HPSC – Lecture 19

Outline:

- Review MPI, reduce and bcast
- MPI send and receive
- Master–Worker paradigm

References:

- `$/codes/mpi`
- MPI Standard
- OpenMPI

MPI — Simple example

```
program test1
    use mpi
    implicit none
    integer :: ierr, numprocs, proc_num,

    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, numprocs, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, proc_num, ierr)

    print *, 'Hello from Process ', proc_num, &
             ' of ', numprocs, ' processes'

    call mpi_finalize(ierr)

end program test1
```

Always need to: `use mpi`,

Start with `mpi_init`,

End with `mpi_finalize`.

Compiling and running MPI code (Fortran)

Try this test:

```
$ cd $UWHPSC/codes/mpi
$ mpif90 test1.f90
$ mpiexec -n 4 a.out
```

You should see output like:

```
Hello from Process number  1  of  4  processes
Hello from Process number  3  of  4  processes
Hello from Process number  0  of  4  processes
Hello from Process number  2  of  4  processes
```

Note: Number of processors is specified with `mpiexec`.

MPI Communicators

All communication takes place in groups of processes.

Communication takes place in some context.

A group and a context are combined in a communicator.

`MPI_COMM_WORLD` is a communicator provided by default that includes all processors.

MPI Communicators

All communication takes place in groups of processes.

Communication takes place in some context.

A group and a context are combined in a communicator.

`MPI_COMM_WORLD` is a communicator provided by default that includes all processors.

`MPI_COMM_SIZE(comm, numprocs, ierr)` returns the number of processors in communicator `comm`.

`MPI_COMM_RANK(comm, proc_num, ierr)` returns the rank of this processor in communicator `comm`.

mpi module

The mpi module includes:

Subroutines such as `mpi_init, mpi_comm_size, mpi_comm_rank, ...`

Global variables such as

`MPI_COMM_WORLD`: a communicator,
`MPI_INTEGER`: used to specify the type of data being sent
`MPI_SUM`: used to specify a type of reduction

Remember: Fortran is case insensitive:
`mpi_init` is the same as `MPI_INIT`.

MPI functions

There are 125 MPI functions.

Can write many program with these 8:

- `MPI_INIT(ierr)` Initialize
- `MPI_FINALIZE(ierr)` Finalize
- `MPI_COMM_SIZE(...)` Number of processors
- `MPI_COMM_RANK(...)` Rank of this processor
- `MPI_SEND(...)` Send a message
- `MPI_RCV(...)` Receive a message
- `MPI_BCAST(...)` Broadcast to other processors
- `MPI_REDUCE(...)` Reduction operation

MPI Reduce

Examples: Compute $l\,x\,l_\infty = \max_i |x_i|$ for a distributed vector:
(each process has some subset of *x* elements)

```
xnorm_proc = 0.d0
! set istart and iend for each process
do i=istart,iend
    xnorm_proc = max(xnorm_proc, abs(x(i)))
    enddo

call MPI_REDUCE(xnorm_proc, xnorm, 1, &
          MPI_DOUBLE_PRECISION, MPI_MAX, 0, &
          MPI_COMM_WORLD,ierr)

if (proc_num == 0) print "norm of x = ", xnorm
```

Processors do not exit from `MPI_REDUCE` until all have called the subroutine.

Normalize the vector *x*: Replace *x* by *x*/|*x*|∞

```
! compute xnorm_proc on each process as before.

call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                MPI_DOUBLE_PRECISION,MPI_MAX, 0, &
                MPI_COMM_WORLD,ierr)
! only Process 0 has the value of xnorm

call MPI_BCAST(xnorm, 1, &
                MPI_DOUBLE_PRECISION, 0, &
                MPI_COMM_WORLD,ierr)
! now every process has the value of xnorm

do i=istart,iend
    x(i) = x(i) / xnorm
    enddo
```

MPI AllReduce

To make a reduction available to *all* processes:

```
call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                MPI_DOUBLE_PRECISION,MPI_MAX, 0, &
                MPI_COMM_WORLD,ierr)
! only Process 0 has the value of xnorm

call MPI_BCAST(xnorm, 1, &
                MPI_DOUBLE_PRECISION, 0, &
                MPI_COMM_WORLD,ierr)
```

One-step alternative: simpler and perhaps more efficient...

```
call MPI_ALLREDUCE(xnorm_proc, xnorm, 1, &
                MPI_DOUBLE_PRECISION,MPI_MAX, &
                MPI_COMM_WORLD,ierr)
```

Remember — no shared memory

Suppose all of vector x is stored on memory of Process 0,

We want to normalize x (using more than one processor),

and replace x by normalized version in memory of Process 0.

Remember — no shared memory

Suppose all of vector *x* is stored on memory of Process 0,

We want to normalize *x* (using more than one processor),

and replace *x* by normalized version in memory of Process 0.

We would have to:

- Send parts of *x* to other processes,
- Compute `xnorm_proc` on each process,
- Use `MPI_ALLREDUCE` to combine into `xnorm`
  and broadcast to all processes,
- Normalize part of *x* on each process,
- Send each part of normalized *x* back to Process 0.

Remember — no shared memory

Suppose all of vector *x* is stored on memory of Process 0,

We want to normalize *x* (using more than one processor),

and replace *x* by normalized version in memory of Process 0.

We would have to:

- Send parts of *x* to other processes,
- Compute `xnorm_proc` on each process,
- Use `MPI_ALLREDUCE` to combine into `xnorm`
  and broadcast to all processes,
- Normalize part of *x* on each process,
- Send each part of normalized *x* back to Process 0.

Communication cost will probably make this much slower than just normalizing all of *x* on Process 0!

Remember — no shared memory

Might be worthwhile if much more work is required for each element of *x*.

Suppose all of vector *x* is stored on memory of Process 0,

Want to solve an expensive differential equation
with different initial conditions given by elements of *x*,

and then collect all results on Process 0.

Remember — no shared memory

Might be worthwhile if much more work is required for each element of *x*.

Suppose all of vector *x* is stored on memory of Process 0,

Want to solve an expensive differential equation
with different initial conditions given by elements of *x*,

and then collect all results on Process 0.

Master–Worker paradigm:

- Process 0 sends different chunks of *x* to Process 1, 2, . . .
- Each process grinds away to solve differential equations
- Each process sends results back to Process 0.

MPI Send and Receive

`MPI_BCAST` sends from one process to all processes.

Often want to send selectively from Process *i* to Process *j*.

Use `MPI_SEND` and `MPI_RECV`.

MPI Send and Receive

`MPI_BCAST` sends from one process to all processes.

Often want to send selectively from Process *i* to Process *j*.

Use `MPI_SEND` and `MPI_RECV`.

Need a way to tag messages so they can be identified.

The parameter `tag` is an integer that can be matched to identify a message.

Tag can also be used to provide information about what is being sent, for example if a Master process sends rows of a matrix to other processes, the `tag` might be the row number.

MPI Send

Send value(s) from this Process to Process `dest`.

General form:

```
call MPI_SEND(start, count, &
              datatype, dest, &
              tag, comm, ierr)
```

where:

- `start`: starting address (variable, array element)
- `count`: number of elements to send
- `datatype`: type of each element
- `dest`: destination process
- `tag`: identifier tag (integer between 0 and 32767)
- `comm`: communicator

MPI Receive

Receive value(s) from Process `source` with label `tag`.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- `source`: source process
- `tag`: identifier tag (integer between 0 and 32767)
- `comm`: communicator
- `status`: integer array of length `MPI_STATUS_SIZE`.

MPI Receive

Receive value(s) from Process `source` with label `tag`.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- `source`: source process
- `tag`: identifier tag (integer between 0 and 32767)
- `comm`: communicator
- `status`: integer array of length `MPI_STATUS_SIZE`.

source could be `MPI_ANY_SOURCE` to match any source.

tag could be `MPI_ANY_TAG` to match any tag.

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print `j = 55`

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print `j = 55`

The tag is 21. (Arbitrary integer between 0 and 32767)

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print `j = 55`

The tag is 21. (Arbitrary integer between 0 and 32767)

Blocking Receive: Processor 3 won't return from `MPI_RECV` until message is received.

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print `j = 55`

The tag is 21. (Arbitrary integer between 0 and 32767)

Blocking Receive: Processor 3 won't return from `MPI_RECV` until message is received.

Run-time error if `num_procs <= 4` (Procs are 0,1,2,3)

Send/Receive example

Pass value of `i` from Processor 0 to 1 to 2 ... to `num_procs-1`

```
if (proc_num == 0) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 1, 21, &
                  MPI_COMM_WORLD, ierr)
    endif

else if (proc_num < num_procs - 1) then
    call MPI_RECV(i, 1, MPI_INTEGER, proc_num-1, 21, &
                  MPI_COMM_WORLD, status, ierr)
    call MPI_SEND(i, 1, MPI_INTEGER, proc_num+1, 21, &
                  MPI_COMM_WORLD, ierr)

else if (proc_num == num_procs - 1) then
    call MPI_RECV(i, 1, MPI_INTEGER, proc_num-1, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "i = ", i
    endif
```

MPI Receive

Receive value(s) from Process `source` with label `tag`.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- `source`: source process
- `tag`: identifier tag (integer between 0 and 32767)
- `comm`: communicator
- `status`: integer array of length `MPI_STATUS_SIZE`.

MPI Receive

Receive value(s) from Process `source` with label `tag`.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- `source`: source process
- `tag`: identifier tag (integer between 0 and 32767)
- `comm`: communicator
- `status`: integer array of length `MPI_STATUS_SIZE`.

source could be `MPI_ANY_SOURCE` to match any source.

tag could be `MPI_ANY_TAG` to match any tag.

MPI Receive — `status` argument

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

Elements of the `status` array give additional useful information about the message received.

In particular,

`status(MPI_SOURCE)` is the source of the message,
May be needed if `source = MPI_ANY_SOURCE`.

`status(MPI_TAG)` is the tag of the message received,
May be needed if `tag = MPI_ANY_TAG`.

Another Send/Receive example

Master (Processor 0) sends *j*th column to Worker Processor *j*, gets back 1-norm to store in `anorm(j)`, *j* = 1, . . . , `ncols`

```
! code for Master (Processor 0):
if (proc_num == 0) then
  do j=1,ncols
    call MPI_SEND(a(1,j), nrows, MPI_DOUBLE_PRECISION,&
                  j, j, MPI_COMM_WORLD, ierr)
    enddo

  do j=1,ncols
    call MPI_RECV(colnorm, 1, MPI_DOUBLE_PRECISION, &
                  MPI_ANY_SOURCE, MPI_ANY_TAG, &
                  MPI_COMM_WORLD, status, ierr)
    jj = status(MPI_TAG)
    anorm(jj) = colnorm
    enddo
  endif
```

Note: Master may receive back in any order!

`MPI_ANY_SOURCE` will match first to arrive.

The tag is used to tell which column's norm has arrived (`jj`).

Send and Receive example — worker code

Master (Processor 0) sends *j*th column to Worker Processor *j*, gets back 1-norm to store in `anorm(j)`, *j* = 1, . . . , `ncols`

```
! code for Workers (Processors 1, 2, ...):

if (proc_num /= 0) then

   call MPI_RECV(colvect, nrows, MPI_DOUBLE_PRECISION,&
                    0, MPI_ANY_TAG, &
                    MPI_COMM_WORLD, status, ierr)

   j = status(MPI_TAG)     ! this is the column number
                           ! (should agree with proc_num)

   colnorm = 0.d0
   do i=1,nrows
       colnorm = colnorm + abs(colvect(i))
       enddo

   call MPI_SEND(colnorm, 1, MPI_DOUBLE_PRECISION, &
                  0, j, MPI_COMM_WORLD, ierr)

   endif
```

Note: Sends back to Process 0 with tag *j*.

Send may be blocking

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(j, 1, MPI_INTEGER, 3, 22, &
                  MPI_COMM_WORLD, status, ierr)
    endif

if (proc_num == 3) then
    j = 66
    call MPI_SEND(j, 1, MPI_INTEGER, 4, 22, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(i, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    endif
```

Both processors *might* get stuck in `MPI_SEND`!

May depend on size of data and send buffer.

Blocking send: `MPI_SSEND`. See documentation

Send may be blocking

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(j, 1, MPI_INTEGER, 3, 22, &
                  MPI_COMM_WORLD, status, ierr)
    endif

if (proc_num == 3) then
    j = 66
    call MPI_SEND(j, 1, MPI_INTEGER, 4, 22, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(i, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    endif
```

Both processors *might* get stuck in `MPI_SEND`!

May depend on size of data and send buffer.

Blocking send: `MPI_SSEND`. See documentation

There are also non-blocking sends and receives:

`MPI_ISEND, MPI_IRECV`

Non-blocking receive

```
call MPI_IRECV(start, count, datatype, &
     source, tag, comm, request, ierror)
```

Additional argument: `request`.

Program continues after initiating receive,

Can later check if it has finished with

```
call MPI_TEST(request, flag, status, ierror)
```

`flag` is logical output variable.

Or can later wait for it to finish with

```
call MPI_WAIT(request, status, ierror)
```