University of Notre Dame

MPI Tutorial

Part 1

Introduction

Laboratory for Scientific Computing

Fall 1998

http://www.lam-mpi.org/tutorials/nd/

lam@lam-mpi.org

MPI Tutorial

- Tutorial Instructors
  - M.D. McNally
  - Jeremy Siek
- Tutorial TA's
  - Jeff Squyres
  - Kinis Meyer
- Many Thanks to
  - Purushotham V. Bangalore, Shane Hebert, Andrew Lumsdaine, Boris Protopopov, Anthony Skjellum, Jeff Squyres, Brian McCandless
  - Nathan Doss, Bill Gropp, Rusty Lusk

Recommended Reading

- Tutorial Home Page:

  `http://www.lam-mpi.org/tutorials/nd/`

- LAM/MPI ND User Guide:

  `http://www.lam-mpi.org/tutorials/lam/`

- MPI FAQ
- *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum
- *MPI: The Complete Reference - 2nd Edition Volume 2 - The MPI-2 Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir
- *MPI Annotated Reference Manual*, by Snir, *et al*

- The LAM companion to “Using MPI...” by Zdzislaw Meglicki
- *Designing and Building Parallel Programs* by Ian Foster.
- A Tutorial/User’s Guide for MPI by Peter Pacheco `(ftp://math.usfca.edu/pub/MPI/mpi.guide.ps)`
- The MPI standard and other information is available at `http://www.mpi-forum.org/`, as well as the source for several implementations.

Course Outline

- Part 1 - Introduction
  - Basics of Parallel Computing
  - Six-function MPI
  - Point-to-Point Communications
  - Collective Communication
- Part 2 - High-Performance MPI
  - Non-blocking Communication
  - Persistent Communication
  - User-defined datatypes
  - MPI Idioms for High-performance

Course Outline cont.

- Part 3 - Advanced Topics
  - Communicators
  - Topologies
  - Attribute caching

Section I

Basics of Parallel Computing

Background

- Parallel Computing
- Communicating with other processes (Message-passing Paradigm)
- Cooperative operations
- One-sided operations
- MPI

Types of Parallel Computing

- Flynn's taxonomy (hardware oriented)

  SISD : **S**ingle **I**nstruction, **S**ingle **D**ata

  SIMD : **S**ingle **I**nstruction, **M**ultiple **D**ata

  MISD : **M**ultiple **I**nstruction, **S**ingle **D**ata

  MIMD : **M**ultiple **I**nstruction, **M**ultiple **D**ata

Types of Parallel Computing

- A programmer-oriented taxonomy

  Data-parallel : Same operations on different data (SIMD)

  Task-parallel : Different programs, different data

  MIMD : Different programs, different data

  SPMD : Same program, different data

  Dataflow : Pipelined parallelism

- All use different data for each worker.
- SPMD and MIMD are essentially the same because any MIMD can be made SPMD.
- MPI is for SPMD/MIMD.
- HPF is an example of a SIMD interface.

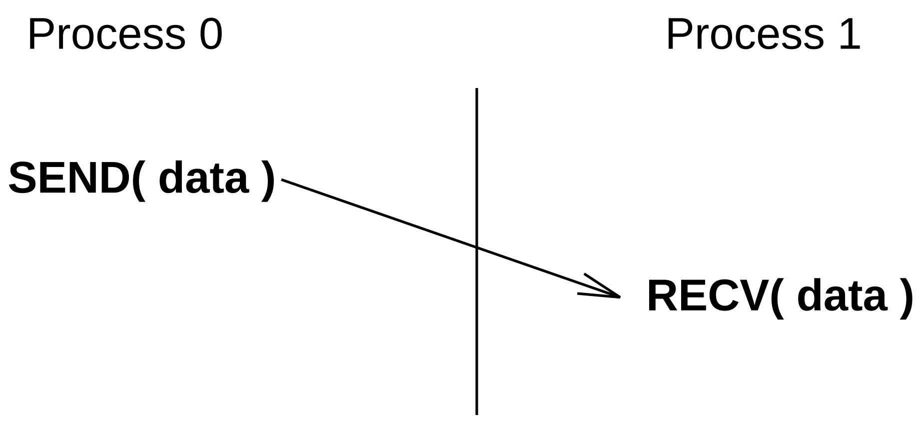
Hardware Models

- Distributed memory (e.g., Intel Paragon, IBM SP, workstation network)
- Shared memory (e.g., SGI Origin 2000, Cray T3D)
- Either may be used with SIMD or MIMD software models.
- Distributed shared memory (e.g., HP/Convex Exemplar) — memory is physically distributed but logically shared

But actually, all memory is distributed.

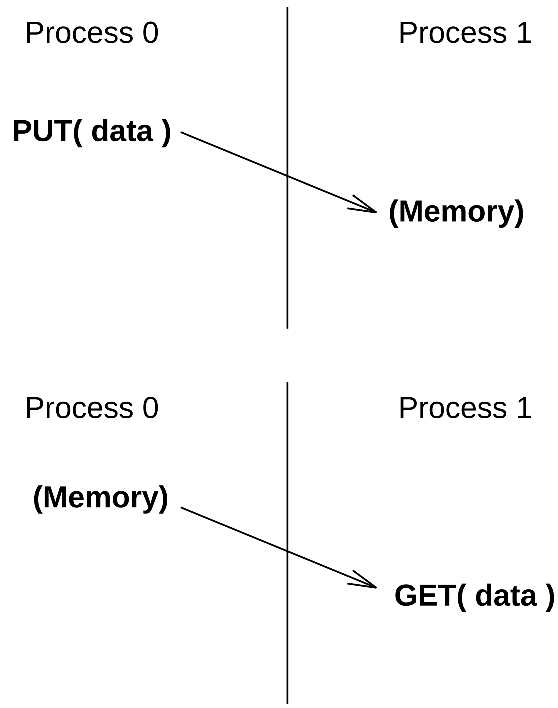
Communicating: Cooperative Operations

- Message-passing is an approach that makes the exchange of data cooperative.
- Data must both be explicitly sent and received.
- An advantage is that any change in the *receiver's* memory is made with the receiver's participation.



Communicating: One-Sided Operations

- One-sided operations between parallel processes include remote memory reads and writes (gets and puts)
- Advantage: data can be accessed without waiting for another process
- Disadvantage: synchronization may be easy or difficult



Lab – Classroom Parallel Computing Exercise

- Goal: Hands-on experience with parallel computing
- Take a piece of paper.
- Algorithm:
  - Write down the number of neighbors that you have
  - Compute average of your neighbor's values
  - Make that your new value
  - Repeat until done

Lab – Questions

- Questions:
  1. How do you get values from your neighbors?
  2. Which step or iteration do they correspond to?
     – Do you know?
     – Do you care?
  3. How do you decide when you are done?

What is MPI?

- A *message-passing library specification*
  - Message-passing model
  - Not a compiler specification
  - Not a specific product
- Specified in C, C++, and Fortran 77
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Two parts: MPI-1 (1.2) and MPI-2 (2.0)

What is MPI? (cont.)

- Designed to permit (unleash?) the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

Motivation for Message Passing

- Message Passing is now mature as programming paradigm
  - Well understood
  - Efficient match to hardware
  - Many applications
- Vendor systems were not portable
- Portable systems are mostly research projects
  - Incomplete
  - Lack vendor support
  - Not at most efficient level

Motivation (cont.)

- Few systems offer the full range of desired features.
  - Modularity (for libraries)
  - Access to peak performance
  - Portability
  - Heterogeneity
  - Safe communication (lexical scoping)
  - Subgroups
  - Topologies
  - Performance measurement tools

Features of MPI

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols on some systems), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

Features of MPI (cont.)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)
- Profiling
  - Hooks allow users to intercept MPI calls to install their own tools
- Environmental
  - Inquiry
  - Error control

Features Not in MPI-1

- Non-message-passing concepts not included:
  - Process management
  - Remote memory transfers
  - Active messages
  - Threads
  - Virtual shared memory
- MPI does not address these issues, but has tried to remain compatible with these ideas (e.g., thread safety as a goal, etc.)
- Some of these features are in MPI-2

Is MPI Large or Small?

- MPI is large. MPI-1 is 128 functions. MPI-2 is 152 functions.
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs can be written with just 6 basic functions.
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.

Where to Use MPI?

- You need a portable parallel program
- You are writing a parallel library
- You have irregular or dynamic data relationships that do not fit a data parallel model
- You care about performance

Where *not* to Use MPI

- You can use HPF or a parallel Fortran 90
- You don’t need parallelism at all
- You can use libraries (which may be written in MPI)
- You need simple threading in a slightly concurrent environment

Section II

Six-function MPI

Getting Started

- Writing MPI programs
- Compiling and linking
- Running MPI programs

Simple MPI C Program

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

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  printf("Hello world\n");
  MPI_Finalize();
  return 0;
}
```

Simple MPI C++ Program

```
#include <iostream.h>
#include "mpi++.h"    //Should really be "mpi.h"

int main(int argc, char **argv)
{
  MPI::Init(argc, argv);
  cout <<  "Hello world" << endl;
  MPI::Finalize();
  return 0;
}
```

Simple MPI Fortran Program

```
program main
include 'mpif.h'
integer ierr

call MPI_INIT(ierr)
print *, 'Hello world'
call MPI_FINALIZE(ierr)

end
```

Commentary

- `#include "mpi.h"` or `#include "mpif.h"` provides basic MPI definitions and types
- All non-MPI routines are local; thus the `printf()` runs on each process

The sample programs have been kept as simple as possible by assuming that all processes can do output. Not all parallel systems provide this feature – MPI provides a way to handle this case.

Commentary

- Starting MPI

```
int MPI_Init(int *argc, char **argv)

void MPI::Init(int& argc, char**& argv)

MPI_INIT(IERR)
INTEGER IERR
```

Commentary

- Exiting MPI

```
int MPI_Finalize(void)

void MPI::Finalize()

MPI_FINALIZE(IERR)
INTEGER IERR
```

C/C++ and Fortran Language Considerations

- `MPI_INIT`: The C version accepts the `argc` and `argv` variables that are provided as arguments to `main()`
- Error codes: Almost all MPI Fortran subroutines have an integer return code as their last argument. Almost all C functions return an integer error code.
- Bindings
  - C: All MPI names have an `MPI_` prefix. Defined constants are in all capital letters. Defined types and functions have one capital letter after the prefix; the remaining letters are lowercase.
  - C++: All MPI functions and classes are in the `MPI` namespace, so instead of refering to `X` as `MPI_X` as one would in C, one writes `MPI::X`.

C/C++ and Fortran Language Considerations (cont.)

- Bindings (cont.)
  - Fortran: All MPI names have an `MPI_` prefix, and all characters are capitals
- Types: Opaque objects are given type names in C. In C++ the opaque objects are C++ object, defined by a set of MPI classes. In Fortran, opaque objects are usually of type `INTEGER` (exception: binary-valued variables are of type `LOGICAL`)
- Inter-language interoperability is not guaranteed (e.g., Fortran calling C or vice-versa)
- Mixed language programming is OK as long as only C or Fortran uses MPI

Running MPI Programs

- On many platforms MPI programs can be started with '`mpirun`'.

  ```
  mpirun N -w hello
  ```

- '`mpirun`' is not part of the standard, but some version of it is common with several MPI implementations. The version shown here is for the LAM implementation of MPI.

  Just as Fortran does not specify how Fortran programs are started, MPI does not specify how MPI programs are started.

Finding Out About the Parallel Environment

- Two of the first questions asked in a parallel program are as follows:
  1. "How many processes are there?"
  2. "Who am I?"
- "How many" is answered with `MPI_COMM_SIZE`; "Who am I" is answered with `MPI_COMM_RANK`.
- The rank is a number between zero and `(SIZE - 1)`.

A Second MPI C Program

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

int main(int argc, char **argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello world! I am %d of %d\n",rank,size);
  MPI_Finalize();
  return 0;
}
```

A Second MPI C++ Program

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

int main(int argc, char **argv)
{
  MPI::Init(argc, argv);
  int rank = MPI::COMM_WORLD.Get_rank();
  int size = MPI::COMM_WORLD.Get_size();
  cout << "Hello world! I am " << rank << " of "
       << size << endl;
  MPI::Finalize();
  return 0;
}
```

A Second MPI Fortran Program

```
program main
include 'mpif.h'
integer rank, size, ierr

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
print *, 'Hello world!  I am ', rank, ' of ', size
call MPI_FINALIZE(ierr)

end
```

MPI_COMM_WORLD

- Communication in MPI takes place with respect to *communicators* (more about communicators later)
- The `MPI_COMM_WORLD` communicator is created when MPI is started and contains all MPI processes
- `MPI_COMM_WORLD` is a useful default communicator — many applications do not need to use any other

LAM MPI

- LAM (Local Area Multicomputer)
  - Public domain implementation of MPI that runs on workstaion clusters
  - Originally written at the Ohio Supercomputing Center, now hosted at `www.lam-mpi.org`
- Need to setup your environment:

  `source ~ccse/mpi/lam_cshrc`

- Need a valid `$HOME/.rhosts` file
  - Copy from web page to `$HOME/.rhosts`
  - Replace `YOUR_AFS_ID` with your AFS id
  - `chmod 644 $HOME/.rhosts`

Introduction to LAM MPI

- Create a text file named hostfile with 4 machine names, one per line (put your hostname first)

  Example:

  ```
  austen.helios.nd.edu
  dickens.helios.nd.edu
  milton.helios.nd.edu
  moliere.helios.nd.edu
  ```

- To start up LAM:

  ```
  lamboot -v hostfile
  ```

- MPI programs can now be run

- When finished, be sure to **SHUT DOWN LAM!!**

  ```
  wipe -v hostfile
  ```

Introduction to LAM MPI (cont.)

- To run an MPI job:

  `mpirun [args] [processors] program [--[prog_args]]`

  – `[args]` can contain `-w` (wait), `-c2c` (faster messages)

  – `[processors]` can be either all processors (`N`), or a range of processors (e.g. `n0-1`)

- To see the status of a running MPI job:

  `mpitask`

- To see outstanding messages in MPI:

  `mpimsg`

- To kill a running MPI job:

  `lamclean -v`

Introduction to LAM MPI (cont.)

- General steps for running under LAM MPI:

Freq.	**Task**	**Command**
Once	Start up LAM MPI	`lamboot -v hostfile`
As needed	Run your program	`mpirun N /path/to/program`
As needed	“Clean up”	`lamclean -v`
Once	Shut down LAM MPI	`wipe -v hostfile`

- You must `lamboot` before any other LAM commands will work
- `lamclean` is used to “clean up” any residuals from an individual run
- Once you `wipe`, no LAM commands will work until you `lamboot` again

Lab – Getting Started

- Objective: Learn how to write, compile, and run a simple MPI program and become familiar with MPI.
- Compile and run the second "Hello world" program in your favorite language (see slides 38, 39, and 40). Try various numbers of processors.
  - Download the `Makefile` from the web page.
  - Be sure to name your file `lab1.c`, `lab1.cc`, or `lab1.f`
  - Compile with "`make lab1c`", "`make lab1cc`", or "`make lab1f`" (depending on your language).
  - Use `lamboot` to start LAM MPI.
  - Use `mpirun` to run your program (use `lamclean` if things go wrong).
  - Use `wipe` when all finished with LAM MPI.
- What does the output look like?

Programming Notes

- MPI C and F77 function index is at:

  `http://www.mpi-forum.org/docs/mpi-11-html/`
  `node182.html#Node182`

- Refer to this during labs for bindings, documents, etc.

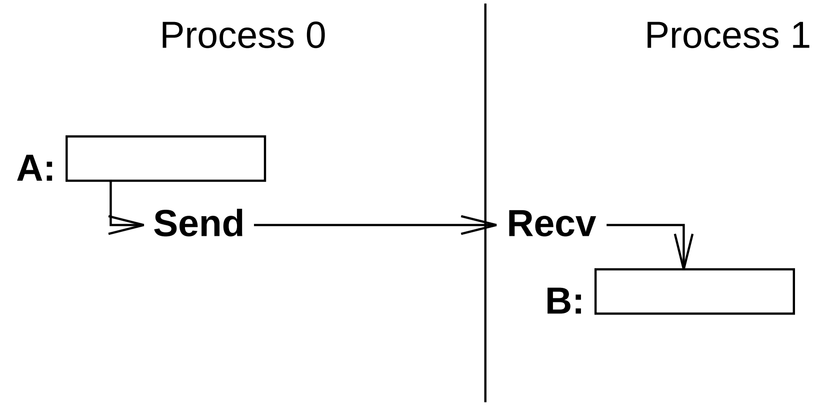
  `http://www.mpi-forum.org/docs/docs.html`

Section III

Point-to-Point Communications

Sending and Receiving Messages

- Basic message passing process



- Questions:
  - To whom is data sent?
  - Where is the data?
  - How much of the data is sent?
  - What type of the data is sent?
  - How does the receiver identify it?

Current Message-Passing

- A typical send might look like:

```
send(dest, address, length)
```

  - `dest` is an integer identifier representing the process to receive the message.
  - (`address, length`) describes a contiguous area in memory containing the message to be sent.

Traditional Buffer Specification

Sending and receiving only a contiguous array of bytes:

- Hides the real data structure from hardware which might be able to handle it directly
- Requires pre-packing of dispersed data
  - Rows of a matrix stored columnwise
  - General collections of structures
- Prevents communications between machines with different representations (even lengths) for same data type, except if user works this out

Generalizing the Buffer Description

- Specified in MPI by *starting address*, *datatype*, and *count*, where datatype is:
  - Elementary (all C and Fortran datatypes)
  - Contiguous array of datatypes
  - Strided blocks of datatypes
  - Indexed array of blocks of datatypes
  - General structure
- Datatypes are constructed recursively.
- Specifications of elementary datatypes allows heterogeneous communication.
- Elimination of length in favor of count is clearer.
- Specifying application-oriented layout of data allows maximal use of special hardware.

MPI C Datatypes

MPI datatype	C datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`

MPI C Datatypes (cont.)

MPI datatype	C datatype
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI C++ Datatypes

MPI datatype	C++ datatype
MPI::CHAR	signed char
MPI::SHORT	signed short int
MPI::INT	signed int
MPI::LONG	signed long int
MPI::UNSIGNED_CHAR	unsigned char
MPI::UNSIGNED_SHORT	unsigned short int
MPI::UNSIGNED	unsigned int

MPI C++ Datatypes (cont.)

MPI datatype	C++ datatype
`MPI::UNSIGNED_LONG`	`unsigned long int`
`MPI::FLOAT`	`float`
`MPI::DOUBLE`	`double`
`MPI::LONG_DOUBLE`	`long double`
`MPI::BYTE`	
`MPI::PACKED`	

MPI Fortran Datatypes

MPI datatype	**Fortran datatype**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Generalizing the Process Identifier

- `destination` has become `(rank, group)`.
- Processes are named according to their rank in the group
- Groups are enclosed in “communicators”
- `MPI_ANY_SOURCE` wildcard permitted in a receive.

Providing Safety

- MPI provides support for safe message passing (e.g. keeping user and library messages separate)
- Safe message passing
  - Communicators also contain "contexts"
  - Contexts can be envisioned as system-managed tags
- Communicators can be thought of as (`group, system-tag`)
- `MPI_COMM_WORLD` contains a "context" and the "group of all known processes"
- Collective and point-to-point messaging is kept separate by "context"

Identifying the Message

- MPI uses the word “tag”
- Tags allow programmers to deal with the arrival of messages in an orderly way
- MPI tags are guaranteed to range from 0 to 32767
- The range will always start with 0
- The upper bound may be larger than 32767. Section 7.1.1 of the standard discusses how to determine if an implementation has a larger upper bound
- `MPI_ANY_TAG` can be used as a wildcard value

MPI Basic Send/Receive

- Thus the basic (blocking) send has become:

```
MPI_SEND(start, count, datatype, dest, tag, comm)
```

- And the receive has become:

```
MPI_RECV(start, count, datatype, source, tag, comm,
         status)
```

- The source, tag, and count of the message actually received can be retrieved from `status`.
- For now, `comm` is `MPI_COMM_WORLD` or `MPI::COMM_WORLD`

MPI Procedure Specification

- MPI procedures are specified using a language independent notation.
- Procedure arguments are marked as

  IN: the call uses but does not update the argument

  OUT: the call may update the argument

  INOUT: the call both uses and updates the argument
- MPI functions are first specified in the language-independent notation
- ANSI C and Fortran 77 realizations of these functions are the language *bindings*

MPI Basic Send

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (nonnegative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

Bindings for Send

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm)

void MPI::Comm::Send(const void* buf, int count,
                     const MPI::Datatype& datatype,
                     int dent, int tag) const;

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COM, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERR
```

MPI Basic Receive

MPI_RECV(buf, count, datatype, src, tag, comm, status)

OUT	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (nonnegative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	src	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

Bindings for Receive

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

void MPI::Comm::Recv(void *buf, int count, const
                    Datatype & datatype, int source,
                    int tag, Status & status) const;

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
         STATUS, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
         STATUS(MPI_STATUS_SIZE), IERR
```

Getting Information About a Message

- The (non-opaque) `status` object contains information about a message

```
/* In C */
MPI_Status status;
MPI_Recv(..., &status);

recvd_tag    = status.MPI_TAG;
recvd_source = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);

/* In C++ */
MPI::Status status;
MPI::COMM_WORLD.Recv(..., status);

recvd_tag = status.Get_tag();
recvd_source = status.Get_source();
recvd_count = status.Get_count(datatype);
```

Getting Information About a Message (cont'd)

- The fields `status.MPI_TAG` and `status.MPI_SOURCE` are primarily of use when `MPI_ANY_TAG` and/or `MPI_ANY_SOURCE` is used in the receive
- The function `MPI_GET_COUNT` may be used to determine how much data of a particular type was received.

Simple C Example

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

int main(int argc, char **argv)
{
  int i, rank, size, dest;
  int to, src, from, count, tag;
  int st_count, st_source, st_tag;
  double data[100];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Process %d of %d is alive\n", rank, size);

  dest = size - 1;
  src = 0;

  if (rank == src) {
    to    = dest;
    count = 100;
    tag   = 2001;
    for (i = 0; i < 100; i++)
      data[i] = i;
    MPI_Send(data, count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD);
  }
  else if (rank == dest) {
    tag   = MPI_ANY_TAG;
    count = 100;
```

```
    from  = MPI_ANY_SOURCE;
    MPI_Recv(data, count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD,
             &status);

    MPI_Get_count(&status, MPI_DOUBLE, &st_count);
    st_source= status.MPI_SOURCE;
    st_tag= status.MPI_TAG;

    printf("Status info: source = %d, tag = %d, count = %d\n",
           st_source, st_tag, st_count);
    printf(" %d received: ", rank);
    for (i = 0; i < st_count; i++)
      printf("%lf ", data[i]);
    printf("\n");
  }

  MPI_Finalize();
  return 0;
}
```

Simple C++ Example

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

int main(int argc, char **argv)
{
  int i, rank, size, dest;
  int to, src, from, count, tag;
  int st_count, st_source, st_tag;
  double data[100];
  MPI::Status status;

  MPI::Init(argc, argv);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();

  cout << "Process " << rank << " of " << size << " is alive" << endl;

  dest = size - 1;
  src = 0;

  if (rank == src) {
    to    = dest;
    count = 100;
    tag   = 2001;
    for (i = 0; i < 100; i++)
      data[i] = i;
    MPI::COMM_WORLD.Send(data, count, MPI::DOUBLE, to, tag);
  }
  else if (rank == dest) {
    tag   = MPI::ANY_TAG;
```

```
    count = 100;
    from  = MPI::ANY_SOURCE;
    MPI::COMM_WORLD.Recv(data, count, MPI::DOUBLE, from, tag, status);
    st_count = status.Get_count(MPI::DOUBLE);
    st_source= status.Get_source();
    st_tag= status.Get_tag();

    cout << "Status info: source = " << st_source << ", tag = " << st_tag
 << ", count = " << st_count << endl;
    cout << rank << " received: ";
    for (i = 0; i < st_count; i++)
      cout << data[i] << " ";
    cout << endl;
  }

  MPI::Finalize();
  return 0;
}
```

Simple Fortran Example

```
      program main
      include 'mpif.h'

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(100)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src = 0
C
      if (rank .eq. src) then
         to     = dest
         count  = 100
         tag    = 2001
         do 10 i=1, 100
 10         data(i) = i
         call MPI_SEND(data, count, MPI_DOUBLE_PRECISION, to,
     +                   tag, MPI_COMM_WORLD, ierr)
      else if (rank .eq. dest) then
         tag   = MPI_ANY_TAG
         count = 100
         from  = MPI_ANY_SOURCE
         call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +                  tag, MPI_COMM_WORLD, status, ierr)
```

```
          call MPI_GET_COUNT(status, MPI_DOUBLE_PRECISION,
     +                          st_count, ierr)
          st_source = status(MPI_SOURCE)
          st_tag    = status(MPI_TAG)
C
          print *, 'Status info: source = ', st_source,
     +             ' tag = ', st_tag, ' count = ', st_count
          print *, rank, ' received', (data(i),i=1,100)
       endif

       call MPI_FINALIZE(ierr)
       end
```

Six Function MPI

MPI is very simple. These six functions allow you to write many programs:

MPI_INIT

MPI_COMM_SIZE

MPI_COMM_RANK

MPI_SEND

MPI_RECV

MPI_FINALIZE

More on LAM MPI

- The `lam_cshrc` script sets up a special alias: `lamrun`
  - For example: `lamrun program`
  - Shortcut for: `mpirun -c2c -O -w N -D ‘pwd‘/program`
- New general steps for running under LAM:

Freq.	**Task**	**Command**
Once	Start up LAM MPI	`lamboot -v hostfile`
As needed	Run your program	`lamrun program`
As needed	“Clean up”	`lamclean -v`
Once	Shut down LAM MPI	`wipe -v hostfile`

Lab – Message Ring

- Objective: Pass a message around a ring `n` times. Use blocking `MPI_SEND` and `MPI_RECV`.
  - Write a program to do the following:
    - Process 0 should read in a single integer (> 0) from standard input
    - Use MPI send and receive to pass the integer around a ring
    - Use the user-supplied integer to determine how many times to pass the message around the ring
    - Process 0 should decrement the integer each time it is received.
    - Processes should exit when they receive a "0".

Section IV

Collective Communication

Collective Communications in MPI

- Communication is coordinated among a group of processes, as specified by communicator
- Message tags are not used.
- All collective operations are blocking
- All processes in the communicator group must call the collective operation
- Three classes of collective operations:
  - Data movement
  - Collective computation
  - Synchronization

Pre-MPI Message-Passing

- A typical (pre-MPI) global operation might look like:

```
broadcast(type, address, length)
```

- As with point-to-point, this specification is a good match to hardware and easy to understand
- But also too inflexible

MPI Basic Collective Operations

- Two simple collective operations:

```
MPI_BCAST(start, count, datatype, root, comm)

MPI_REDUCE(start, result, count, datatype,
           operation, root, comm)
```

- The routine `MPI_BCAST` sends data from one process to all others.
- The routine `MPI_REDUCE` combines data from all processes returning the result to a single process.

MPI_BCAST

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer
IN	count	number of entries in buffer
IN	datatype	data type of buffer
IN	root	rank of broadcast root
IN	comm	communicator

MPI_BCAST Binding

```
int MPI_Bcast(void* buffer, int count,
              MPI_Datatype datatype, int root,
              MPI_Comm comm )

void MPI::Comm::Bcast(void* buffer, int count,
                      const MPI::Datatype& datatype,
                      int root) const = 0

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT,
          COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_REDUCE

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer
IN	count	of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	reduce operation
IN	root	rank of root process
IN	comm	communicator

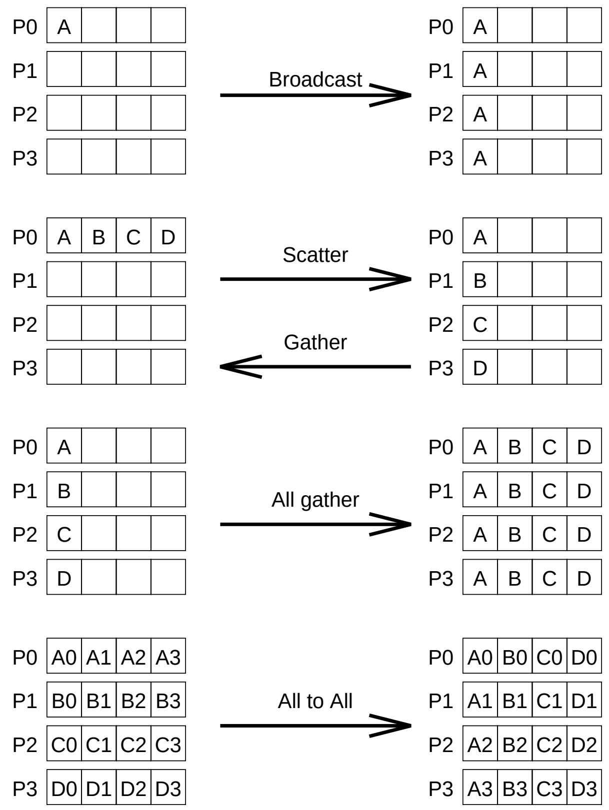
Binding for MPI_REDUCE

```
int MPI_Reduce(void* sendbuf, void* recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)

void MPI::Comm::Reduce(const void* sendbuf, void*
                       recvbuf, int count, const
                       MPI::Datatype& datatype,
                       const MPI::Op& op,
                       int root) const = 0

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
           ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

MPI Basic Collective Operations

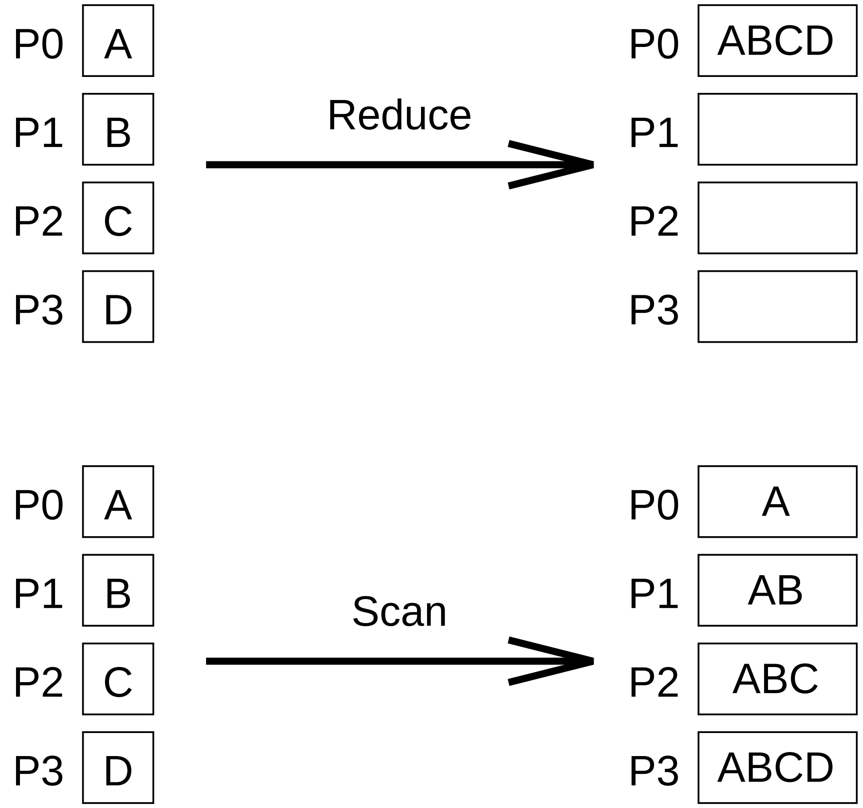
- Broadcast and reduce are very important mathematically
- Many scientific programs can be written with just
  MPI_INIT
  MPI_COMM_SIZE
  MPI_COMM_RANK
  MPI_SEND
  MPI_RECV
  MPI_BCAST
  MPI_REDUCE
  MPI_FINALIZE
- Some won't even need send and receive

Available Collective Patterns



- Schematic representation of collective data movement in MPI

Available Collective Computation Patterns



- Schematic representation of collective data movement in MPI

MPI Collective Routines

- Many routines:

MPI_ALLGATHER	MPI_ALLGATHERV	MPI_ALLREDUCE
MPI_ALLTOALL	MPI_ALLTOALLV	MPI_BCAST
MPI_GATHER	MPI_GATHERV	MPI_REDUCE
MPI_REDUCESCATTER	MPI_SCAN	MPI_SCATTER
MPI_SCATTERV		

- `All` versions deliver results to all participating processes.
- `V` versions allow the chunks to have different sizes.
- `MPI_ALLREDUCE`, `MPI_REDUCE`, `MPI_REDUCESCATTER`, and `MPI_SCAN` take both built-in and user-defined combination functions.

Built-in Collective Computation Operations

MPI Name	Operation
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	Product
`MPI_SUM`	Sum
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or (xor)
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise xor
`MPI_MAXLOC`	Maximum value and location
`MPI_MINLOC`	Minimum value and location

Defining Your Own Collective Operations

```
MPI_OP_CREATE(user_function, commute, op)
MPI_OP_FREE(op)
```

```
user_function(invec, inoutvec, len, datatype)
```

The user function should perform:

```
inoutvec[i] = invec[i] op inoutvec[i];
```

for `i` from `0` to `len-1`.

`user_function` can be non-commutative (e.g., matrix multiply).

MPI_SCATTER

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer
IN	sendcount	number of elements sent to each process
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process
IN	comm	communicator

MPI_SCATTER Binding

```
int MPI_Scatter(void* sendbuf, int sendcount,
                MPI_Datatype sendtype, void* recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)

void MPI::Comm::Scatter(const void* sendbuf,
                       int sendcount,
                       const MPI::Datatype& sendtype,
                       void* recvbuf, int recvcount,
                       const MPI::Datatype& recvtype,
                       int root) const = 0
```

MPI_SCATTER Binding (cont.)

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
            RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR
```

MPI_GATHER

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements for any single receive
IN	recvtype	data type of recv buffer elements
IN	root	rank of receiving process
IN	comm	communicator

MPI_GATHER Binding

```
int MPI_Gather(void* sendbuf, int sendcount,
               MPI_Datatype sendtype, void* recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)

void MPI::Comm::Gather(const void* sendbuf,
                      int sendcount,
                      const MPI::Datatype& sendtype,
                      void* recvbuf, int recvcount,
                      const MPI::Datatype& recvtype,
                      int root) const = 0
```

MPI_GATHER Binding (cont.)

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
           RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR
```

Synchronization

`MPI_BARRIER(comm)`

Function blocks until all processes in “comm” call it

```
int MPI_Barrier(MPI_Comm comm )

void Intracomm::Barrier() const

MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```

Simple C Example

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

main(int argc, char **argv)
{
  int rank, size, myn, i, N;
  double *vector, *myvec, sum, mysum, total;

  MPI_Init(&argc, &argv );

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* In the root process read the vector length, initialize
     the vector and determine the sub-vector sizes */
  if (rank == 0) {
    printf("Enter the vector length : ");
    scanf("%d", &N);
    vector = (double *)malloc(sizeof(double) * N);
    for (i = 0, sum = 0; i < N; i++)
      vector[i] = 1.0;
    myn = N / size;
  }

  /* Broadcast the local vector size */
  MPI_Bcast(&myn, 1, MPI_INT, 0, MPI_COMM_WORLD );
  /* allocate the local vectors in each process */
  myvec = (double *)malloc(sizeof(double)*myn);
  /* Scatter the vector to all the processes */
```

```
  MPI_Scatter(vector, myn, MPI_DOUBLE, myvec, myn, MPI_DOUBLE,
      0, MPI_COMM_WORLD );

  /* Find the sum of all the elements of the local vector */
  for (i = 0, mysum = 0; i < myn; i++)
    mysum += myvec[i];

  /* Find the global sum of the vectors */
  MPI_Allreduce(&mysum, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );

  /* Multiply the local part of the vector by the global sum */
  for (i = 0; i < myn; i++)
    myvec[i] *= total;

  /* Gather the local vector in the root process */
  MPI_Gather(myvec, myn, MPI_DOUBLE, vector, myn, MPI_DOUBLE,
     0, MPI_COMM_WORLD );

  if (rank == 0)
    for (i = 0; i < N; i++)
      printf("[%d] %f\n", rank, vector[i]);

  MPI_Finalize();
  return 0;
}
```

Lab – Image Processing – Sum of Squares

- Objective: Use collective operations to find the root mean square of the pixel values in an image
- Write a program to do the following:
  - Process 0 should read in an image (using provided functions)
  - Use collective operations to distribute the image among the processors
  - Each processor should calculate the sum of the squares of the values of its sub-image
  - Use a collective operation to calculate the global sum of squares
  - Process 0 should return the global root mean square