

# HIGH-PERFORMANCE COMPUTING IN APPLIED MATHEMATICS

MTH 410/510

## COMMUNICATIONS: MPI\_Send and MPI\_Recv

```
program main
include 'mpif.h'
integer rank, size, ierr

call MPI_INIT(ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

print*, 'Hello world! I am processor ', rank, ' of ', size

call MPI_FINALIZE(ierr)
stop
end
```

No communication among processors is involved.  
Message passing among processors is carried out by the MPI  
functions MPI\_Send and MPI\_Recv

## The structure of a message: data + envelope

- Data structure of the message:

buf = initial address of send buffer % actual data transmitted

count = number of entries to send % integer - how big is the data

datatype = datatype of each message entry % e.g., integer, real, ...

- The message envelope:

dest = rank of receiver % integer - where to send

source = the rank of sender % integer - who is the sender  
(MPI\_ANY\_SOURCE)

tag = message tag % integer used for sorting messages  
(MPI\_ANY\_TAG)

comm = communicator % the route used for communication. Must be matched in send/receive. E.g., MPI\_COMM\_WORLD

## C programming: MPI\_Send and MPI\_Recv

```
int MPI_Send(
    void* message,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm)
```

```
int MPI_Recv(
    void* message,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status* status)
```

Most common datatypes:

MPI datatype

MPI\_CHAR

MPI\_INT

MPI\_FLOAT

MPI\_DOUBLE

C datatype

signed char

signed int

float

double

## Fortran programming: MPI\_Send and MPI\_Recv

- MPI\_SEND(buf,count,datatype,dest,tag,comm,ierror)
- MPI\_RECV(buf,count,datatype,source,tag,comm,status,ierror)

buf = initial address of  
send/receive buffer

count = number of entries to  
send/receive

datatype = datatype of each  
message entry

dest = rank of destination

source = rank of source

tag = message tag

comm = communicator

ierror = return error code

### Most common datatypes

MPI\_INTEGER

MPI\_REAL

MPI\_DOUBLE\_PRECISION

MPI\_COMPLEX

MPI\_LOGICAL

MPI\_CHARACTER

## Receive a message from any source and/or with any tag (wildcards)

- Ignore source by using the `MPI_ANY_SOURCE` wildcard:  
call `MPI_RECV(buf,count,datatype,MPI_ANY_SOURCE,`  
& `tag,comm,status, ierror)`
- Ignore tag by using the `MPI_ANY_TAG` wildcard:  
call `MPI_RECV(buf,count,datatype,source,MPI_ANY_TAG,`  
& `comm,status, ierror)`
- Ignore tag and source:  
call `MPI_RECV(buf,count,datatype,MPI_ANY_SOURCE,`  
& `MPI_ANY_TAG, comm,status, ierror)`
- In general, avoid using wildcards.

**MPI\_SEND** and **MPI\_RECV** are blocking communication routines.

- **MPI\_SEND is a blocking routine:**

The call to **MPI\_SEND** does not return control to the calling program or routine, until the buffer containing the data to be copied unto the receiving process can be safely overwritten.

Assures that the message being sent is not "corrupted" before the sending is complete.

- **MPI\_RECV is a blocking routine:**

The call does not return control to the calling program until the data to be received has in fact been received.

- **Blocking functions may cause the system to "hung":**

A "Hung" condition occurs when one or more processes reach a call to an MPI block-receive routine, but the message never arrives. The process will wait indefinitely and no error message will be generated.

- **Must be carefully programmed!**

## I do not hang

An example of an message exchange that will not "hang":

```
if      (myid == 0) then
    call mpi_send(a,1,mpi_real,1>tag,MPI_COMM_WORLD,ierr)
    call mpi_recv(b,1,mpi_real,1>tag,MPI_COMM_WORLD,
&      status,ierr)
elseif (myid == 1) then
    call mpi_recv(a,1,mpi_real,0>tag,MPI_COMM_WORLD,
&      status,ierr)
    call mpi_send(b,1,mpi_real,0>tag,MPI_COMM_WORLD,ierr)
end if
```

## I hang (a lot!)

An example of an message exchange that **will** "hang":

```
if      (myid == 0) then
    call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
    call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ierr)
&      status,ierr)
elseif (myid == 1) then
    call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
&      status,ierr)
    call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ierr)
end if
```

## Exercise 1: Example of communication using send/receive

Consider the following task:

- process 0 reads input  $x$  and sends it to all other processes
- each process receives  $x$ , modifies it, then sends it back to process 0
- process 0 receives updated value  $y$  from all other processes and prints it

## Initialization

```
program example1
include 'mpif.h'
integer my_rank,np, ierror
integer i, dest, source
integer tag
integer status(MPI_STATUS_SIZE)
real x, y
call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierror)
tag = 0
```

Process 0 reads x and sends it to all other processes

```
if (my_rank == 0) then
```

```
!—process 0 reads and sends data to all other processes
```

```
  print*, 'input x'
```

```
  read(*,*) x
```

```
  do dest=1,np-1
```

```
    call MPI_SEND(x,1,MPI_REAL,dest>tag,
```

```
&      MPI_COMM_WORLD,ierror)
```

```
  end do
```

```
else
```

```
! each process receives x from process 0
```

```
  source = 0
```

```
  call MPI_RECV(x,1,MPI_REAL,source>tag,
```

```
&      MPI_COMM_WORLD,status,ierror)
```

```
end if
```

Each process modifies  $x$  and sends it to process 0

```
if (my_rank == 0) then
  do source=1,np-1
    call MPI_RECV(y,1,MPI_REAL,source,tag,
&           MPI_COMM_WORLD,status,ierror)
    print*,'Process ',my_rank,' received y = ', y,
&           'from process ',source
  end do
else
  print*,'I am process ', my_rank,' I received x = ', x
  x = x*my_rank
  dest = 0
  call MPI_SEND(x,1,MPI_REAL,dest,tag,
&           MPI_COMM_WORLD,ierror)
end if
call MPI_FINALIZE(ierr)
stop
end
```

## Compile and execute

```
mpif90 -o example1 example1.f
```

```
mpirun -np 4 example1
```

input x

5

I am process 3 I received x = 5.000000

I am process 1 I received x = 5.000000

I am process 2 I received x = 5.000000

Process 0 received y = 5.000000 from process 1

Process 0 received y = 10.00000 from process 2

Process 0 received y = 15.00000 from process 3

## The MPI broadcast function MPI\_Bcast

- A **broadcast** is a **collective communication** in which a single process sends the same data to every process in the communicator.
- The root process broadcasts the data in buffer to all the processes in the communicator.
- All processes must call MPI\_BCAST with the same root value.

C:            MPI\_Bcast(buffer,count,datatype, root,comm)

Fortran: MPI\_BCAST(buffer,count,datatype, root,comm,ierror)

buffer = initial address of buffer

count = number of entries in buffer (integer)

datatype = datatype of buffer

root = rank of broadcasting process (integer)

comm = communicator

ierror =return error code (integer)



```
call MPI_BCAST(x,1,MPI_REAL,0,MPI_COMM_WORLD,ierr)
if (my_rank .ne. 0) then
    print*,'I am process ', my_rank,' I received x = ', x
    x = x*my_rank
end if
```

! sum all x values in variable y on process 0

```
call MPI_Reduce(x, y, 1, MPI_REAL, MPI_SUM, 0,
& MPI_COMM_WORLD,ierr)
call MPI_Reduce(x, z, 1, MPI_REAL, MPI_PROD, 1,
& MPI_COMM_WORLD,ierr)
if (my_rank == 0) then
    print*,'I am process ',my_rank,' sum of x is = ', y
    print*,'I am process ',my_rank,' prod of x is = ', z
elseif (my_rank == 1) then
    print*,'I am process ',my_rank,' prod of x is = ', z
    print*,'I am process ',my_rank,' sum of x is = ', y
! this will output some garbage. Why?
end if
```

## Compile and execute

```
mpif90 -o example1_bcast example1_bcast.f
```

```
mpirun -np 5 example1
```

input x

1

I am process 1 I received x = 1.000000

I am process 3 I received x = 1.000000

I am process 2 I received x = 1.000000

I am process 4 I received x = 1.000000

I am process 0 sum of x is = 11.00000

I am process 0 prod of x is = 1.4012985E-45

I am process 1 prod of x is = 24.00000

I am process 1 sum of x is = -1.999474

## The MPI scatter function MPI\_Scatter

C: MPI\_Scatter(send\_data, send\_count, send\_type,  
recv\_data, recv\_count, recv\_type, root, comm)

Fortran: MPI\_SCATTER(send\_data, send\_count, send\_type,  
recv\_data, recv\_count, recv\_type, root, comm, ierror)

- MPI\_Scatter splits the data referenced by *send\_data* on the process with rank *root* into *nproc* segments, each of which consists of *send\_count* of type *send\_type*.
- The first segment is sent to process 0, the second to process 1, etc ...
- In most cases *send\_count* and *recv\_count* are the same and *send\_type* and *recv\_type* are the same.
- All parameters *root* and *comm* must be the same in all processes

The Dot Product  $a = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x(i) * y(i)$

Serial function dot product

```
real function sdot(n,x,y)
integer n, i
real x(n), y(n)
sdot = 0.0
do i=1,n
    sdot = sdot + x(i)*y(i)
end do
return
end
```

The Dot Product  $a = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x(i) * y(i)$

Parallel dot product

...

```
call MPI_SCATTER(x,local_n,MPI_REAL,loc_x,local_n,MPI_REAL,0,  
& MPI_COMM_WORLD,ierr)
```

```
call MPI_SCATTER(y,local_n,MPI_REAL,loc_y,local_n,MPI_REAL,0,  
& MPI_COMM_WORLD,ierr)
```

```
local_a = sdot(local_n,loc_x,loc_y)
```

! sum all loc\_a values in variable a on process 0

```
call MPI_REDUCE(local_a, a, 1, MPI_REAL, MPI_SUM, 0,  
& MPI_COMM_WORLD,ierr)
```

...

## MPI functions so far

- 1 MPI\_INIT(ierr)
- 2 MPI\_COMM\_RANK(MPI\_COMM\_WORLD, rank, ierr)
- 3 MPI\_COMM\_SIZE(MPI\_COMM\_WORLD, size, ierr)
- 4 MPI\_SEND(buf,count,datatype,dest,tag,comm,ierror)
- 5 MPI\_RECV(buf,count,datatype,source,tag,comm,status,ierror)
- 6 MPI\_BCAST(buffer,count,datatype, root,comm,ierror)
- 7 MPI\_SCATTER(send\_data,send\_count,send\_type,  
recv\_data,recv\_count,recv\_type, root, comm, ierror)
- 8 MPI\_REDUCE(operand,result, count, datatype,  
operator, root, comm, ierror)
- 9 MPI\_FINALIZE(ierr)

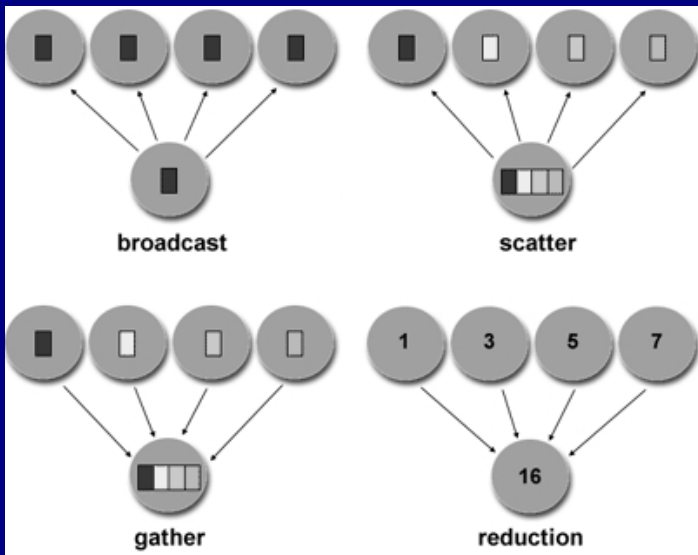
## MPI gather function MPI\_Gather

C: MPI\_Gather(send\_data, send\_count, send\_type, recv\_data,  
recv\_count, recv\_type, root, comm)

Fortran: MPI\_GATHER(send\_data, send\_count, send\_type, recv\_data,  
recv\_count, recv\_type, root, comm, ierror)

- MPI\_Gather collects the data referenced by *send\_data* from each process in the communicator *comm* and stores the data *in process rank order* on the process with rank *root* in the memory referenced by *recv\_data*.
- Data from process 0 is followed by the data from process 1, followed by the data from process 2, etc ...
- Data referenced by *send\_data* on each process consists of *send\_count* elements, each of which has type *send\_type*

## Collective communication functions



...

```
if (my_rank == 0) then
  do i=1,n
    x(i) = rand()
  end do
end if
local_n = n/np
call MPI_SCATTER(x,local_n,MPI_REAL,loc_x,local_n,MPI_REAL,0,
& MPI_COMM_WORLD,ierror)
loc_x = loc_x*my_rank
call MPI_GATHER(loc_x,local_n,MPI_REAL,y,local_n,MPI_REAL,0,
& MPI_COMM_WORLD,ierror)
if (my_rank == 0) then
  do i=1,n
    print*, 'x(',i,') = ', x(i), 'y(',i,')= ',y(i)
  end do
end if
```

...

## The extended MPI\_Reduce function MPI\_Allreduce

- MPI\_Reduce(operand,result, count, datatype, operator, root, comm)
- MPI\_Allreduce(operand,result, count, datatype, operator, comm)

After the function returns all the processes in the communicator will have the result stored in the memory referenced by *result*.

## The extended MPI\_Gather function MPI\_Allgather

- MPI\_Gather(send\_data, send\_count, send\_type, recv\_data, recv\_count, recv\_type, root, comm)
- MPI\_Allgather(send\_data, send\_count, send\_type, recv\_data, recv\_count, recv\_type, comm)

This has the effect of gathering the contents of each process's *send\_data* into each process's *recv\_data*

## Example 1 revisited

```
call MPI_BCAST(x,1,MPI_REAL,0,MPI_COMM_WORLD,ierr)
if (my_rank .ne. 0) x = x*my_rank
```

! sum all x values in variable y in all processes

```
call MPI_ALLREDUCE(x, y, 1, MPI_REAL, MPI_SUM,
& MPI_COMM_WORLD,ierr)
```

! multiply all x values in variable z in all processes

```
call MPI_ALLREDUCE(x, z, 1, MPI_REAL, MPI_PROD,
& MPI_COMM_WORLD,ierr)
```

```
if (my_rank == 0) then
```

```
  print*,'I am process ',my_rank,' sum of x is = ', y
```

```
  print*,'I am process ',my_rank,' prod of x is = ', z
```

```
elseif (my_rank == 1) then
```

```
  print*,'I am process ',my_rank,' prod of x is = ', z
```

```
  print*,'I am process ',my_rank,' sum of x is = ', y
```

! this will NOT output garbage. Why?

```
end if
```

## Numerical Integration: Parallel implementation of quadrature methods

Consider the problem of evaluating

$$\int_a^b f(x) dx$$

using composite trapezoidal rule:

- divide the interval  $[a, b]$  in  $n$  segments

$$h = (b - a)/n, \quad x_i = a + i * h, \quad i = 0, 1, \dots, n$$

- approximation formula

$$\int_a^b f(x) dx \approx h [f(x_0)/2 + f(x_1) + \dots + f(x_{n-1}) + f(x_n)/2]$$

Serial algorithm :

$I = \text{serialtrap}(a, b, n)$

$h = (b - a)/n$

$I = f(a)/2 + f(b)/2$

for  $i = 1, n - 1$

$x = a + i * h$

$I = I + f(x)$

end

$I = h * I$

Parallel algorithm:

$I = \text{paralleltrap}(a, b, n)$

$local\_n = n/p$

$local\_a = a + my\_rank * local\_n * h$

$local\_b = local\_a + local\_n * h$

$local\_I = \text{serialtrap}(local\_a, local\_b, n)$

$MPI\_Reduce(local\_I, I, \dots, MPI\_SUM \dots)$

## Taking Timings: MPI\_Barrier(comm), MPI\_Wtime(), MPI\_Wtick()

- MPI\_Barrier is used to synchronize the processes
- Each process in the communicator is blocked until every process has called the MPI\_Barrier.
- MPI\_Wtime returns a double precision value that represents the number of seconds elapsed since some point in the past.
- The precision of MPI\_Wtime may be found by calling MPI\_Wtick

```
...
call MPI_BARRIER(MPI_COMM_WORLD,ierror)
start = MPI_WTIME()
do i=1,20000000
    x = rand()
end do
call MPI_BARRIER(MPI_COMM_WORLD,ierror)
finish = MPI_WTIME()
resolution = MPI_WTICK()
if (my_rank .eq. 0) then
    print *,'elapsed wall-clock time =',finish-start,'resolution=',resolution
end if
```

## Homework Assignment:

- 1 (10 points) Write a program for the following ring-type communication:
  - process 0 reads  $x$  from keyboard and sends it to process 1.
  - process 1 increments  $x$  by its rank (=1) and sends it to process 2.
  - in general process  $i$  receives  $x$  from process  $i - 1$ , increments  $x$  by  $i$ , then sends it to process  $i + 1$
  - process 0 receives  $x$  from process  $np - 1$  and prints the value on the screen.
- 2 (15 points) Implement in parallel the composite Simpson's rule:  
 $h = (b - a)/n$ ,  $n$  even number  
 $x_i = a + i * h, i = 0, 1, \dots, n$

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[ f(a) + 2 \sum_{i=1}^{(n/2)-1} f(x_{2i}) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + f(b) \right]$$

- Test run: evaluate  $\pi$  using  $n=10000$  on 10 processors