Introduction to MPI

Ekpe Okorafor

School of Parallel Programming & Parallel Architecture for HPC

ICTP

October, 2014

Topics

- Introduction
- MPI Model and Basic Calls
- MPI Communication
- Summary

Topics

- Introduction
- MPI-1.x Model and Basic Calls
- MPI Communication
- Summary

Parallel Programming Models

- Data Parallelism
  - Each processor performs the same task on different data
- Task Parallelism
  - Each processor performs a different task on the same data
- Most applications fall between these two

Single Program Multiple Data

- SPMD: dominant programming model for shared and distributed memory machines.
  - One source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code start simultaneously and communicate and sync with each other periodically

- MPMD: more general, and possible in hardware, but no system/programming software enables it

Shared Memory Programming: OpenMP

- Shared memory systems (SMPs, cc-NUMAs) have a single address space:
  - applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - shared memory codes are mostly data parallel, ‘SPMD’ kinds of codes
  - OpenMP is the standard for shared memory programming (compiler directives)
  - Vendors offer native compiler directives

Distributed Memory Programming: MPI

- Distributed memory systems have separate address spaces for each processor
  - Local memory accessed faster than remote memory
  - Data must be manually decomposed
  - MPI is the standard for distributed memory programming (library of subprogram calls)
  - Older message passing libraries include PVM and P4; all vendors have native libraries such as SHMEM (T3E) and LAPI (IBM)

Opening Remarks

- Context: distributed memory parallel computers
- We have communicating sequential processes, each with their own memory, and no access to another process's memory
  - A fairly common scenario from the mid 1980s (Intel Hypercube) to today
  - Processes interact (exchange data, synchronize) through message passing
  - Initially, each computer vendor had its own library and calls
  - First standardization was PVM
    - Started in 1989, first public release in 1991
    - Worked well on distributed machines
    - A library, not an API
  - Next was MPI

Message Passing Interface (MPI)

- What is it?
  - An open standard library interface for message passing, ratified by the MPI Forum
  - Version: 1.0 (1994), 1.1 (1995), 1.2 (1997), 1.3 (2008)
  - Version: 2.0 (1997), 2.1 (2008), 2.2 (2009)
  - Version: 3.0 (2012)

- MPI Implementations
  - OpenMPI (www.open-mpi.org)
    - OpenMPI 1.4.x, 1.6.x
  - MPICH2 (www.mpich.org)
    - MPICH2, MVAPICH2, IntelMPI

Message Passing Interface (MPI)

- MPI is a **Library for** Message-Passing
  - Not built in to compiler
  - Function calls that can be made from any compiler, many languages
  - Just link to it
  - Wrappers: mpicc, mpif77
- MPI is a Library for **Message-Passing**
  - Communication/coordination between tasks done by sending and receiving messages.
  - Each message involves a function call from each of the programs.
  - Three basic sets of functionality
    - **Pairwise** or **point-to-point** communication via messages
    - **Collective** operations via messages
    - Efficient routines for getting data from memory into messages and vice versa

Size of MPI Library

- Many, many functions (>200)
- Not nearly so many concepts
- You will probably use 10 – 12 for now, use more as needed.
- Below are the most frequently used

```
MPI_Init ()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Send()
MPI_Recv()
MPI_Finalize()
```

Topics

- Introduction
- MPI Model and Basic Calls
- MPI Communication
- Summary

MPI : Basics

- Every MPI program must contain the preprocessor directive

```
#include <mpi.h>
```

- The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.
- mpi.h is usually found in the “include” directory of most MPI installations.

```
...
#include <mpi.h>
...
MPI_Init(&Argc,&Argv);
...
...
MPI_Finalize();
...
```

MPI: Initializing MPI Environment

Function:	MPI_init()
int MPI_Init(int *argc, char **argv)	

Description:

Initializes the MPI execution environment. MPI_init() must be called before any other MPI functions can be called and it should be called only once. It allows systems to do any special setup so that MPI Library can be used. ***argc*** is a pointer to the number of arguments and ***argv*** is a pointer to the argument vector. On exit from this routine, all processes will have a copy of the argument list.

```
...
#include “mpi.h”
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Init.html

MPI: Terminating MPI Environment

Function: MPI_Finalize()

int MPI_Finalize()

Description:

Terminates MPI execution environment. All MPI processes must call this routine before exiting. MPI_Finalize() need not be the last executable statement or even in main; it must be called at some point following the last call to any other MPI function.

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Finalize.html

MPI Hello World

- C source file for a simple MPI Hello World

```
#include <iostream>
#include <mpi.h>

int main( int argc, char **argv)
{
     int err;
     err  =  MPI_Init(&argc,&argv);
     std::cout << "Hello World!\n";
     err  =  MPI_Finalize();
     return 0;
}
```

Include header files

Initialize MPI Context

Finalize MPI Context

Building an MPI Executable

- Not specified in the standard
  - Two normal options, dependent on implementation

  - Library version

    ```
    g++ -Iheaderdir -Llibdir mpicode.cc -lmpich
    ```

    - User knows where header file and library are, and tells compiler

  - Wrapper version

    ```
    mpi++ -o executable mpicode.cc
    ```

    - Does the same thing, but hides the details from the user

  - You can do either one, but don't try to do both!

Building an MPI Executable

- Library version

```
g++ -m64 -O2 -fPIC -Wl,-z,noexecstack -o hello
hello.cc -I/usr/include/mpich2-x86_64
-L/usr/lib64/mpich2/lib -L/usr/lib64/mpich2/lib
-Wl,-rpath,/usr/lib64/mpich2/lib -lmpich -lopa
-lpthread -lrt
```

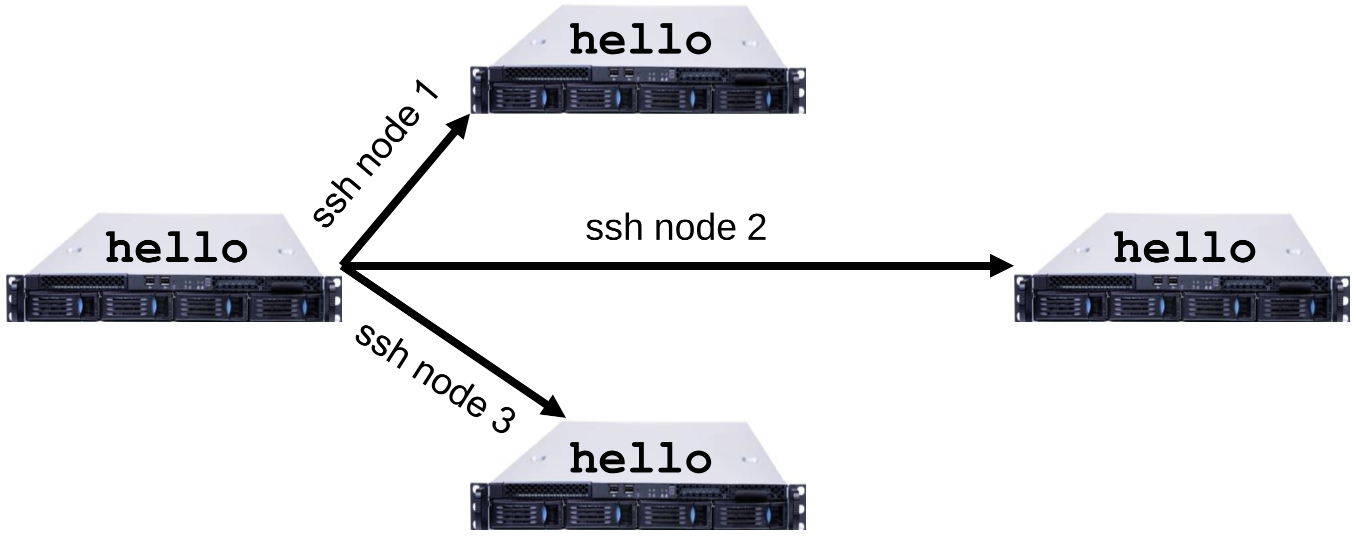
OR

- Wrapper version

```
Mpic++ -o hello hello.cc
```

Running an MPI Executable

- What **`mpirun`** does
  - Launches n processes, assigns each an MPI rank and starts the program
  - For multinode run, has a list of nodes, ssh's to each node and launches the program



Running an MPI Executable

- Number of processes
  - Number of processes to use is always equal to the number of processors
  - But not necessarily
  - On your nodes what happens when you run this?

```
$ mpirun -p 24 hello
```

MPI Communicators

- Communicator is an internal object
  - MPI provides functions to interact with it
- Default communicator is MPI_COMM_WORLD
  - All processes are member of it
  - It has a size (the number of processes)
  - Each process has a rank within it
  - Can think of it as an ordered list of processes
- Additional communicator can co-exist
- A process can belong to more than one communicator
- Within a communicator, each process has a unique rank

A Sample MPI program (Fortran)

```
...
INCLUDE mpif.h
...
CALL MPI_INITIALIZE(IERR)
...
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, SIZE, IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
...
CALL MPI_FINALIZE(IERR)
...
```

A Sample MPI program (C)

```
...
#include <mpi.h>
...
err = MPI_Init(&Argc,&Argv);
...
err = MPI_Comm_size(MPI_COMM_WORLD, &size);
err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

A Sample MPI program (C)

- Mandatory in any MPI code
- Defines MPI-related parameters

```
...
#include <mpi.h>
...
err = MPI_Init(&Argc,&Argv);
...
err = MPI_Comm_size(MPI_COMM_WORLD, &size);
err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

A Sample MPI program (C)

- Must be called in any MPI code by all processes once and only once before any other MPI calls

```
...
#include <mpi.h>
...
err = MPI_Init(&Argc,&Argv);
...
err = MPI_Comm_size(MPI_COMM_WORLD, &size);
err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

A Sample MPI program (C)

- Must be called in any MPI code by all processes once and only once after all other MPI calls

```
...
#include <mpi.h>
...
err = MPI_Init(&Argc,&            ;
...
err = MPI_Comm_size    I_COMM_WORLD, &size);
err = MPI_Comm_ra     (MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

A Sample MPI program (C)

- Returns the number of processes (size) in the communicator (MPI_COMM_WORLD)

```
...
#include <mpi.h>
...
err = MPI_Init(&Argc       rgv);
...
err = MPI_Comm_size(MPI_COMM_WORLD, &size);
err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

A Sample MPI program (C)

- Returns the rank of this process (rank) in the communicator (MPI_COMM_WORLD)
- Has unique return value per process

```
...
#include <mpi.h>
...
err = MPI_Init(&Argc       gv);
...
err = MPI_Comm_size(MPI_COMM_WORLD, &size);
err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

A Complete MPI Example

```
#include <iostream>
#include <mpi.h>
int main(int argc, char **argv)
{
  int err, size, rank;

  err = MPI_Init(&argc, &argv);      /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    std::cout<<"MPI initialization failed!\n";
    return(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) { /* root process */
    std::cout<<"I am the root\n";
  } else {
    std::cout<<"I am not the root\n";
  }
  std::cout<<"My rank is"<<rank<<"of"<<size
  <<"\n";
  err = MPI_Finalize();
  return(0);
}
```

Output (with 3 processes):

```
I am not the root
My rank is 2 of 3
I am the root
My rank is 0 of 3
I am not the root
My rank is 1 of 3
```

Topics

Point-to-point Communication

- How two processes interact
- Most flexible communication in MPI
- Two basic functions
  - Send and receive
- **With these two functions, and the four functions we already know, you can do everything in MPI**
  - But there's probably a better way to do a lot things, using other functions

MPI: Send & Receive

```
err = MPI_Send(sendptr, count, MPI_TYPE,
destination,tag, Communicator)
err = MPI_Recv(rcvptr, count, MPI_TYPE,
source, tag,Communicator, MPI_status)
```

- `sendptr/rcvptr:` pointer to message
- `count:` number of elements in ptr
- `MPI TYPE:` one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- `destination/source:` rank of sender/reciever
- `tag:` unique id for message pair
- `Communicator:` MPI COMM WORLD or user created
- `status:` receiver status (error, source, tag)

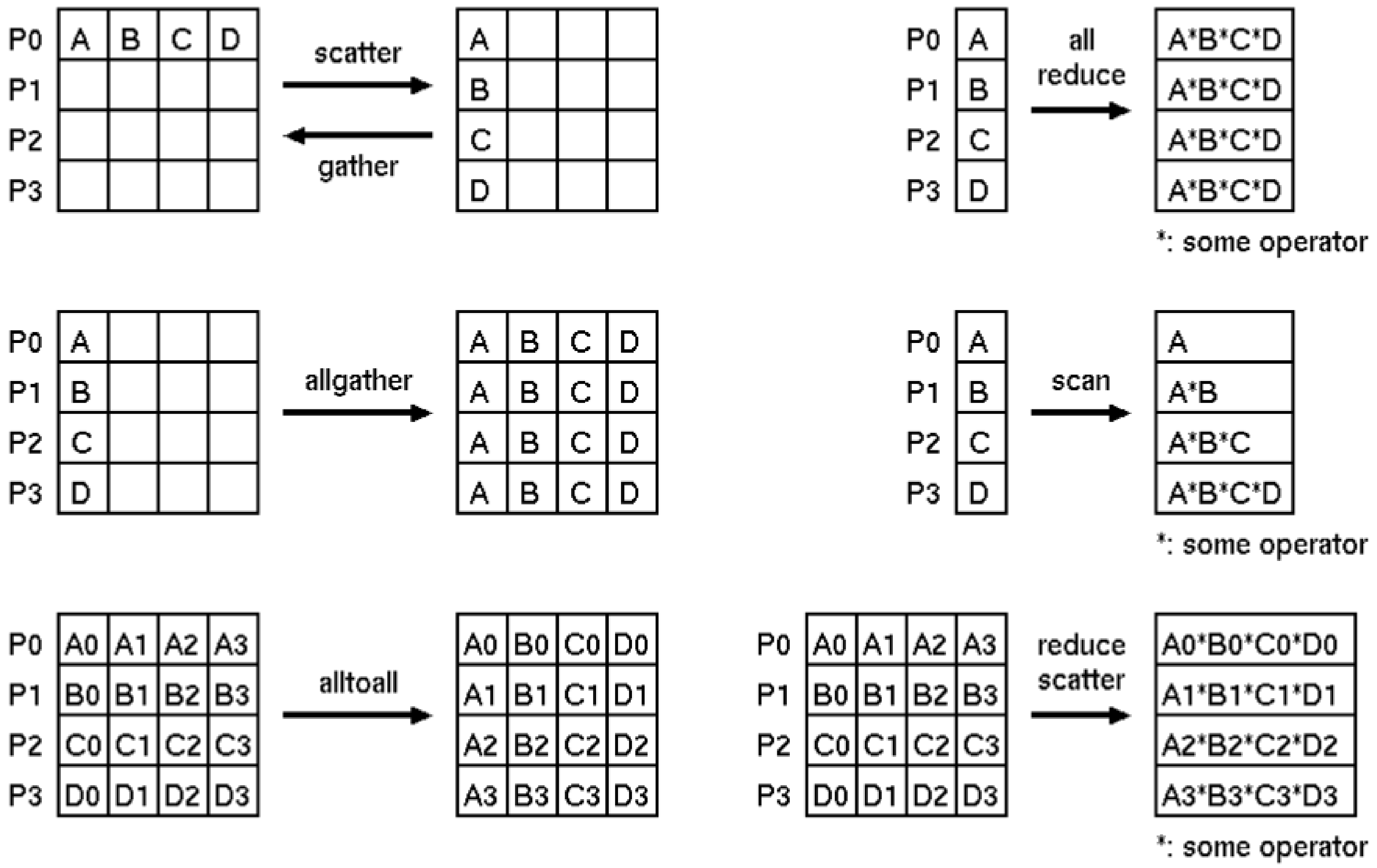
Collective Communication

- How a group of processes interact
  - “group” here means processes in a communicator
  - A group can be as small as one or two processes
    - One process communicating with itself isn't interesting
    - Two processes communicating are probably better handled through point-to-point communication
- Most efficient communication in MPI
- All collective communication is blocking

Collective Communication Types

- Three types of collective communication
  - Synchronization
    - Example: **barrier**
  - Data movement
    - Examples: **broadcast, gather**
  - Reduction (computation)
    - Example: **reduce**
- All of these could also be done with point-to-point communications
  - Collective operations give better performance and better productivity

Collective Operations



Topics

- Introduction
- MPI Model and Basic Calls
- MPI Communication
- Summary

Summary

- Basic MPI Components
  - **`#include <mpi.h>`** : MPI library details
  - **`MPI Init(&argc, &argv);`** : MPI Initialization, must come first
  - **`MPI Finalize()`**: Finalizes MPI, must come last
  - **`err`** : Returns error code

- Communicator Components
  - **`MPI_COMM_WORLD`** : Global Communicator
  - **`MPI_Comm_rank(MPI_COMM_WORLD, &rank)`**: Get current tasks rank
  - **`MPI_Comm_size(MPI_COMM_WORLD, &size)`**: Get communicator size

- MPI Communication
  - Point-to-point: (Send & Receive)
  - Collective: (Synchronization, Data movement & Reduction)