Introduction to MPI Programming – Part 1

Wei Feinstein, Le Yan

HPC@LSU
Outline

• Introduction
• MPI program basics
• Point-to-point communication
Why Parallel Computing

As computing tasks get larger and larger, may need to enlist more computer resources

- Bigger: more memory and storage
- Faster: each processor is faster
- More: do many computations simultaneously
Memory system models for parallel computing

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

- Distributed
Message Passing

Any data to be shared must be explicitly transferred from one to another

Communication medium: concrete network,…

Entities: MPI processes
Why MPI?

• There are already network communication libraries
• Optimized for performance
• Take advantage of faster network transport
  • Shared memory (within a node)
  • Faster cluster interconnects (e.g. InfiniBand)
  • TCP/IP if all else fails
• Enforces certain guarantees
  • Reliable messages
  • In-order message arrival
• Designed for multi-node technical computing
MPI History

- 1980-1990
- 1994: MPI-1
- 1998: MPI-2
- 2012: MPI-3
Message Passing Interface

• MPI defines a standard API for message passing
  – The standard includes
    • What functions are available
    • The syntax of those functions
    • What the expected outcome is when calling those functions
  – The standard does NOT include
    • Implementation details (e.g. how the data transfer occurs)
    • Runtime details (e.g. how many processes the code run with etc.)
• MPI provides C/C++ and Fortran bindings
Various MPI Implementations

• OpenMPI: open source, portability and simple installation and config
• MPICH: open source, portable
• MVAPICH2: MPICH derivative InfiniBand, iWARP and other RDMA-enabled interconnects (GPUs)
• Intel MPI (IMPI): vendor-supported MPICH from Intel
High or low level programming?

- High level compared to other network libraries
  - Abstract transport layer
  - Supply higher-level operations
- Low level for scientists
  - Handle problem decomposition
  - Manually write code for every communications among processes
More about MPI

• MPI provides interface to libraries
  • APIs and constants
  • Binding to Fortran/C
  • Several third-party bindings for Python, R and more other languages
  • Run MPI programs (e.g. mpiexec)
Let’s try it

- $\texttt{whoami}$
- $\texttt{mpiexec -np 4 whoami}$
What just happened?

- mpiexec launched 4 processes
- Each process ran `whoami`
- Each ran independently
- Usually launch no more MPI processes than #processors
- Use multiple nodes:
  ```
  mpiexec --hostfile machine.lst --np/-npp 4 app.exe
  ```
Outline of a MPI Program

1. Initialize communications
   - `MPI_INIT` initializes the MPI environment
   - `MPI_COMM_SIZE` returns the number of processes
   - `MPI_COMM_RANK` returns this process’s index (rank)

2. Communicate to share data between processes
   - `MPI_SEND` sends a message
   - `MPI_RECV` receives a message

3. Exit in a “clean” fashion when MPI communication is done
   - `MPI_FINALIZE`
Hello World (C)

```c
#include "mpi.h"

int main(int argc, char* argv[]){
    int nprocs, myid;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    printf("Hello World from process %d/%d \n", myid, nprocs);

    MPI_Finalize();
    ...
}
```

Header file

Initialization

Computation and communication

Termination
Hello World (C)

```
#include "mpi.h"

int main(int argc, char* argv[]){
    int nprocs, myid;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    printf("Hello World from process %d/%d
", myid, nprocs);

    MPI_Finalize();
    ...
}
```
Hello World (Fortran)

include "mpif.h"

integer::nprocs, ierr, myid
integer::status(mpi_status_size)

call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
call mpi_comm_rank(mpi_comm_world, myid, ierr)

write(*, '("Hello World from process ",I3, "/",I3)') myid, nprocs

call mpi_finalize(ierr)

...
Hello World (Fortran)

```
include "mpif.h"

integer::nprocs, ierr, myid
integer::status(mpi_status_size)

call mpi_init(ierr)

call mpi_comm_size(mpi_comm_world, nprocs, ierr)

call mpi_comm_rank(mpi_comm_world, myid, ierr)

write(*, '("Hello World from process ",I3," / ",I3)') myid, nprocs

call mpi_finalize(ierr)
```

Header file

Initialization

```bash
[wfeinste@shelob1 hello]$ mpif90 hello.f90
[wfeinste@shelob1 hello]$ mpirun -np 4 ./a.out
```

Hello World from process 3 / 4
Hello World from process 0 / 4
Hello World from process 1 / 4
Hello World from process 2 / 4

Termination
Naming Signature (C/Fortran)

• Function name convention
  – C: MPI_Xxxx(arg1,...)
  – Fortran: mpi_xxx (not case sensitive)

• Error handles
  If rc/ierr == MPI_SUCCESS, then the call is successful.
  • C: int rc = MPI_Xxxx(arg1,...)
  • Fortran: call mpi_some_function(arg1,...,ierr)
A communicator is an identifier associated with a group of processes.

```c
MPI_Comm_size(MPI_Comm comm, int *nprocs);
MPI_Comm_rank(MPI_Comm comm, int *myid);
```
Communicators (2)

• A communicator is an identifier associated with a group of processes
  – Can be regarded as the name given to an ordered list of processes
  – Each process has a unique rank, which starts from 0 (usually referred to as “root”)
  – It is the context of MPI communications and operations
    • For instance, when a function is called to send data to all processes, MPI needs to understand what “all” means
Communicators (3)

• MPI_COMM_WORLD: the default communicator contains all processes running a MPI program

• There can be many communicators
  e.g., MPI_Comm_split(MPI_Comm comm, int color, int, kye, MPI_Comm* newcomm)

• A process can belong to multiple communicators
  – The rank is usually different
Communicator Information

• Rank: unique id of each process
  – C: `MPI_Comm_Rank(MPI_Comm comm, int *rank)`
  – Fortran: `MPI_COMM_RANK(COMM,RANK,ERR)`

• Get the size/processes of a communicator
  – C: `MPI_Comm_Size(MPI_Comm comm, int *size)`
  – Fortran: `MPI_COMM_SIZE(COMM,SIZE,ERR)`
Compiling MPI Programs

• Not a part of the standard
  – Could vary from platform to platform
  – Or even from implementation to implementation on the same platform
  – mpicc/mpicxx/mpif77/mpif90: wrappers to compile MPI code and auto link to startup and message passing libraries.
## MPI Compilers

<table>
<thead>
<tr>
<th>Language</th>
<th>Script Name</th>
<th>Underlying Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>gcc</td>
</tr>
<tr>
<td></td>
<td>mpiicc</td>
<td>icc</td>
</tr>
<tr>
<td></td>
<td>mpipgcc</td>
<td>pgcc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC</td>
<td>g++</td>
</tr>
<tr>
<td></td>
<td>mpiicpc</td>
<td>icpc</td>
</tr>
<tr>
<td></td>
<td>mpipgCC</td>
<td>pgCC</td>
</tr>
<tr>
<td>Fortran</td>
<td>mpif90</td>
<td>f90</td>
</tr>
<tr>
<td></td>
<td>mpgfortran</td>
<td>gfortran</td>
</tr>
<tr>
<td></td>
<td>mpiifort</td>
<td>ifort</td>
</tr>
<tr>
<td></td>
<td>mpipgf90</td>
<td>pgf90</td>
</tr>
</tbody>
</table>
Compiling and Running MPI Programs

• On Shelob:
  – Compile
    • C: `mpicc -o <executable name> <source file>`
    • Fortran: `mpif90 -o <executable name> <source file>`
  – Run
    • `mpirun -hostfile $PBS_NODEFILE -np <number of procs> <executable name> <input parameters>`
About Exercises

- Exercises
  - Track a: Process color
  - Track b: Matrix multiplication
  - Track c: Laplace solver

- Your tasks:
  - Fill in blanks to make MPI programs work under /exercise directory
  - Solutions are provided in /solution directory
Exercise a1: 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 b1: Matrix Multiplication

\[ C_{1,1} = \sum_{i=1}^{n} (A_{1,i} \times B_{i,1}) \]
Exercise b1: Matrix Multiplication

```c
for(i=0;i<row;i++){
    for(j=0;j<col;j++){
        sum=0;
        for(k=0;k<n;k++)
            sum=sum+a[i][k]*b[k][j];
        c[i][j]=sum;       //final matrix
    }
}
```
Exercise b1: Matrix Multiplication

• 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 c1: Laplace Solver version 1

\[ P_{x,y} = \frac{(D_{x-1,y} + D_{x,y-1} + D_{x+1,y} + D_{x,y+1})}{4} \]
Exercise c1: Laplace Solver version 1

• Goal: Distribute the work load among processes in 1-d manner
  e.g. 4 MPI processes (color coded) to share the work load
Exercise c1: Laplace Solver version 1

$X$, showing decomposition by color

$X_{local}$ for Blue processor

Ghost points

Ghost points
Exercise c1: Laplace Solver version 1

• 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)
MPI Functions

- Environment management functions
  - Initialization and termination
- Point-to-point communication functions
  - Message transfer from one process to another
- Collective communication functions
  - Message transfer involving all processes in a communicator
Point-to-point Communication

CPU 1
Process 0
send
data

CPU 2
Process 1
receive
data
Point-to-point Communication

• Blocking send/receive
  – 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)
Send/Receive

int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);

int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status)

• A MPI message consists of two parts
  – Message itself: data body
  – Message envelope: routing info

• status: information of the message that is received
Example: Gathering Array Data

• Gather some array data from each process and place it in the memory of the root process.
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, 0, mpi_comm_world, status, ierr)
enddo
write(*,*) "The content of the array:" write(*,*) array
else
call mpi_send(array, 2, mpi_integer, 0, 0, mpi_comm_world, 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 value(s) stored in the variable can be **safely overwritten**
  – When a **blocking receive** returns, the data has been received and is **ready to be used**
Deadlock (1)

Deadlock occurs when both processes awaits the other to make progress

```
// 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
```

- Guaranteed deadlock!
- Both receives wait 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
```

• No deadlock!
• P0 receives the data first, then sends the data to P1
• There will be performance penalty due to serialization of potentially concurrent operations.
Deadlock (3)

• And this one?

```cpp
// 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 send returns, then we are OKAY - most MPI implementations buffer the message, so a send could return even before the matching receive is posted.
• If the message is too large to be buffered, deadlock will occur.
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 tasks between the initialization and completion
  – Should be used whenever possible
Non-blocking Operations (asynchronous)

- Separate 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 Point-to-point Communication

- **MPI_ISEND function**
  - 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 function**
  - 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, REQ, 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

```fortran
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)
```
Exercise a2: 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
  • 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 b2: Matrix Multiplication

• Modify b1 so that each process sends its partial results to the root process
  – The root process should have the whole matrix
• Validate the result at the root process
Exercise c2: Laplace Solver

- Goal: develop a working MPI Laplace solver using c1
  - Distribute the workload in 1D 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
Why MPI?

• Standardized
  – With efforts to keep it evolving (MPI 3.0)
• 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
  – De Facto programming model for distributed memory machines
• Nearly every big academic or commercial simulation or
  data analysis running on multiple nodes uses MPI
  directly or indirectly
Continue...

- MPI Part 2: Collective communications
- MPI Part 3: Understanding MPI applications