Introduction to Message Passing Interface (MPI)

Le Yan
Scientific Computing Consultant
User Services
LONI High Performance Computing
Outline

• Introduction – what is MPI and why MPI?
• Basic MPI functions
  – Environment and communicator management
  – Point to point communication
  – Collective communication
• How to parallelize serial programs
Message Passing

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

• Most message passing programs use the single program multiple data (SPMD) model
  – Each processor executes the same set of instructions
  – Parallelization is achieved by letting each processor operate on a different piece of data
MPI: Message Passing Interface

• MPI defines a **standard** API for message passing
  – What’s in the standard:
    • A core set of functions
    • Both the syntax and semantics of these functions
  – What’s not in the standard:
    • How to compile and link the code
    • How many processes on which the code will run

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

• Portability
  – MPI implementations are available on almost all platforms

• Explicit parallelization
  – Users have control on when, where and how the data transmit occurs

• Scalability
  – Not limited by the number of processors on one computation node, as opposed to shared-memory parallel models
MPI Functions

• Environment and communicator management functions
  – Initialization and termination
  – Communicator setup

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

• Point-to-point communication functions
  – Message transfer from one process to another
A sample MPI program

```c
#include 'mpif.h'
...
call mpi_init(ierr)
...
call mpi_comm_size(comm,size,ierr)
call mpi_comm_rank(comm,rank,ierr)
...
call mpi_finalize(ierr)
...```

---

LONI High Performance Computing Workshop, University of Louisiana at Lafayette
November 2, 2010
Header file

... include 'mpif.h'
... call mpi_init(ierr)
... call mpi_comm_size(comm,size,ierr)
call mpi_comm_rank(comm,rank,ierr)
... call mpi_finalize(ierr)

- Defines MPI-related parameters
- Must be included
- C/C++: mpi.h
Initialization

```c
... include 'mpif.h'
... call mpi_init(ierr)
... call mpi_comm_size(comm,size,ierr)
call mpi_comm_rank(comm,rank,ierr)
... call mpi_finalize(ierr)
```

- Must be called before any other MPI calls;
- `MPI_Init()` for C and `MPI::Init()` for C++
... include 'mpif.h'
...
call mpi_init(ierr)
...
call mpi_comm_size(comm,size,ierr)
call mpi_comm_rank(comm,rank,ierr)
...
call mpi_finalize(ierr)

- Must be called after all other MPI calls;
- MPI_Finalize() for C and MPI::Finalize() for C++
Communicator size

... include 'mpif.h'
...
call mpi_init(ierr)
...
call mpi_comm_size(comm, size, ierr)
call mpi_comm_rank(comm, rank, ierr)
...
call mpi_finalize(ierr)

- Return the number of processes (size) in a communicator (comm);
- MPI_Comm_size for C and MPI::Comm::Get_size for C++
... include 'mpif.h'
... call mpi_init(ierr)
... call mpi_comm_size(comm,size,ierr)
call mpi_comm_rank(comm,rank,ierr)
... call mpi_finalize(ierr)

• Return the rank of the current processes (rank) in a communicator (comm);
• Allow us to make the behavior of each process different by using the value of its rank;
• MPI_Comm_rank for C and MPI::Comm::Get_rank for C++
Example

```fortran
include 'mpif.h'
call mpi_init(ierr)
call mpi_comm_size(comm,size,ierr)
call mpi_comm_rank(comm,rank,ierr)
if (rank.eq.0) then
    print(*,*) 'I am the root'
    print(*,*) 'My rank is',rank
else
    print(*,*) 'I am not the root'
    print(*,*) 'My rank is',rank
endif

call mpi_finalize(ierr)
```

Output (assume 3 processes):

```
I am not the root
My rank is 2
I am the root
My rank is 0
I am not the root
My rank is 1
```
Communicators

- A communicator is an identifier associated with a group of processes
  - Can think of it as an ordered list of processes (a mapping from MPI processes to physical processes)
  - Each process has a unique id (rank) within a communicator
    - Ex: if there are 8 processes in a communicator, their ranks will be 0, 1, ..., 7.
  - It is the context of any MPI communication
    - Unless a context is specified, MPI cannot understand “get this message to all processes” or “get this message from process #1 to process #2”. 
More on communicators

- **MPI_COMM_WORLD**: default communicator contains all processes

- More than one communicators can co-exist
  - Useful when communicating among a subset of processes

- A process can belong to different communicators
  - Ex: A physical process can be proc #4 in comm1 and proc #0 in comm2
  - An analogy is that a person can have different identities under different contexts
Point-to-point communication

• Process to process communication (two processes are involved)

• There are two types of point-to-point communication
  – Blocking
  – Non-blocking
Blocking

- The call will wait until the data transfer process is over.
  - The sending process will wait until all data are transferred from the send buffer to the system buffer.
  - The receiving process will wait until all data are transferred from the system buffer to the receive buffer.
- All collective communications are blocking
Non-blocking

- Returns immediately after the data transfer is initiated.
- More efficient than blocking procedures.
- Could cause problems
  - When send and receive buffers are updated before the transfer is over, the result might not be the one expected.
  - Example provided in the hands-on lab.
Examples

• Transfer data from process 0 to process 1

  • Blocking send and receive

    IF (myrank==0) THEN
      CALL MPI_SEND(sendbuf,count,datatype,destination,tag,comm,ierror)
    ELSEIF (myrank==1) THEN
      CALL MPI_RECV(recvbuf,count,datatype,source,tag,status,comm,ierror)
    ENDIF

  • Non-blocking send and receive

    IF (myrank==0) THEN
      CALL MPI_ISEND(sendbuf,count,datatype,destination,tag,comm,ireq,ierror)
    ELSEIF (myrank==1) THEN
      CALL MPI_IRecv(recvbuf,count,datatype,source,tag,comm,ireq,ierror)
    ENDIF
    CALL MPI_WAIT(ireq,istatus,ierror)
Data exchange between 2 processes

- We can do two separate send-receive pairs
  - Inefficient
- Simultaneous send-receive
  - Efficient
  - Possible deadlock
    - One processor is waiting for a message from the other, which is also waiting for a message from the first – nothing will happen and your job will be killed when the queue time runs out (MPI does not have timeout!!)
    - Something to avoid

IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf,...)
  CALL MPI_RECV(recvbuf,...)
ELSEIF (myrank==1) THEN
  CALL MPI_SEND(sendbuf,...)
  CALL MPI_RECV(recvbuf,...)
ENDIF
Deadlock

IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf,...)
  CALL MPI_RECV(recvbuf,...)
ELSEIF (myrank==1) THEN
  CALL MPI_SEND(sendbuf,...)
  CALL MPI_RECV(recvbuf,...)
ENDIF
Solution for deadlock

IF (myrank==0) THEN
   CALL MPI_ISEND(sendbuf,...)
   CALL MPI_RECV(recvbuf,...)
   CALL MPI_WAIT(ireq,...)
ELSEIF (myrank==1) THEN
   CALL MPI_ISEND(sendbuf,...)
   CALL MPI_RECV(recvbuf,...)
   CALL MPI_WAIT(ireq,...)
ENDIF

• Non-blocking send
  – Process 0: Start sending; then start receiving while the data is being sent;
  – Process 1: Start sending; then start receiving while the data is being sent;
Collective communication

- Collective communications are communications that involve all processes in a communicator
- There are three types of collective communications
  - Data movement
    - Example: mpi_bcast
  - Reduction (computation)
    - Example: mpi_reduce
  - Synchronization
    - Example: mpi_barrier
Broadcast

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
</tr>
<tr>
<td>P2</td>
<td>A</td>
</tr>
<tr>
<td>P3</td>
<td>A</td>
</tr>
</tbody>
</table>

- Send data from one process (called root) to all other processes in the same communicator.
- Called by all processes in the communicator using the same arguments
Example

PROGRAM bcast
INCLUDE 'mpif.h'
INTEGER imsg(4)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
  DO i=1,4
    imsg(i) = i
  ENDDO
ELSE
  DO i=1,4
    imsg(i) = 0
  ENDDO
ENDIF
PRINT *, 'Before: ', imsg
CALL MP_FLUSH(1)
CALL MPI_BCAST(imsg, 4, MPI_INTEGER,
                0, MPI_COMM_WORLD, ierr)
PRINT *, 'After: ', imsg
CALL MPI_FINALIZE(ierr)
END

Output

0: Before: 1 2 3 4
1: Before: 0 0 0 0
2: Before: 0 0 0 0
0: After: 1 2 3 4
1: After: 1 2 3 4
2: After: 1 2 3 4
• Collects data from all processes in the communicator to the root process (the data have to be of the same size).
• Called by all processes in the communicator using the same arguments
Example

PROGRAM gather
INCLUDE 'mpif.h'
INTEGER irecv(3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
isend = myrank + 1
CALL MPI_GATHER(isend, 1, MPI_INTEGER,
                & irecv, 1, MPI_INTEGER,
                & 0, MPI_COMM_WORLD, ierr)
IF (myrank==0) THEN
    PRINT *,'irecv =',irecv
ENDIF
CALL MPI_FINALIZE(ierr)
END

Output
0: irecv = 1 2 3
Reduction

- Similar to gather: collects data from all processes
- Then perform some operation on the collected data.
- Called by all processes in the communicator using the same arguments
Reduction operations

- Summation and production
- Maximum and minimum
- Max and min location
- Logical (AND & OR & XOR)
- Bitwise (AND & OR & XOR)
- User defined
  - Subroutine mpi_op_create
Examples

PROGRAM reduce
INCLUDE 'mpif.h'
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
isum = 0
ista = myrank * 3 + 1
iend = ista + 2
DO i=ista,iend
   isum = isum + i
ENDDO
CALL MPI_REDUCE(isum, itmp, 1, MPI_INTEGER, MPI_SUM, 0, &
                 MPI_COMM_WORLD, ierr)
isum = itmp
IF (myrank==0) THEN
   PRINT *, 'isum =', isum
ENDIF
CALL MPI_FINALIZE(ierr)
END

Output (with 3 processors)
0: isum = 45
Some other collective communication

**Scatter**

- **P0**: A
- **P1**: B
- **P2**: C
- **P3**: D

- **P0**: A
- **P1**: B
- **P2**: C
- **P3**: D

- **P0**: A' B'C'D
- **P1**: A'B'C'D
- **P2**: A'B'C'D
- **P3**: A'B'C'D

*: some operator

**Gather**

- **P0**: A
- **P1**: B
- **P2**: C
- **P3**: D

- **P0**: A
- **P1**: B
- **P2**: C
- **P3**: D

- **P0**: A
- **P1**: A'B
- **P2**: A'B'C
- **P3**: A'B'C'D

*: some operator

**Allgather**

- **P0**: A A B C D
- **P1**: A B C D
- **P2**: A B C D
- **P3**: A B C D

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>P2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>P3</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

- **P0**: A
- **P1**: A'B
- **P2**: A'B'C
- **P3**: A'B'C'D

*: some operator

**Alltoall**

- **P0**: A0 A1 A2 A3
- **P1**: B0 B1 B2 B3
- **P2**: C0 C1 C2 C3
- **P3**: D0 D1 D2 D3

- **P0**: A0 B0 C0 D0
- **P1**: A1 B1 C1 D1
- **P2**: A2 B2 C2 D2
- **P3**: A3 B3 C3 D3

- **P0**: A0 B0 C0 D0
- **P1**: A1 B1 C1 D1
- **P2**: A2 B2 C2 D2
- **P3**: A3 B3 C3 D3

- **P0**: A0 B0 C0 D0
- **P1**: A1 B1 C1 D1
- **P2**: A2 B2 C2 D2
- **P3**: A3 B3 C3 D3

*: some operator

**Scan**

- **P0**: A
- **P1**: A'B
- **P2**: A'B'C
- **P3**: A'B'C'D

*: some operator

**Reduce scatter**

- **P0**: A0 A1 A2 A3
- **P1**: B0 B1 B2 B3
- **P2**: C0 C1 C2 C3
- **P3**: D0 D1 D2 D3

- **P0**: A0 B0 C0 D0
- **P1**: A1 B1 C1 D1
- **P2**: A2 B2 C2 D2
- **P3**: A3 B3 C3 D3

*: some operator
Synchronization

• Called by all processes in a communicator
• Blocks each process in the communicator until all processes have called it.
• It can slow down the program remarkably, so do not use it unless really necessary
Steps to parallelize a serial program

• Make sure the serial program works

• Identify which part of your code needs to be parallelized
  – Which part consumes most of the CPU time
  – Which part can be parallelized

• Decide the details
  – How loops are parallelized
  – What data has to be transmitted between processes (the less the better)
Example

! The serial version

Program summation_ser
...

hide=0.
do i=1,n
  do j=1,n
    <compute some_result>
    total=total+some_result
  enddo
enddo
...

! The parallel version

Program summation_par

include 'mpif.h'
...
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myrank,ierr)
iwork=(n-1)/nprocs+1
ista=min(myrank*iwork+1,n+1)
iend=min(ista+iwork-1,10)
total_proc=0.
do i=ista,iend
  do j=1,n
    <compute some_result>
    total_proc=total_proc+some_result
  enddo
enddo

call mpi_reduce(total_proc,total,1,mpi_real8,mpi_sum,0,mpi_comm_world,ierr)
...
call mpi_finalize(ierr)
...

In Case of n=10, nprocs=4

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Alternative

! The serial version
Program summation_ser
...
total=0.
do i=1,n
   do j=1,n
      <compute some_result>
      total=total+some_result
   enddo
enddo
...

! The parallel version
Program summation_par_v2
include 'mpif.h'
...
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myrank,ierr)
total_proc=0.
do i=1+myrank,n,nprocs
   do j=1,n
      <compute some_result>
      total_proc=total_proc+some_result
   enddo
enddo
call mpi_reduce(total_proc,total,1,mpi_real8,mpi_sum,0,mpi_comm_world,ierr)
...
call mpi_finalize(ierr)
...

In Case of n=10, nprocs=4
<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
References

• Internet
  – http://www.mpi-forum.org
  – http://www.mcs.anl.gov/mpi
  – http://docs.loni.org
  – http://www.hpc.lsu.edu/help

• Books
  – *Using MPI*, by W. Gropp, E. Lusk and A. Skjellum
  – *Using MPI-2*, by W. Gropp, E. Lusk and A. Skjellum
  – *Parallel programming with MPI*, by P. Pacheco
  – *Practical MPI Programming*, IBM Redbook
Hand-on Labs

• How to get the lab material
  – Log in any cluster of your choice
  – Type the following commands:
    ➢ `cp -r ~lyan1/traininglab/mpilab .`
    ➢ `cd mpilab`

• What’s in it
  – A README file
    ➢ How to compile and run a MPI code
  – Two directories corresponding to different languages
    ➢ C
    ➢ Fortran
Overview of the sample programs

- **hello.f90, hello.c**
  - Each process prints a “Hello, world!” message.

- **bcast.f90, bcast.c**
  - An example of the broadcast collective communication.

- **allgatherv.f90, allgatherv.c**
  - An example of the allgatherv collective communication.

- **reduceprod.f90, reduceprod.c**
  - An example of the reduce collective communication.
Overview of the sample programs

- **pointcomm.f90, pointcomm.c**
  - Examples of blocking and non-blocking point-point communications, and the potential problem of non-blocking communication

- **pointbcast.f90, pointbcast.c**
  - Use point-point communication to perform a data transfer equivalent to the bcast collective communication

- **paraloop.f90, paraloop.c**
  - Two basic techniques to parallelize a DO loop: block and cyclic