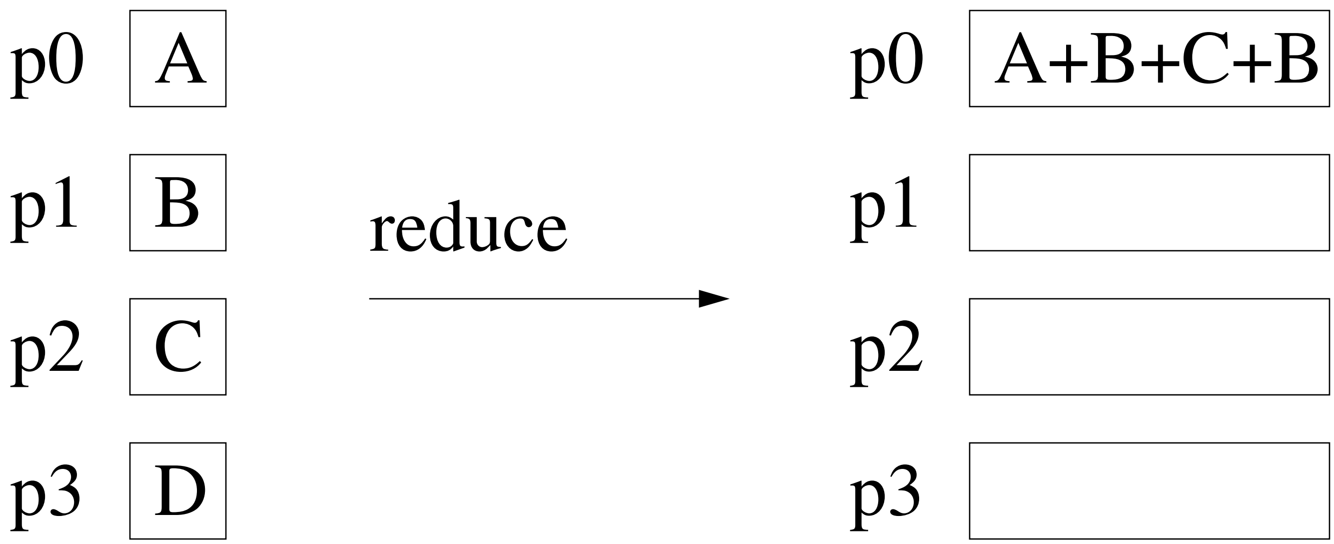


subroutine MPI_Bcast(array, count, datatype, root, comm, ierror)

<data type> array(*) ! data array to be broadcasted
integer count ! # of elements in the array
integer datatype ! MPI_REAL, MPI_INTEGER, MPI_BYTE.

integer root ! root processor on which data to be broadcarsted
integer comm ! communicator, the scope of broadcast
integer ierror ! return error message

Collective Operations



subroutine MPI_Reduce(sent_array, recv_array, count, datatype, op, root, comm, ierror)

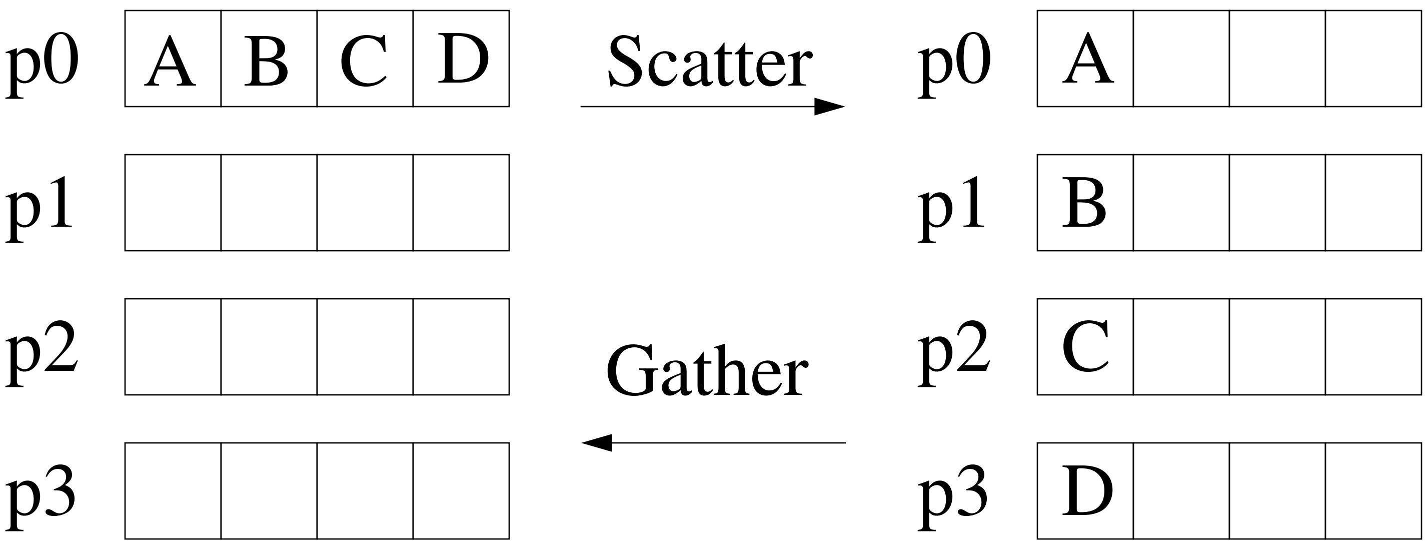
<data type> sent_array(*) ! data array on whch reduction operated
<data type> recv_array(*) ! data array containing reduction results

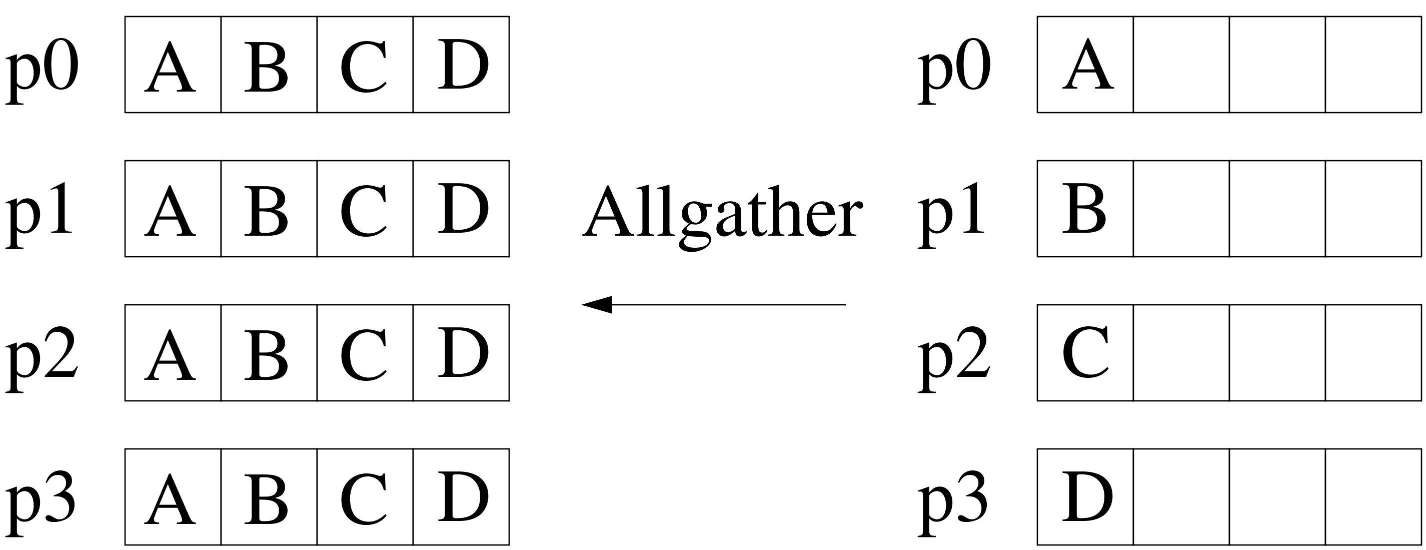
integer count ! # of elements in the array
integer datatype ! MPI_REAL, MPI_INTEGER, MPI_BYTE.

integer op ! operation, MPI_SUM, MPI_MAX, MPI_AND
integer root ! root processor on which data to be broadcarsted
integer comm ! communicator, the scope of broadcast
integer ierror ! return error message

```
C An example of MPI_Bcast and MPI_Reduce. Calculate the number pi=3.1416...

      implicit none
      include 'mpif.h'

      real PI15DT
      parameter      (PI15DT = 3.14159265358979)
      real tmp, pi, h, sum, x, f
      integer nIntv, myPE, numprocs, i, ierr

      f(x) = 4.e0 / (1.e0 + x*x) !statement function to integrate

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      print *, "myPE is ", myPE, " of total ", numprocs, " PEs"

      if ( myPE .eq. 0 ) then
        write(6,*) 'Enter the number of intervals: (0 quits)'
        read(5,*) nIntv
      endif

      call MPI_BCAST(nIntv,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

      if ( nIntv .LE. 0 ) stop ! nIntv must be > 0. Otherwise quit.

      h = 1.0e0/nIntv
      sum  = 0.0e0
      do i = myPE+1, nIntv, numprocs
        x = h * (i - 0.5e0)
        sum = sum + f(x)
      enddo
      sum = h* sum

      call MPI_REDUCE(sum,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     &    MPI_COMM_WORLD,ierr)

      if (myPE .eq. 0) write(6, 90) pi, pi-PI15DT
90    format('  pi =', F18.15, '  Error is: ', F18.15)

      call MPI_FINALIZE(ierr)
      end
```

Collective Operations





Processor groups and communicator in MPI

- Need
  - ~ group scope for collective communication
  - ~ define abstract communication space for modular programming, library, etc.
- Overlapped and non-overlapped communicators
- Manipulation of communicators

```
      program mpi_group_test
      implicit none
      include 'mpif.h'

      integer myPE,totPE,totPEcols,PEcol,PErow
      integer size1,size2,rank1,rank2,sum1,sum2
      integer COMM1,COMM2
      integer color,key
      integer ierr

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myPE, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, totPE, ierr )

c      write(6,*)' my_pe=', myPE, ' totPE=',totPE
      if(myPE .EQ. 0) write(6,*)' totPE=',totPE

      totPEcols = NINT(SQRT(totPE*1.0))
      PEcol = MOD(myPE, totPEcols)
      PErow = myPE / totPEcols

C ---------   split MPI_COMM_WORLD --------------
      color = PErow
      key = 0
      call MPI_COMM_SPLIT(MPI_COMM_WORLD, color,  key,
     &                                 COMM1, ierr)
      call MPI_ALLREDUCE(myPE, sum1, 1, MPI_INTEGER,
     &                                    MPI_SUM,  COMM1, ierr)

      call MPI_COMM_SIZE(COMM1, size1,ierr )
      call MPI_COMM_RANK(COMM1, rank1,ierr )
```

```
      if(myPE .EQ. 0) write(6,*)' COMM1: '
      write(6,100) myPE, COMM1, size1, rank1, sum1
100   format('PE=',i2,' COMM1=',i1,' size1=',i1,' rank1=',i1,' sum1=',
     &  i3)
      call flush(6)
      call MPI_BARRIER( MPI_COMM_WORLD, ierr )

C ---------  split MPI_COMM_WORLD ------------
      color = PEcol
      key = 0
      call MPI_COMM_SPLIT(MPI_COMM_WORLD, color,  key,
     &                                   COMM2, ierr)
      call MPI_ALLREDUCE(myPE, sum2, 1, MPI_INTEGER,
     &                                  MPI_SUM, COMM2, ierr)

      call MPI_COMM_SIZE(COMM2, size2, ierr )
      call MPI_COMM_RANK(COMM2,  rank2, ierr )
      if(myPE .EQ. 0) write(6,*)' COMM2: '
      write(6,200) myPE, COMM2, size2, rank2, sum2
200   format('PE=',i2,' COMM2=',i1,' size2=',i1,' rank2=',i1,' sum2=',
     &  i3)

      call MPI_FINALIZE(ierr)
      end
~
```

Output obtained by running the codes on totPE= 16


```
COMM1:
PE= 0   COMM1=2   size1=4   rank1=0    sum1=  6
PE= 1   COMM1=2   size1=4   rank1=1    sum1=  6
PE= 2   COMM1=2   size1=4   rank1=2    sum1=  6
PE= 3   COMM1=2   size1=4   rank1=3    sum1=  6

PE= 4   COMM1=2   size1=4   rank1=0    sum1= 22
PE= 5   COMM1=2   size1=4   rank1=1    sum1= 22
PE= 6   COMM1=2   size1=4   rank1=2    sum1= 22
PE= 7   COMM1=2   size1=4   rank1=3    sum1= 22

PE= 8    COMM1=2   size1=4   rank1=0   sum1= 38
PE= 9    COMM1=2   size1=4   rank1=1   sum1= 38
PE=10   COMM1=2   size1=4   rank1=2   sum1= 38
PE=11   COMM1=2   size1=4   rank1=3   sum1= 38

PE=12   COMM1=2   size1=4   rank1=0    sum1= 54
PE=13   COMM1=2   size1=4   rank1=1    sum1= 54
PE=14   COMM1=2   size1=4   rank1=2    sum1= 54
PE=15   COMM1=2   size1=4   rank1=3    sum1= 54

COMM2:
PE= 0    COMM2=3    size2=4    rank2=0    sum2= 24
PE= 4    COMM2=3    size2=4    rank2=1    sum2= 24
PE= 8    COMM2=3    size2=4    rank2=2    sum2= 24
PE=12   COMM2=3    size2=4    rank2=3    sum2= 24

PE= 1   COMM2=3    size2=4     rank2=0    sum2= 28
PE= 5   COMM2=3    size2=4     rank2=1    sum2= 28
PE= 9   COMM2=3    size2=4     rank2=2    sum2= 28
PE=13  COMM2=3    size2=4     rank2=3    sum2= 28

PE= 2    COMM2=3    size2=4    rank2=0    sum2= 32
PE= 6    COMM2=3    size2=4    rank2=1    sum2= 32
PE=10   COMM2=3    size2=4    rank2=2    sum2= 32
PE=14   COMM2=3    size2=4    rank2=3    sum2= 32

PE= 3    COMM2=3    size2=4    rank2=0    sum2= 36
PE= 7    COMM2=3    size2=4    rank2=1    sum2= 36
PE=11   COMM2=3    size2=4    rank2=2    sum2= 36
PE=15   COMM2=3    size2=4    rank2=3    sum2= 36
```

Processor groups and communicator in MPI

Define abstract communication space for modular programming, library, etc.

MPI_Send(array, count, datatype,
dest_pro, tag, **comm**, ierror)

MPI_Recv(array, count, datatype,
src_pro, tag, **comm**, status, ierror)

~ dest_pro/src_pro refers to comm

~ dest_pro = 2 means different logical proc on different communicator

subroutine task1234(a,b,..., **comm**)

~ **comm** defines an indenpendent communication space, i.e. dest_pro =1 etc..

~ Allocate 64 PEs, use 32 PEs for one task, and the rest 32 PEs for another task.

~ Use global variables or common block for a suit of subroutines or libraries

Processor groups and communicator in MPI

Manipulation of communicators

Most commonly used:

MPI_Comm_Split(in_comm, color, key, out_comm, ierror)
MPI_Comm_Rank(comm, rank, error)
MPI_Comm_Size(comm, size, ierror)

The rest are not necessary:

MPI_Comm_Compare(comm1, comm2, result, ierror)
MPI_Comm_DUP(in_comm, out_comm, ierror)
MPI_Comm_Create(in_comm, in_**group**, out_comm, ierror)

There are 9 MPI routines for manipulate group contexts.

A Summary of MPI routines

There are 130 MPI routines. The most commonly used are the following:

MPI_Init
MPI_Finalize

MPI_Comm_Rank
MPI_Comm_Size

MPI_Send
MPI_Recv / MPI_Irecv

MPI_Bcast
MPI_Reduce / MPI_Allreduce

MPI_Comm_Split

MPI_Barrier

MPI_Wtime -- wall-clock, in seconds
MPI_Type_Size -- byte size of the data object (sizeof in "C")

Most important: SEND and RECV

MPI Code Examples

To Compile on T3E:

```
f90 hello.f -o hello
```

To run on T3E (on 4 PEs) :

```
mpprun -n4 ./hello
```

All MPI codes are in /usr/local/pkg/usg/examples/ on T3E.
Type “cd $examples” and you are there.

Hello World: MPI/hello.f
A simple code intruducing 4 basic MPI routines.

PI: MPI/pi.f
Compute the number pi using many PEs collectively.

Group of processors: MPI/group.f
Use MPI_COMM_SPLIT to group PEs into different subgroups.

2D heat distribution: Heat1d/
Solve a 2D Laplace equation using 1D domain decomposition.

3D FFT: 3dFFT/
Real-to-complex and complex-to-real 3D FFFs