Introduction to Distributed Memory Programming with MPI

Pawel Pomorski, University of Waterloo, SHARCNET
ppomorsk@sharcnet.ca

June 16 , 2015

Essentials of MPI

What is parallel computing?

- Using many computers linked together by a communication network to efficiently perform a computation that would not be possible on a single computer
- Single computers have stagnated in performance, computing power advances must be achieved today through parallelism.
- Parallelization cannot be handled for the user by the compiler.
- Various approaches to parallelization since 1990s

Parallelization techniques

- ▶ Message passing - most popular method which explicitly passes data from one computer to another as they compute in parallel.
- ▶ Assumes that each computer has its own memory not shared with the others, so all data exchange has to occur through explicit procedures.
- ▶ Contrast to shared memory processors, which include the relatively recent consumer multicore processors, where processes running on the cores can share memory and use threads.
- ▶ Can still use message passing in shared memory architectures.

What is MPI?

- **M**essage **P**assing **I**nterface
- Language-independent communications protocol
- Portable, platform independent, de facto standard for parallel computing on distributed memory systems
- Various implementations exist (MPICH, Open MPI, LAM, vendor versions)
- Many popular software libraries have parallel MPI versions
- Principal drawback: it is very difficult to design and develop programs using message passing
- "assembly language of parallel computing"
- Evolving: MPI-1 -> MPI-2 -> MPI-3

What is MPI?

- ▶ MPI is not a new programming language.
- ▶ It is a collection of functions and macros, or a library that can be used in C programs (also C++, Fortran, Python etc.)
- ▶ Most MPI programs are based on SPMD model - Single Program Multiple Data. This means that the same executable in a number of processes, but the input data makes each copy compute different things.
- ▶ All MPI identifiers begin with MPI_
- ▶ Each MPI function returns an integer which is an error code, but the default behavior of MPI implementations is to abort execution of the whole program if an error is encountered. This default behavior can be changed.

Preliminaries

- A process is an instance of a program, can be created or destroyed
- MPI uses a statically allocated group of processes - their number is set at the beginning of program execution, no additional processes created (unlike threads)
- Each process is assigned a unique number or rank, which is from 0 to p-1, where p is the number of processes
- Number of processes is not necessarily number of processors; a processor may execute more than one process
- Generally, to achieve the close-to-ideal parallel speedup each process must have exclusive use of one processor core.
- Running MPI programs with one processor core is fine for testing and debugging, but of course will not give parallel speedup.

Blocking communication

- Assume that process 0 sends data to process 1
- In a blocking communication, the sending routine returns only after the buffer it uses is ready to be reused
- Similarly, in process 1, the receiving routine returns after the data is completely stored in its buffer
- Blocking send and receive: MPI_Send and MPI_Recv
- MPI_Send: sends data; does not return until the data have been safely stored away so that the sender is free to access and overwrite the send buffer
- The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.
- MPI Recv: receives data; it returns only after the receive buffer contains the newly received message

Message structure

Each message consists of two parts:

1. Data transmitted
2. Envelope, which contains:

   - rank of the receiver
   - rank of the sender
   - a tag
   - a communicator

Receive does not need to know the exact size of data arriving but it must have enough space in its buffer to store it.

MPI program structure

- Include mpi.h
- Initialize MPI environment (MPI_Init)
- Do computations
- Terminate MPI environment (MPI_Finalize)

MPI program structure

```c
#include "mpi.h"

int main(int argc, char* argv[])
{
/* ... */

/* This must be the first MPI call */
 MPI_Init(&argc, &argv);

/* Do computation */

 MPI_Finalize();
/* No MPI calls after this line */

/* ... */

   return 0;
}
```

MPI program structure - quick Fortran example

Note the additional ierr argument in Fortran

```fortran
      program main
      include "mpif.h"
      integer ierr
c ...

c This must be the first MPI call
      call MPI_INIT(ierr)

c  Do computation

      call MPI_FINALIZE(ierr)

c No MPI calls after this line
c ...
      stop
      end
```

MPI_Send

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

- buf - beginning of the buffer containing the data to be sent
- count - number of elements to be sent (not bytes)
- datatype - type of data, e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR
- dest - rank of the process, which is the destination for the message
- tag - number, which can be used to distinguish among messages
- comm - communicator: a collection of processes that can send messages to each other, e.g. MPI_COMM_WORLD MPI_COMM_WORLD: all the processes running when execution begins

Returns error code

Predefined MPI datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
etc.	etc.
MPI_BYTE	no direct C equivalent
MPI_PACKED	no direct C equivalent

In addition, user-defined datatypes are possible.

MPI_Recv

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

- buf - beginning of the buffer where data is received
- count - number of elements to be received (not bytes)
- datatype - type of data, e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR
- source - rank of the process from which to receive message
- tag - number, which can be used to distinguish among messages
- comm - communicator
- status - information about the data received, e.g. rank of source, tag, error code. Replace with MPI_STATUS_IGNORE if never used.
- Returns error code
- Wildcards are possible for source and tag (eg. MPI_ANY_SOURCE)

MPI_Comm_rank

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- comm - communicator
- rank - of the calling process in group comm

MPI_Comm_size

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- comm - communicator
- size - number of processes in group comm

First program

Adapted from P. Pacheco, Parallel Programming with MPI

```c
/* greetings.c
 * Send a message from all processes with rank != 0
 * to process 0.
 * Process 0 prints the messages received.  */
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
  int         my_rank;      /* rank of process      */
  int         p;            /* number of processes  */
  int         source;       /* rank of sender       */
  int         dest;         /* rank of receiver     */
  int         tag = 0;      /* tag for messages     */
  char        message[100]; /* storage for message  */
  MPI_Status  status;       /* status for receive   */
```

First program continued

```
/* Start up MPI */
  MPI_Init(&argc, &argv);
  /* Find out process rank  */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  /* Find out number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (my_rank != 0)
    {
      /* Create message */
      sprintf(message, "Greetings from process %d!",
              my_rank);
      dest = 0;
      /* Use strlen+1 so that '\0' gets transmitted */
      MPI_Send(message, strlen(message)+1, MPI_CHAR,
               dest, tag, MPI_COMM_WORLD);
    }
```

First program continued

```c
    else
      { /* my_rank == 0 */
        for (source = 1; source < p; source++)
          {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                     MPI_COMM_WORLD, &status);
            printf("%s\n", message);
          }
      }

    /* Shut down MPI */
    MPI_Finalize();

    return 0;
}
```

Compilation and execution

Ubuntu Linux example:

1. Install necessary packages: mpich-bin,libmpich1.0-dev
2. Make sure ssh client and server installed
3. ssh localhost - needs to work without password. Might need ssh key set up.
4. compile code with:

```
mpicc hello_world.c -o test.x
```

5. Execute:

```
mpirun -np 8 test.x
```

-np option specifies number of processes, which will all run on localhost

Compilation and execution

SHARCNET example:

All the preliminaries taken care of. After getting an account, choose a suitable cluster, log in.

Compile code with:

```
mpicc greetings.c -o greetings.x
```

mpicc is a script wrapper for C compiler that automatically links the necessary MPI libraries, add the -v flag to see details

It is not possible to use mpirun directly. Instead, your executable must be submitted as a job to the queueing system via:

```
sqsub -q mpi -n 8 -r 20m -o out.dat greetings.x
```

-r 20m - estimates runtime of 20 minutes

-n 8 - specifies 8 MPI processes will be used

MPI in Python

- mpi4py (MPI for Python) provides bindings for MPI in Python
- object oriented, more user friendly, will automatically determine many of the needed arguments to MPI calls
- `http://mpi4py.scipy.org/`
- `http://mpi4py.scipy.org/docs/mpi4py.pdf`
- Execute with: mpirun -np 4 python program.py

First program in Python using mpi4py

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
p = comm.Get_size()

if my_rank != 0:
    message = "Greetings from process "+str(my_rank)
    comm.send(message, dest=0)
else :
    for procid in range(1,p):
        message = comm.recv(source=procid)
        print "process 0 receives message from process",\
              procid,":",message
```

Example: Numerical integration

Trapezoid rule for integrating $\int_a^b = f(x)dx$

with $h = (b-a)/n$ is

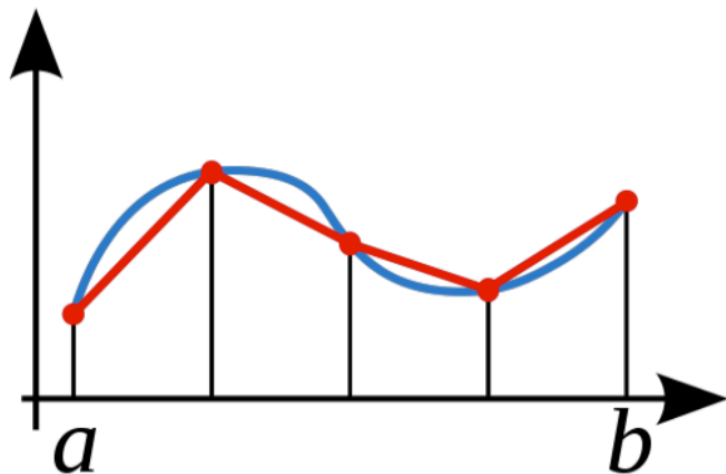$$f(x) \approx \frac{h}{2}(f(x_0) + f(x_n)) + h\sum_{i=1}^{n-1} f(x_i)$$

where $x_i = a + ih, i = 0, 1, ..., n$

Given p processes, each process can work on n/p intervals

Note: for simplicity will assume n/p is an integer

process	interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
...	...
p-1	$[a + (p-1)\frac{n}{p}h, b]$

Example: Numerical integration with trapezoid rule

Parallel trapezoid

Assume $f(x) = x^2$

Of course could have chosen any desired (integrable) function here.

Write function f(x) in

```c
/* func.c */

float f(float x)
{
  return x*x;
}
```

Serial trapezoid rule

```c
/* traprule.c */

extern float f(float x); /* function we're integrating */

float Trap(float a, float b, int n, float h) {
  float integral;   /* Store result in integral  */
  float x;
  int i;

  integral = (f(a) + f(b))/2.0;
  x = a;
  for ( i = 1; i <= n-1; i++ )
    {
      x = x + h;
      integral = integral + f(x);
    }
  return integral*h;
}
```

Parallel trapezoid rule

```
/* trap.c -- Parallel Trapezoidal Rule
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *     using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *     1.  Each process calculates "its" interval of
 *         integration.
 *     2.  Each process estimates the integral of f(x)
 *         over its interval using the trapezoidal rule.
 *     3a. Each process != 0 sends its integral to 0.
 *     3b. Process 0 sums the calculations received from
 *         the individual processes and prints the result.
 *
 *         The number of processes (p) should evenly divide
 *         the number of trapezoids (n = 1024)
 */
```

Main program

```
int main(int argc, char** argv) {
    int          my_rank;    /* My process rank          */
    int          p;          /* The number of processes  */
    float        a = 0.0;    /* Left endpoint            */
    float        b = 1.0;    /* Right endpoint           */
    int          n = 1024;   /* Number of trapezoids     */
    float        h;          /* Trapezoid base length    */
    float        local_a;    /* Left endpoint my process */
    float        local_b;    /* Right endpoint my process */
    int          local_n;    /* Number of trapezoids     */
    float        integral;   /* Integral over my interval */
    float        total=-1;   /* Total integral           */
    int          source;     /* Process sending integral */
    int          dest = 0;   /* All messages go to 0     */
    int          tag = 0;    MPI_Status  status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```c
    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */
    /* Length of each process' interval of
       integration = local_n*h. */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);
    /* Add up the integrals calculated by each process */
    if (my_rank == 0)
      {
        total = integral;
        for (source = 1; source < p; source++)
          {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                    MPI_COMM_WORLD, &status);
          printf("PE %d <- %d, %f\n", my_rank,source,
                  integral);
          total = total + integral;
          }
      }
```

```c
    else
        {
        printf("PE %d -> %d, %f\n", my_rank, dest, integral);
        MPI_Send(&integral, 1, MPI_FLOAT, dest,
                    tag, MPI_COMM_WORLD);
        }
    /* Print the result */
    if (my_rank == 0)
        {
        printf("With n = %d trapezoids, our estimate\n",
                n);
        printf("of the integral from %f to %f = %f\n",
                a, b, total);
        }

    MPI_Finalize();
    return 0;
}
```

Summary

To write many MPI parallel programs you only need:

- MPI_Init
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv
- MPI_Finalize

Understanding Communications

Concepts

Buffering

Safe programs

Non-blocking communications

Buffering

Suppose we have

```
if (rank==0)
    MPI_Send(sendbuf,...,1,...)
if (rank==1)
    MPI_Recv(recvbuf,...,0,...)
```

These are blocking communications, which means they will not return until the arguments to the functions can be safely modified by subsequent statements in the program.

Assume that process 1 is not ready to receive

There are 3 possibilities for process 0:

1. stops and waits until process 1 is ready to receive
2. copies the message at sendbuf into a system buffer (can be on process 0, process 1 or somewhere else) and returns from MPI_Send
3. fails

Buffering

As long as buffer space is available, (b) is a reasonable alternative

An MPI implementation is permitted to copy the message to be sent into internal storage, but it is not required to do so

What if not enough space is available?

- In applications communicating large amounts of data, there may not be enough memory (left) in buffers
- Until receive starts, no place to store the send message
- Practically, (a) results in a serial execution

A programmer should not assume that the system provides adequate buffering

Example

Consider a program executing

Process 0	Process 1
MPI_Send to process 1	MPI_Send to process 0
MPI_Recv from process 1	MPI_Recv from process 0

Such a program may work in many cases, but it is certain to fail
for message of some size that is large enough

Possible solutions

Ordered send and receive - make sure each receive is matched with send in execution order across processes

This matched pairing can be difficult in complex applications. An alternative is to use **MPI_Sendrecv**. It performs both send and receive such that if no buffering is available, no deadlock will occur

Buffered sends. MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left in buffer) via **MPI_Bsend**

Nonblocking communication. Initiated, then program proceeds while the communication is ongoing, until a check that communication is completed later in the program. **Important**: in this case you must make certain that you do not modify the data until you are certain communication has completed.

MPI_Sendrecv

```
int MPI_Sendrecv( void *sendbuf, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status )
```

Combines:

- MPI_Send - send data to process with rank=dest
- MPI_Recv - receive data from process with rank=source

Source and dest may be the same

MPI_Sendrecv may be matched by ordinary MPI_Send or MPI_Recv

Performs Send and Recv, and organizes them in such a way that even in systems with no buffering program won't deadlock

MPI_Sendrecv_replace

```
int MPI_Sendrecv_replace( void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag,
        MPI_Comm comm, MPI_Status *status )
```

- Sends and receives using a single buffer
- Process sends contents of buf to dest, then replaces them with data from source
- If source=dest, then function swaps data between process which calls it and process source

Safe programs

- A program is safe if it will produce correct results **even if the system provides no buffering**.
- Need safe programs for portability.
- Most programmers expect the system to provide some buffering, hence many unsafe MPI programs are around.
- Write safe programs using matching send with receive, MPI_Sendrecv, allocating own buffers, nonblocking operations

Nonblocking communications

- nonblocking communications are useful for **overlapping** communication with computation, and ensuring safe programs
- a nonblocking operation requests the MPI library to perform an operation (when it can)
- nonblocking operations do not wait for any communication events to complete
- nonblocking send and receive: return almost immediately
- can safely modify a send (receive) buffer only after send (receive) is completed
- "wait" routines will let program know when a nonblocking operation is done

MPI_Isend

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
        int dest, int tag , MPI_Comm comm,
        MPI_Request *request)
```

- buf - starting address of buffer
- count - number of entries in buffer
- datatype - data type of buffer
- dest - rank of destination
- tag - message tag
- comm - communicator
- request - communication request (out)

MPI_Irecv

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
        int source , int tag ,MPI_Comm comm,
        MPI_Request *request)
```

- buf - starting address of buffer (out)
- count - number of entries in buffer
- datatype - data type of buffer
- source - rank of source
- tag - message tag
- comm - communicator
- request - communication request (out)

Wait routines

```
int MPI_Wait (MPI_Request *request , MPI_Status *status)
```

Waits for MPI_Isend or MPI_Irecv to complete

- ▶ request - request (in), which is out parameter in MPI_Isend and MPI_Irecv
- ▶ status - status output, replace with MPI_STATUS_IGNORE if not used

Wait routines

Other routines include:

- ▶ MPI_Waitall waits for all given communications to complete
- ▶ MPI_Waitany waits for any of given communications to complete
- ▶ MPI_Test tests for completion of send or receive, i.e returns true if completed, false otherwise
- ▶ MPI_Testany tests for completion of any previously initiated communication in the input list

MPI_Waitall

```
int MPI_Waitall (int count,
                 MPI_Request array_of_requests[],
                 MPI_Status array_of_statuses[])
```

Waits for all given communications to complete

- ▶ count - list length
- ▶ array_of_requests - each request is an output parameter in MPI_Isend and MPI_Irecv
- ▶ array_of_statuses - array of status objects, replace with MPI_STATUSES_IGNORE if never used

Example: Communication between processes in ring topology
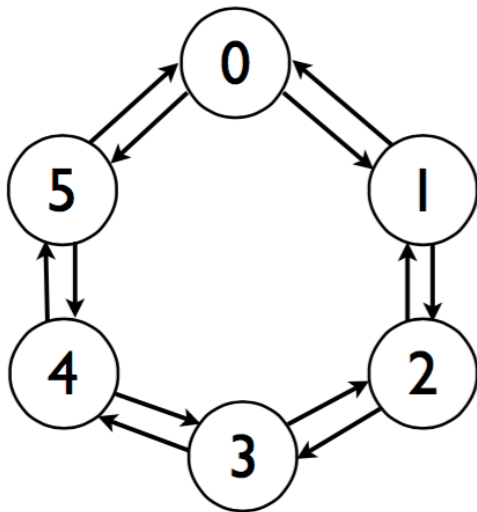
Example - Communication between processes in ring topology

- With blocking communications it is not possible to write a simple code to accomplish this data exchange.
- For example, if we have MPI_Send first in all processes, program will get stuck as there will be no matching MPI_Recv to send data to
- Nonblocking communication avoids this problem

Ring topology example

```c
/* nonb.c */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) {
  int numtasks, rank, next, prev,
    buf[2], tag1=1, tag2=2;

  tag1=tag2=0;
  MPI_Request reqs[4];
  MPI_Status stats[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Ring topology example

```
prev = rank-1;
next = rank+1;
if (rank == 0)              prev = numtasks - 1;
if (rank == numtasks - 1) next = 0;
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
          MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2,
          MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2,
          MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1,
          MPI_COMM_WORLD, &reqs[3]);
MPI_Waitall(4, reqs, stats);
printf("Task %d communicated with tasks %d & %d\n",
       rank,prev,next);
MPI_Finalize();
return 0; }
```

MPI_Test

```
int MPI_Test ( MPI_Request *request,int *flag,
        MPI_Status *status)
```

- request - (input) communication handle, which is output parameter in MPI_Isend and MPI_Irecv
- flag - true if operation completed (logical)
- status - status output, replace with MPI_STATUS_IGNORE if not used

MPI_Test - can be used to test if communication completed, can be called multiple times, in combintation with nonblocking send/receive, to control execution flow between processes

Non-deterministic workflow

```c
if (my_rank == 0){
(... do computation ...)
/* send signal to other processes */
 for (proc = 1; proc < nproc; proc++){
  MPI_Send(&buf, 1, MPI_INT, proc, tag, MPI_COMM_WORLD)
} }
else{
/* initiate nonblocking receive */
 MPI_Irecv(&buf,1, MPI_INT, 0, tag, MPI_COMM_WORLD,&reqs);
 for(i = 0; i <= Nlarge; i++){
/* test if Irecv completed */
  MPI_Test(&reqs, &flag, &status);
      if(flag){
         break;    /* terminate loop */
      }
      else{
         (... do computation ...)
      } } }
```

Summary for Nonblocking Communications

- nonblocking send can be posted whether a matching receive has been posted or not
- send is completed when data has been copied out of send buffer
- nonblocking send can be matched with blocking receive and vice versa
- communications are initiated by sender
- a communication will generally have lower overhead if a receive buffer is already posted when a sender initiates a communication

Collective communications

Introduction

Collective communication involves all the processes in a communicator

We will consider:

- Broadcast
- Reduce
- Gather
- Scatter

Reason for use: convenience and speed

Broadcast

Broadcast: a single process sends data to all processes in a communicator

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

- buffer starting address of buffer (in/out)
- count number of entries in buffer
- datatype data type of buffer root
- rank - rank of broadcast root
- comm - communicator

MPI_Bcast

- MPI_Bcast sends a copy of the message on process with rank root to each process in comm
- must be called in each process
- data is sent in root and received by all other processes
- buffer is 'in' parameter in root and 'out' parameter in the rest of processes
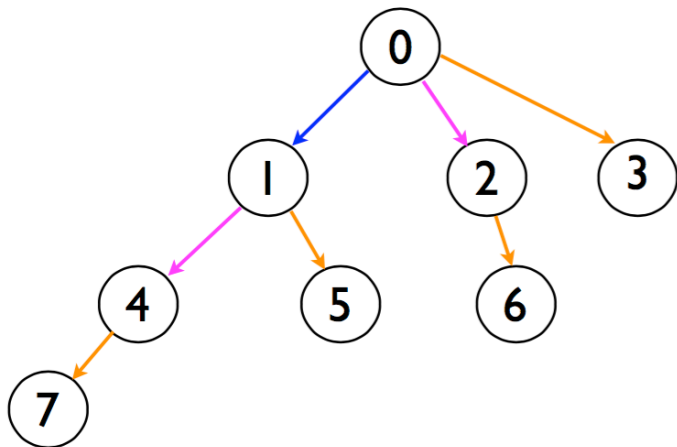- cannot receive broadcasted data with MPI_Recv

Broadcast - poor implementation

- Serial, 7 time steps needed
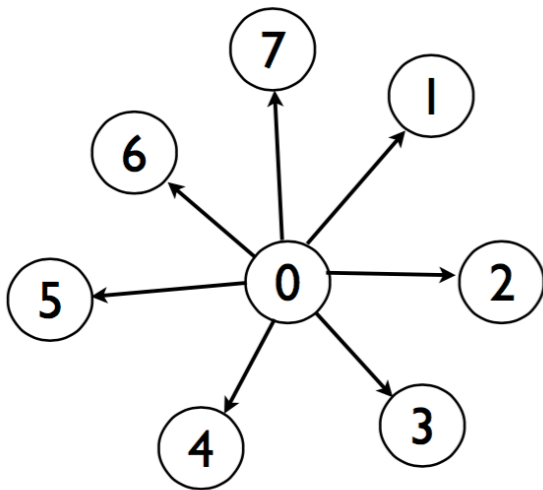
Broadcast - actual, parallel implementation

- ▶ Parallel, 3 time steps needed

Example: reading and broadcasting data

Code adapted from P. Pacheco, PP with MPI

```
/* getdata2.c */

/* Function Get_data
 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes using three calls to MPI_Bcast.
 */
```

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

void Get_data(float* a_ptr, float* b_ptr, int* n_ptr,
              int my_rank)
{
  if (my_rank == 0)
    {
      printf("Enter a, b, and n\n");
      scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
  MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(n_ptr, 1, MPI_INT,   0, MPI_COMM_WORLD);
}
```

Reduce

Data from all processes are combined using a binary operation

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

- sendbuf - address of send buffer
- recvbuf - address of receive buffer, significant only at root
- count - number of entries in send buffer
- datatype - data type of elements in send buffer
- op - reduce operation; predefined, e.g. MPI_MIN, MPI_SUM, or user defined
- root - rank of root process
- comm - communicator
- Must be called in all processes in a communicator, BUT result only available in root process

Example - trapezoid with reduce

Code adapted from P. Pacheco, PP with MPI

```c
/* redtrap.c */
#include <stdio.h>
#include "mpi.h"
extern void Get_data2(float* a_ptr, float* b_ptr,
                      int* n_ptr, int my_rank);
extern float Trap(float local_a, float local_b,
                  int local_n, float h);

int main(int argc, char** argv)
{
    int         my_rank, p;
    float       a, b, h;
    int         n;
    float       local_a, local_b, local_n;
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral           */
```

```c
Get_data2(&a, &b, &n, my_rank);

  h = (b-a)/n;
  local_n = n/p;

  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
  integral = Trap(local_a, local_b, local_n, h);

  /* Add up the integrals calculated by each process */
  MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
             MPI_SUM, 0, MPI_COMM_WORLD);

  if (my_rank == 0)
    {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %f\n",
             a, b, total);
    }
```
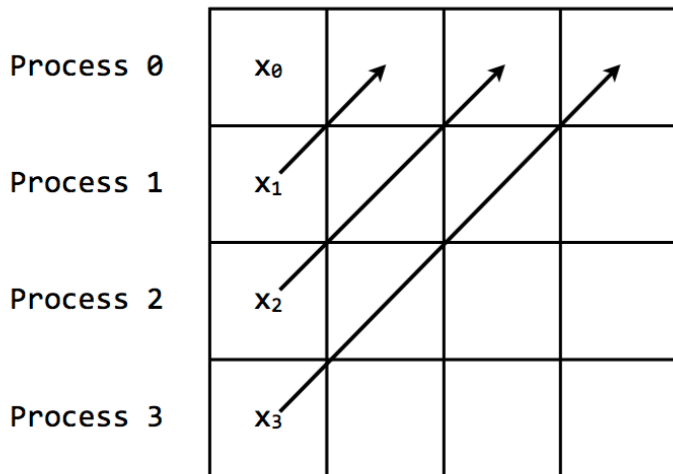
Allreduce

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

Similar to MPI Reduce except the result is returned to the receive buffer of each process in comm

Gather

Gather

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

- Gathers together data from a group of processes sendbuf - starting address of send buffer
- sendcount - number of elements in send buffer
- sendtype - data type of send buffer elements
- recvbuf - address of receive buffer (significant only at root)
- recvcount - number of elements for any single receive (significant only at root)
- recvtype - data type of recv buffer elements (significant only at root)
- root - root rank of receiving process
- comm - communicator

Gather

- MPI_Gather collects data, stored at sendbuf, from each process in comm and stores the data on root at recvbuf
- Data is received from processes in order, i.e. from process 0, then from process 1 and so on
- Usually sendcount, sendtype are the same as recvcount, recvtype
- root and comm must be the same on all processes
- The receive parameters are significant only on root
- Amount of data sent/received must be the same

Allgather

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

- Use if gathered data needs to be available to all processes.
- The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

Scatter
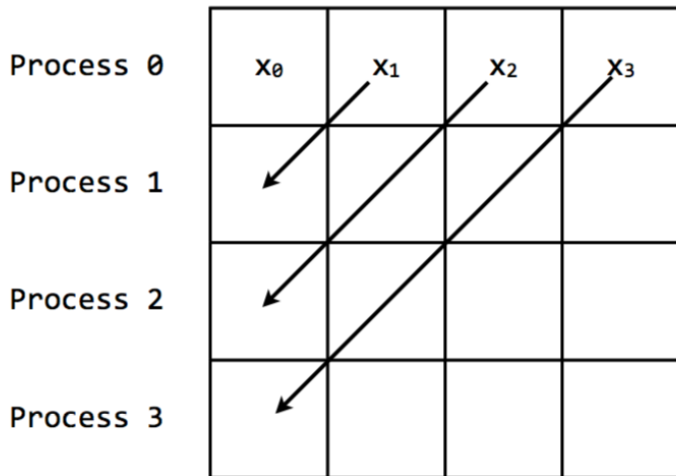
Scatter

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

- Sends data from one process to all other processes in a communicator
- sendbuf starting address of send buffer (significant only at root)
- sendcount - number of elements sent to each process (significant only at root)
- sendtype - data type of send buffer elements (significant only at root)
- recvbuf - address of receive buffer
- recvcount - number of elements for any single receive
- recvtype - data type of recv buffer elements
- root - rank of sending process
- comm - communicator

Scatter

MPI_Scatter splits data at sendbuf on root into **p** segments, each of **sendcount** elements, and sends these segments, in order, to processes **0, 1, ..., p-1**

Inverse operation to MPI_Gather

The outcome is as if the root executed n send operations,

```
MPI_Send(sendbuf + i * sendcount * extent(sendtype),
         sendcount, sendtype, i, ...)
```

and each process executed a receive,

```
MPI_Recv(recvbuf, recvcount, recvtype, i, ...).
```

Amount of data sent must be equal to amount of data received.

Parallel Matrix Multiplication
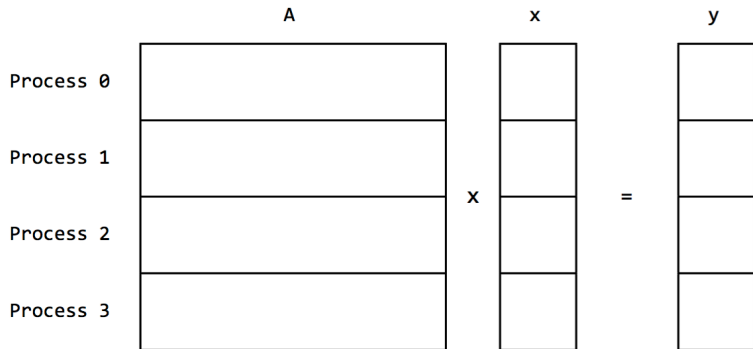
Ax=y - data distributed on 4 processes

Figure 1: Matrix multiply

Example - parallel matrix times vector

```
/* Code adapted from P. Pacheco, PP with MPI
 * parallel_mat_vect.c -- computes a parallel
 * matrix-vector product.
 * Matrix is distributed by block rows.
 * Vectors are distributed by blocks.
 * Input:
 *     m, n: order of matrix
 *     A, x: the matrix and the vector to be multiplied
 * Output:
 *     y: the product vector
 * Notes:
 *     1.  Local storage for A, x, and y
 *         is statically allocated.
 *     2.  Number of processes (p) should evenly
 *         divide both m and n.
 */
```

```c
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"
int main(int argc, char* argv[]) {
    int             my_rank, p;
    LOCAL_MATRIX_T  local_A;
    float           global_x[MAX_ORDER];
    float           local_x[MAX_ORDER];
    float           local_y[MAX_ORDER];
    int             m, n;
    int             local_m, local_n;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == 0) {
        printf("Enter the order of the matrix (m x n)\n");
        scanf("%d %d", &m, &n); }
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
    local_m = m/p;
    local_n = n/p;
    Read_matrix("Enter the matrix",
                local_A, local_m, n, my_rank, p);
    Print_matrix("We read",
                 local_A, local_m, n, my_rank, p);
    Read_vector("Enter the vector",
                local_x, local_n, my_rank, p);
    Print_vector("We read",
                 local_x, local_n, my_rank, p);
    Parallel_matrix_vector_prod(local_A, m, n, local_x,
                                global_x, local_y, local_m,
                                local_n);
    Print_vector("The product is", local_y, local_m,
                 my_rank, p);
    MPI_Finalize();
    return 0;
}
```

```c
/* matvec.h */
#define MAX_ORDER 100
typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];
void Read_matrix(char* prompt, LOCAL_MATRIX_T local_A,
                 int local_m, int n, int my_rank, int p);
void Read_vector(char* prompt, float local_x[],
                 int local_n, int my_rank, int p);
void Parallel_matrix_vector_prod(LOCAL_MATRIX_T local_A,
                                 int m,
                                 int n, float local_x[],
                                 float global_x[],
                                 float local_y[],
                                 int local_m, int local_n);
void Print_matrix(char* title, LOCAL_MATRIX_T local_A,
                  int local_m, int n, int my_rank, int p);
void Print_vector(char* title, float local_y[],
                  int local_m, int my_rank, int p);
```

```c
/* parmatvec.c */
#include "mpi.h"
#include "matvec.h"
void Parallel_matrix_vector_prod
( LOCAL_MATRIX_T  local_A, int m, int n,
  float local_x[], float global_x[], float local_y[],
  int    local_m, int local_n) {
  /* local_m = m/p, local_n = n/p */
  int i, j;
  MPI_Allgather(local_x, local_n, MPI_FLOAT,
                global_x, local_n, MPI_FLOAT,
                MPI_COMM_WORLD);
  for (i = 0; i < local_m; i++) {
      local_y[i] = 0.0;
      for (j = 0; j < n; j++)
        local_y[i] = local_y[i] +
          local_A[i][j]*global_x[j];
    }
}
```

```c
/* readvec.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"
void Read_vector(char *prompt,float local_x[],int local_n,
                 int my_rank, int p) {
   int    i;
   float temp[MAX_ORDER];
   if (my_rank == 0)
     {
       printf("%s\n", prompt);
       for (i = 0; i < p*local_n; i++)
         scanf("%f", &temp[i]);
     }
   MPI_Scatter(temp, local_n, MPI_FLOAT,
               local_x, local_n, MPI_FLOAT,
               0, MPI_COMM_WORLD);
}
```

```c
/* readmat.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"
void Read_matrix(char *prompt, LOCAL_MATRIX_T  local_A,
                 int local_m, int n, int my_rank,int p) {
  int i, j;
  LOCAL_MATRIX_T  temp;
  for (i = 0; i < p*local_m; i++)
    for (j = n; j < MAX_ORDER; j++)
      temp[i][j] = 0.0; /* Initialize temp with zeroes */
  if (my_rank == 0) {
      printf("%s\n", prompt);
      for (i = 0; i < p*local_m; i++)
        for (j = 0; j < n; j++)
          scanf("%f",&temp[i][j]); }
  MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT,
  local_A,local_m*MAX_ORDER,MPI_FLOAT,0,MPI_COMM_WORLD);}
```

```c
/* printvec.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"
void Print_vector(char *title, float  local_y[],
                  int local_m, int my_rank, int p) {
  int i;
  float temp[MAX_ORDER];
  MPI_Gather(local_y, local_m, MPI_FLOAT,
             temp, local_m, MPI_FLOAT,
             0, MPI_COMM_WORLD);
  if (my_rank == 0)
    {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++)
        printf("%4.1f ", temp[i]);
      printf("\n");
    }
}
```

```c
/* printmat.c */
#include <stdio.h>
#include "mpi.h"
#include "matvec.h"
void Print_matrix(char *title, LOCAL_MATRIX_T  local_A,
                  int local_m, int n, int my_rank, int p){
  int    i, j;
  float temp[MAX_ORDER][MAX_ORDER];
  MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT,
             temp, local_m*MAX_ORDER, MPI_FLOAT,
             0, MPI_COMM_WORLD);
  if (my_rank == 0) {
    printf("%s\n", title);
    for (i = 0; i < p*local_m; i++) {
        for (j = 0; j < n; j++)
          printf("%4.1f ", temp[i][j]);
        printf("\n");
      }
  }
}
```

Summary for Collective Communications

- Amount of data sent must match amount of data received
- Blocking versions only
- No tags: calls are matched according to order of execution
- A collective function can return as soon as its participation is complete

Further MPI features to explore

- Communicators
- Topologies
- User defined datatypes
- Parallel input/output operations
- Parallel algorithms
- Parallel libraries (eg. Scalapack)