Introduction to MPI

Fall 2010

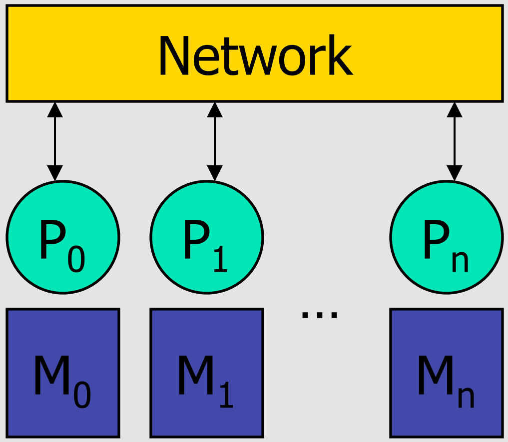
Types of programming models

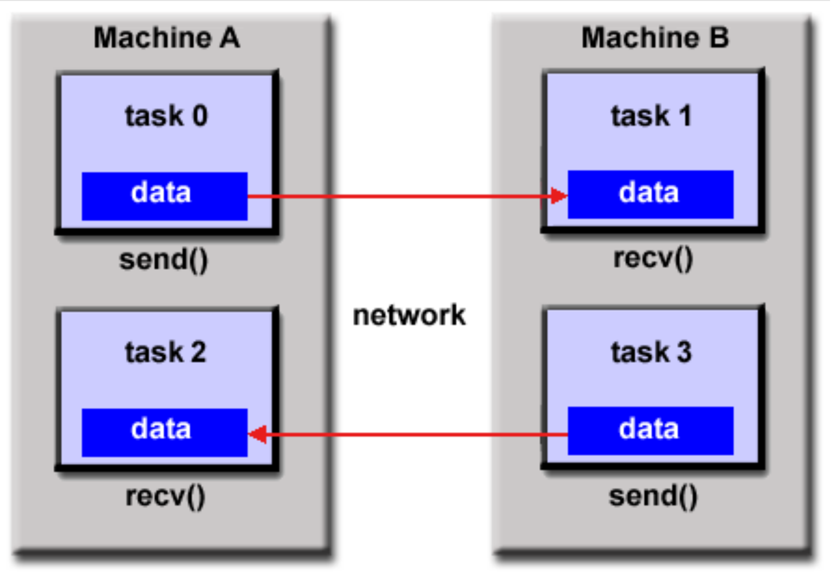
- Shared Memory
  - Global memory space
- Data Parallel
  - Shared or Distributed memory splitting a larger data structure
- Message Passing
  - MPI, PVM, libraries using message passing
- Threads
  - One process, one memory space, multiple threads
- Hybrid
  - Ex: Threads and Message Passing

- Above models are NOT specific to a particular type of machine or memory architecture
  - can (theoretically) be implemented on any underlying hardware.

Message Passing Model

- In the message passing model, each process has its own memory, and messages are sent between processes to exchange data over a network.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.
- Suitable mainly for SPMD/MIMD machines





Outline

- Background of MPI
- Overview of MPI functions
- Point-to-point communication
- Collective communication

What?: Message Passing Interface

- A message-passing library specification
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- Standards (NOT implementation)
- Designed for high performance on both massively parallel machines and on workstation clusters.
- Is widely available, with both free available (e.g., MPICH, Open MPI) and vendor-supplied implementations
- Was developed by a broadly based vendors, implementors and users

Who? MPI Forum

- Early vendor systems were not portable
- Early portable systems (PVM etc) were mainly research efforts
  - Did not address the full spectrum of issues
  - Lacked vendor support
  - Were not implemented at the most efficient level

When? (1992) – 1994 – Now

- MPI 1.0, MPI 1.2, MPI 2
- The MPI Forum organized in 1992 with broad participation by
  - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - Portability library writers: PVM, p4
  - Users: application scientists and library writers

How?

- A set of processes communicate by send/recv msgs
- Each process computes its local data

More Features:

- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Collective operations for scalable global communications
- …

Why?

Portability; Efficiency; Functionality

- Goal of large-scale scientific computing:
  - deliver computing performance to applications
- Deliverable computing power (in flops):
  - Pflops
- Independent research projects contribute new ideas to programming modes, languages, and libraries
  - Most make a prototype available and encourage use by others
  - Users require commitment, support, portability
  - Not all research groups can provide this
- Failure to achieve critical mass of users can limit impact of research

Where?

- The Standard itself:

  http://www.mpi-forum.org

- Info:

  http://www.mcs.anl.gov/mpi

- Implementations:
  - MPICH: http://www.mcs.anl.gov/mpi/mpich
  - Open MPI: http://www.open-mpi.org/

- MPI implementations
  - MPICH:
    MPICH2, MPICH-GM, MPICH-G2, MPICH-VMI,
    MPICH for Windows
  - LAM/MPI
  - Open MPI (combine FT-MPI, LAM/MPI etc)

- Programming languages
  - Fortran
  - C
  - C++
  - Others: Python

Programming With MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
    - MPI module for Fortran 90 (optional)

- First Program:
  - Create 3 processes in a simple MPI job
  - Write out process number

Example: ~mpich2-1.0.7/examples/hellow.c

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

int main( int argc, char *argv[] )
{
    int rank;
    int size;

    MPI_Init( 0, 0 );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?

- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

Program Basis

- #include "mpi.h"
- How to compile
  - mpicc –o <program_name> <src1> <src2> …
    - e.g. mpicc –o hellow hellow.c

- How to run (on a single machine)
  - mpiexec –n <proc_num> <program>
    - e.g. mpiexec –n 3 ./hellow

Basic concepts

- Process can be collected into groups
- Each message is sent in a context, and must be received in the same context
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called MPI_COMM_WORLD

Basic concepts

- MPI datatype
  - Predefined:

    MPI_INT, MPI_FLOAT, MPI_DOUBLE
  - User-defined
- Point-to-point message passing
  - Message tag
- Collective message passing
  - One-to-all, all-to-one, all-to-all

MPI functions overview:
more than 125 functions

MPI is simple:

many parallel programs can be written using just these six functions, only two of which are non-trivial:

- MPI_Init: initiate a MPI computation
- MPI_Finalize: terminate an MPI computation
- MPI_Comm_Size: determine number of processes
- MPI_Comm_Rank: determine my process id

- MPI_Send: send a message
- MPI_Recv: receive a message

MPI functions overview: more than 125 functions

MPI is simple:
alternative set of 6 functions:

- MPI_Init: initiate a MPI computation
- MPI_Finalize: terminate an MPI computation
- MPI_Comm_Size: determine number of processes
- MPI_Comm_Rank: determine my process id

- MPI_BCAST: broadcast a message to all processes
- MPI_REDUCE: reduce values on all processes to a single value

Basic functions

- int MPI_Init(int *argc, char ***argv)
- int MPI_Finalize()

- int MPI_Comm_size (MPI_Comm comm, int *size)
  - ret = MPI_Comm_size(MPI_COMM_WORLD, &size);
- int MPI_Comm_rank (MPI_Comm comm, int *rank)
  - ret = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

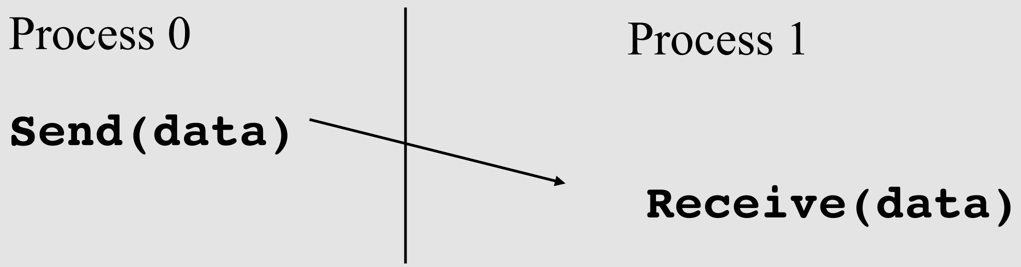
- int MPI_Barrier (MPI_Comm comm)
  - ret = MPI_Barrier(MPI_COMM_WORLD);

Point-to-point communication

- Blocking message passing
  - int MPI_Send(void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm)
  - int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)
- Non-blocking message passing
  - int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Request *request)
  - int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Request *request)
  - int MPI_Wait (MPI_Request *request, MPI_Status *status)

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific *tag*, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

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

- The message buffer is described by (**`buf, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.

- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(rbuf, count, datatype, source, tag, comm, status)

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`status`** contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

Send-Receive Summary

- Send to matching Receive

MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

- Datatype
  - Basic for heterogeneity
  - Derived for non-contiguous
- Contexts
  - Message safety for libraries
- Buffering
  - Robustness and correctness

Retrieving Further Information

- **Status** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..
  status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of "wild card" tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application

Collective Operations in MPI

- Collective operations are called by all processes in a communicator.

- **MPI_BCAST**
  distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE**
  combines data from all processes in communicator and returns it to one process.

- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency. But not always…

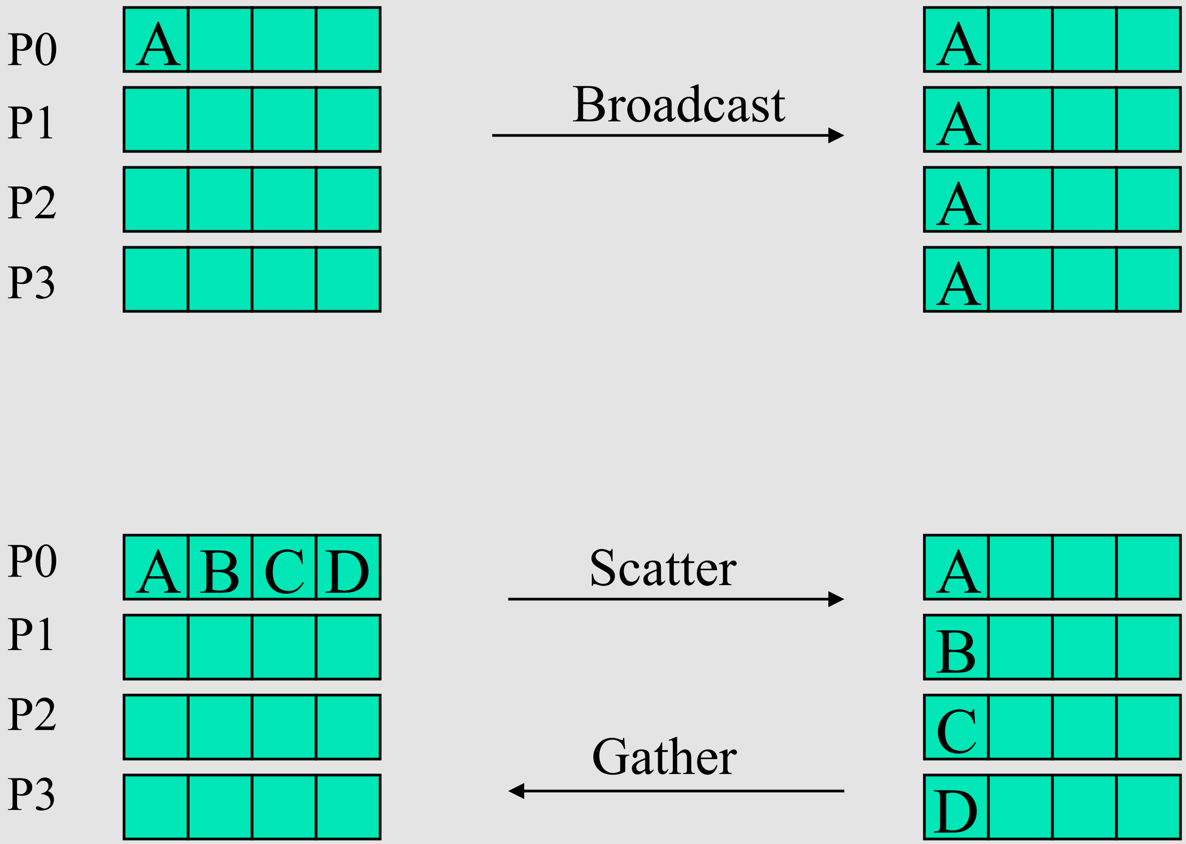
MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using MPI's topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations:
  - synchronization,
  - data movement,
  - collective computation.

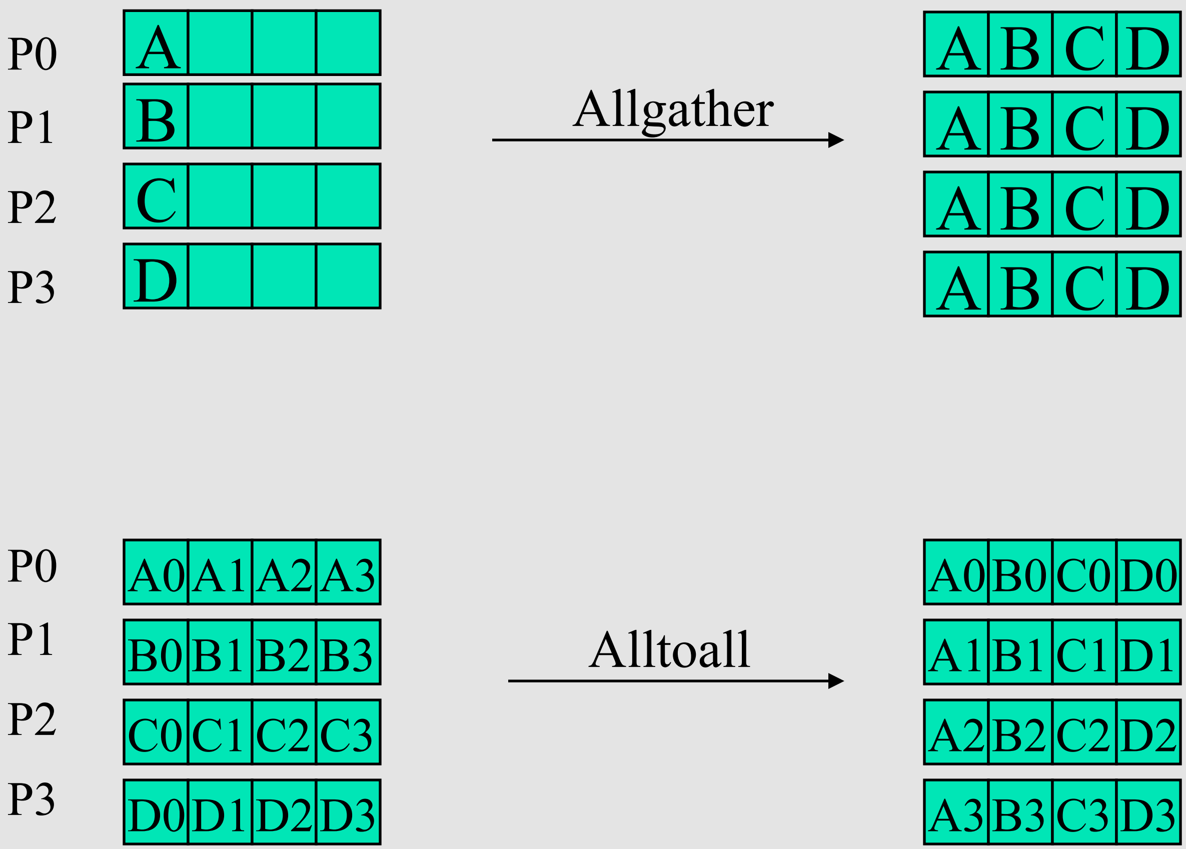
Synchronization

- **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.

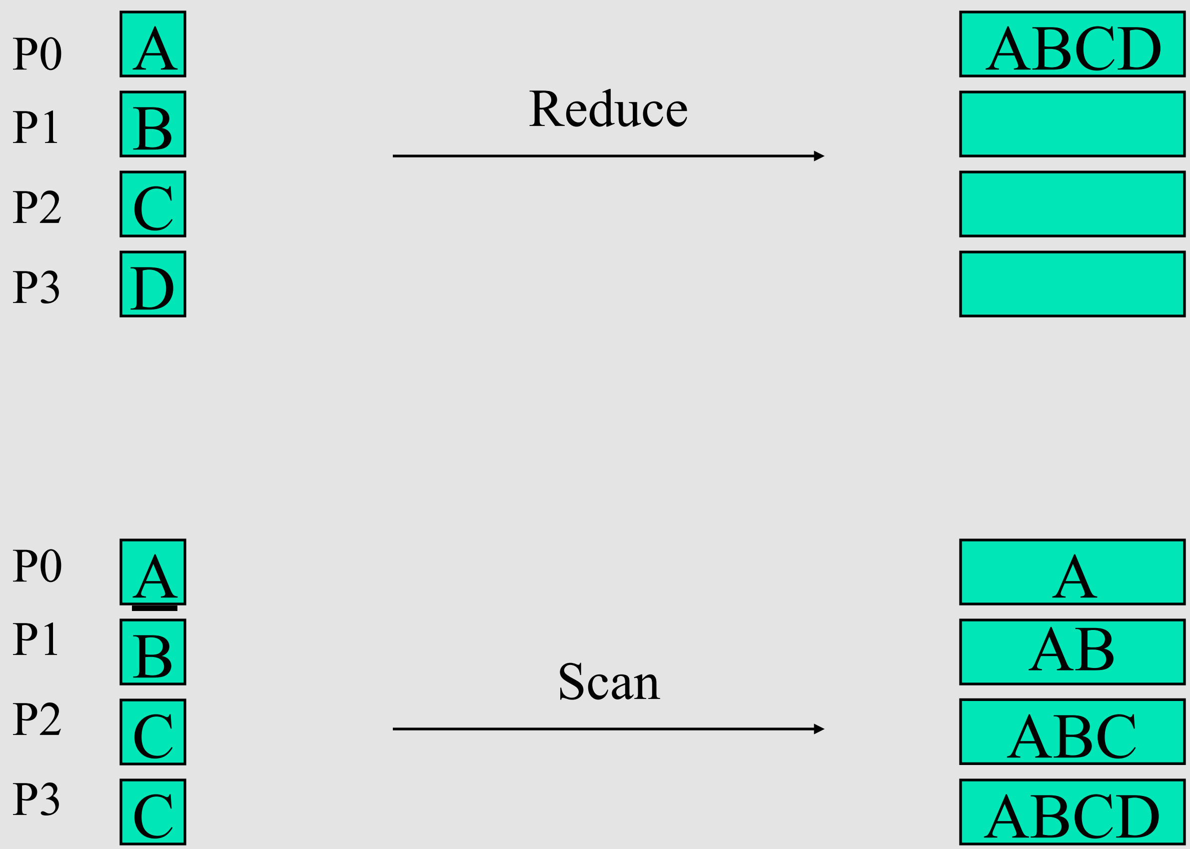
Collective Data Movement



More Collective Data Movement



Collective Computation



MPI Collective Routines

- Many Routines:

  Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, ReduceScatter, Scan, Scatter, Scatterv
- **All** versions deliver results to all participating processes.
- V versions allow the hunks to have different sizes.
- **Allreduce**, **Reduce**, **ReduceScatter**, and **Scan** take both built-in and user-defined functions.

MPI Built-in Collective Computation Operations

`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
`MPI_Band`	Binary and
`MPI_Bor`	Binary or
`MPI_Bxor`	Binary exclusive or
`MPI_Maxloc`	Maximum and location
`MPI_Minloc`	Minimum and location

Collective communication

- Involves coordinated communication within a group of processes
- All collective routines block until they are locally complete

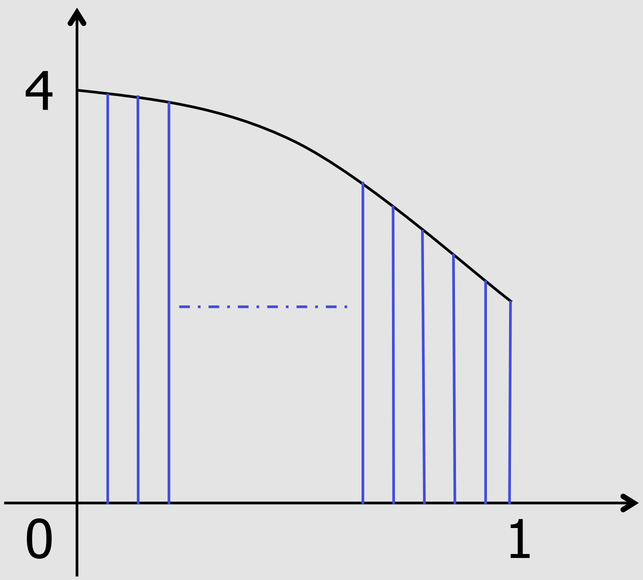
- int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
- int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Example: Calculating Pi

- One way to calculate Pi:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

- Calculating Pi via numerical integration
  - Divide interval up into subintervals
  - Assign subintervals to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get Pi



Example: PI in C (1/2)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, width, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done) {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }

     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

input/output data

root process

Example: PI in C (2/2)

```
    width = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = width * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = width * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

input location

output data

operation

root process

Examples

```
void main(int argc, char **argv) {
    int msg[100];
    int tag = 1;
    …
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    …
    if (rank == 0 ) {
      MPI_Send(msg, msgsize, MPI_INT, 1, tag, MPI_COMM_WORLD)
    } else if (rank == 1) {
      MPI_Recv(msg, msgsize, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    }
    MPI_Bcast(msg, msgsize, MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Finalize();
}
```

Summary: Using MPI

- The Message Passing Interface is:
  - a library for parallel communication
  - a system for launching parallel jobs (mpirun/mpiexec)
  - a community standard
- Launching jobs is easy
  - mpiexec -n 3 ./hellow
- You should never have to make MPI calls when using PETSc
  - Almost never

Summary: MPI Concepts

- Communicator
  - A context (or scope) for parallel communication (“Who can I talk to")
  - There are two defaults:
    - yourself (MPI_COMM_SELF),
    - and everyone launched (MPI_COMM_WORLD)
  - Can create new communicators by splitting existing ones

- Point-to-point communication
  - Happens between two processes (e.g., MatMult())

- Reduction or scan operations
  - Happens among all processes (e.g., VecDot())

Homework 2

1. Install MPI and PETSc
2. Test the installation of MPI
3. Test the installation of PETSc
4. Read Numerical Linear Algebra, by Trefethen and Bau:
   Lecture 1: Matrix-Vector Multiplication