Lecture 9

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 of broadcast root

comm - communicator

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

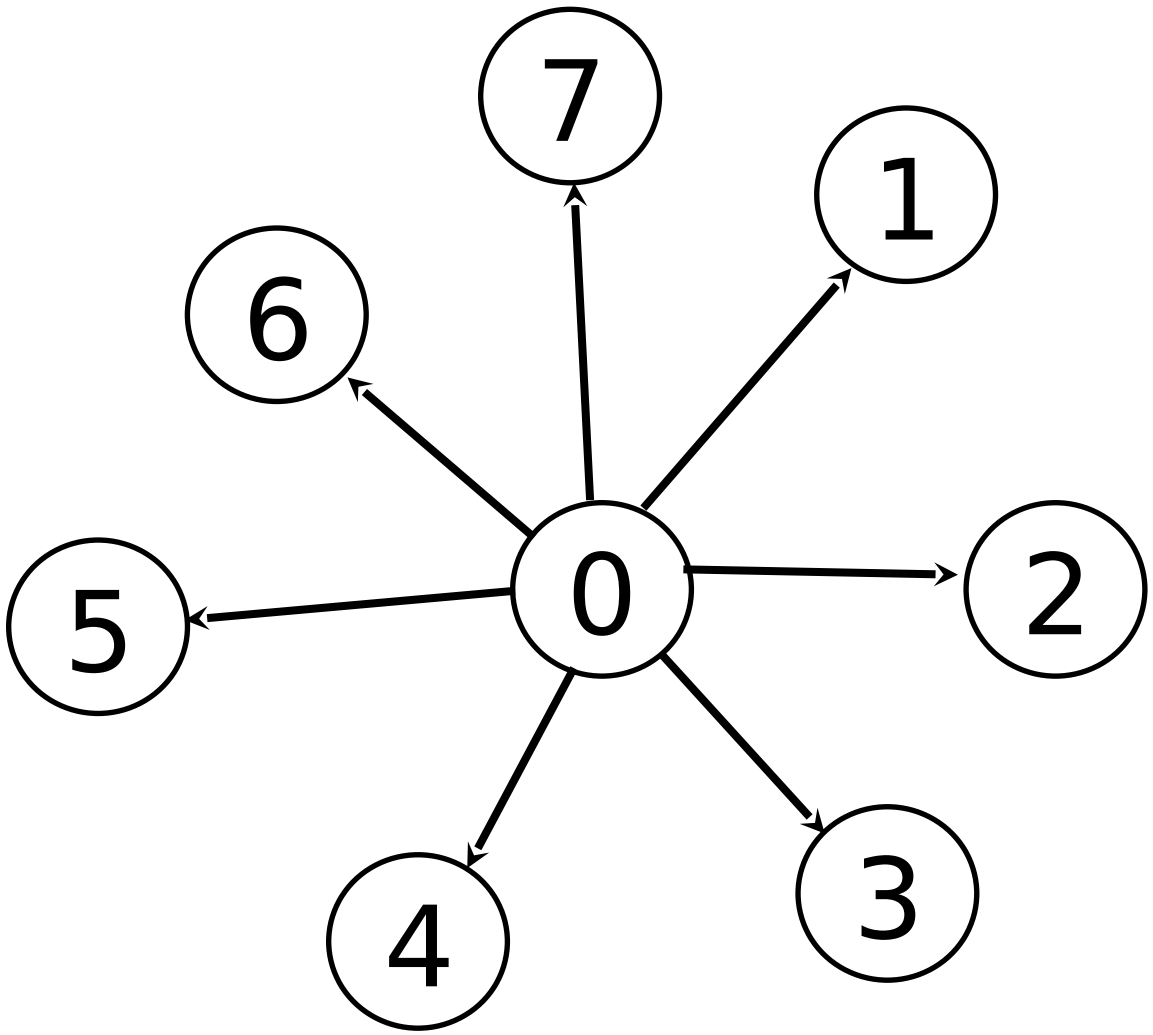
Must be called in each process.

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.

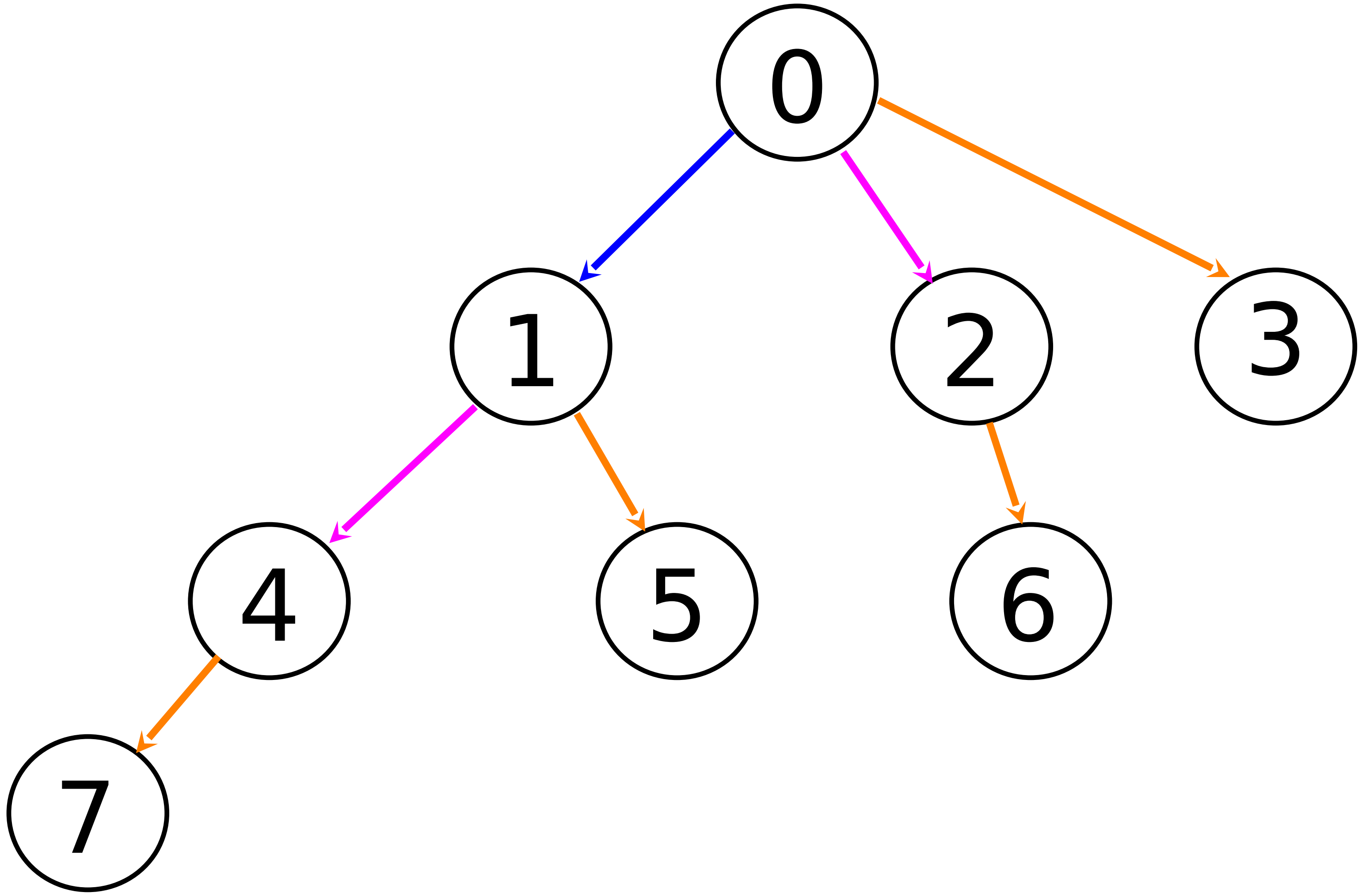
Cannot receive broadcasted data with MPI_Recv .

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.

MPI Reduce operations

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical “and” across the elements.
- MPI_LOR - Performs a logical “or” across the elements.
- MPI_BAND - Performs a bitwise “and” across the bits of the elements.
- MPI_BOR - Performs a bitwise “or” across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC

- MPI_2INT : pair of ints
- MPI_SHORT_INT : short and int
- MPI_LONG_INT : long and int
- MPI_LONG_DOUBLE_INT : long double and int
- MPI_FLOAT_INT : float and int
- MPI_DOUBLE_INT : double and int

MPI_MINLOC example

```
int myrank, minrank;
float minval;

struct {
  float value;
  int rank;
} in, out;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

in.value = A;
in.rank = myrank;

MPI_Reduce( in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
/* At this point, the answer resides on process root */
if (myrank == root) {
/* read answer out */
  minval = out.value;
  minrank = out.rank;
}
```

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);
```

~syam/ces745/mpi/collective

```
  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 vectors
 *  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);
```

~/ces745/mpi/collective/parallel_dot.c

```
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]);

          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 */
```

```c
/*****************************************************************/
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 */
```

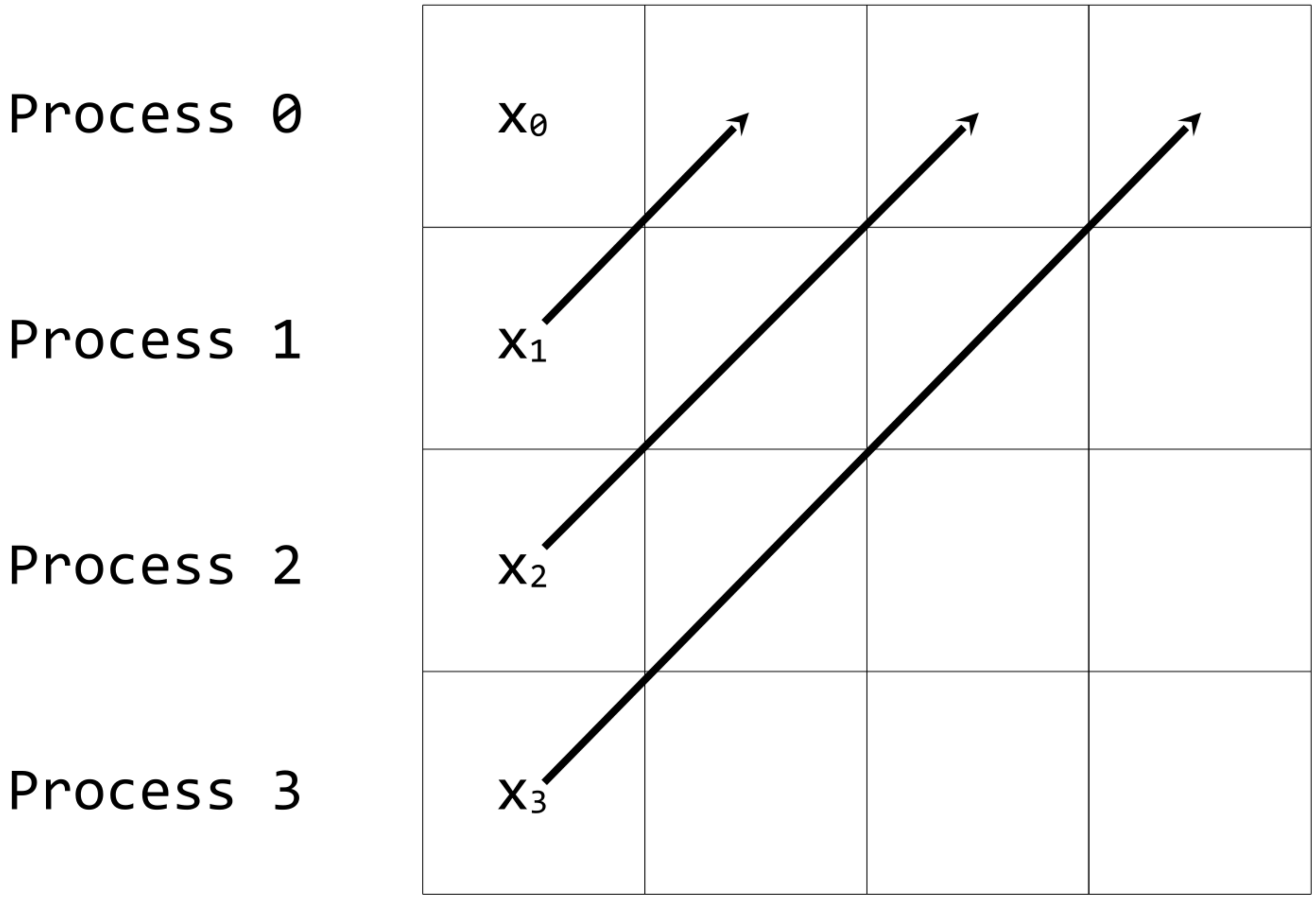
Allreduce

---

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 *recvbuf* 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.

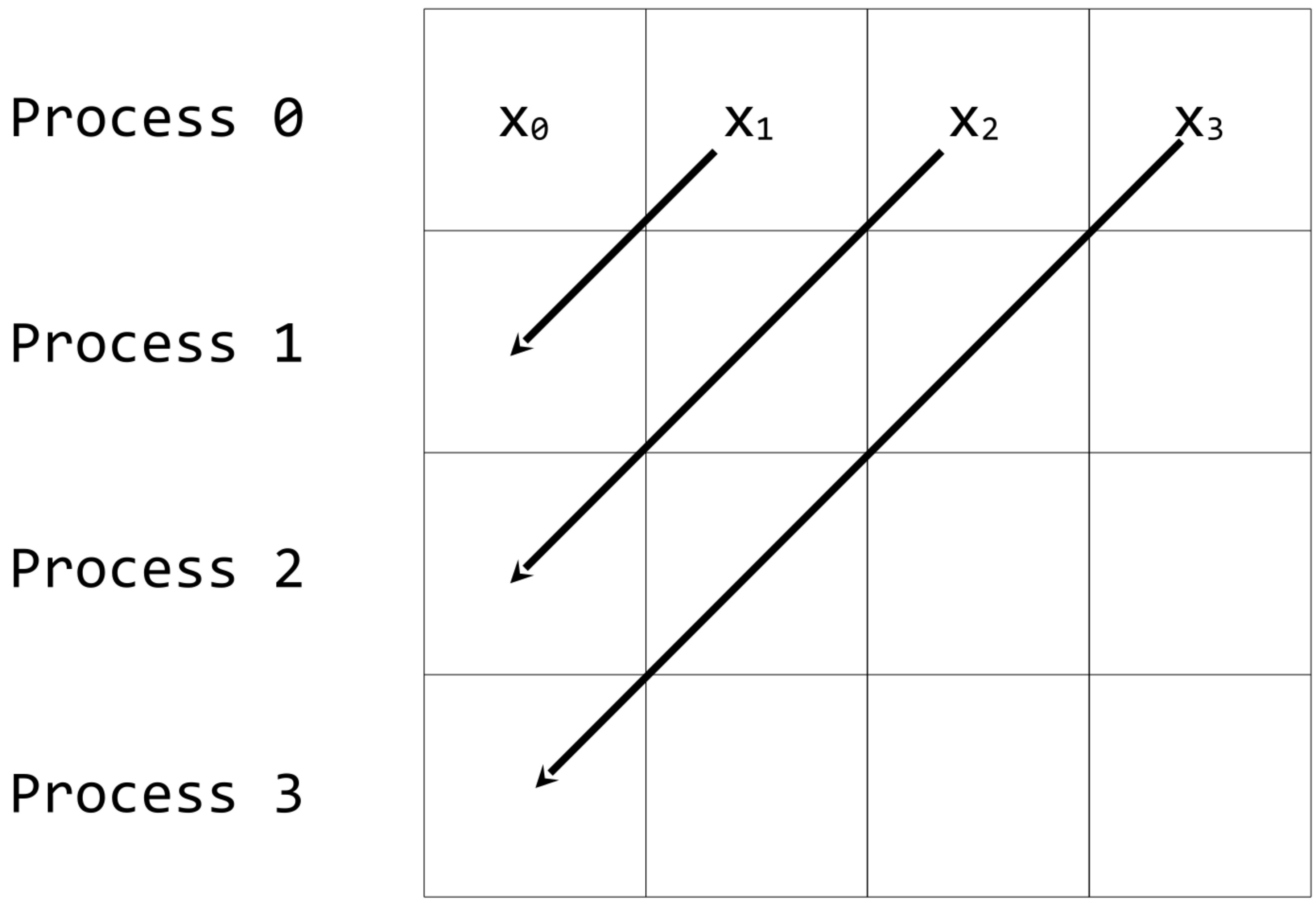
If gathered data needs to be available to all processes, use:

---

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, …)
```

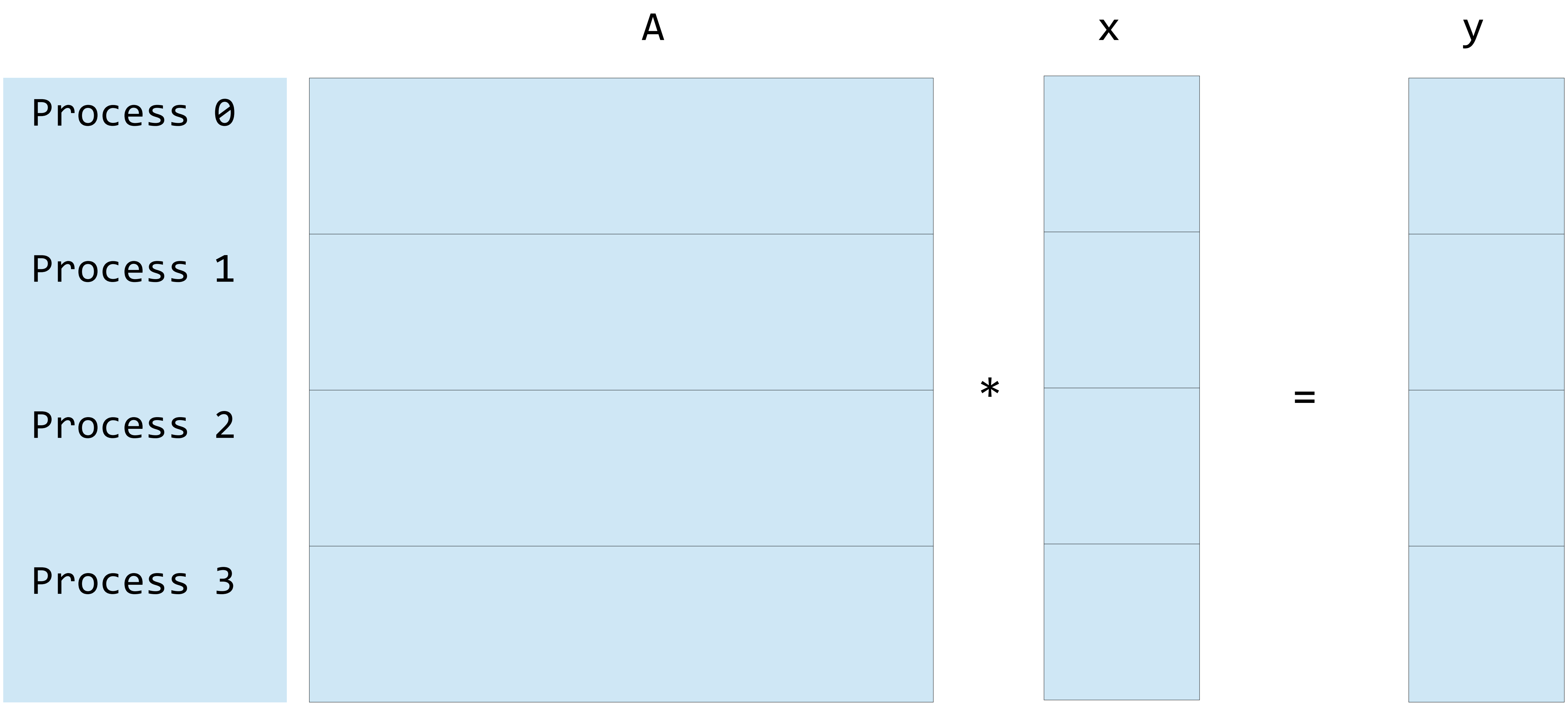
and each process executed a receive,

```
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.
 */
```

~syam/ces745/mpi/collective/parallel_mat_vect.c

```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;

  /* 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);}
```

```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");
      }
  }
}
```

Some details

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.

Communicators

Outline

- Communicators, groups, contexts
- When to create communicators
- Some group and communicator operations
- Examples

Communicators, groups, contexts

Processes can be collected into groups.

A group is an *ordered* set of processes.

- Each process has a unique rank in the group.
- Ranks are from 0 to $p - 1$, where p is the number of processes in the group.

A communicator consists of a:

- group

- context, a system-defined object that uniquely identifies a communicator

Every communicator has a unique context and every context has a unique communicator.

Two distinct communicators will have different contexts, even if they have identical underlying groups.

Each message is sent in a context, and must be received in the same context.

Communicators, groups, contexts. Cont.

A process is identified by its rank in the group associated with a communicator.

MPI_COMM_WORLD is a default communicator, whose group contains all initial processes.

A process can create and destroy groups at any time without reference to other processes—local to the process.

The group contained within a communicator is agreed across the processes at the time when the communicator is created.

Two types of communicators exist:

Intra-communicator is a collection of processes that can send messages to each other and engage in collective communications. This is the more important type, and this lecture will focus on those.

Inter-communicator are for sending messages between processes of disjoint intra-communicators. These are less important for most MPI programming tasks, so we will not be covering them.

When to create a new communicator?

To achieve modularity; e.g. a library can exchange messages in one context, while an application can work within another context. (Use of tags is not sufficient, as we need to know the tags in other modules.)

To restrict a collective communication to a subset of processes.

To create a virtual topology that fits the communication pattern better.

Some group and communicator operations

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
```

Returns a handle to the group associated with *comm* .

Obtain group for existing communicator: MPI_Comm -> MPI_Group .

MPI_Group_incl

```
int MPI_Group_incl (MPI_Group group , int n, int *ranks,
          MPI_Group *new_group)
```

Creates a new group from a list of processes in old group (a subset).

E.g., can be used to reorder the elements of a group.

The number of processes in the new group is *n* .

The processes to be included are listed at *ranks* .

Process *i* in *new_group* has rank *rank[i]* in *group* .

MPI_Comm_create

```
int MPI_Comm_create (MPI_Comm comm, MPI_Group new_group ,
             MPI_Comm *new_comm)
```

Associates a context with new group and creates *new_comm* .

All the processes in *new_group* belong to the group of the underlying *comm* .

This is a collective operation.

All process in comm must call MPI_Comm_create, so all processes choose a single context for the new communicator.

Example

Code from N. Nedialkov

~syam/ces745/mpi/communicators/comm.c

```
/* comm.c */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define NPROCS 8

int main(int argc, char *argv[])
{
  int rank, new_rank,
    sendbuf, recvbuf,
    Numtasks;
  int ranks1[4]={0,1,2,3};
  int ranks2[4]={4,5,6,7};

  MPI_Group  orig_group, new_group;
  MPI_Comm   new_comm;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

```
if (numtasks != NPROCS && rank==0)
  {
    printf("Must specify MP_PROCS = %d. Terminating.\n",
           NPROCS);
    MPI_Finalize();
    Exit(0);
  }

/* store the global rank in sendbuf */
sendbuf = rank;

/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < numtasks/2)
  /* if rank = 0,1,2,3, put original processes 0,1,2,3
      into new_group */
  MPI_Group_incl(orig_group, 4, ranks1, &new_group);
else
  /* if rank = 4,5,6,7, put original processes 4,5,6,7
     into new_group */
  MPI_Group_incl(orig_group, 4, ranks2, &new_group);
```

```
  /* Create new communicator and then perform collective
communications */
  MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);

  /* new_comm contains a group with processes 0,1,2,3
     on processes 0,1,2,3 */
  /* new_comm contains a group with processes 4,5,6,7
     on processes 4,5,6,7 */
  MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT,
                MPI_SUM, new_comm);

  /* new_rank is the rank of my process in the new group */
  MPI_Group_rank (new_group, &new_rank);

  printf("rank= %d newrank= %d recvbuf= %d\n",
         rank,new_rank,recvbuf);

  MPI_Finalize();

  return 0;
}
```

MPI_Comm_split

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,
          MPI_Comm *comm_out)
```

Partitions the group associated with *comm* into disjoint subgroups, one for each value of *color*.

Each subgroup contains all processes marked with the same color.

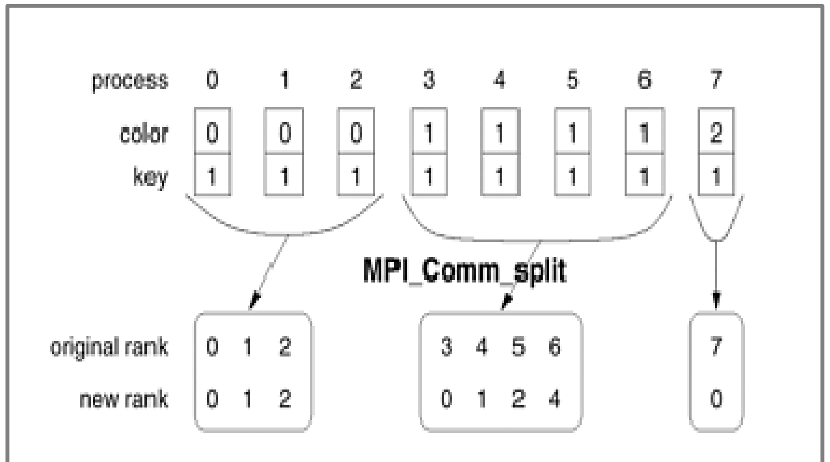
Within each subgroup, processes are ranked in order defined by the value of *key*.

Ties are broken according to their rank in the old group.

A new communicator is created for each subgroup and returned in *comm_out*.

Although a collective operation, each process is allowed to provide different values for color and key.

The value of color must be greater than or equal to 0.



$N=q*q$ processes arranged on a q by q grid ($q=3$ in this example)

Row 0	0	1	2
Row 1	3	4	5
Row 2	6	7	8

Need to define communicator for each row of processes i.e. processes {0,1,2}, {3,4,5} and {6,7,8}, for easy communication between them.

MPI_Comm_split provides an easy way to do this, more convenient than using MPI_Group_incl and MPI_Comm_create .

Example

Code from P. Pacheco, PP with MPI

```
/* comm_split.c -- build a collection of q
   communicators using MPI_Comm_split
 * Input: none
 * Output:  Results of doing a broadcast across each of
            the q communicators.
 * Note:  Assumes the number of processes, p = q^2
 */
#include <stdio.h>
#include "mpi.h"
#include <math.h>

int main(int argc, char* argv[])
{
  int       p, my_rank;
  MPI_Comm  my_row_comm;
  int       my_row, my_rank_in_row;
  int       q, test;

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

~syam/ces745/mpi/communicators/comm_split.c

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

  q = (int) sqrt((double) p);
 /* my_rank is rank in MPI_COMM_WORLD.
      q*q = p */
  my_row = my_rank/q;
  MPI_Comm_split (MPI_COMM_WORLD, my_row, my_rank,
                 &my_row_comm);

  /* Test the new communicators */
  MPI_Comm_rank (my_row_comm, &my_rank_in_row);
  if (my_rank_in_row == 0) test = my_row;
  else test = 0;

  MPI_Bcast (&test, 1, MPI_INT, 0, my_row_comm);

  printf("Process %d > my_row = %d,"
         "my_rank_in_row = %d, test = %d\n",
         my_rank, my_row, my_rank_in_row, test);
  MPI_Finalize();
  return 0;
}
```

Implementation

Groups and communicators are opaque objects.

The details of their internal representation depend on the particular implementation, and so they cannot be directly accessed by the user.

To use these objects, the user accesses a handle that references the opaque object, and the opaque objects are manipulated by special MPI functions.