Overview

Message Passing Programming

- A message passing program is executed by **multiple processes**.
  Each of these processes can execute a **different program** (MPMD).
  Simplification: Each process executes the **same program**.
  (**SPMD - single program multiple data**).
- Each process has a **separate address space**. Processes can exchange data by **sending messages**.
  Basic exchange operation: send/receive.
- Messages are sent using **explicit communication operations**.
- Communication libraries providing standardized interfaces are used.
  PVM (Parallel Virtual Machine), **MPI** (Message Passing Interface)
  $\rightsquigarrow$ portable programs

Overview

MPI – Message Passing Interface – Introduction

- MPI interfaces are available for C, FORTRAN 77 and C++ (MPI-2)
  In the following, we concentrate on the interface for C;
  further information: `www.mpi-forum.org`
- An MPI program consists of a **fixed number of processes**.
  The number of processes is determined at program start.
- Each processor executes a single process of the program.
- Program execution is based on the **SPMD model**;
  Processes are distinguished using the unique process number that is assigned at program start.
- MPI supports the following **communication operations**:
  - **point-to-point communication operations** to exchange data between two processes;
  - **global communication operations** to exchange data between multiple processes;
    global/collective communication operations

MPI – Classification of the Communication Operations

- Distinction on the basis of the **local view** of the processes:
  - **Blocking operation:** Control returns to the calling process not until all resources used by the sender/receiver to execute the operation are freed;
    $\rightarrow$ Resources can be reused immediately.
  - **Non-blocking operation:** Control may return to the calling process before the completion of the operation;
    $\rightarrow$ Resources used for the operation (e.g. buffers) may not be manipulated before the operation has signaled its completion to the calling process by setting appropriate flags.
- Global communication operations are always blocking in MPI.
- Distinction on the basis of the **global view** of the processes:
  - **Synchronous communication:** Transmission of the message only takes place, if sender and receiver participate at the same time at the communication.
  - **Asynchronous communication:** Sender can transmit data without ensuring that the receiver is ready to receive the data.

Starting and Initializing an MPI Program

- Execute an MPI program using the command:
  mpirun -np 4 <program name> <arguments>
- Initialization of the MPI runtime library:
  MPI_Init (&argv, &argc)
- Obtaining the local process number:
  MPI_Comm_rank (MPI_Comm, &num)
- Obtaining the total number of processes of the program:
  MPI_Comm_size (MPI_Comm, &count)
- Processes are combined into groups to coordinate the participation in the communication operations executed.
  The **processes of a group** can exchange data using **communicators**.
  Each **communication operation** requires a **communicator** as a parameter.
  The default communicator *MPI_COMM_WORLD* comprises all processes.

Overview

Sending Data using MPI Point-to-Point Communication

Basic form of a data exchange: operation of the **sender** in MPI:

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

- **smessage** Send buffer containing the elements to be sent successively;
- **count** Number of elements to be sent;
- **datatype** Data type common to all elements to be sent;
- **dest** Rank of the target process that should receive the data;
- **tag** Additional message tag (between 0 and 32767) to distinguish different messages of the same sender;
- **comm** Communicator of the underlying processor group;
- Data types available in MPI:
  - *MPI_{CHAR, SHORT, INT, LONG, BYTE, UNSIGNED, FLOAT, DOUBLE}*
  - *MPI_UNSIGNED_{CHAR, SHORT, LONG}*, *MPI_LONG_{DOUBLE}*, *MPI_PACKED*

Receiving Data with MPI Point-to-Point Communication (1)

The following point-to-point receive operation corresponds to `MPI_Send`:

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

- **rmessage** Buffer of adequate size to receive the message;
- **count** Upper limit of the number of elements to accept;
- **datatype** Data type of the elements to be received;
- **source** Rank of the process from which to receive a message;
- **tag** Message tag of the message to be received;
- **comm** Communicator of the underlying processor group;
- **status** Data structure to be filled with information on the message received;

`MPI_Send()` and `MPI_Recv()` are **blocking** and **asynchronous** operations.

Receiving Data with MPI Point-to-Point Communication (2)

- Arbitrary messages can be received using:
  - **source**=*MPI_ANY_SOURCE* to receive a message from an arbitrary sender;
  - **tag**=*MPI_ANY_TAG* to receive a message with an arbitrary tag.
  - The sender/tag of the message received can be obtained by inspecting **status**.*MPI_*{*SOURCE, TAG*} after completion of the receive operation.
- The number of data elements transmitted to the receiver can be obtained from the data structure `status`:

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count_ptr)
```

  - **status** Pointer to the data structure returned by the corresponding call to `MPI_Recv()`.
  - **count_ptr** Address of a variable in which the number of elements received are returned.

Example: Message Passing from Process 0 to Process 1

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

int main (int argc, char *argv[]) {
   int my_rank, p, source, dest, tag=0;
   char msg [20];
   MPI_Status status;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
   if (my_rank == 0) {
      strcpy (msg, "Hello");
      MPI_Send (msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
   }
   if (my_rank == 1)
      MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Finalize ();
}
```

Implementation of Message Transfers in MPI

Internally a message transfer is implemented in **three steps**:

1. The data elements are copied from the **send buffer** `smessage` into a **system buffer** and the message to be sent is **assembled** by adding a **header** with information on the **sender** and the **receiver** of the message, the **tag**, and the **communicator**.
2. The message is sent via the **network** of the parallel platform **from the sender to the receiver**.
3. The **receiver** `dest` copies the data elements from the **system buffer** used to receive the message into the **receive buffer** `rmessage`.

Semantics of Blocking, Asynchronous Communication Operations

- An `MPI_Recv()` operation can also be **started** when the corresponding `MPI_Send()` operation has **not yet been started**.
  The `MPI_Recv()` operation **blocks** until the **receive buffer** specified contains the data elements sent.
- An `MPI_Send()` operation can also be **started** when the corresponding `MPI_Recv()` operation has **not yet been started**.
  The `MPI_Send()` operation **blocks** until the **send buffer** specified can be **reused**, i.e., new data elements can be copied to the buffer.

Implementation Variants of Blocking MPI Operations

a) Direct copy without a system buffer
The message is copied **directly** without using an intermediate **system buffer** into the **receive buffer** of another process.
→ The `MPI_Send()` operation is **blocked** until the **entire** message has been copied into the **receive buffer**.
In particular, control returns not until the corresponding `MPI_Recv()` operation has been **started**.

b) Intermediate buffering with a system buffer
The sender first copies the message to an internal **system buffer** .
The **sending process** can continue its operation as soon as the **copy operation** of the **sending side** is completed
(even before the start of the corresponding `MPI_Recv()` operation).
Advantage: the sender is **blocked** only for a short period of time;
Drawback: additional memory space for the system buffer;
additional execution time for copying data into the system buffer.

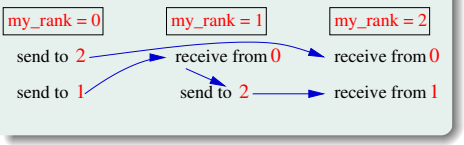
Order of MPI Communication Operations

- Each MPI implementation guarantees that messages of the same process do **not overtake** each other:
  If a sender sends two messages **one after another** to the **same receiver** and **both messages fit** to the first `MPI_Recv()` called by the receiver, the MPI runtime system ensures that the **first message sent will always be received first**.
- The involvement of a **third process** may disturb the order and may lead to a **violation** of the delivery order desired.

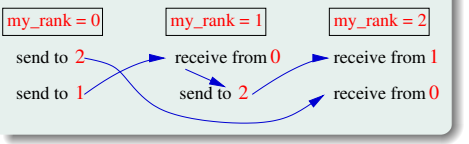
Example: Violation of the Delivery Order

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0 ) {
  MPI_Send(sendbuf1, count, MPI_INT, 2, tag, comm);
  MPI_Send(sendbuf2, count, MPI_INT, 1, tag, comm); }
else if (my_rank == 1) {
MPI_Recv(recvbuf1, count, MPI_INT, 0, tag, comm, &status);
MPI_Send(recvbuf1, count, MPI_INT, 2, tag, comm); }
else if (my_rank == 2) {
MPI_Recv(recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
MPI_Recv(recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
}
```

Expected delivery order:



Reversed delivery order:



Overview

Deadlocks with Blocking Point-to-Point Communication

- Careless usage of send and receive operation may lead to **deadlocks**.
- **Example:** Program fragment that leads to a **deadlock**

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm); }
else if (my_rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm); }
```

- Sequence of communication:

  Process 0 starts MPI_Recv() and process 1 starts MPI_Recv()
  → Process 0 wants to start MPI_Send() but has to wait for the termination of MPI_Recv()
  → this MPI_Recv() waits for the execution of MPI_Send() by process 1
  → this depends on the termination of MPI_Recv() by process 1
  → this MPI_Recv() waits for the execution of MPI_Send() by process 0

Deadlocks depending on the Implementation

- The occurence of a deadlock might also depend on the implementation of the MPI runtime system (usage of system buffers):
- **Example:** Implementation dependent **deadlock**

```
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status); }
else if (my_rank == 1) {
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status); }
```

- MPI implementation **with a system buffer** $\rightarrow$ **correct execution without a deadlock**. The messages sent are copied from the send buffer `sendbuf` into a system buffer. Control returns after the copy operation immediately to the sender.
- MPI implementation **without a system buffer** $\rightarrow$ **deadlock occurs**. Neither of the two processes can complete its `MPI_Send()` operation since the other process cannot execute the corresponding `MPI_Recv()`.

Secure Implementation with Blocking Communication

- An MPI program is called **secure** if its correctness does not depend on assumptions about the **existence of system buffers** or the **size of system buffers**.
  The MPI standard does **not specify** the provision of system buffers.
- In **absence of a system buffer** the operation must have **exclusive access to the program buffer** until the transfer is completed.
- **Example:** Program fragment without **deadlocks**

```
if (my_rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Secure Communication with $p > 2$ Processes (1)

- A secure implementation of point-to-point communication with more than two processes each sending and receiving a message is achieved by an **exact specification in which order** the send and receive operations are to be performed.
- **Example:** MPI program with p processes:
  Process i sends a message to process $(i+1) \bmod p$, $0 \leq i \leq p-1$
  Process i receives a message from process $(i-1) \bmod p$
  Thus, the messages are sent in a **logical ring**.
- A **secure implementation** cen be obtained by the following rule:
  Processes with an **even number** send first and then receive;
  Processes with an **odd number** receive first and then send.
- Exchange scheme for an **even number of processes**

Time	Process 0	Process 1	Process 2	Process 3
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 3	`MPI_Recv()` from 2
2	`MPI_Recv()` from 3	`MPI_Send()` to 2	`MPI_Recv()` from 1	`MPI_Send()` to 0

Secure Communication with $p > 2$ Processes (2)

- This scheme also leads to a secure implementation for an **odd number of processes**.
- Exchange scheme for an **odd number of processes**

Time	Process 0	Process 1	Process 2
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 0
2	`MPI_Recv()` from 2	`MPI_Send()` to 2	-wait-
3		-wait-	`MPI_Recv()` from 1

- Some communication operations like the `MPI_Send()` operation of process 2 can be **delayed** because the receiver calls the corresponding `MPI_Recv()` operation at a **later time**. But a **deadlock cannot occur**.

Data Exchange with MPI_Sendrecv() (1)

Situation: Each process **sends** and **receives** data

```
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)
```

- **sendbuf** Send buffer in which the data elements to be sent are stored;
- **sendcount** Number of data elements to be sent;
- **sendtype** Data type of the elements to be sent;
- **dest** Rank of the target process to which the data elements are sent;
- **sendtag** Tag for the message to be sent;
- **recvbuf** Receive buffer for the message to be received;
- **recvcount** Maximum number of data elements to be received;
- **recvtype** Data type of the data elements to be received;
- **source** Rank of the process from which the message is expected;
- **recvtag** Tag of the message to be received;
- **comm** Communicator used for the communication;
- **status** Data structure to store information on the message received.

Data Exchange with MPI_Sendrecv() (2)

- **Advantage** of MPI_Sendrecv(): The runtime system guarantees deadlock freedom.
- **Prerequisit**: sendbuf and recvbuf must be **disjoint, non-overlapping memory locations**.
- Messages of **different lengths** and **different data types** may be exchanged.
- If send and receive buffers are **identical**, the following MPI operation may be used:

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

- **buffer** Buffer that is used as both send and receive buffer.
- **Prerequisit:** The **number count** and the **data type type** of the data elements to be sent and to be received have to be **identical**.
  The runtime system is responsible for the temporary storage in system buffers when needed.

Overview

Non-blocking Communication with MPI

- **Blocking communication operations** do not fully utilize the communication hardware of parallel platforms that often operates asynchronously;
  $\rightarrow$ **Avoiding waiting times** with **non-blocking** operations.
- Properties of **non-blocking communication operations**
  - **Control** is **returned to the caller** immediately without blocking; the **communication operation** is only **initiated**.
  - The runtime system sends the message to the receiver **asynchronously**; $\rightarrow$ the program can perform other computations concurrently to the send operation;
  - **However:** The send buffer specified cannot be reused safely as long as the send operation is in progress.
  - The **sending process** is **notified** by the runtime system when the send operation is terminated using a `request` structure.

Non-blocking Send and Receive Operations

- Non-blocking **send operation**

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

  The parameters have the same meaning as for `MPI_Send()`.
  `MPI_Request` denotes an opaque object that **identifies a specific non-blocking communication operation**.

- Non-blocking **receive operation**

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

  Starting the receive operation informs the runtime system that the **receive buffer** specified is ready to **receive data**.

Status of Non-blocking Communication Operations

- Querying the **status of a non-blocking communication operation**

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

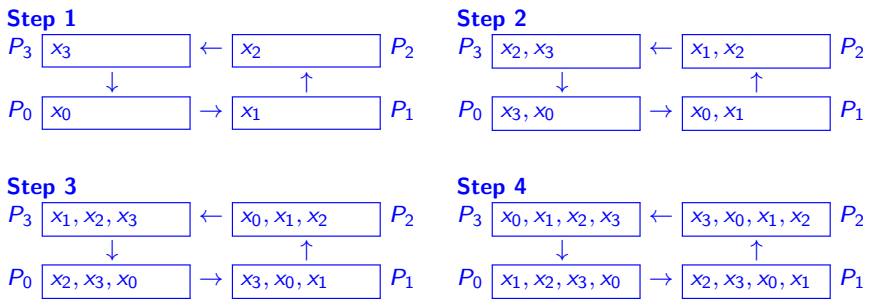
  - The call returns **flag = 1** if the send or receive communication operation specified by `request` has been completed; **flag = 0** denotes that the operation **is still in progress**.
  - If `MPI_Test()` is called for a **receive operation** that is completed the parameter `status` contains information on the message received as described by `MPI_Recv()`.

- Waiting for the **completion of a communication operation** The following MPI operation **blocks the calling process** until the send or receive operation specified by `request` is **completed**.

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

Example: Collection of Information from p Processes (1)

- Each of the p processes available stores the **same number of data elements in consecutive memory locations**. The data elements of each process should be made available to all other processes.
- The processes are logically arranged in a **ring**; Implementation using $p - 1$ steps:
- Illustration for $p = 4$ processes



Example: Collection of Information from p Processes (2)

Assumptions: Local data is provided in array **x**; Entire data is collected in array **y**; Blocking communication operations are used.

```
void Gather_ring (float x[], int blocksize, float y[]) {
   int i, p, my_rank, succ, pred, send_offset, recv_offset;
   MPI_Status status;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<blocksize; i++) y[i+my_rank * blocksize] = x[i];
   succ = (my_rank+1) % p;
   pred = (my_rank-1+p) % p;
   for (i=0; i<p-1; i++) {
      send_offset = ((my_rank-i+p) % p) * blocksize;
      recv_offset = ((my_rank-i-1+p) % p) * blocksize;
      MPI_Send (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
                MPI_COMM_WORLD);
      MPI_Recv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
                MPI_COMM_WORLD, &status);
}}
```

Deadlock freedom is ensured only if the MPI runtime system uses **system buffers** that are large enough.

Example: Collection of Information from p Processes (3)

Assumption: Non-blocking communication operations are used.

```
void Gather_ring (float x[], int size, float y[]) {
   int i, p, my_rank, succ, pred, send_offset, recv_offset;
   MPI_Status status;
   MPI_Request send_requ, recv_requ;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<size; i++) y[i+my_rank * size] = x[i];
   succ = (my_rank+1) % p; pred = (my_rank-1+p) % p;
   send_offset = my_rank * size;
   recv_offset = ((my_rank-1+p) % p) * size;
   for (i=0; i<p-1; i++) {
      MPI_Isend (y+send_offset, size, MPI_FLOAT, succ, 0,
                 MPI_COMM_WORLD, &send_requ);
      MPI_Irecv (y+recv_offset, size, MPI_FLOAT, pred, 0,
                 MPI_COMM_WORLD, &recv_requ);
      send_offset = ((my_rank-i-1+p) % p) * size;
      recv_offset = ((my_rank-i-2+p) % p) * size;
      MPI_Wait (&send_requ, &status);
      MPI_Wait (&recv_requ, &status);
}}
```

Synchronous Communication Mode

- The communication operations described until now use the **standard mode** of communication: The **runtime system** decides whether outgoing messages are **buffered** in a system buffer or transmitted directly **without buffering** to the receiver.
- Alternative: **Synchronous Mode:** A **send operation** will be **completed** not before the corresponding receive operation has started with the **reception of data**.
  $\rightarrow$ **Synchronization** between sender and receiver.
- **blocking send operation in synchronous mode:** `MPI_Ssend()`
  (Parameters identical to `MPI_Send()`)
- **non-blocking send operation in synchronous mode:** `MPI_Issend()`
  (Parameters identical to `MPI_Isend()`)
  There is **no synchronisation** between `MPI_Issend()` and the corresponding `MPI_Irecv()`; **Synchronization** between sender and receiver is performed when the **sender** calls `MPI_Wait()`.

Buffered Communication Mode

- **Buffered Mode:** Control will be **returned** by the send operation even if the corresponding **receive operation has not yet been started**.
  → The runtime system must **buffer** the outgoing message.
  - **blocking send operation in buffered mode:** `MPI_Bsend()`
    (Parameters identical to `MPI_Send()`)
  - **non-blocking send operation in buffered mode:** `MPI_Ibsend()`
    (Parameters identical to `MPI_Isend()`)
- The **buffer space** to be used by the runtime system must be **provided by the programmer**. **Provision of a buffer**:

  ```
  int MPI_Buffer_attach (void *buffer, int buffersize)
  ```

  buffersize is the size of the buffer `buffer` in **bytes**.
- **Detaching a buffer** previously provided:

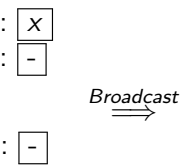
  ```
  int MPI_Buffer_detach (void *buffer, int *size)
  ```

- For **receive operations**, MPI provides the **standard mode** only.

Overview

Broadcast Operation

- All **global** communication operations are **blocking** in MPI.
- **Broadcast Operation:** The **root process** `root` sends the **same data block** to **all** other processes of the group.
  For a broadcast, **each process** has to call the following MPI function:

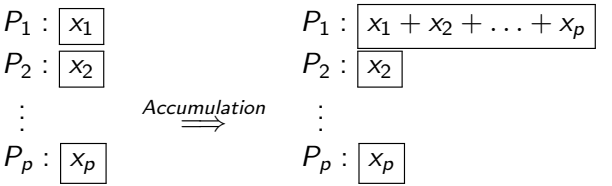



```
int MPI_Bcast (void *message,
               int count,
               MPI_Datatype type,
               int root,
               MPI_Comm comm)
```

- The **root process** provides the **data block** to be sent in the parameter `message`.
- All other processes provide in `message` their **receive buffer**.
- Control is returned to the calling process as soon as **its fraction** of the communication operation is completed.
  → `MPI_Bcast` does **not provide explicit synchronisation**.
- Each process has to specify the **same root process** `root`.

Accumulation Operation (1)

- There is a duality between a single broadcast and a **single accumulation operation**:
  - Each participating process provides a **block of data**.
  - Additionally, a **binary reduction operation** is provided.
  - The root process collects the data blocks provided by the participating processes and accumulates the individual data elements using the reduction operation specified.
- **Illustration** using the sum $(+)$ as the reduction operation:



Accumulation Operation (2)

- MPI provides the following predefined **reduction operations**:
  - arithmetical: $MPI\_\{MAX, MIN, SUM, PROD, MINLOC, MAXLOC\}$;
  - logical: $MPI\_\{LAND, BAND, LOR, BOR, LXOR, BXOR\}$;
  - $MPI\_\{MAXLOC, MINLOC\}$ additionally return the **index** attached by the **process** with the maximum or minimum value respectively;
  - Data types for a pair of values: $MPI\_\{FLOAT, \ldots, LONG\}\_INT$, MPI_2INT
- Syntax of the accumulation operation in MPI:

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

Accumulation Operation (3)

- Additional reduction operations can be defined by the user with:

```
int MPI_Op_create (MPI_User_function *function,
                   int commute,
                   MPI_Op *op)
```

- The argument `function` specifies a **user-defined function** which must define the following four parameters:
  `void *in, void *out, int *len, MPI_Datatype *type.`
- The parameter `commute` specifies whether the function is **commutative** (`commute = 1`) or not (`commute = 0`).
- The call of `MPI_Op_create()` returns a reduction operation `op` which can then be used as parameter of `MPI_Reduce()`.

Accumulation Operation (4)

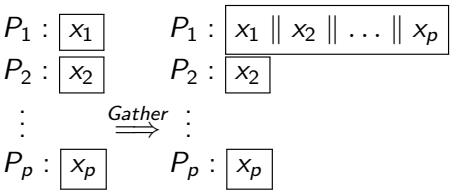
- **Example**: MPI program for the parallel computation of the **scalar product** (dot product)

```
int j, m, p, local_m, my_rank;
float local_dot, dot;
float local_x[100], local_y[100];

MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_size (MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf("%d", &m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j<local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce (local_dot, &dot, 1, MPI_FLOAT, MPI_SUM,
            0, MPI_COMM_WORLD);
MPI_Finalize();
```

Gather Operation (1)

- Impact of a **gather operation**: each of the participating n processes provides a block of data that is **collected at the root process**.
- Illustration and syntax of a gather operation




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

- **sendcount**: Number of data elements with data type **sendtype** to be sent;
- **sendbuf**: send buffer that is provided by each participating process;
- **recvbuf**: receive buffer provided by the root process **root** that is large enough to hold all data elements sent.

Gather Operation (2)

- Each data block provided has to comprise the **same number** of elements with the **same data type**.
  Data blocks will be stored equally spaced at the root process with offset **recvcount**.
  Each process has to specify the same **root process** `root`.
- **Example** for `MPI_Gather()`:
  - Each process provides **100 integer values**.
  - The **root process** collects the data blocks in its receive buffer.

```
MPI_Comm comm;
int sendbuf[100], my_rank, root = 0, gsize, *rbuf;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   rbuf = (int *) malloc (gsize*100*sizeof(int));
}
MPI_Gather (sendbuf, 100, MPI_INT, rbuf, 100,
            MPI_INT, root, comm);
```

Gatherv Operation (1)

- more general vector-based *MPI_Gatherv* operation:
  each process can provide a **different number** of elements.

```
int MPI_Gatherv (void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int *recvcounts,
                 int *displs,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
```

- **sendcount**: number of data elements to be sent;
- **recvcounts**: array, where `recvcounts[i]` denotes the number of elements provided by process `i`;
- **displs**: array that specifies the positions of the data blocks in **recvbuf**;
- **recvcounts** and **displs** are only significant at the **root process**.
- **Overlaps** in the receive buffer must not occur;
  $\rightarrow$ $displs_{root}[i+1] \geq displs_{root}[i] + sendcount_i$ with $recvcounts_{root}[i] = sendcount_i$

Gatherv Operation (2)

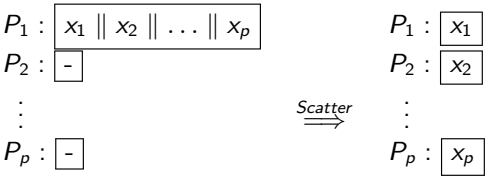
- **Example** for the use of `MPI_Gatherv()`:
  - Each process provides **100 integer values**.
  - The blocks received are stored in the receive buffer such that there is a **free gap of 10 elements** between two blocks.

```
int my_rank, root = 0, gsize, sbuf[100];
int *rbuf, *displs, *rcounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   rbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   rcounts = (int *) malloc(gsize*sizeof(int));
   for (i = 0; i < gsize; i++) {
      displs[i] = i*stride;
      rcounts[i] = 100; }
}
MPI_Gatherv (sbuf, 100, MPI_INT, rbuf, rcounts, displs,
             MPI_INT, root, comm);
```

- An **error** occurs for `stride<100`, since this would lead to **overlapping entries** in the receive buffer of the root process.

Scatter Operation (1)

- **Scatter:** The root process provides a data block (with the same size but possibly different elements) for each participating process.
- **Illustration**:



- Syntax of the MPI operation

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

- **sendbuf** is the send buffer provided by the root process **root** which contains a data block with **sendcount** elements of data type **sendtype** for each process of communicator **comm**.

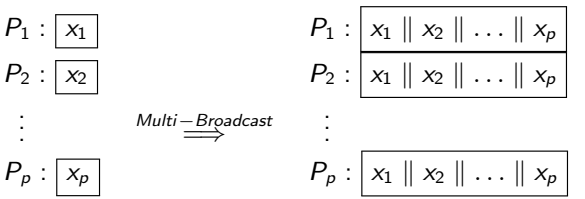
Scatter Operation (2)

- In the send buffer, the blocks are ordered in **rank order of the receiving processes**.
- **flexible number of data elements** per receiver: vector-based version **MPI_Scatterv**.
- **Example**: Process 0 sends **100 elements** to each process i such that there is a gap of **10 elements** between neighboring send blocks.

```
MPI_Comm comm;
int rbuf[100], my_rank, root=0, gsize;
int *sbuf, *displs, *scounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   sbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   scounts = (int *) malloc(gsize*sizeof(int));
   for (i=0; i<gsize; i++) {
      displs[i] = i*stride;
      scounts[i]=100; }
}
MPI_Scatterv (sbuf, scounts, displs, MPI_INT, rbuf,
              100, MPI_INT, root,comm);
```

Allgather Operation (1)

- **Multi-broadcast operation:** Each process sends **the same** block of data to each other process
  $\rightarrow$ Each process performs a **single-broadcast operation**.
- **Illustration**:



- Syntax of the MPI operation

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

- **sendbuf**: **send buffer** provided by each of the participating processes.

Allgather Operation (2)

- Each process provides a **receive buffer recvbuf** in which all received data blocks are collected in **rank order of the sending processes**.
- A multi-broadcast does **not have a distinguished root process**.
- **Example:** each process contributes a **send buffer** with 100 integer values which are made available by a multi-broadcast operation to all processes:

```
int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

- Syntax of the vector-based MPI operation **MPI_Allgatherv**:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm)
```

Allgather Operation (2)

- Each process provides a **receive buffer recvbuf** in which all received data blocks are collected in **rank order of the sending processes**.
- A multi-broadcast does **not have a distinguished root process**.
- **Example:** each process contributes a **send buffer** with 100 integer values which are made available by a multi-broadcast operation to all processes:

```
int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100,
               MPI_INT, comm);
```

- Syntax of the vector-based MPI operation **MPI_Allgatherv**:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm)
```

Multi-accumulation Operation (1)

- **Each** process provides a data block of the same size.
- The data blocks are accumulated with a **reduction operation** $\rightarrow$ multi-accumulation equals a **single-accumulation** with a subsequent **broadcast**.
- The available **reduction operations** are the same as for `MPI_Reduce()`.
- **Illustration**:

$$\begin{array}{l} P_0 : x_0 \\ P_1 : x_1 \\ \vdots \\ P_{p-1} : x_{p-1} \end{array} \quad \overset{Multi-Accumulation(+)}{\Longrightarrow} \quad \begin{array}{l} P_0 : x_0 + x_1 + \ldots + x_{p-1} \\ P_1 : x_0 + x_1 + \ldots + x_{p-1} \\ \vdots \\ P_{p-1} : x_0 + x_1 + \ldots + x_{p-1} \end{array}$$

Multi-accumulation Operation (2)

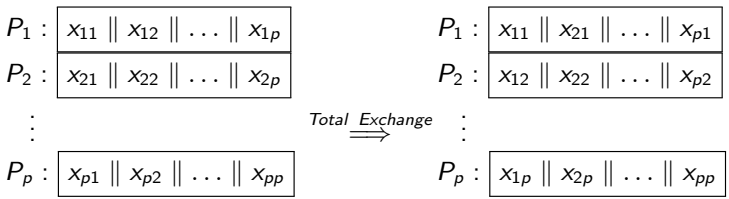
- Syntax of the MPI operation

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

- **sendbuf** is the **local buffer** in which each process provides its local data;
- **recvbuf** is the local buffer of each process in which the accumulated result is **collected**.

Total Exchange – MPI_Alltoall() (1)

- **Each** process provides a **different** block of data for **each** other process.
- Each process collects the data blocks provided for this particular process.
- **Illustration**:



Total Exchange – MPI_Alltoall() (2)

- Syntax of the MPI operation

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

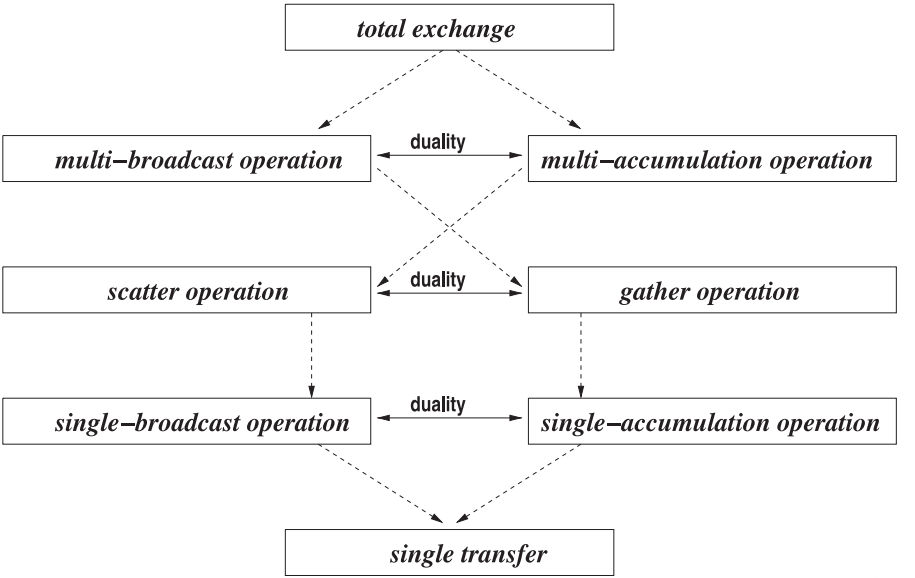
- **sendbuf** is the **send buffer** in which each process provides for each process a block of data with **sendcount** elements of type **sendtype**; The blocks are **arranged in rank order** of the target processes.
- Each process provides a receive buffer **recvbuf** in which the data blocks received from the other processes are stored;
  The blocks received are **stored in rank order** of the sending processes in communicator **comm**.

Total Exchange – MPI_Alltoall() (3)

- Syntax of the more general **vector-based version** for data blocks of different sizes

```
int MPI_Alltoallv (void *sendbuf,
                   int *scounts,
                   int *sdispls,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int *rcounts,
                   int *rdispls,
                   MPI_Datatype recvtype,
                   MPI_Comm comm)
```

Hierarchy of Communication Operations



Overview

Example: Matrix-Vector Multiplication – Overview

- Multiplication of a dense $n \times m$ matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ with a vector $\mathbf{b} \in \mathbb{R}^m$:

$$c_i = \sum_{j=1}^{m} a_{ij} b_j, \quad i = 1, ..., n,$$

- Computation of n **scalar products** of the rows $\mathbf{a}_1, ..., \mathbf{a}_n$ of $\mathbf{A}$ with the vector $\mathbf{b}$:

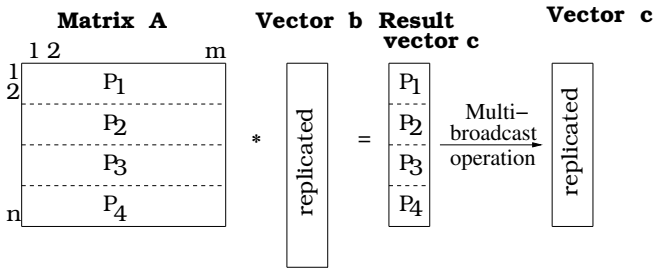
$$\mathbf{A} \cdot \mathbf{b} = \begin{pmatrix} (\mathbf{a}_1, \mathbf{b}) \\ \vdots \\ (\mathbf{a}_n, \mathbf{b}) \end{pmatrix},$$

- **Linear combination** of the columns $\tilde{\mathbf{a}}_1, ..., \tilde{\mathbf{a}}_m$ of $\mathbf{A}$, where $\mathbf{b} \in \mathbb{R}^n$ contains the coefficients of this linear combination

$$\mathbf{A} \cdot \mathbf{b} = \sum_{j=1}^{m} b_j \tilde{\mathbf{a}}_j \, .$$

Example: Matrix-Vector Multiplication – Distribution of the Scalar Products

- **Data distribution**: Distribution of **A** and **b** on the processors:
  - **row-oriented** blockwise distribution of **A**:
    processor P_k stores rows $\mathbf{a}_i$ for $i = n/p \cdot (k-1) + 1, \ldots, n/p \cdot k$ in its local memory
  - **replicated** storage of **b**



- The final result should be available on **all processors** $\rightarrow$ Re-distribution using a **multi-broadcast operation**;

Example: Matrix-Vector Multiplication – Program Sketch I

- **row-oriented** distribution of the matrix
  - each processor stores a **local array** `local_A` of dimension `local_n` $\times$ `m`;
  - initialization of the local array of processor P_k:

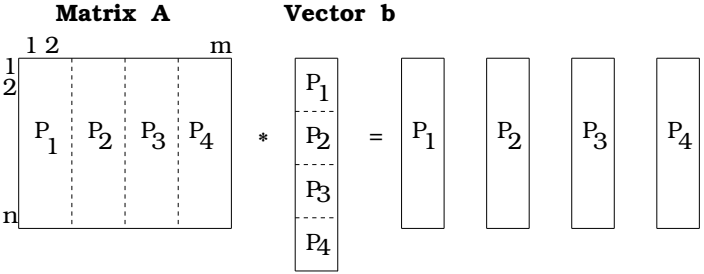
$$\texttt{local\_A[i][j]} = \texttt{A[i} + (\texttt{k} - 1) * \texttt{n/p][j]}$$

- Program Sketch I

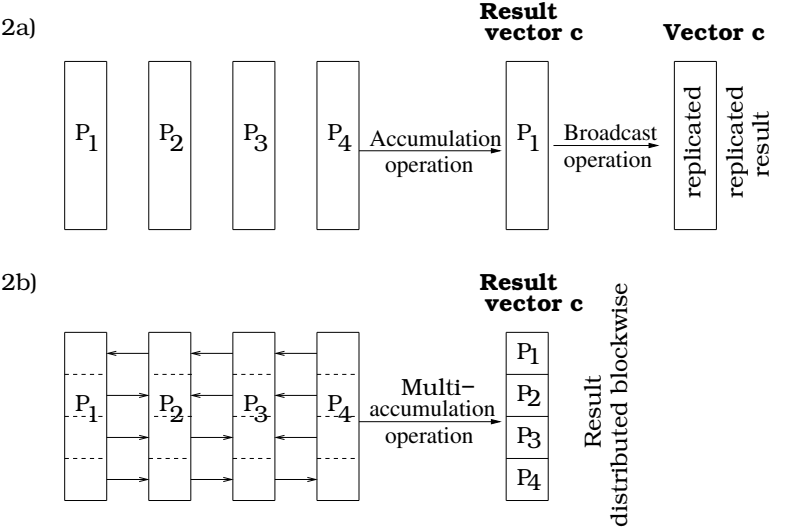
```
local_n = n/p;
for (i=0; i<local_n; i++)
   local_c[i] = 0;
for (i=0; i<local_n; i++)
   for (j=0; j<m; j++)
      local_c[i] = local_c[i] + local_A[i][j] * b[j];
MPI_Allgather( local_c, local_n, MPI_DOUBLE,
               global_c, local_n, MPI_DOUBLE, comm);
```

Example: Matrix-Vector Multiplication – Distribution of the Linear Combinations (1)

- **Data distribution**:
  - **column-oriented** blockwise distribution of **A**:
    processor P_k stores the columns $\tilde{\mathbf{a}}_i$ with
    $i = m/p \cdot (k-1) + 1, \ldots, m/p \cdot k$
  - blockwise distribution of **b**



Example: Matrix-Vector Multiplication – Distribution of the Linear Combinations (2)



Example: Matrix-Vector Multiplication – Program Sketch II

- **Data distribution**: **column-oriented** blockwise distribution of the matrix **A**:
  - each processor stores a **local array** `local_A` of dimension `n` $\times$ `local_m`;
  - P_k computes the **partial linear combination**

$$\mathbf{d}_k = \sum_{j=m/p\cdot(k-1)+1}^{m/p\cdot k} b_j \tilde{\mathbf{a}}_j.$$

- Program Sketch II

```
local_m=m/p;
for (i=0; i<n; i++) d[i] = 0;
for (j=0; j<local_m; j++)
   for (i=0; i<n; i++)
      d[i] = d[i] + local_b[j] * local_A[i][j];
MPI_Reduce (d, c, n, MPI_DOUBLE, MPI_SUM, 0,
            comm);
MPI_Bcast (c, n, MPI_DOUBLE, 0, comm);
```

Overview

Deadlocks with Global Communication Operations

- Global communication operations in MPI are always **blocking**; Thus, **without the use of a system buffer** these operations **synchronise** the participating processes.
- **Program fragment** with **two** participating processes:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Bcast (buf2, count, type, 1, comm); break;
case 1: MPI_Bcast (buf2, count, type, 1, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

- The runtime system detects an **error** if it matches the first calls to `MPI_Bcast()` of each process since **different root processes** are specified.
- A **deadlock** occurs if the MPI runtime system matches the `MPI_Bcast()` operations with **the same root process** and **no system buffers are used** or the system buffers **are too small**.
- The error or deadlock can be **avoided** by letting the participating processes call the matching `MPI_Bcast()` operation in the same order.

Deadlocks with Mixed Global and Point-to-point Communication

- **Deadlocks** can also occur when **mixing global communication operations** and **point-to-point communication**:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, 0, tag, comm,
                  &status);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

- If **no system buffers** are used, a **deadlock** occurs.
- The deadlock can be avoided by calling the **corresponding** communication operations in the **same order**.

Synchronization Behavior of Global Communication Operations (1)

- The **synchronization behavior** of collective communication operations depends on the use of system buffers by the MPI runtime system.
- **Example** that possibly leads to different execution orders

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
                  comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
                  comm, &status);
        break;
case 2: MPI_Send (buf2, count, type, 1, tag, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

Synchronization Behavior of Global Communication Operations (2)

- Execution order that may occur **with or without system buffers**:

process$_0$		**process$_1$**		**process$_2$**
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
`MPI_Bcast()`		`MPI_Bcast()`		`MPI_Bcast()`
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		

- Execution order that may only occur **with system buffers**:

process$_0$		**process$_1$**		**process$_2$**
`MPI_Bcast()`				
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		
		`MPI_Bcast()`		
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
				`MPI_Bcast()`

- **Conclusion:** a **non-deterministic program behavior** may result when system buffers are used.
- Secure **synchronization** of all processes of a communicator:

```
MPI_Barrier (MPI_Comm comm)
```

Overview

Process Groups and Communicators

- A **process group** is an ordered set of processes of a parallel program. Each process of a group gets a uniquely defined process number(**rank**).
- A process may be a member of **multiple groups** and may have different ranks in each of these groups.
- For the programmer, a group is an object of type `MPI_Group` which can only be accessed via a **handle**.
- **Global communication operations** can be **restricted** to previously defined groups.
- Each process group is associated with a **communication domain** that is *locally* represented by **communicators**.
  MPI_COMM_WORLD is the predefined communicator for the **global process group**.
- **Intra-communicators** support the execution of arbitrary collective communication operations on a **single** group of processes.
- **Inter-communicators** support the execution of point-to-point communication operations **between two process groups**.

Operations on Process Groups (1)

- The corresponding **process group** to a given communicator `comm` can be obtained by calling

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

  MPI_GROUP_EMPTY denotes the **empty process group**.

- New process groups can be constructed based on existing groups:
- **Union** of two existing groups `group1` and `group2`:

```
int MPI_Group_union (MPI_Group group1,
                     MPI_Group group2,
                     MPI_Group *new_group)
```

  The processes in `group1` **keep their ranks** from `group1` and the processes in `group2` which are not in `group1` get **subsequent ranks** in consecutive order.

Operations on Process Groups (2)

- The **intersection** of two groups is obtained by calling

```
int MPI_Group_intersection (MPI_Group group1,
                            MPI_Group group2,
                            MPI_Group *new_group)
```

  where the **process order** from `group1` is kept for `new_group`. The processes in `new_group` get **successive ranks** starting from 0.

- The **set difference** of two groups is obtained by calling

```
int MPI_Group_difference (MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group),
```

  where the process order from `group1` is kept as well.

Operations on Process Groups (3)

- **Construction of a subset** of an existing group:

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

  - `ranks` is an integer array with `p` entries.
  - The call creates a **new group** `new_group` with `p` processes which have **ranks** from `0` to `p-1`.
  - Process `i` is the process which has rank `ranks[i]` in group `group`.
  - The group `group` must contain **at least** `p` processes and the values `ranks[i]` must be **valid process ranks** in `group` which are **different from each other**.

- **Deletion of processes** from a group:

```
int MPI_Group_excl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group)
```

  The **new group** `new_group` is generated by **deleting** the processes with ranks `ranks[0]`,...,`ranks[p-1]` from `group`.

Operations to Obtain Information about Process Groups

- The **size of a process group** group can be obtained by calling

```
int MPI_Group_size (MPI_Group group, int *size)
```

The group size is returned in *size.
The **rank of the calling process** in a group group can be obtained by calling:

```
int MPI_Group_rank (MPI_Group group, int *rank)
```

- To check whether two **process groups** describe the **same process group** the function

```
int MPI_Group_compare (MPI_Group group1,
                       MPI_Group group2, int *res)
```

can be used. Possible return values *res:

  - *res = MPI_IDENT: the groups group1 and group2 contain the **same processes in the same order**;
  - *res = MPI_SIMILAR: both groups contain the **same processes** but group1 uses a different order than group2;
  - *res = MPI_UNEQUAL: the two groups contain **different processes**.

Operations on Communicators (1)

- Generation of a new **intra-communicator** to a given group of processes:

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

  - `group` must specify a process group which is a **subset** of the process group associated with communicator `comm`.
  - **All** processes of `comm` must call `MPI_Comm_create()` with **the same group** as an argument.
  - Result of the call: each calling process which is a member of group `group` obtains a **pointer to the new communicator** `new_comm`.
  - Processes not belonging to `group` get `MPI_COMM_NULL` as return value in `new_comm`.

Operations on Communicators (2)

- A **Splitting** of a communicator can be obtained by calling:

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

  - The process group associated with communicator `comm` is partitioned into **a number of disjoint subgroups** that equals the number of **different values** specified in `color`.
  - Each **subgroup** contains all processes that specify **the same value** for `color`.
  - The **rank order** of the processes within a subgroup is defined by the argument `key`.
  - If two processes specify **the same value** for `key` the order of the original group is used.
  - If a process specifies `color = MPI_UNDEFINED`, it is **not a member of any of the subgroups** generated.
  - Each participating process gets a pointer `new_comm` to the communicator of that subgroup which the process belongs to.

Operations on Communicators (3)

- **Example**: We consider a group of **10 processes** each of which calls the operation `MPI_Comm_split()` with the following argument values:

process	a	b	c	d	e	f	g	h	i	j
rank	0	1	2	3	4	5	6	7	8	9
color	0	$\perp$	3	0	3	0	0	5	3	$\perp$
key	3	1	2	5	1	1	1	2	1	0

- This call generates **three subgroups**:
  {f, g, a, d}, {e, i, c} and {h}
- The groups contain the processes in the indicated order.

Overview

Process Topologies

Idea: The **arrangement** of the processes in a **grid structure** facilitates data exchanges in some applications.
Definition of a **virtual Cartesian grid structure of arbitrary dimension**:

```
int MPI_Cart_create (MPI_Comm comm,
                     int ndims,
                     int *dims,
                     int *periods,
                     int reorder,
                     MPI_Comm *new_comm).
```

- `comm` is the original communicator **without topology**;
- `ndims` specifies the **number** of dimensions of the grid to be created;
- `dims` is an integer array with `ndims` elements where `dims[i]` denotes the **total number of processes in dimension** i.
- The array `periods` of size `ndims` specifies for each dimension whether the grid is **periodic** (entry 1) or not (entry 0) in this dimension.
- For `reorder = false`, the processes in `new_comm` have the **same rank** as in `comm`.

Process Topologies – Example

Example: Let `comm` be a communicator with **12 processes**.
Using the initializations `dims[0] = 3`, `dims[1] = 4`, `period[0] = period[1] = 0`, `reorder = 0`, the call

```
MPI_Cart_create (comm, 2, dims, period, reorder, &new_comm)
```

generates a **virtual** 3×4 **grid** with the following group ranks and coordinates:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Process Topologies – Distribution of the Processes

The following function helps the programmer to select a balanced distribution of the processes for the different dimensions:

```
int MPI_Dims_create (int nnodes, int ndims,int *dims)
```

- `ndims` is the **number of dimensions** in the grid to be defined;
- `nnodes` is the **total number of processes** in the grid;
- `dims` is an integer array of size `ndims`.
  In the case `dims[i] = 0` is specified for the call, `dims[i]` contains the **number of processes** in dimension `i` after the call.
  The function tries to assign the **same number of processes** to each dimension.
- The number of processes in a dimension *i* can be **fixed** by setting `dims[i]` to the desired number of processes **before the call**.
  The MPI runtime system sets the entries of the **other, non-initialized** entries of `dims` accordingly.

Process Topologies – Translation of Cartesian Coordinates

- When defining a **virtual topology**, each process has a **group rank**, and also a position in the virtual grid which can be expressed by its **Cartesian coordinates**.
- **Translation** of **Cartesian coordinates** into **group ranks**:

```
int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

  The call translates the **Cartesian coordinates** of a process provided in the array `coords` into the **group rank** according to the virtual grid associated with `comm`.
- **Translation** of **group ranks** into **Cartesian coordinates**:

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int dims,
                     int *coords)
```

- `rank` contains the process number; `dims` denotes the **number of dimensions** in the virtual grid defined for communicator `comm`. The **Cartesian coordinates** of the process are returned in the array `coords`.

Process Topologies – Determining Neighboring Processes

- A typical communication pattern in many grid-based algorithms is that processes communicate with **their neighboring processes in a specific dimension**.
- **Determining the neighboring processes** in each dimension of the grid:

```
int MPI_Cart_shift (MPI_Comm comm, int dir, int dspl,
                    int *rk_src, int *rk_dst).
```

- `dir` specifies the **dimension** for which the neighboring process should be determined. The parameter `dspl` specifies the **displacement desired**.
  positive value: request neighbors in **upward direction**;
  negative value: request neighbors in **downward direction**.
- The result of the call is that `rk_dst` contains the **group rank of the neighboring process** in the specified dimension and distance; `rk_src` returns the **rank of the process** for which the calling process is the neighbor in the specified dimension and distance.

Process Topologies – Example (1)

We consider **12 processes** that are arranged in a 3×4 grid structure with **periodic connections**. Each process stores a **floating-point value** which is exchanged with the neighboring process in dimension 0:

```
int coords[2], dims[2], periods[2], source, dest;
int my_rank, reorder = 0;
float a, b;
MPI_Comm comm_2d;
MPI_Status status;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
dims[0] = 3; dims[1] = 4; periods[0] = periods[1] = 1;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods,
                 reorder, &comm_2d);
MPI_Cart_coords (comm_2d, my_rank, 2, coords);
MPI_Cart_shift  (comm_2d, 0, coords[1], &source, &dest);
a = my_rank;
MPI_Sendrecv (&a, 1, MPI_FLOAT, dest, 0, &b, 1, MPI_FLOAT,
              source, 0, comm_2d, &status);
```

- The operation `MPI_Cart_shift()` is used to determine the communication partners `dest` and `source` for the operation `MPI_Sendrecv()`.

Process Topologies – Example (2)

- The **distance between the neighbors** in dimension 0 **increases** with the coordinates in dimension 1 since `displs = coord` is used;
  $\rightarrow$ in **each column** of the grid a **different exchange** is executed.
- The following diagram illustrates the exchange. For each process, its **rank**, its **Cartesian coordinates**, and its communication partners in the form **source**|**dest** are given in this order.

0 (0,0) 0|0	1 (0,1) 9|5	2 (0,2) 6|10	3 (0,3) 3|3
4 (1,0) 4|4	5 (1,1) 1|9	6 (1,2) 10|2	7 (1,3) 7|7
8 (2,0) 8|8	9 (2,1) 5|1	10 (2,2) 2|6	11 (2,3) 11|11

- For example, for the process with `rank=5` it is `coords[1]=1` and therefore `source = 9` (lower neighbor in dimension 0) and `dest = 1` (upper neighbor in dimension 0).

Process Topologies – Subgrids (1)

A **virtual topology** can be partitioned into **subgrids**:

```
int MPI_Cart_sub (MPI_Comm comm, int *remain_dims,
                  MPI_Comm *new_comm)
```

- `comm` is the communicator for which the **virtual topology** has been defined;
- `new_comm` denotes the **new communicator** for which the new topology as a **subgrid of the original grid** is defined.
- The **subgrid selection** is controlled by the array `remain_dims` which contains an entry for each dimension of the original grid.
  Setting `remain_dims[i]=1` means that the *i*th dimension **is kept** in the subgrid;
  `remain_dims[i]=0` means that the *i*th dimension **is dropped** in the subgrid.
- If a dimension `i` does not exist in the subgrid, the size of dimension `i` defines the **number of subgrids** that have been generated for this dimension.

Process Topologies – Subgrids (2)

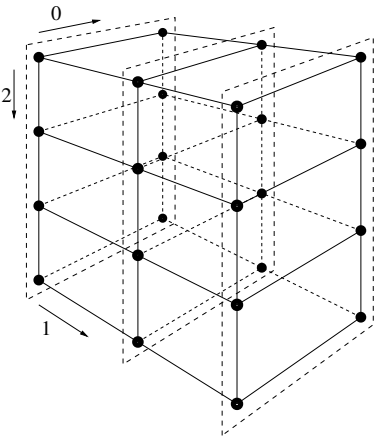
- A call to `MPI_Cart_sub()` generates a **new communicator** `new_comm` for each calling process, representing the corresponding subgroup of the subgrid to which the calling process belongs.
- The **dimensions of the different subgrids** result from the dimensions for which `remain_dims[i]` has been set to `1`.
- The **total number of different communicators** generated is defined by the product of the number of processes in all dimensions `i` for which `remain_dims[i]` has been set to `0`.

Process Topologies – Example for Subgrids

- **Example:** let `comm` be a communicator for which a $(2 \times 3 \times 4)$ virtual grid topology has been defined. Calling

  ```
  int MPI_Cart_sub (comm_3d, remain_dims, &new_comm)
  ```

  with `remain_dims = (1,0,1)` generates **three different communicators** each representing a (2×4) grid:



Process Topologies – Inquiring Information about a Virtual Topology

Information on the **virtual topology** that has been defined for a communicator can be inquired using the following two **MPI functions**:

- **Number of dimensions** of the virtual grid:

```
int MPI_Cartdim_get (MPI_Comm comm, int *ndims)
```

- **Cartesian coordinates** of the **calling process** within the virtual grid associated with communicator `comm` can be obtained by calling

```
int MPI_Cart_get (MPI_Comm comm, int maxdims, int *dims,
                  int *periods, int *coords)
```

  where `maxdims` is the number of **dimensions** of the virtual topology, and `dims`, `periods`, and `coords` are **arrays** of size `maxdims`.
  - The arrays `dims` and `periods` have the same meaning as for `MPI_Cart_create()`.
  - The array `coords` is used to **return the coordinates**.

Timings and Aborting Processes

- Measurement of the **execution times** of program parts:

  ```
  double MPI_Wtime (void)
  ```

- typical usage:

```
start = MPI_Wtime();
part_to_measure();
end = MPI_Wtime();
```

- `MPI_Wtime()` returns an **absolute time**, not the system time
- **Abortion** of the execution of all processes of a **communicator**:

  ```
  int MPI_Abort (MPI_Comm comm, int error_code)
  ```