Introduction to MPI Programming – Part 1
Outline

• Introduction – what is MPI and why MPI
• MPI program structure
• Point-to-point communication
Memory system models

• Different ways of sharing data among processors
  – Distributed Memory
  – Shared Memory
  – Other memory models
    • Hybrid model
    • PGAS (Partitioned Global Address Space)
Message Passing

• Context: distributed memory parallel computers
  – Each processor has its own memory space and cannot access the memory of other processors
  – Any data to be shared must be explicitly transferred from one to another
Distributed memory model

• Each process has its own address space
  – Data is local to each process
• Data sharing achieved via explicit message passing (through network)
• Example: MPI (Message Passing Interface)
Shared memory model

• All threads can access the global address space
• Data sharing achieved via writing to/reading from the same memory location
• Example: OpenMP
Message Passing Interface

• MPI defines a standard API for message passing
  – What’s in the standard
    • The syntax and semantics of a core set of functions
  – What’s not in the standard
    • Implementation details
    • Runtime details (how many processes the code will run with etc.)

• MPI provides C/C++ and Fortran bindings
Why MPI?

• Standardized
  – With efforts to keep it evolving (MPI 3.0 draft came out in 2010)

• Portability
  – MPI implementations are available on almost all platforms

• Scalability
  – In the sense that it is not limited by the number of processors that can access the same memory space

• Popularity
  – Many packages are based on MPI
When NOT to use MPI

• Not suitable for small-scale loop level parallelization
  – Shared memory parallelism and accelerator are better
MPI Functions

• Point-to-point communication functions
  – Message transfer from one process to another

• Collective communication functions
  – Message transfer involving all processes in a communicator

• Environment management functions
  – Initialization and termination
  – Process group and topology
program hello
...
includ "mpif.h"
integer :: nprocs,myid,ierr
...
call mpi_init(ierr)
...
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myid,ierr)
Write(*,'("There are",I3," processes")') nprocs
write(*,'("Process",I3," says Hello World!")') myid
...
call mpi_finalize(ierr)
...
MPI Program Structure

```fortran
program hello
  ...
  include "mpif.h"
  integer :: nprocs, myid, ierr
  ...
  call mpi_init(ierr)
  ...
  call mpi_comm_size(mpi_comm_world, nprocs, ierr)
  call mpi_comm_rank(mpi_comm_world, myid, ierr)
  write(*,'("There are",I3," processes")') nprocs
  write(*,'("Process",I3," says Hello World!")') myid
  ...
  call mpi_finalize(ierr)
  ...
end program hello
```

Header file

- Initialization
- Computation and communication
- Termination

```
[lyan1@qb563 ex]$ mpirun -np 4 ./a.out
There are 4 processes.
There are 4 processes.
There are 4 processes.
There are 4 processes.
Process 3 says Hello World!
Process 1 says Hello World!
Process 0 says Hello World!
Process 2 says Hello World!
```
C vs. Fortran

- **Header file**
  - C: `mpi.h`
  - Fortran: `mpif.h`

- **Function names**
  - C: `MPI_Some_Function`
  - Fortran: `mpi_some_function` *(not case sensitive)*

- **Error handles**
  - C returns the error value, while Fortran passes it as an argument
    - C: `int err = MPI_Some_Function(arg1, arg2, ..., argN)`
    - Fortran: `call mpi_some_function(arg1, arg2, ..., argN, ierr)`
Initialization and Termination

• Initialization
  – Must be called before any other MPI calls
  – C: MPI_Init()
  – Fortran: MPI_INIT(ierr)

• Termination
  – Clean up data structures, terminate incomplete calls etc.
  – C: MPI_Finalize()
  – Fortran: MPI_FINALIZE(ierr)
Communicators (1)

• A communicator is an identifier associated with a group of processes
  – Can be regarded as an ordered list of processes
  – Each process has a unique rank, which starts from 0 (root)
  – It is the context of MPI communicators and operations
    • When a function is called to send data to all processes, MPI needs to understand what “all” means
Communicators (2)

• MPI_COMM_WORLD: the default communicator that contains all processes running the MPI program
• There can be many communicators
• A process can belong to multiple communicators
  – The rank is usually different
Getting Communicator Information

• Get the rank of a communicator
  – C: MPI_Comm_Rank(MPI_Comm comm, int *rank)
  – Fortran: MPI_COMM_RANK(COMM,RANK,ERR)

• Get the size in a communicator
  – C: MPI_Comm_Size(MPI_Comm comm, int *size)
  – Fortran: MPI_COMM_SIZE(COMM,SIZE,ERR)
Compiling and Running MPI Programs

• Not a part of the standard
  – Could vary from platform to platform
  – Or even from implementation to implementation on the same platform

• On Super Mike 2 and LONI Linux systems:
  – Compile
    • C: `mpicc -o <executable name> <source file>`
    • Fortran: `mpif90 -o <executable name> <source file>`
  – Run
    • `mpirun -machinefile $PBS_NODEFILE -np <number of procs> <executable name> <input parameters>`
Exercise 1a: Process Color

- Write a MPI program where
  - Processes with odd rank print to screen “Process x has the color green”
  - Processes with even rank print to screen “Process x has the color red”
Exercise 1b: Matrix Multiplication version 1

• Goal: Distribute the work load among processes in 1-d manner
  – Each process initializes its own copy of A and B
  – Then processes part of the workload
    • Need to determine how to decompose (which process deals which rows or columns)
    • Assume that the dimension of A and B is a multiple of the number of processes (need to check this in the program)
  – Validate the result at the end
Exercise 1c: Laplace Solver version 0

• Goal: Distribute the work load among processes in 1-d manner
  – Find out the size of sub-matrix for each process
  – Let each process report which part of the domain it will work on, e.g. “Process x will process column (row) x through column (row) y.”
• Row-wise (C) or column-wise (Fortran)
Point-to-point Communication

• Communication between a pair of processes, so two functions calls are required
  – The sending process calls the MPI_SEND function
    • C: int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
    • Fortran: MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
  – The receiving process calls the MPI_RECV function
    • C: int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status);
    • Fortran: MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS, IERR)

• The function arguments characterize the message being transferred
A MPI message consists of two parts

- Message body
  - Buffer: starting location in memory for outgoing data (send) or incoming data (receive)
  - Data type: type of data to be sent or received
  - Count: number of items of type datatype to be sent or received

- Message envelope
  - Destination (source): rank of the destination (source) of the message
  - Tag: what MPI uses to match messages between processes
  - Communicator

The `status` argument contains information on the message that is received

- Only for MPI_RECV
Example: Gathering Array Data

- Goal: gather some array data from each process and place it in the memory of the root process
Example: Gathering Array Data

```fortran
... integer, allocatable :: array(:)

! Initialize MPI
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
call mpi_comm_rank(mpi_comm_world, myid, ierr)

! Initialize the array
allocate(array(2*nprocs))
array(1) = 2*myid
array(2) = 2*myid+1

! Send data to the root process
if (myid.eq.0) then
    do i=1,nprocs-1
        call mpi_recv(array(2*i+1), 2, mpi_integer, i, i, status, ierr)
    enddo
    write(*,*) "The content of the array:", array
else
    call mpi_send(array, 2, mpi_integer, 0, myid, ierr)
endif
```
Example: Gathering Array Data

```
...  
integer, allocatable :: array(:)
! Initialize MPI
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
call mpi_comm_rank(mpi_comm_world, myid, ierr)
! Initialize the array
allocate(array(2*nprocs))
array(1) = 2*myid
array(2) = 2*myid + 1
! Send data to the root process
if (myid.eq.0) then
  do i = 1, nprocs - 1
    call mpi_recv(array(2*i+1), 2, mpi_integer, i, i, status, ierr)
  enddo
  write(*,*) "The content of the array:"
  write(*,*) array
else
  call mpi_send(array, 2, mpi_integer, 0, myid, ierr)
endif

[lyan1@qb563 ex]$ mpirun -np 4 ./a.out
The content of the array:
    0        1        2        3        4        5
    6        7
    do i = 1, nprocs - 1
      call mpi_recv(array(2*i+1), 2, mpi_integer, i, i, status, ierr)
    enddo
    write(*,*) "The content of the array:"
    write(*,*) array
else
  call mpi_send(array, 2, mpi_integer, 0, myid, ierr)
endif
```
Blocking Operations

• MPI_SEND and MPI_RECV are blocking operations
  – They will not return from the function call until the communication is completed
  – When a blocking send returns, the send buffer can be safely overwritten
  – When a blocking receive returns, the data has been received and is ready to use
Deadlock (1)

- Deadlock occurs when both processes awaits the other to make progress

```c
// Exchange data between two processes
If (process 0)
    Receive data from process 1
    Send data to process 1
If (process 1)
    Receive data from process 0
    Send data to process 0
```

This is a guaranteed deadlock because both receives will be waiting for data, but no send can be called until the receive returns.
Deadlock (2)

• How about this one?

```c
// Exchange data between two processes
If (process 0)
  Receive data from process 1
  Send data to process 1
If (process 1)
  Send data to process 0
  Receive data from process 0
```
Deadlock (2)

• How about this one?

```c
// Exchange data between two processes
If (process 0)
  Receive data from process 1
  Send data to process 1
If (process 1)
  Send data to process 0
  Receive data from process 0
```

No deadlock will occur – process 0 will receive the data first, then send the data to process 1; However, there will be performance penalty because we turn concurrent operations into sequential.
Deadlock (3)

• And this one?

```c
// Exchange data between two processes
If (process 0)
  Send data to process 1
  Receive data from process 1
If (process 1)
  Send data to process 0
  Receive data from process 0
```
Deadlock (3)

• And this one?

```c
// Exchange data between two processes
if (process 0)
    send data to process 1
    receive data from process 1

if (process 1)
    send data to process 0
    receive data from process 0
```

It depends. If one of the sends returns, then we are OKAY - most MPI implementations buffer the message, so a send could return even before the matching receive is posted. However, if this is not the case or the message is too large to be buffered, deadlock will occur.
Non-blocking Operations (1)

• Non-blocking operations separate the initialization of a send or receive from its completion

• Two calls are required to complete a send or receive
  – Initialization
    • Send: MPI_ISEND
    • Receive: MPI_IRECV
  – Completion: MPI_WAIT
Non-blocking Operations (2)

- **MPI_ISEND**
  - C: `int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  - Fortran: `MPI_ISEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, REQ, IERR)`
- **MPI_IRecv**
  - C: `int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *request);`
  - Fortran: `MPI_Irecv(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, REQUEST, IERR)`
- **MPI_WAIT**
  - C: `int MPI_Wait(MPI_Request *request, MPI_Status *status);`
  - Fortran: `MPI_WAIT(REQUEST, STATUS, IERR)`
Example: Exchange Data with Non-blocking calls

integer reqids, reqidr
integer status(mpi_status_size)

if (myid.eq.0) then
  call mpi_isend(to_p1,n,mpi_integer,1,100,mpi_comm_world,reqids,ierr)
  call mpi_irecv(from_p1,n,mpi_integer,1,101,mpi_comm_world,reqidr,ierr)
elseif (myid.eq.1) then
  call mpi_isend(to_p0,n,mpi_integer,0,101,mpi_comm_world,reqids,ierr)
  call mpi_irecv(from_p0,n,mpi_integer,0,100,mpi_comm_world,reqidr,ierr)
endif

call mpi_wait(status,reqids,ierr)
call mpi_wait(status,reqidr,ierr)
Blocking vs. Non-blocking

• Blocking operations are data corruption proof, but
  – Possible deadlock
  – Performance penalty

• Non-blocking operations allow overlap of completion and computation
  – The process can work on other things between the initialization and completion
  – Should be used whenever possible
Exercise 2a: Find Global Maximum

• Goal: Find the maximum in an array
  – Each process handle part of the array
  – Every process needs to know the maximum at the end of program

• Hints
  – This can be done in two steps
    • Step 1: each process send the local maximum to the root process to find the global maximum
    • Step 2: the root process send the global maximum to all other processes
Exercise 2b: Matrix Multiplication

Version 2

• Modify version 1 so that each process sends its partial results to the root process
  – The root process should have the whole matrix of C
• Then validate the result at the root process
Exercise 2c: Laplace Solver Version 1

- **Goal:** develop a working MPI Laplace solver
  - Distribute the workload in a one-dimensional manner
  - Initialize the sub-matrix at each process and set the boundary values
  - At the end of each iteration
    - Exchange boundary data with neighbors
    - Find the global convergence error and distribute to all processes