MPI

https://rc.colorado.edu/training/downloads

August 13, 2017
Timothy Brown

Research Computing
UNIVERSITY OF COLORADO **BOULDER**

Overview

Background

MPI

Programming Models

Data Types

Communications

Point to Point Communications

Collective Communications

Topologies

Parallel IO

Overview

Parallelism

Parallelism can be achieved across many levels

- ▸ Nodes – MPI

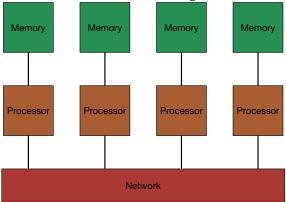- ▸ Threads – OpenMP

- ▸ Instructions – ILP

  I1: add R1, R2, R3
  I2: sub R4, R1, R5
  I3: xor R10, R2, R11

- ▸ Data – SIMD

Distributed Memory Model

- All processors see a different view of data.
- Processors interact and synchronize by passing messages.

Message Passing I

- ► Most natural and efficient paradigm for distributed-memory systems.
- ► Two-sided, send and receive communication between processes.
- ► Efficiently portable to shared-memory or almost any other parallel architecture: "assembly language of parallel computing" due to universality and detailed, low-level control of parallelism.

Message Passing II

- ► Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary.
- ► Sometimes deemed tedious and low-level, but thinking about locality promotes:
  - ► good performance
  - ► scalability
  - ► portability

Overview

Programming

- MPI (Message Passing Interface).
- Message passing standard, universally adopted library of communication routines callable from C, C++, Fortran, Java, (Python).
- 125+ functions—I will introduce a small subset of functions.

MPI

- ▶ MPI has been developed in three major stages, MPI 1, MPI 2 and MPI 3.
  - ▶ MPI 1 – 1994
  - ▶ MPI 2 – 1996
  - ▶ MPI 3 – 2012
- ▶ MPI Forum
  `http://www.mpi-forum.org/docs/docs.html`
- ▶ MPI Standard
  `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`
- ▶ *Using MPI* and *Using Advanced MPI*
  `http://www.mcs.anl.gov/research/projects/mpi/usingmpi/`
- ▶ Online MPI tutorial
  `http://mpitutorial.com/beginner-mpi-tutorial/`

MPI 1

- ► Features of MPI-1 include:
  - ► Point-to-point communication.
  - ► Collective communication process.
  - ► Groups and communication domains.
  - ► Virtual process topologies.
  - ► Environmental management and inquiry.
  - ► Profiling interface bindings for Fortran and C.

MPI 2

- ► Additional features of MPI-2 include:
    - ► Dynamic process management input/output.
    - ► One-sided operations for remote memory access (update or interrogate).
    - ► Memory access bindings for C++.

MPI 3

- ▶ Updates, corrections and clarifications of MPI-3 include:
  - ▶ Non-blocking collectives.
  - ▶ New one-sided communication operations.
  - ▶ Fortran 2008 bindings.

Implementations

- MPICH `ftp://ftp.mcs.anl.gov/pub/mpi`
- OpenMPI `http://www.open-mpi.org/`
- Intel MPI
  `https://software.intel.com/en-us/intel-mpi-library`
- SGI
- Cray
- IBM

Overview

Programming Models

- Single Program Multiple Data (SPMD)
  - Same program runs on each process.
- Multiple Programs Multiple Data (MPMD)
  - Different programs runs on each process.

Programming in MPI

```
_____ C _____
#incude <mpi.h>

int ierr = 0;
ierr = MPI_Init(&argc,
                &argv);
...
ierr = MPI_Finalize();
```

```
_____ Fortran _____
use mpi

integer :: ierr
ierr = 0
call MPI_Init(ierr)
...
call MPI_Finalize(ierr)
```

- ▶ C returns error codes as function values.
- ▶ Fortran uses the last argument (ierr).

Execution

Once your program has compiled you can run it with:

- ▶ Through a batch system (SLURM for example).
  login01 ~$ srun –n 1 ./a.out
- ▶ Through mpiexec.
  node0001 ~$ mpiexec –n 1 ./a.out

Login to Janus

1. **Log in to Janus.**
   ```
   laptop ~$ ssh user0000@tutorial-login.rc.colorado.edu
   ```


2. **Load the slurm and intel module.**
   ```
   [user0000@tutorial-login ~]$ ml slurm gcc openmpi
   ```

3. **Start a compute job.**
   ```
   [user0000@tutorial-login ~]$ sinteractive        \
                         -t 02:00:00 -N 1            \
                         --reservation=rmacc-tutorials
   ```

Communicator

- A collection of processors working on some part of a parallel job.
- Used as a parameter for most MPI calls.
- Processors within a communicator are assigned ranks 0 to $n-1$.
- MPI_COMM_WORLD includes all of the processors in your job.
- Can create subsets of MPI_COMM_WORLD

```Fortran
program hello
use mpi

implicit none

integer :: ierr
integer :: id, nprocs
ierr = 0
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, id, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
call MPI_Finalize(ierr)
end program hello
```

- Determine the process rank (id).
- Total number of processes (nprocs).

Additional Functions

- Determine the processor name (hostname)
  `MPI_Get_processor_name(node, nlen, ierr)`
- Determine the MPI version (and sub-version)
  `MPI_Get_version(ver, sub_ver, ierr)`

Note: All MPI calls *should* have man-pages.

Exercise

- **Write a hello world program that outputs:**
  - Rank
  - Processor name
  - MPI version

  There is a template in `mpi_ver.f90`.

All of the examples taught are online, in a github repository:
`https://github.com/ResearchComputing/RMACC_2015`

Overview

Data Types

Fortran	Optional	C
MPI_CHARACTER		MPI_CHAR
MPI_COMPLEX	8, 16, 32	MPI_DOUBLE
MPI_DOUBLE_COMPLEX		MPI_FLOAT
MPI_DOUBLE_PRECISION		MPI_INT
MPI_INTEGER	1, 2, 4, 8	MPI_LONG_DOUBLE
MPI_LOGICAL	1, 2, 4, 8	MPI_LONG_LONG
MPI_REAL	2, 4, 8, 16	MPI_LONG_LONG_INT
		MPI_SHORT
		MPI_SIGNED_CHAR
		MPI_UNSIGNED
		MPI_UNSIGNED_CHAR
		MPI_UNSIGNED_LONG
		MPI_UNSIGNED_LONG_LONG
		MPI_UNSIGNED_SHORT
		MPI_WCHAR

Overview

Communications

- ► Bytes transferred from one processor to another.
- ► Specify destination, data buffer, and message ID (called a tag).
- ► Synchronous send: send call does not return until the message is sent.
- ► Asynchronous send: send call returns immediately, send occurs during other calculation ideally.
- ► Synchronous receive: receive call does not return until the message has been received (may involve a significant wait).
- ► Asynchronous receive: receive call returns immediately. When received data is needed, call a wait subroutine.
- ► Asynchronous communication used in attempt to overlap communication with computation.

Overview
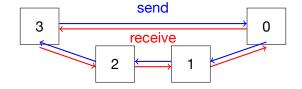
Point to Point Communications

Sending Data In A Ring

Sending n elements to the left (rank +1) receive N elements from the right (rank -1).

- Store them in an array of size $nprocs \times n$.
- Sum up and print the local results.

Blocking Send & Receive

- Blocking Send `MPI_Send()`
  - Does not return until the message data and envelope have been buffered in matching receive buffer or temporary system buffer.
  - Can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver.
  - MPI buffers or not, depending on availability of space.
  - **Non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.
- Blocking Receive `MPI_Recv()`
  - The opposite of `MPI_Send()`.
  - Does not return until the message data has been copied into the destination buffer.
  - Stalls progress of program **but**:
    - Blocking sends and receives enforce process synchronization.
    - Enforces consistency of data.

Blocking Send

```
MPI_Send(buf, count, data_type, dest, tag, comm, ierr)
```

buf	Beginning address of data.
count	Length of the source array (in elements).
data_type	MPI type of data.
dest	Processor number of destination.
tag	Message tag.
comm	Communicator.
ierr	Error status.

Example:
```
MPI_Send(data, 10, MPI_INT, 1, 1, MPI_COMM_WORLD, ierr)
```

Blocking Receive

```
MPI_Recv(buf, count, data_type, source, tag, &
         comm, status, ierr)
```

buf	Beginning address of data.
count	Length of the source array (in elements).
data_type	MPI type of data.
dest	Processor number of destination.
tag	Message tag.
comm	Communicator.
status	Status object.
ierr	Error status.

Example:

```
MPI_Recv(data, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &
         stat, ierr)
```

Receive Wildcards & Status

- Can receive from any source, using `MPI_ANY_SOURCE`.
  ```
  MPI_Recv(data, 10, MPI_INT, MPI_ANY_SOURCE, &
           1, MPI_COMM_WORLD, stat, ierr)
  ```
- Can receive from any tag, using `MPI_ANY_TAG`.
  ```
  MPI_Recv(data, 10, MPI_INT, MPI_ANY_SOURCE, &
           MPI_ANY_TAG, MPI_COMM_WORLD, stat, ierr)
  ```
- Can ignore the status, using `MPI_STATUS_IGNORE`.
  ```
  MPI_Recv(data, 10, MPI_INT, MPI_ANY_SOURCE, &
           MPI_ANY_TAG, MPI_COMM_WORLD,         &
           MPI_STATUS_IGNORE, ierr)
  ```

Status Structure

- In C it is a typedef structure.
- In Fortran it is an array.

```
                    C
typedef struct _MPI_Status {
  int count;
  int cancelled;
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
} MPI_Status;

MPI_Status status;
source = status.MPI_SOURCE;
```

```
                 Fortran
integer :: status(MPI_STATUS_SIZE)
source = status(MPI_SOURCE)
tag    = status(MPI_TAG)
```

For Success

- ▸ Sender must specify a valid destination rank.
- ▸ Receiver must specify a valid source rank.
- ▸ The communicator must be the same.
- ▸ Tags must match.
- ▸ Message data types must match.
- ▸ Receiver's buffer must be large enough.

Exercise

- **Write a ring program that outputs:**
  - The total number of processors
  - Rank
  - Local sum
- **Run the program with different array sizes:**
  - $N = 10$
  - $N = 100000$

There is a template in `ring.f90`.

Deadlocks

A process waiting for a condition that will never become true.

- Easy to write send/receive code that deadlocks
  - Two processes: both receive before send.
  - Send tag does not match receive tag.
  - Process sends message to wrong destination process.

Non-Blocking Send & Receive

Non-Blocking Send `MPI_Isend()` and Receive `MPI_Irecv()`.

- ► Same syntax as `MPI_Send()`/`MPI_Recv()` with the addition of a request handle.
- ► Request handle (`integer` in Fortran; `MPI_Request` in C) is used to check for completeness of the send.
- ► These calls returns immediately.
- ► Data in the buffer may not be accessed until the user has completed the send/receive operation.
- ► The send is completed by a successful call to `MPI_Test()` or a call to `MPI_Wait()`.

Non-Blocking Wait

Non-Blocking Wait for ISend completion `MPI_Wait()`

`MPI_Wait(requset, status, ierr)`

- ▸ Request is the handle returned by the non-blocking send or receive call.
- ▸ Upon return, status holds source, tag, and error code information.
- ▸ This call does not return until the non-blocking call referenced by request has completed.
- ▸ Upon return, the request handle is freed.
- ▸ If request was returned by a call to `MPI_ISend()`, return of this call indicates nothing about the destination process.

Wait for any specified send or receive to complete.

```
MPI_Waitany(count, requsets, index, status, ierr)
```

- ▶ Requests is an array of handles returned by nonblocking send or receive calls.
- ▶ Count is the number of requests.
- ▶ This call does not return until a non-blocking call referenced by one of the requests has completed.
- ▶ Upon return, index holds the index into the array of requests of the call that completed.
- ▶ Upon return, status holds source, tag, and error code information for the call that completed.
- ▶ Upon return, the request handle stored in `requests[index]` is freed.

Wait for all specified send or receive to complete.

```
MPI_Waitall(count, requsets, status, ierr)
```

- ▶ Requests is an array of handles returned by nonblocking send or receive calls.
- ▶ Count is the number of requests.
- ▶ This call does not return until all non-blocking call referenced by one of the requests has completed.
- ▶ Upon return, status holds source, tag, and error code information for all the call that completed.
- ▶ Upon return, the request handle stored in requests is freed.

Tests for the completion of a specific send or receive.

`MPI_Test(request, flag, status, ierr)`

- ► `Request` is a handle returned by a non-blocking send or receive call.
- ► Upon return, `flag` will have been set to true if the associated non-blocking call has completed. Otherwise it is set to false.
- ► If `flag` returns true, the request handle is freed and status contains source, tag, and error code information.
- ► If request was returned by a call to `MPI_Isend()`, return with flag set to true indicates nothing about the destination process.

Exercise

- **Fix the ring program so that it is no longer in a deadlock.**
- **Run the program with different array sizes:**
  - $N = 10$
  - $N = 100000$

There is a template in `ring_fixed.f90`.

Overview

Collective Communications

- ▶ Synchronization points:
- ▶ One to all:
    - ▶ Broadcast data to all ranks.
    - ▶ Scatter (all and parts).
- ▶ All to one:
    - ▶ Gather from a group.
    - ▶ Gather varying amounts from all.
    - ▶ Reduce values on all.
- ▶ All to all
    - ▶ Gather and distribute to all.
    - ▶ Gather and distribute various amounts to all.
    - ▶ Combines values from all processes and distribute.

Synchronization

- Synchronization between MPI processes in a group.

```
MPI_Barrier(comm, ierr)
```

   comm Communicator.

     ierr Error status.

Example:
```
MPI_Barrier(MPI_COMM_WORLD, ierr)
```

Broadcast

Broadcasts a message from the process with rank root to all other processes of the group.

```
MPI_Bcast(buf, count, data_type, root, comm, ierr)
```

buf Beginning address of data.

count Length of the source array (in elements).
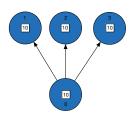
data_type MPI type of data.

root Rank of broadcast root.

comm Communicator.

ierr Error status.

Example:
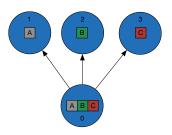
```
MPI_Bcast(data, 10, MPI_INT, 0, MPI_COMM_WORLD, ierr)
```

Exercise

- **Write a program that broadcasts a files information:**
  - Rank
  - File size

  There is a template in bstat.c.

Scatter I


```
MPI_Scatter(sendbuf, sendcount, sendtype, &
            recvbuf, recvcount, recvtype, &
            root, comm, ierr)
```

Scatter II

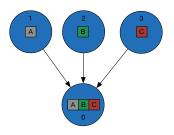|	
sendbuf	Beginning address of send data.
sendcount	Number of elements sent to each process.
sendtype	Datatype of send buffer elements.
recvbuf	Address of receive buffer.
recvcount	Number of elements in receive buffer.
recvtype	Datatype of receive buffer elements.
root	Rank of sending process.
comm	Communicator.
ierr	Error status.

Gather I


```
MPI_Gather(sendbuf, sendcount, sendtype, &
           recvbuf, recvcount, recvtype, &
           root, comm, ierr)
```

Gather II

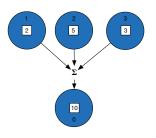|	
sendbuf	Beginning address of send data.
sendcount	Number of elements sent to each process.
sendtype	Datatype of send buffer elements.
recvbuf	Address of receive buffer.
recvcount	Number of elements for any single receive.
recvtype	Datatype of receive buffer elements.
root	Rank of receiving process.
comm	Communicator.
ierr	Error status.

Reduction I


```
MPI_Allgather(sendbuf, recvbuf, count, &
              datatype, op, root, comm, &
              ierr)
```
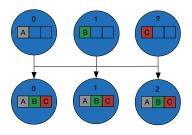
Reduction II

sendbuf Beginning address of send data.

recvbuf Address of receive buffer.

count Number of elements in the send buffer.

datatype Datatype of elements.

op Reduce operation.

root Rank of receiving process.

comm Communicator.

ierr Error status.

Reduction Operations

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

In addition, users may define their own operations that can be overloaded to operate.

All Gather I

MPI_Allgather(sendbuf, sendcount, sendtype, &
          recvbuf, recvcount, recvtype, &
          root, comm, request, ierr)

All Gather II

sendbuf	Beginning address of send data.
sendcount	Number of elements in the send buffer.
sendtype	Datatype of send buffer elements.
recvbuf	Address of receive buffer.
recvcount	Number of elements for any single receive.
recvtype	Datatype of receive buffer elements.
root	Rank of receiving process.
comm	Communicator.
ierr	Error status.

Overview

Topologies

A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.[1]

- ▶ Distribute Processors.
- ▶ Cartesian Topology.
- ▶ Processor Coordinates.

---

[1]MPI 1.1 Documentation

Distribute Processors

Create a balanced distribution of processes per coordinate direction.

MPI_Dims_create(nnodes, ndims, dims, ierr)

nnodes Number of nodes in a grid.

ndims Number of Cartesian dimensions.

dims Integer array of size ndims specifying the number of processors in each dimension.

ierr Error status.

Cartesian Topology.

Makes a new communicator to which Cartesian topology information has been attached.

```
MPI_Cart_create(comm_old, ndims, dims, periods, &
                reorder, comm_cart, ierr)
```

nnodes Input communicator.

ndims Number of Cartesian dimensions.

dims Integer array of size ndims specifying the number of processors in each dimension.

period Logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension.

reorder Ranking may be reordered (true) or not (false).

cart_comm Communicator with new Cartesian topology.

ierr Error status.

Processor Coordinates.

Determines process coords in Cartesian topology given rank in group.

`MPI_Cart_coords(comm, rank, maxdims, coords, ierr)`

comm Communicator with Cartesian structure.

rank Rank of a process within group of comm.

maxdims Length of vector coord in the calling program.

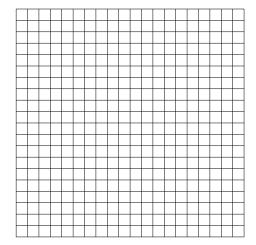coords Integer array containing the Cartesian coordinates of specified processors.

ierr Error status.

Example

Problem

- ▶ Global domain of 20×20.
- ▶ 4 MPI processors.

Steps in the solution:

- ▶ Domain decomposition in MPI.
- ▶ Generate local matrix.

The full solution is in `hdf_pwrite.f90`.

Global Grid:

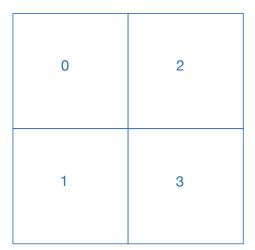Split the process up across the dimensions.

```fortran
integer, parameter :: ndims = 2     ! Number of dims
integer            :: id            ! Rank/ID
integer            :: nprocs        ! Number of procs
integer            :: m_dims(ndims) ! MPI Cart dims

m_dims = (/ 0, 0 /)
call MPI_Comm_rank(MPI_COMM_WORLD, id, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
call MPI_Dims_create(nprocs, ndims, m_dims, ierr)

if (id .eq. 0) then
    write(6, *) 'Processor dims/decomposition: ' m_dims
end if
```

Create a new communicator for this decomposition, also obtain our new rank.

```fortran
logical              :: is_periodic(ndims)
logical              :: reorder
integer              :: MPI_COMM_2D

is_periodic = .false.
reorder     = .false.

call MPI_Cart_create(MPI_COMM_WORLD, ndims, m_dims, &
                     is_periodic, reorder,          &
                     MPI_COMM_2D, ierr)
call MPI_Comm_rank(MPI_COMM_2D, id, ierr)
```
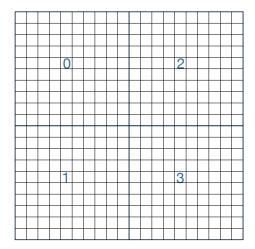
Query our new processor Cartesian coordinates.

```fortran
integer              :: coords(ndims)

call MPI_Cart_coords(MPI_COMM_2D, id, ndims, &
                     coords, ierr)
```

Global grid with our new processor mapping:

Decompose the global grid to our local grid.

```fortran
integer, parameter :: g_N = 20 ! Global grid size
integer :: n(ndims)            ! Local grid size

do i = 1, ndims
    n(i) = g_N / m_dims(i)
end do
allocate(ld(n(1),n(2)), stat=ierr)
```

Global grid with our processor mapping and local grid allocations.

Overview

Parallel IO

- MPI-IO
- Parallel HDF5
  - MPI Properties
  - Hyperslabs

MPI IO

MPI provides a set of file IO routines.

- `MPI_File_open()`
- `MPI_File_set_view()`
- `MPI_File_read_at()`

Be warned, these write binary data to the file.
The file will be:

- Endian dependent.
- Application specific.

Please consider using a higher level library.

HDF5

HDF (Hierarchical Data Format) version 5.
This is really a talk all to it's self.
The easiest way to think about a HDF5 file is that it is a file system in a file. It can contain:

- ▶ Groups (directories/folders)
- ▶ Links (internal and external)
- ▶ Metadata
- ▶ Data
- ▶ Images
- ▶ Tables

Questions?

Online Survey

<Timothy.Brown-1@colorado.edu>

License

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/

When attributing this work, please use the following text: "MPI", Research Computing, University of Colorado Boulder, 2015. Available under a Creative Commons Attribution 4.0 International License.