

# The Message Passing Interface for Parallel Applications

McMaster University 2012

Paweł Pomorski

University of Waterloo, SHARCNET

[ppomorsk@sharcnet.ca](mailto:ppomorsk@sharcnet.ca)

PDF of slides and course information at:

[http://ppomorsk.sharcnet.ca/CES\\_713.html](http://ppomorsk.sharcnet.ca/CES_713.html)

# 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 parallization since 1990s

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?

## Message Passing Interface

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”

MPI = MPI-1 - standard “original” version

MPI-2 - superset of MPI-1, various useful extensions

Extensions not part of standard also available

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 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

Processes can be created and destroyed

MPI assumes statically allocated 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 (i.e.  $n$  times faster with  $n$  processes), each process must have exclusive use of one processor core. Possible on multicore processor if  $p$  does not exceed number of cores

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
- Do computations
- Terminate MPI environment

Initialize with `MPI_Init`

Terminate with `MPI_Finalize`

## MPI program structure

---

```
#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 differences in the way MPI functions called

Almost all Fortran MPI calls have additional `ierr` argument, which has the same meaning as the return value of the function in C

---

```
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.	
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*

```
/* 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 */
```

```
/* 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);
}
```

```
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. To set up keys:

```
ssh-keygen -t rsa  
cd .ssh  
cp id_rsa.pub authorized_keys
```

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.

1. compile code with: `mpicc hello_world.c -o test.x`

`mpicc` is a script, use it with `-v` option to see what is actually executed

2. 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 --test -n 8 -r 20m -o out.dat test.x
```

`--test` means the job will have a 1 hour limit but should start quickly. Only one test job can be active at one time.

`-r 20m` estimates runtime of 20m

`-n 8` specifies 8 processors

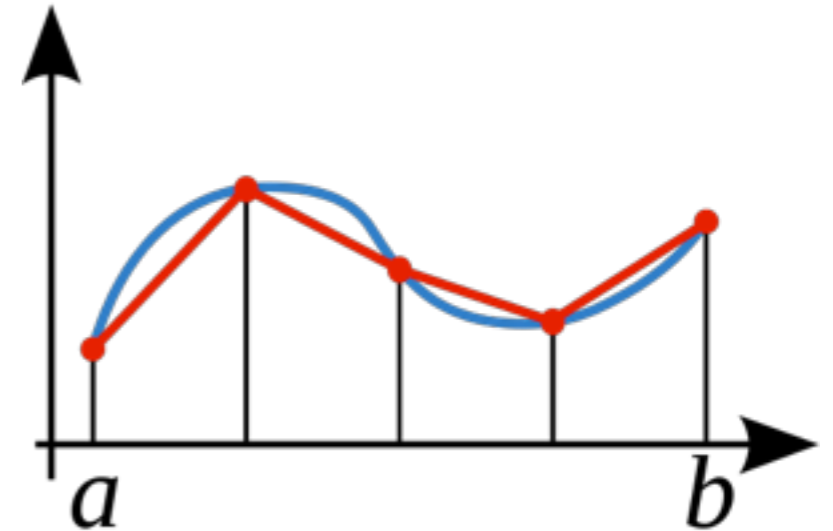
# Example: Numerical integration

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  $x_i = a + ih, i = 0, 1, \dots, n$



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	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
$\vdots$	
$p - 1$	$[a + (p - 1)\frac{n}{p}h, b]$

# Parallel trapezoid integration

Assume  $f(x) = x^2$

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

Write function  $f(x)$  in

---

```
/* func.c */  
  
float f(float x)  
{  
    return x*x;  
}
```

## Trapezoid rule

---

```
/* 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;
}
```



```
/* 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  $\neq 0$  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)
 */
```

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

```
extern float Trap(float a, float b, int n, float h);
```

```

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 for
                           /* my calculation           */
    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);

```

```
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;
    }
}
```

```
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;
}
```

# I/O

We want to read a,b, and n from the standard input

Function Get\_data reads a,b and n

Cannot be called in each process

Process 0 calls Get\_data, which sends these data to processes 1,2,...,p-1

The same scheme applies if we read from a file

---

Most supercomputing clusters are equipped with parallel storage hardware so writing to or reading from files need not be coded in parallel by the user explicitly

It is possible for each process to read to or write from file, as long as they don't interfere with each other (i.e. it's best to use a separate file for each process)

All processes can write to standard output as already illustrated

```
/* getdata.c
```

```
* 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.
```

```
*/
```

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

```
#include "mpi.h"
```

```
void Get_data( float* a_ptr, float* b_ptr, int* n_ptr,
              int my_rank, int p )
{
    int source = 0, dest, tag;
    MPI_Status status;

    if (my_rank == 0)
    {
        printf("Rank %d: Enter a, b, and n\n", my_rank);
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

        for (dest = 1; dest < p; dest++)
        {
            tag = 0;
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
                    MPI_COMM_WORLD);

            tag = 1;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
                    MPI_COMM_WORLD);

            tag = 2;
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
                    MPI_COMM_WORLD);
        }
    }
}
```

```
else
{
    tag = 0;
    MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD, &status);
    tag = 1;
    MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD, &status);
    tag = 2;
    MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
             MPI_COMM_WORLD, &status);
}
}
```



# Parallel program with input

---

```
/* get_data.c -- Parallel Trapezoidal Rule,
   uses basic Get_data function for input.
*/
#include <stdio.h>
#include "mpi.h"

extern void Get_data(float* a_ptr, float* b_ptr,
                   int* n_ptr, int my_rank, int p);
extern float Trap(float a, float b, int n, float h);

int main(int argc, char** argv)
{
    int          my_rank, p;
    float        a, b, h;
    int          n;
    float        local_a, local_b;
    int          local_n;

    float        integral;
    float        total=-1;
    int          source, dest = 0, tag = 0;
    MPI_Status   status;
```

```
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

Get_data(&a, &b, &n, my_rank, p);

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. So my interval
 * starts at: */
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);
        total = total + integral;
    }
}
else
    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;
}
```

# Example makefile

```
CC = gcc
MPICC = mpicc
CFLAGS = -Wall -O2 -g

OBJECTS1 = trap.o func.o traprule.o
OBJECTS2 = func.o traprule.o iotrap.o getdata.o

all: partrap iopartrap

partrap: $(OBJECTS1)
    $(MPICC) -o partrap $(OBJECTS1)

iopartrap: $(OBJECTS2)
    $(MPICC) -o iopartrap $(OBJECTS2)

trap.o: trap.c
    $(MPICC) $(CFLAGS) -c trap.c

traprule.o: traprule.c
    $(MPICC) $(CFLAGS) -c traprule.c

func.o: func.c
    $(MPICC) $(CFLAGS) -c func.c

getdata.o: getdata.c
    $(MPICC) $(CFLAGS) -c getdata.c

iotrap.o: iotrap.c
    $(MPICC) $(CFLAGS) -c iotrap.c

clean:
    rm $(OBJECTS1) $(OBJECTS2)
```

# 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

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:

(a) stops and waits until process 1 is ready to receive

(b) copies the message at sendbuf into a system buffer (can be on process 0, process 1 or somewhere else) and returns from MPI\_Send

(c) fails



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_Recv from process 1	MPI_Send to process 0 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. With buffering, a send may return before a matching receives is posted. With no buffering, communication is deferred until a place for receiving is provided. **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

That is, compute while communicating data

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

The user can modify a send (receive) buffer only after send (receive) is completed

There are “wait” routines to figure out when a nonblocking operation is done

# MPI\_Isend

Performs nonblocking send

---

```
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

Performs nonblocking receive

---

```
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

---

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[])
```

---

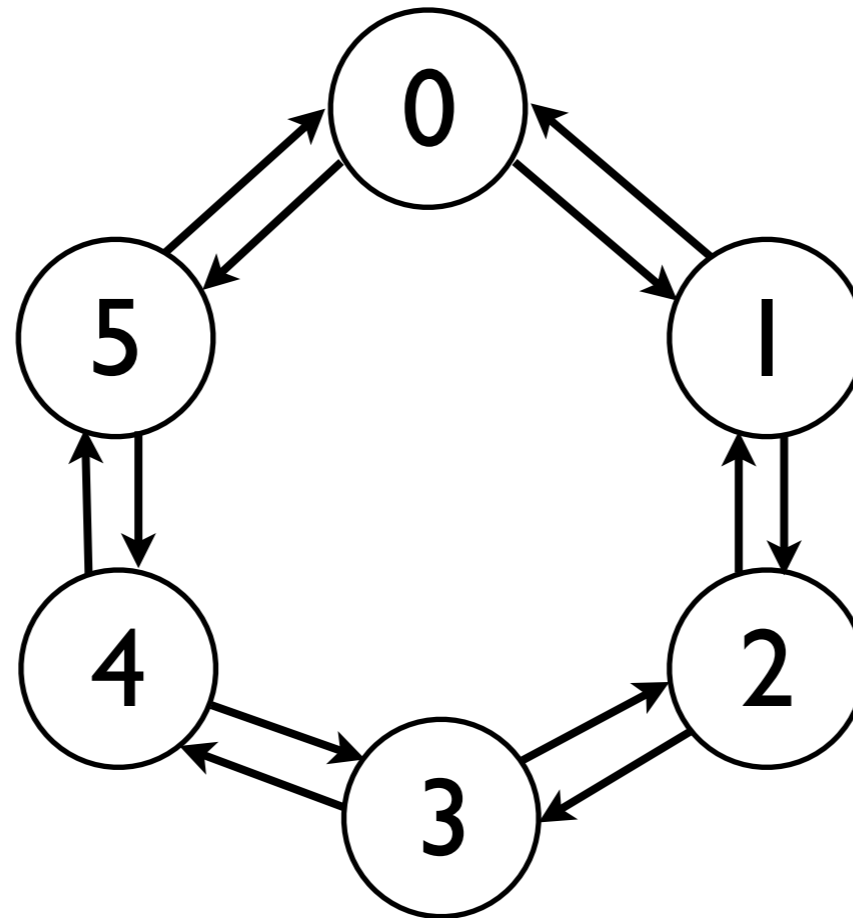
Waits for all given communications to complete

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



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

From [https://computing.llnl.gov/tutorials/mpi/samples/C/mpi\\_ringtopo.c](https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_ringtopo.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);
```

```
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 combination with nonblocking send/receive, to control execution flow between processes

```
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

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 sends a copy of the message on process with rank root to each process in comm

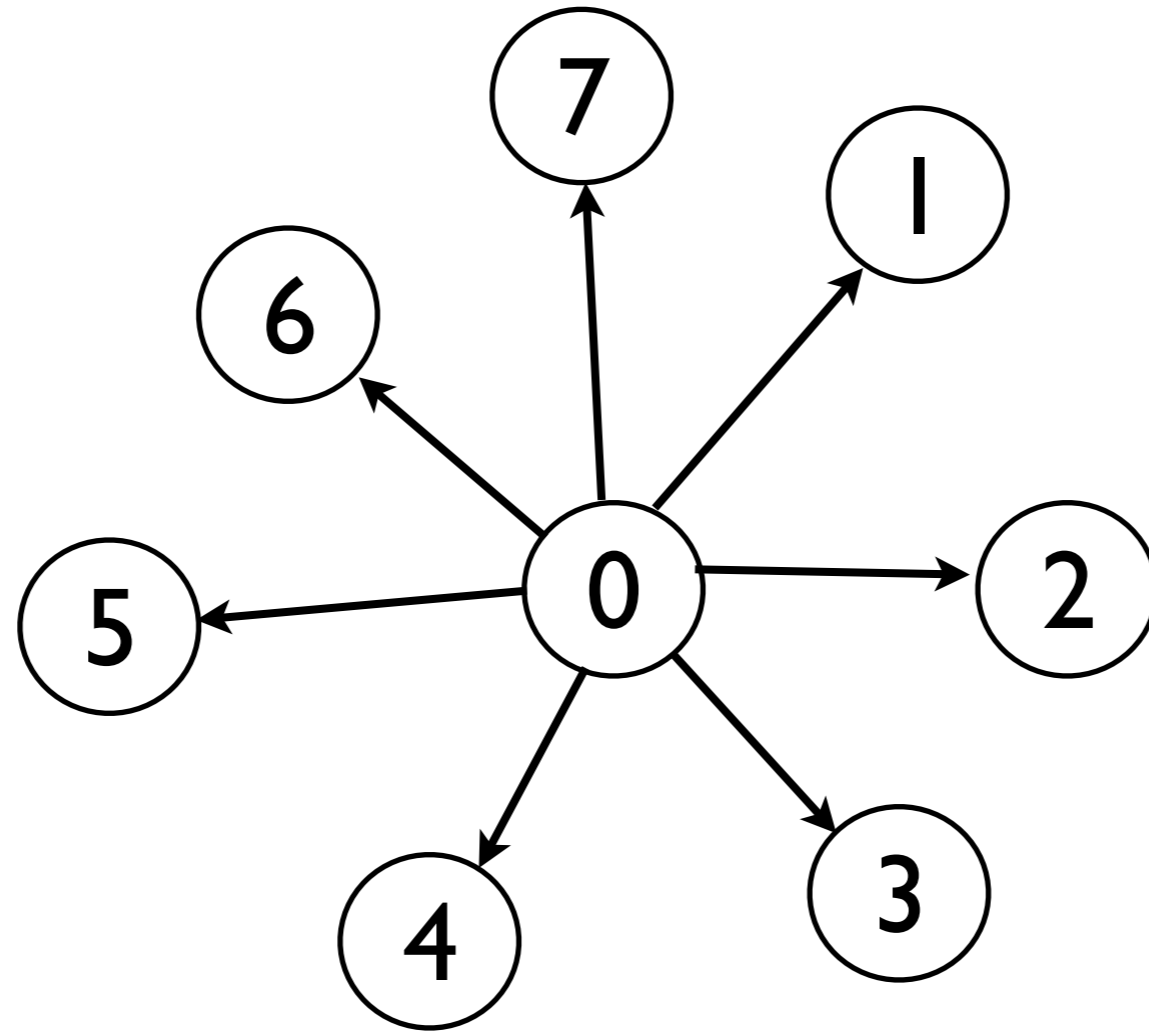
Must be called in each process

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

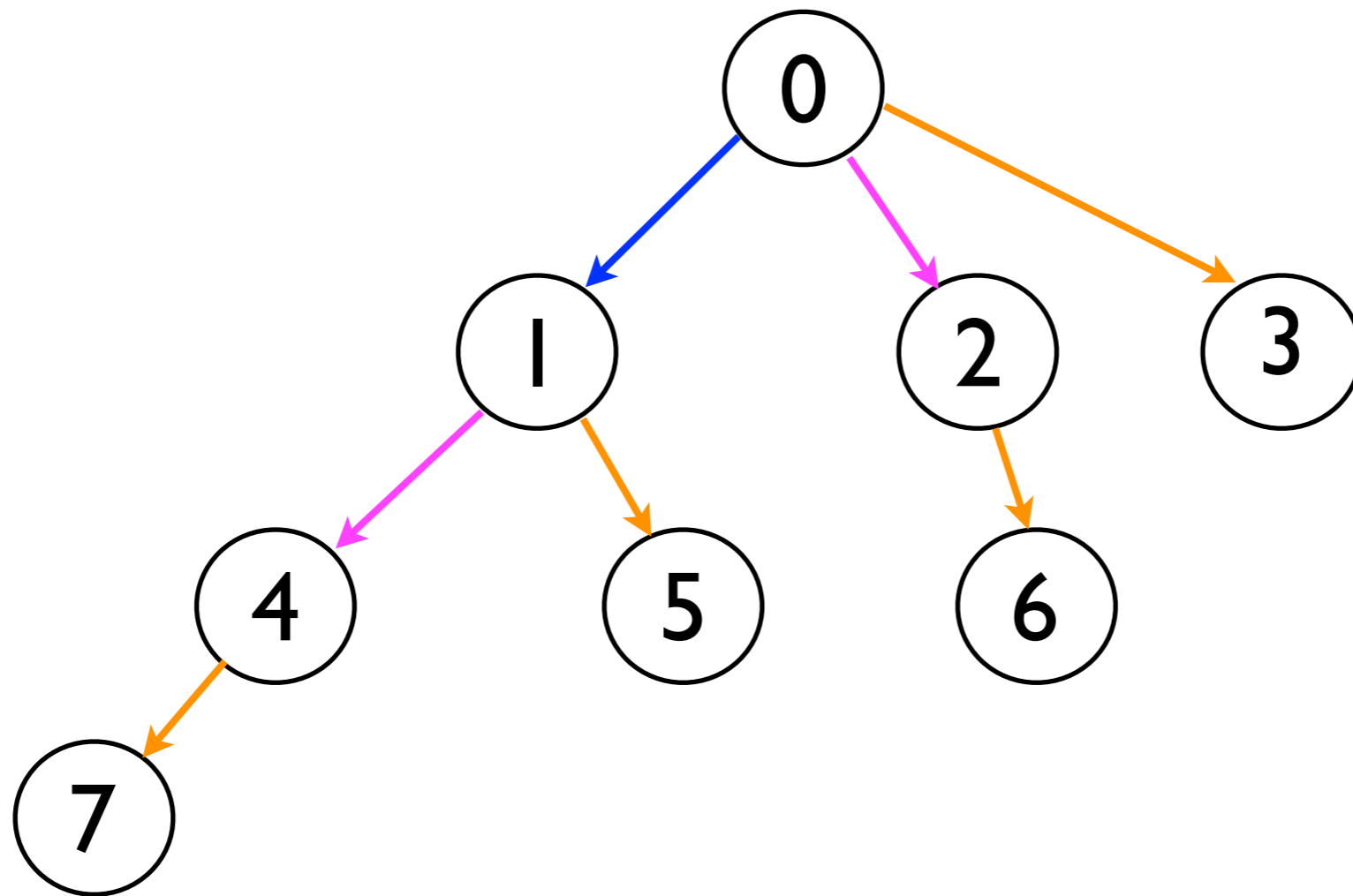
Cannot receive broadcasted data with MPI\_Recv

# Broadcast - poor implementation



Not parallel, 7 time steps needed

# Broadcast - actual, parallel implementation



3 time steps needed

# Example: reading and broadcasting data

Code adapted from P. Pacheco, PP with MPI

```
/* getdata2.c */

/* Function Get_data2
 * 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.
 */
```



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

void Get_data2(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

```
/* 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 */
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
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);
}

MPI_Finalize();
return 0;
}
```

# Example: Dot Product

Adapted from P. Pacheco, PP with MPI

```
/* parallel_dot.c -- compute a dot product of a vector
 * distributed among the processes.
 * Uses a block distribution of the vectors.
 * Input:
 *     n: global order of vectors
 *     x, y: the vectors
 *
 * Output:
 *     the dot product of x and y.
 *
 * Note: Arrays containing vectors are statically allocated.
 * Assumes n, the global order of the vectors, is divisible
 * by p, the number of processes.
 */
#include <stdio.h>
#include "mpi.h"

#define MAX_LOCAL_ORDER 100

void Read_vector(char* prompt, float local_v[], int n_bar,
                int p, int my_rank);
float Parallel_dot(float local_x[], float local_y[],
                  int n_bar);
```

```
main(int argc, char* argv[])
{
    float  local_x[MAX_LOCAL_ORDER];
    float  local_y[MAX_LOCAL_ORDER];
    int    n;
    int    n_bar; /* = n/p */
    float  dot;
    int    p, my_rank;

    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 vectors\n");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    n_bar = n/p;

    Read_vector("the first vector",
               local_x, n_bar, p, my_rank);
```

```

Read_vector("the second vector",
            local_y, n_bar, p, my_rank);

dot = Parallel_dot(local_x, local_y, n_bar);

if (my_rank == 0)
    printf("The dot product is %f\n", dot);

MPI_Finalize();
}

/*****
void Read_vector(
    char*   prompt      /* in */,
    float  local_v[]   /* out */,
    int     n_bar       /* in */,
    int     p           /* in */,
    int     my_rank     /* in */)
{
    int i, q;
    float temp[MAX_LOCAL_ORDER];
    MPI_Status status;

```

```

if (my_rank == 0)
{
    printf("Enter %s\n", prompt);
    for (i = 0; i < n_bar; i++)
        scanf("%f", &local_v[i]);
    for (q = 1; q < p; q++)
    {
        for (i = 0; i < n_bar; i++)
        {
            scanf("%f", &temp[i]);
            printf("temp %f\n", temp[i]);
        }
        MPI_Send(temp, n_bar, MPI_FLOAT, q, 0,
                 MPI_COMM_WORLD);
    }
}
else
    MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0,
             MPI_COMM_WORLD, &status);
} /* Read_vector */

```



```

/*****
float Serial_dot(float x[],float y[],int n)
{
    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */
/*****
float Parallel_dot(float local_x[],float local_y[],int n_bar)
{
    float  local_dot;
    float  dot = 0.0;
    float  Serial_dot(float x[],float y[],int m);

    local_dot = Serial_dot(local_x,local_y,n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
              MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */

```

# Allreduce

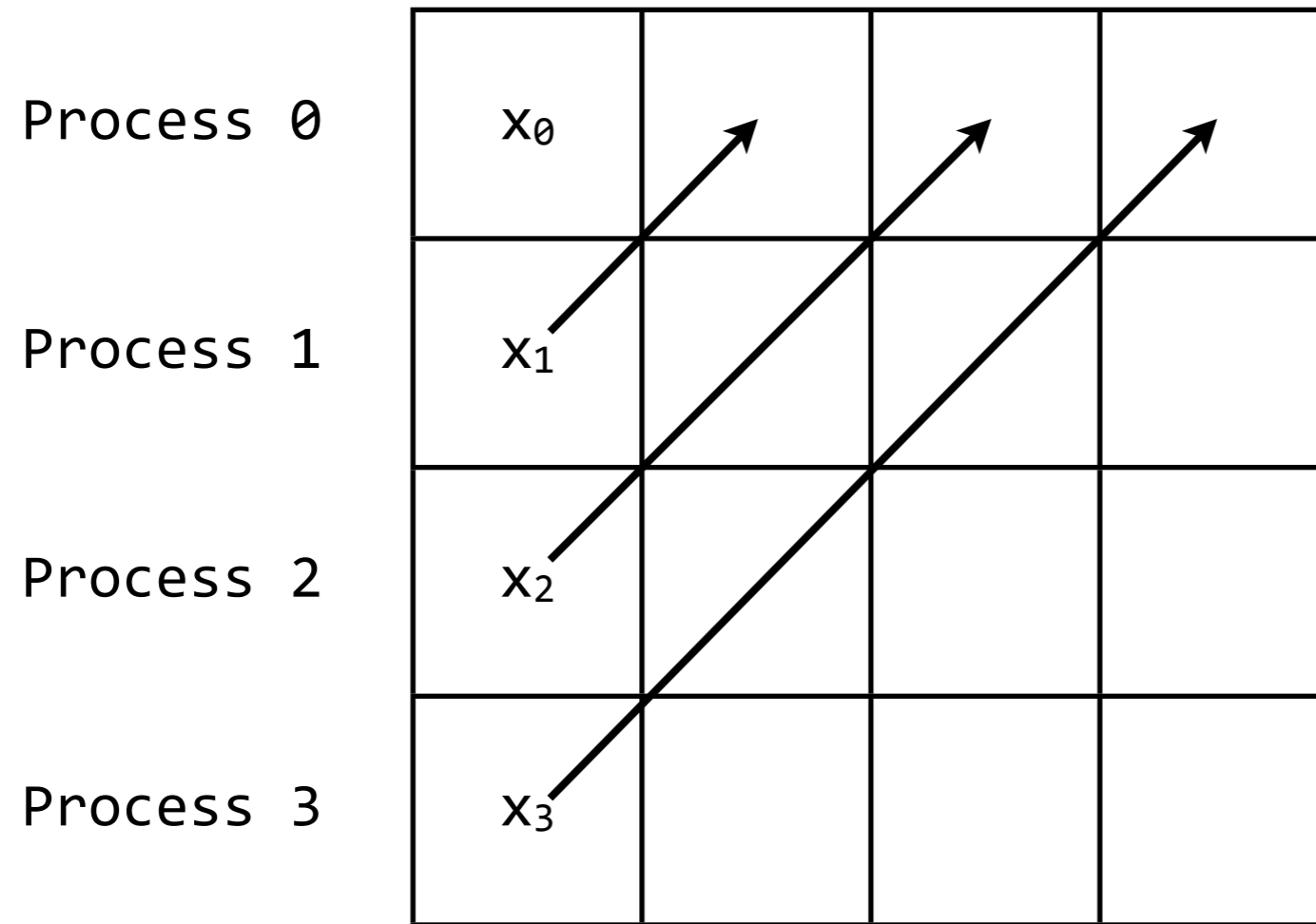
---

```
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

Gathers together data from a group of processes

---

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

---

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

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

If gathered data needs to be available to all processes, use:

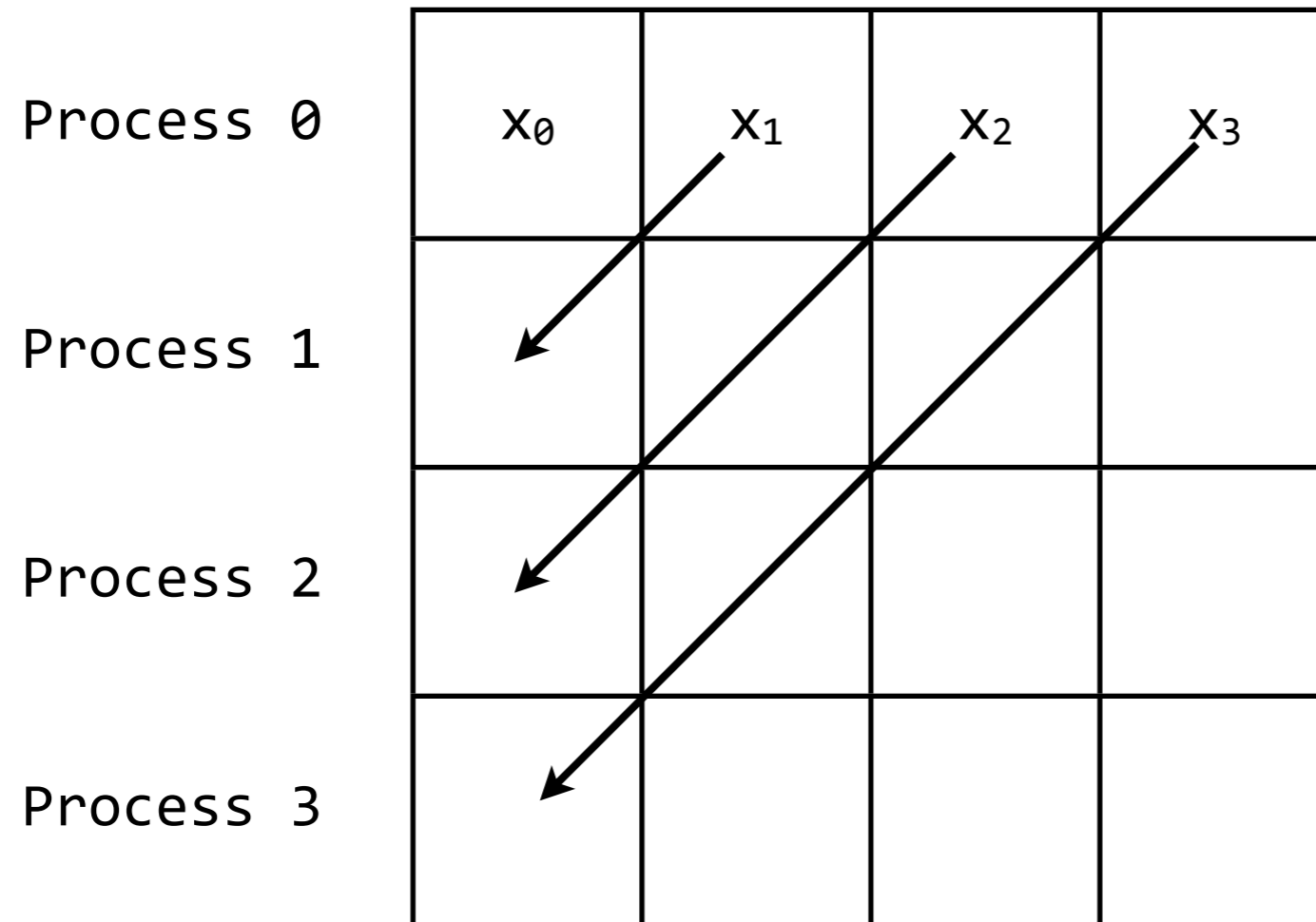
---

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

---

The block of data sent from the  $j$ th process is received by every process and placed in the  $j$ th block of the buffer `recvbuf`.

# Scatter



# Scatter

Sends data from one process to all other processes in a communicator

---

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

---

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



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

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, ...)
```

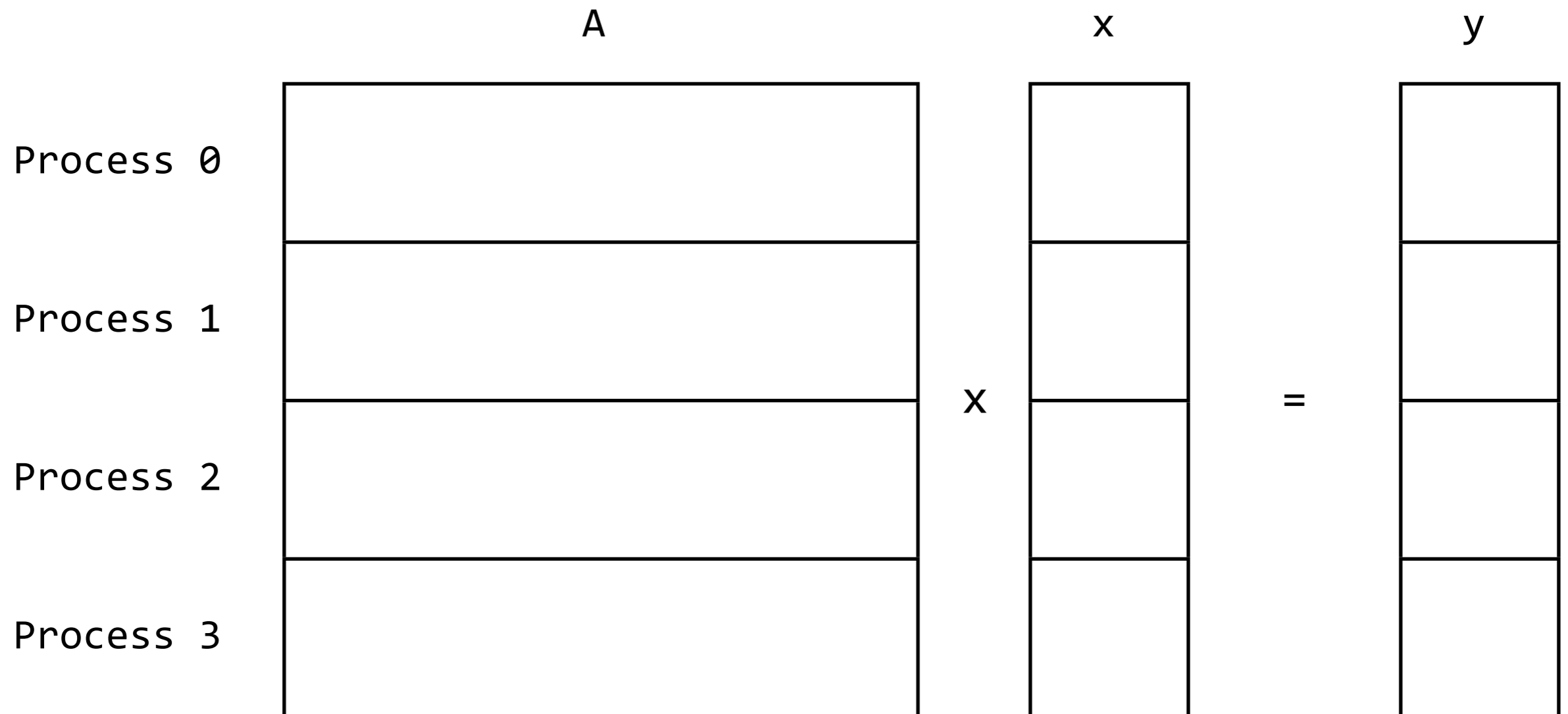
and each process executed a receive,

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

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

# Parallel Matrix Multiplication

$Ax=y$  - data distributed on 4 processes



# 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.
 */
```

```
#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;
}
```

```
/* 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);
```

```

/* 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];
  }
}

```

```
/* 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);
}
```



```

/* 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;

    /* Fill dummy entries in temp with zeroes */
    for (i = 0; i < p*local_m; i++)
        for (j = n; j < MAX_ORDER; j++)
            temp[i][j] = 0.0;

    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);
}

```

```
/* 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");
    }
}
```

```
/* 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

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