Parallel Processing

WS 2020/21

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 22, 2020

Parallel Processing

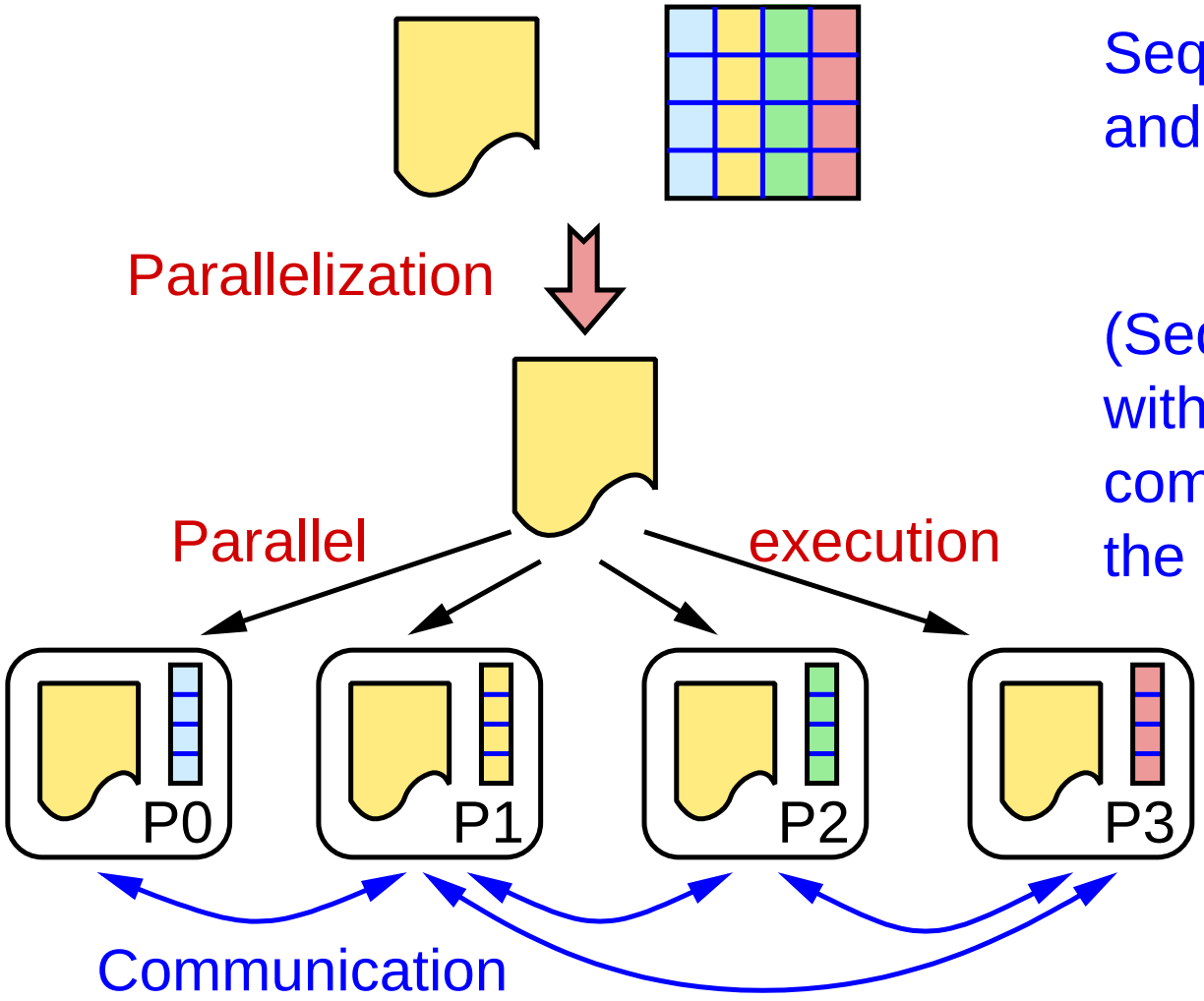
WS 2020/21

3 Parallel Programming with Message Passing

Contents

- Typical approach
- MPI (Message Passing Interface)
  - core routines
  - simple MPI programs
  - point-to-point communication
  - complex data types in messages
  - communicators
  - collective operations
  - further concepts

Data partitioning with SPMD model



Sequential program and data partitioning

(Sequential) node program with message passing performs computations for a part of the data

Identical copies of the program, executed in parallel using multiple processes

Activities when creating a node program

- Adjustment of array declarations
  - node program stores only a part of the data
  - (assumption: data are stored in arrays)
- Index transformation
  - global index $\leftrightarrow$ (process number, local index)
- Work partitioning
  - each process executes the computations on its part of the data
- Communication
  - when a process needs non-local data, a suitable message exchange must be programmed

About communication

- When a process needs data: the owner of the data must send them explicitly
  - exception: one-sided communication (☞ **3.2.7**)
- Communication should be merged as much as possible
  - one large message is better than many small ones
  - data dependences must not be violated

Sequential execution

```
a[1] = ...;
a[2] = ...;
a[3] = a[1]+...;
a[4] = a[2]+...;
```

Parallel execution

Process 1

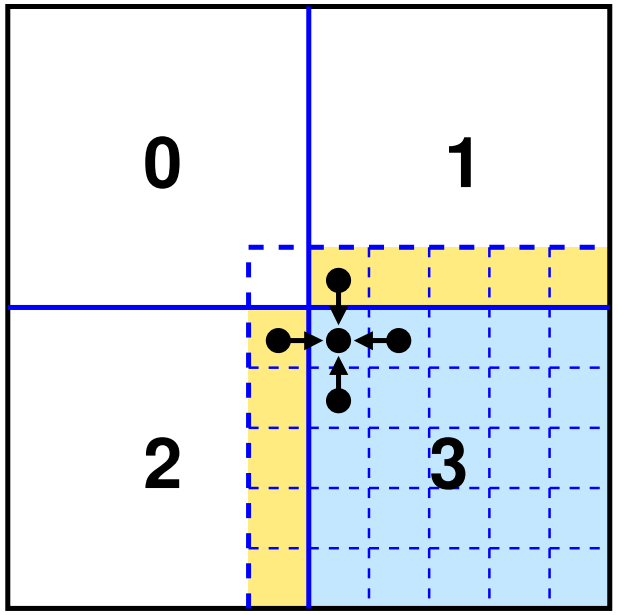
```
a[1] = ...;
a[2] = ...;
send(a[1],a[2]);
```

Process 2

```
recv(a[1],a[2]);
a[3] = a[1]+...;
a[4] = a[2]+...;
```

About communication ...

- Often the node program allocates an overlapping buffer region (**ghost region** / **ghost cells**) for non-local data
- Example: Jacobi iteration



Partitioning of the matrix into 4 parts

Each process allocates an additional row/column at the borders of its sub−matrix

Data exchange at the end of each iteration

History and background

- At the beginning of the parallel computer era (late 1980's):
  - many different communication libraries (NX, PARMACS, PVM, P4, ...)
  - parallel programs are not easily portable
- Definition of an informal standard by the MPI forum
  - 1994: MPI-1.0
  - 1997: MPI-1.2 and MPI-2.0 (considerable extensions)
  - 2009: MPI 2.2 (clarifications, minor extensions)
  - 2012/15: MPI-3.0 und MPI-3.1 (considerable extensions)
  - documents at `http://www.mpi-forum.org/docs`
- MPI only defines the API (i.e., the programming interface)
  - different implementations, e.g., MPICH2, OpenMPI, ...

Programming model

- Distributed memory, processes with message passing
- SPMD: one program code for all processes
  - but different program codes are also possible
- MPI-1: static process model
  - all processes are created at program start
    - program start is standardized since MPI-2
  - MPI-2 also allows to create new processes at runtime
- MPI is thread safe: a process is allowed to create additional threads
  - hybrid parallelization using MPI and OpenMP is possible
- Program terminates when all its processes have terminated

3.2.1 Core routines

- MPI-1.2 has 129 routines (and MPI-2 even more ...)
- However, often only 6 routines are sufficient to write relevant programs:
  - `MPI_Init` – MPI initialization
  - `MPI_Finalize` – MPI cleanup
  - `MPI_Comm_size` – get number of processes
  - `MPI_Comm_rank` – get own process number
  - `MPI_Send` – send a message
  - `MPI_Recv` – receive a message

MPI_Init

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

INOUT **argc**	Pointer to **argc** of **main()**
INOUT **argv**	Pointer to **argv** of **main()**
Result	**MPI_SUCCESS** or error code

- Each MPI process must call `MPI_Init`, before it can use other MPI routines
- Typically:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ...
```

- `MPI_Init` may also ensure that all processes receive the command line arguments

MPI_Finalize

```
int MPI_Finalize()
```

- Each MPI process must call `MPI_Finalize` at the end
- Main purpose: deallocation of resources
- After that, no other MPI routines must be used
  - in particular, no further `MPI_Init`
- `MPI_Finalize` does **not** terminate the process!

MPI_Comm_size

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

IN	**comm**	Communicator
OUT	**size**	Number of processes in **comm**

- Typically: `MPI_Comm_size(MPI_COMM_WORLD, &nprocs)`
  - returns the number of MPI processes in `nprocs`

MPI_Comm_rank

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

IN	**comm**	Communicator
OUT	**rank**	Number of processes in **comm**

- Process number (“rank”) counts upward, starting at 0
  - only differentiation of the processes in the SPMD model

Communicators

- A communicator consists of
  - a process group
    - a subset of all processes of the parallel application
  - a communication context
    - to allow the separation of different communication relations (☞ **3.2.5**)
- There is a predefined communicator `MPI_COMM_WORLD`
  - its process group contains all processes of the parallel application
- Additional communicators can be created as needed (☞ **3.2.5**)

MPI_Send

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

IN	**buf**	(Pointer to) the data to be sent (send buffer)
IN	**count**	Number of data elements (of type **dtype**)
IN	**dtype**	Data type of the individual data elements
IN	**dest**	Rank of destination process in communicator **comm**
IN	**tag**	Message tag
IN	**comm**	Communicator

- Specification of data type: for format conversions
- Destination process is always relative to a communicator
- *Tag* allows to distinguish different messages (or message types) in the program

MPI_Send ...

- `MPI_Send` blocks the calling process, until all data has been read from the send buffer
  - send buffer can be reused (i.e., modified) immediately after `MPI_Send` returns
- The MPI implementation decides whether the process is blocked until
  a) the data has been copied to a system buffer, or
  b) the data has been received by the destination process.
  - in some cases, this decision can influence the correctness of the program! (☞ slide 286)

MPI_Recv

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

OUT	**buf**	(Pointer to) receive buffer
IN	**count**	Buffer size (number of data elements of type **dtype**)
IN	**dtype**	Data type of the individual data elements
IN	**source**	Rank of source process in communicator **comm**
IN	**tag**	Message tag
IN	**comm**	Communicator
OUT	**status**	Status (among others: actual message length)

- Process is blocked until the message has been completely received and stored in the receive buffer

MPI_Recv ...

- `MPI_Recv` only receives a message where
  - sender,
  - message tag, and
  - communicator

  match the parameters
- For source process (sender) and message tag, wild-cards can be used:
  - `MPI_ANY_SOURCE`: sender doesn't matter
  - `MPI_ANY_TAG`: message tag doesn't matter

MPI_Recv ...

- Message must not be larger than the receive buffer
  - but it may be smaller; the unused part of the buffer remains unchanged
- From the return value `status` you can determine:
  - the sender of the message: `status.MPI_SOURCE`
  - the message tag: `status.MPI_TAG`
  - the error code: `status.MPI_ERROR`
  - the actual length of the received message (number of data elements): `MPI_Get_count(&status, dtype, &count)`

Simple data types (MPI_Datatype)

MPI	C	MPI	C
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short	MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long	MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float		
MPI_DOUBLE	double	MPI_LONG_DOUBLE	long double
MPI_BYTE	Byte with 8 bits	MPI_PACKED	Packed data*

*☞ 3.2.4

Example: typical MPI program skeleton (☞ 03/rahmen.cpp)

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main (int argc, char **argv)
{
    int i;
    int myrank, nprocs;
    int namelen;
    char name[MPI_MAX_PROCESSOR_NAME];

    /* Initialize MPI and set the command line arguments */
    MPI_Init(&argc, &argv);

    /* Determine the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

3.2.2 Simple MPI programs ...

```
    /* Determine the own rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    /* Determine the node name */
    MPI_Get_processor_name(name, &namelen);

    /* flush is used to enforce immediate output */
    cout << "Process " << myrank << "/" << nprocs
         << "started on " << name << "\n" << flush;

    cout << "-- Arguments: ";
    for (i = 0; i<argc; i++)
        cout << argv[i] << " ";
    cout << "\n";

    /* finish MPI */
    MPI_Finalize();

    return 0;
}
```

Starting MPI programs: `mpiexec`

- `mpiexec -n 3 myProg arg1 arg2`
  - starts `myProg arg1 arg2` with 3 processes
  - the specification of the nodes to be used depends on the MPI implementation and the hardware/OS plattform
- Starting the example program using MPICH:
  `mpiexec -n 3 -machinefile machines ./rahmen a1 a2`
- Output:

```
Process 0/3 started on bslab02.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 2/3 started on bslab03.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 1/3 started on bslab06.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
```

Example: ping pong with messages (☞ 03/pingpong.cpp)

```
int main (int argc, char **argv)
{
  int i, passes, size, myrank;
  char *buf;
  MPI_Status status;
  double start, end;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  passes = atoi(argv[1]); // Number of repetitions
  size = atoi(argv[2]);   // Message length
  buf = new char[size];
```

3.2.2 Simple MPI programs ...

```
if (myrank == 0) {      /* Proccess 0 */
  start = MPI_Wtime(); // Get the current time
  for (i=0; i<passes; i++) {
    /* Send a message to process 1, tag = 42 */
    MPI_Send(buf, size, MPI_CHAR, 1, 42, MPI_COMM_WORLD);
    /* Wait for the answer, tag is not relevant */
    MPI_Recv(buf, size, MPI_CHAR, 1, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
  }
  end = MPI_Wtime(); // Get the current time
  cout << "Time for one message: "
       << ((end - start) * 1e6 / (2 * passes)) << "us\n";
  cout << "Bandwidth: "
       << (size*2*passes/(1024*1024*(end-start))) << "MB/s\
}
```

3.2.2 Simple MPI programs ...

```
  else {   /* Process 1 */

    for (i=0; i<passes; i++) {
      /* Wait for the message from process 0, tag is not relevant */
      MPI_Recv(buf, size, MPI_CHAR, 0, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);

      /* Send back the answer to process 0, tag = 24 */
      MPI_Send(buf, size, MPI_CHAR, 0, 24, MPI_COMM_WORLD);
    }
  }

  MPI_Finalize();
  return 0;
}
```

Example: ping pong with messages ...

- Results (on the XEON cluster):
  - `mpiexec -n 2 ... ./pingpong 1000 1`
    ```
    Time for one message: 50.094485 us
    Bandwidth: 0.019038 MB/s
    ```
  - `mpiexec -n 2 ... ./pingpong 1000 100`
    ```
    Time for one message: 50.076485 us
    Bandwidth: 1.904435 MB/s
    ```
  - `mpiexec -n 2 ... ./pingpong 100 1000000`
    ```
    Time for one message: 9018.934965 us
    Bandwidth: 105.741345 MB/s
    ```
- (Only) with large messages the bandwidth of the interconnection network is reached
  - XEON cluster: 1 GBit/s Ethernet ($\hat{=}$ 119.2 MB/s)

Additional MPI routines in the examples:

```
int MPI_Get_processor_name(char *name, int *len)
```

OUT	**name**	Pointer to buffer for node name
OUT	**len**	Length of the node name
Result		**MPI_SUCCESS** or error code

- The buffer for node name should have the length `MPI_MAX_PROCESSOR_NAME`

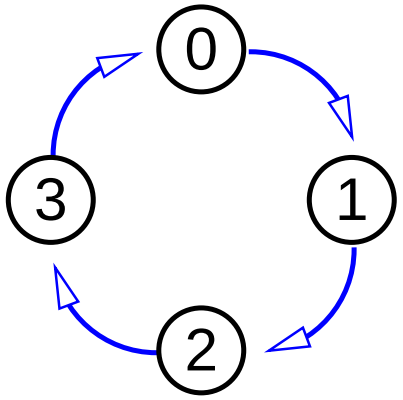
```
double MPI_Wtime()
```

Result	Current wall clock time in seconds

- for timing measurements
- in MPICH2: time is synchronized between the nodes

Example: sending in a closed cycle (☞ 03/ring.cpp)

```
int a[N];
...
MPI_Send(a, N, MPI_INT, (myrank+1) % nprocs,
         0, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_INT,
         (myrank+nprocs-1) % nprocs,
         0, MPI_COMM_WORLD, &status);
```



- Each process first attempts to send, before it receives
- This works **only if** MPI buffers the messages
- But `MPI_Send` can also block until the message is received
  - deadlock!

Example: sending in a closed cycle (correct)

- Some processes must first receive, before they send

```
int a[N];
...
if (myrank % 2 == 0) {
   MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
   MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
}
else {
   MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
   MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
}
```

- Better: use non-blocking operations

Non-blocking communication

- `MPI_Isend` and `MPI_Irecv` return immediately
  - before the message actually has been sent / received
  - result: request object (`MPI_Request`)
  - send / receive buffer must bot be modified / used, until the communication is completed
- `MPI_Test` checks whether communication is completed
- `MPI_Wait` blocks, until communication is completed
- Allows to overlap communication and computation
- can be “mixed” with blocking communication
  - e.g., send usgin `MPI_Send`, receive using `MPI_Irecv`

Example: sending in a closed cycle with `MPI_Irecv`

(☞ 03/ring2.cpp)

```
int sbuf[N];
int rbuf[N];
MPI_Status status;
MPI_Request request;
...
// Set up the receive request
MPI_Irecv(rbuf, N, MPI_INT, (myrank+nprocs-1) % nprocs, 0,
          MPI_COMM_WORLD, &request);
// Sending
MPI_Send(sbuf, N, MPI_INT, (myrank+1) % nprocs, 0,
         MPI_COMM_WORLD);
// Wait for the message being received
MPI_Wait(&request, &status);
```

3.2.4 Complex data types in messages

- So far: only arrays can be send as messages
- What about complex data types (e.g., structures)?
  - z.B. `struct bsp { int a; double b[3]; char c; };`
- MPI offers two mechanisms
  - **packing and unpacking** the individual components
    - use `MPI_Pack` to pack components into a buffer one after another; send as `MPI_PACKED`; extract the components again using `MPI_Unpack`
  - **derived data types**
    - `MPI_Send` gets a pointer to the data structure as well as a description of the data type
    - the description of the data type must be created by calling MPI routines

Derived data types

- MPI offers constructors, which can be used to define own (derived) data types:
  - for contiguous data: `MPI_Type_contiguous`
    - allows the definition of array types
  - for non-contiguous data: `MPI_Type_vector`
    - e.g., for a column of a matrix or a sub-matrix
  - for structures: `MPI_Type_create_struct`
- After a new data type has been created, it must be “announced”: `MPI_Type_commit`
- After that, the data type can be used like a predefined data type (e.g., MPI_INT)

3.2.4 Complex data types in messages ...

MPI_Type_vector: non-contiguous arrays

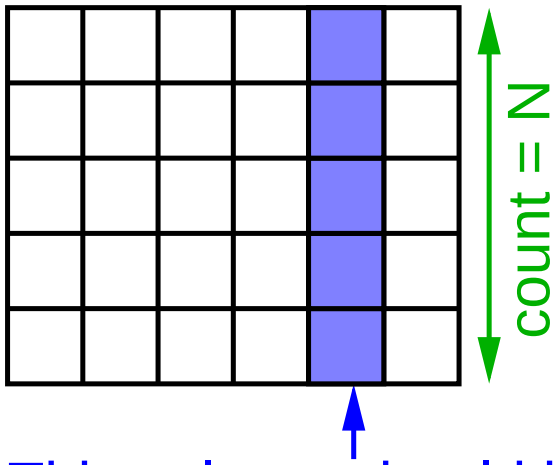
```
int MPI_Type_vector(int count, int blocklen, int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

IN	**count**	Number of data blocks
IN	**blocklen**	Length of the individual data blocks
IN	**stride**	Distance between successive data blocks
IN	**oldtype**	Type of the elements in the data blocks
OUT	**newtype**	Newly created data type

- Summarizes a number of data blocks (described as arrays) into a new data type
- However, the result is more like a new **view** onto the existing data than a new data **type**

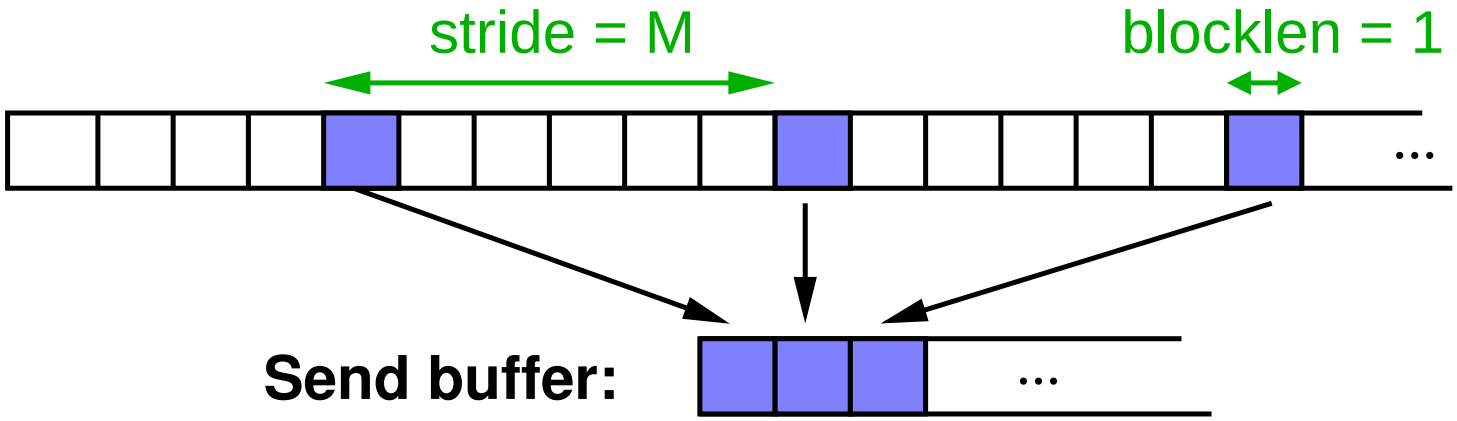
Example: transferring a column of a matrix

Matrix: a[N][M]



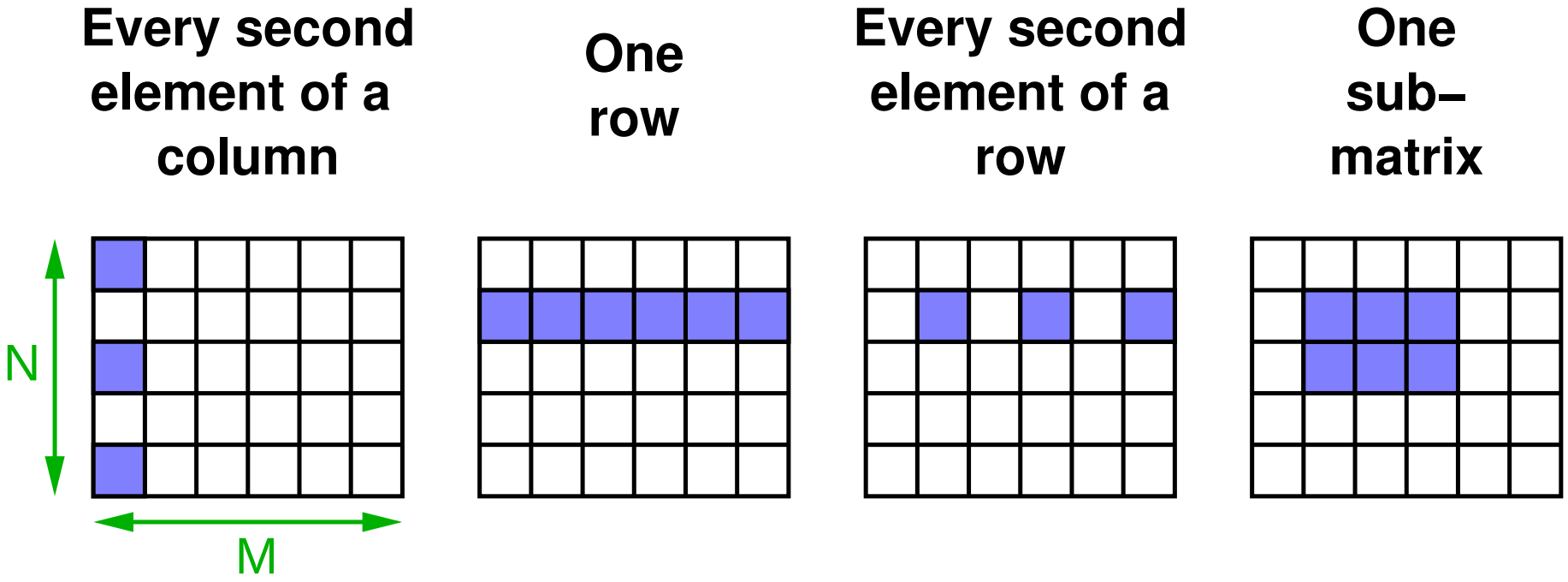
This column should be sent

Memory layout of the matrix:




```
MPI_type_vector(N, 1, M, MPI_INT, &column);
MPI_Type_commit(&column);
// Transfer the column
if (rank==0) MPI_Send(&a[0][4], 1, column, 1, 0, comm);
else MPI_Recv(&a[0][4], 1, column, 0, 0, comm, &status);
```

Additional options of `MPI_Type_vector`



count	N / 2	1	M	M / 2	2
blocklen	1	M	1	1	3
stride	2 * M	x	1	2	M

Remarks on `MPI_Type_vector`

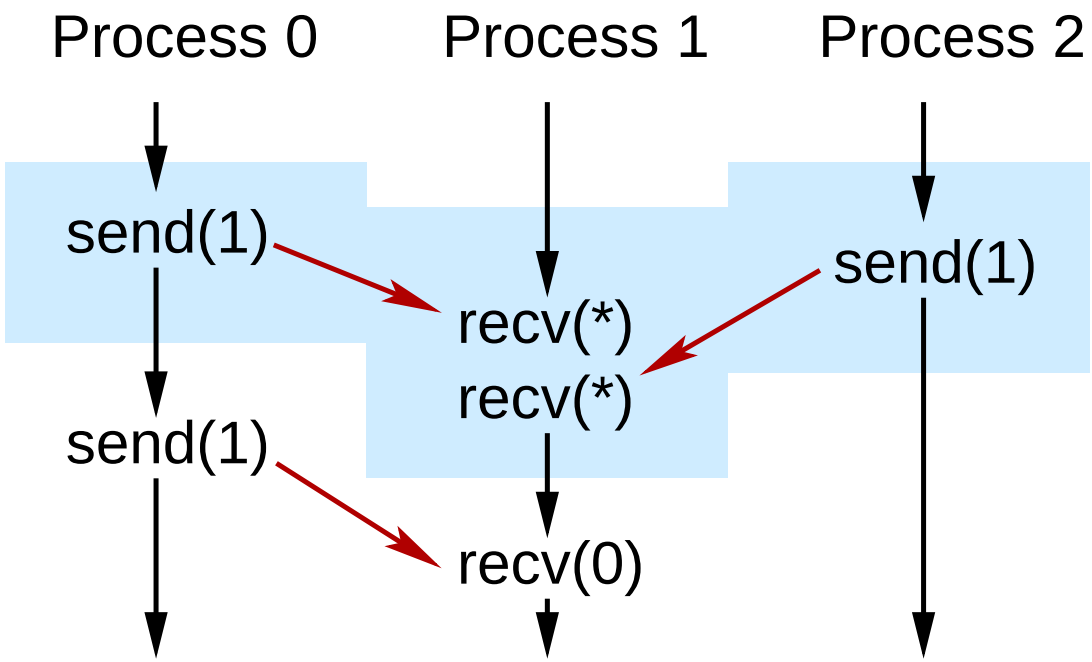
- The receiver can use a different data type than the sender
- It is only required that the number of elements and the sequence of their types is the same in the send and receive operations
- Thus, e.g., the following is possible:
  - sender transmits a column of a matrix
  - reciever stores it in a one-dimensional array

```
int a[N][M], b[N];
MPI_type_vector(N, 1, M, MPI_INT, &column);
MPI_Type_commit(&column);
if (rank==0) MPI_Send(&a[0][4], 1, column, 1, 0, comm);
else MPI_Recv(b, N, MPI_INT, 0, 0, comm, &status);
```

How to select the best approach

- Homogeneous data (elements of the same type):
  - contiguous (stride 1): standard data type and `count` parameter
  - non-contiguous:
    - stride is constant: `MPI_Type_vector`
    - stride is irregular: `MPI_Type_indexed`
- Heterogeneous data (elements of different types):
  - large data, often transmitted: `MPI_Type_create_struct`
  - few data, rarely transmitted: `MPI_Pack` / `MPI_Unpack`
  - structures of variable length: `MPI_Pack` / `MPI_Unpack`

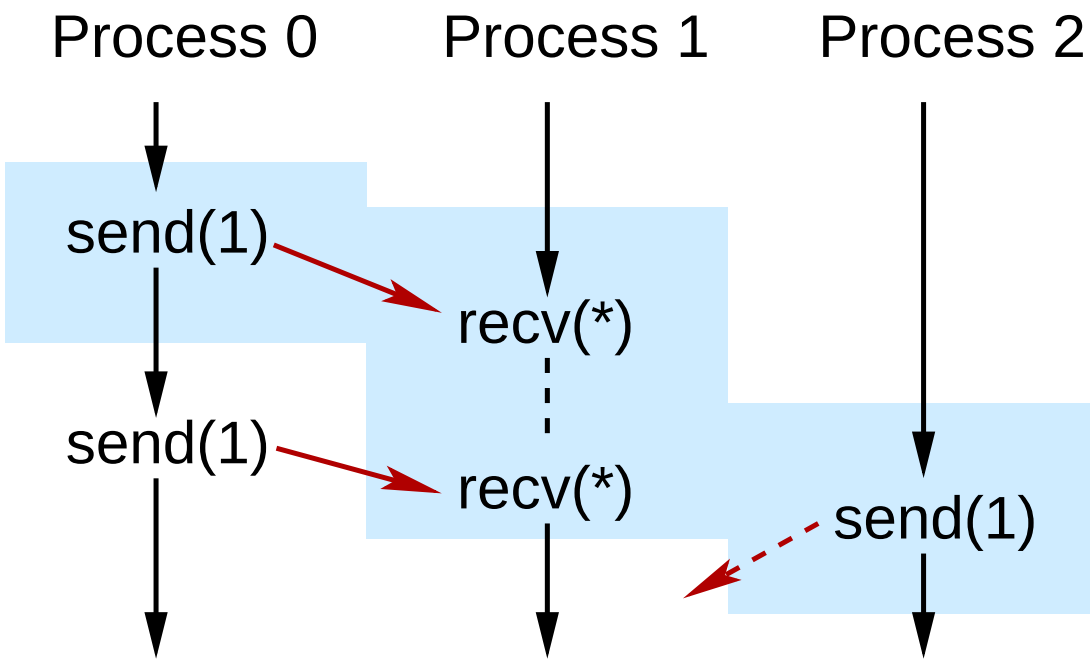
Motivation: problem of earlier communication libraries



: Code in a parallel (e.g. numerical) library

Process 1 should receive from 0 and 2 (in arbitrary order)

Motivation: problem of earlier communication libraries



: Code in a parallel (e.g. numerical) library

If process 2 is ‘late’ for some reason: communication fails!

- Message tags are not a reliable solution
  - tags might be chosen identically by chance!
- Required: different communication contexts

3.2.5 Communicators ...

- Communicator = process group + context
- Communicators support
  - working with process groups
    - task parallelism
    - coupled simulations
    - collective communication with a subset of all processes
  - communication contexts
    - for parallel libraries
- A communicator represents a communication domain
  - communication is possible only within the same domain
    - no wild-card for communicator in `MPI_Recv`
  - a process can belong to several domains at the same time

Creating new communicators

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color
                   int key, MPI_Comm *newcomm)
```

- Collective operations
  - all processes in `comm` must execute them concurrently
- `MPI_Comm_dup` creates a copy with a new context
- `MPI_Comm_split` splits `comm` into several communicators
  - one communicator for each value of `color`
  - as the result, each process receives the communicator to which it was assigned
  - `key` determines the order of the new process ranks

Example for MPI_Comm_split

- *Multi-physics code*: air pollution
  - one half of the processes computes the airflow
  - the other half computes chemical reactions
- Creation of two communicators for the two parts:

```
MPI_Comm_split(MPI_COMM_WORLD, myrank%2, myrank, &comm)
```

Process	`myrank`	Color	Result in `comm`	Rank in C_0	Rank in C_1
P0	0	0	C_0	0	–
P1	1	1	C_1	–	0
P2	2	0	C_0	1	–
P3	3	1	C_1	–	1

3.2.6 Collective operations

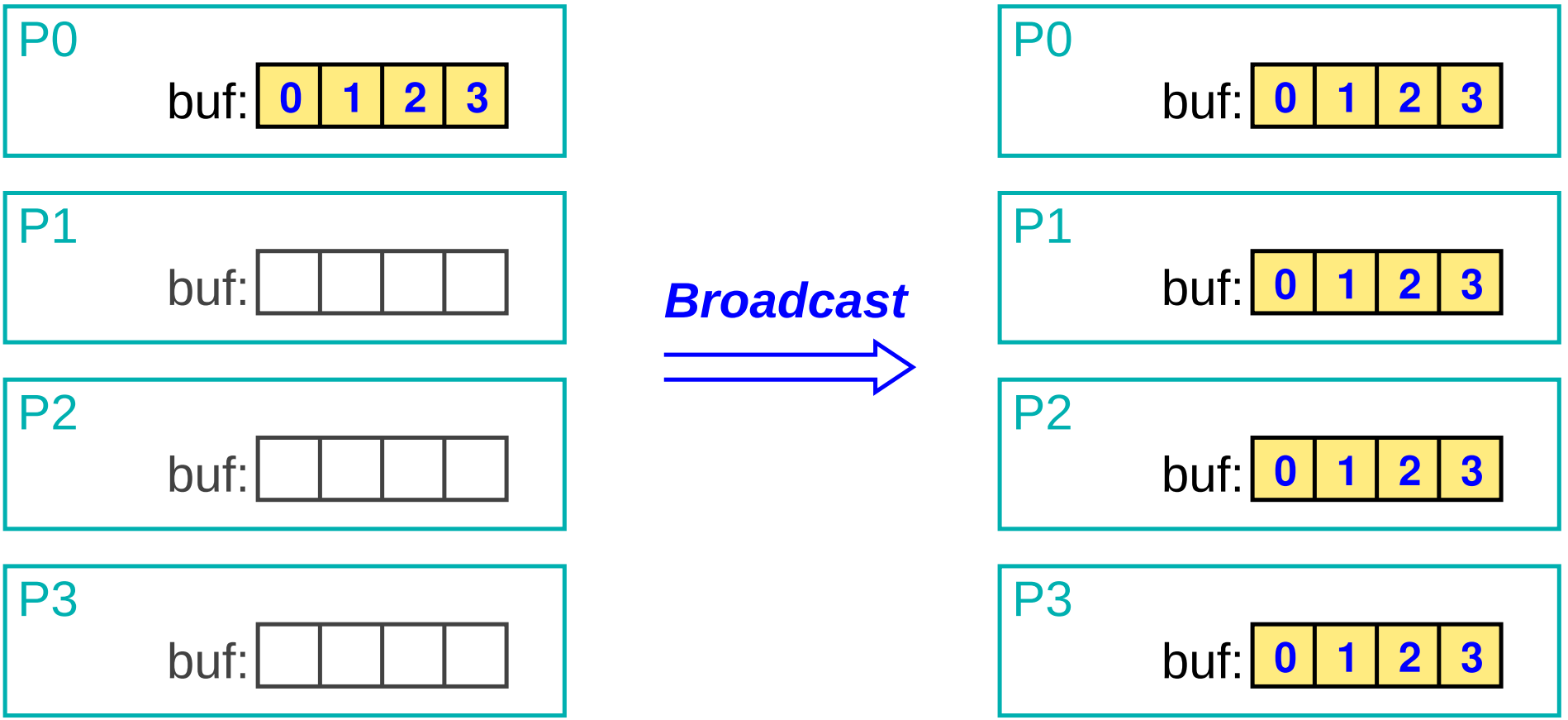
- Collective operations in MPI
  - must be executed concurrently by all processes of a process group (a communicator)
  - are blocking
  - do not neccessarily result in a global (barrier) synchronisation, however
- Collective synchronisation and communication functions
  - barriers
  - communication: broadcast, scatter, gather, ...
  - reductions (communication with aggregation)

MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

- Barrier synchonization of all processes in `comm`
- With message passing, barriers are actually not really necessary
  - synchonization is achieved by message exchange
- Reasons for barriers:
  - more easy understanding of the program
  - timing measurements, debugging output
  - console input/output ??
  - MPI-2: MPI I/O, one-sided communication

Collective communication: broadcast



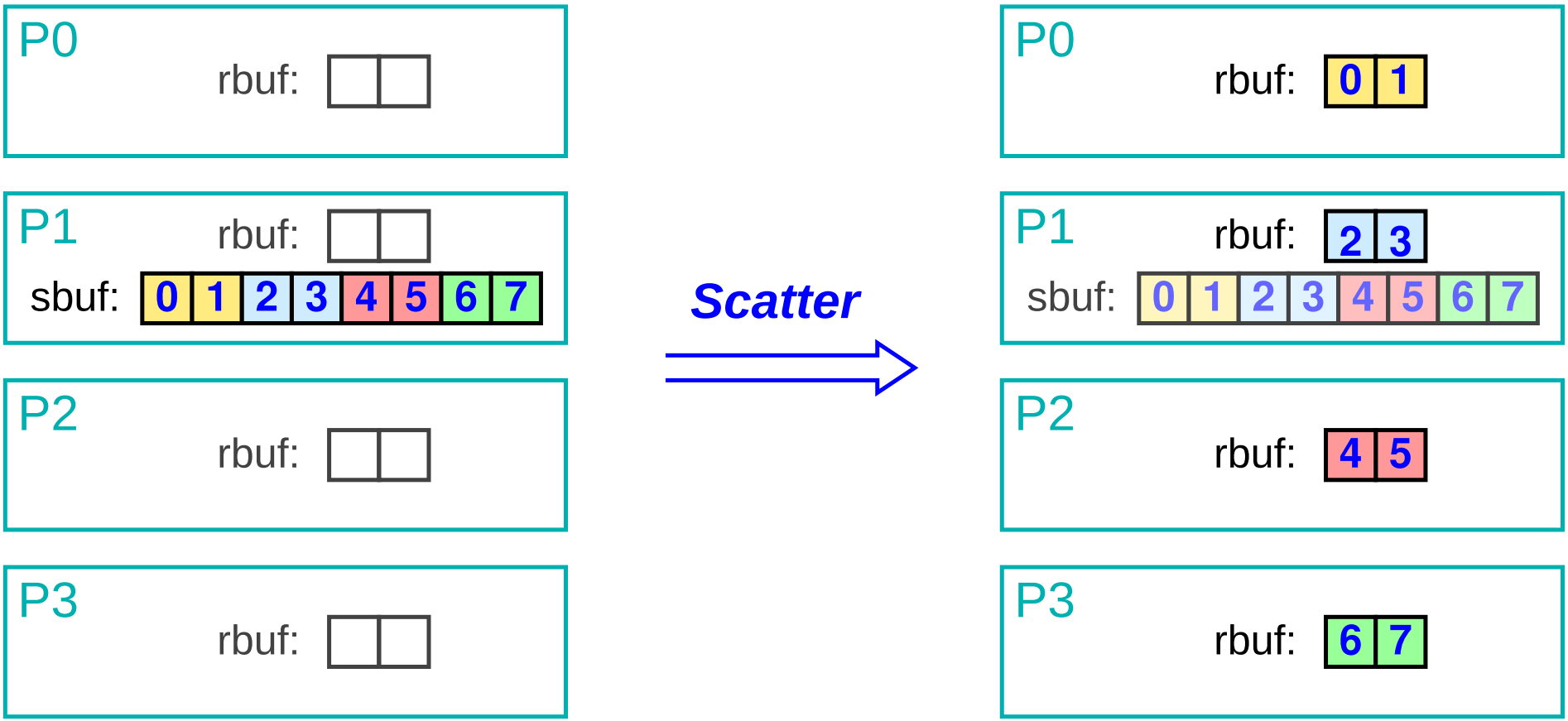
MPI_Bcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype,
              int root, MPI_Comm comm)
```

IN **root** Rank of the sending process

- Buffer is sent by process `root` and reveived by all others
- Collective, blocking operation: no tag necessary
- `count`, `dtype`, `root`, `comm` must be the same in all processes

Collective communication: scatter

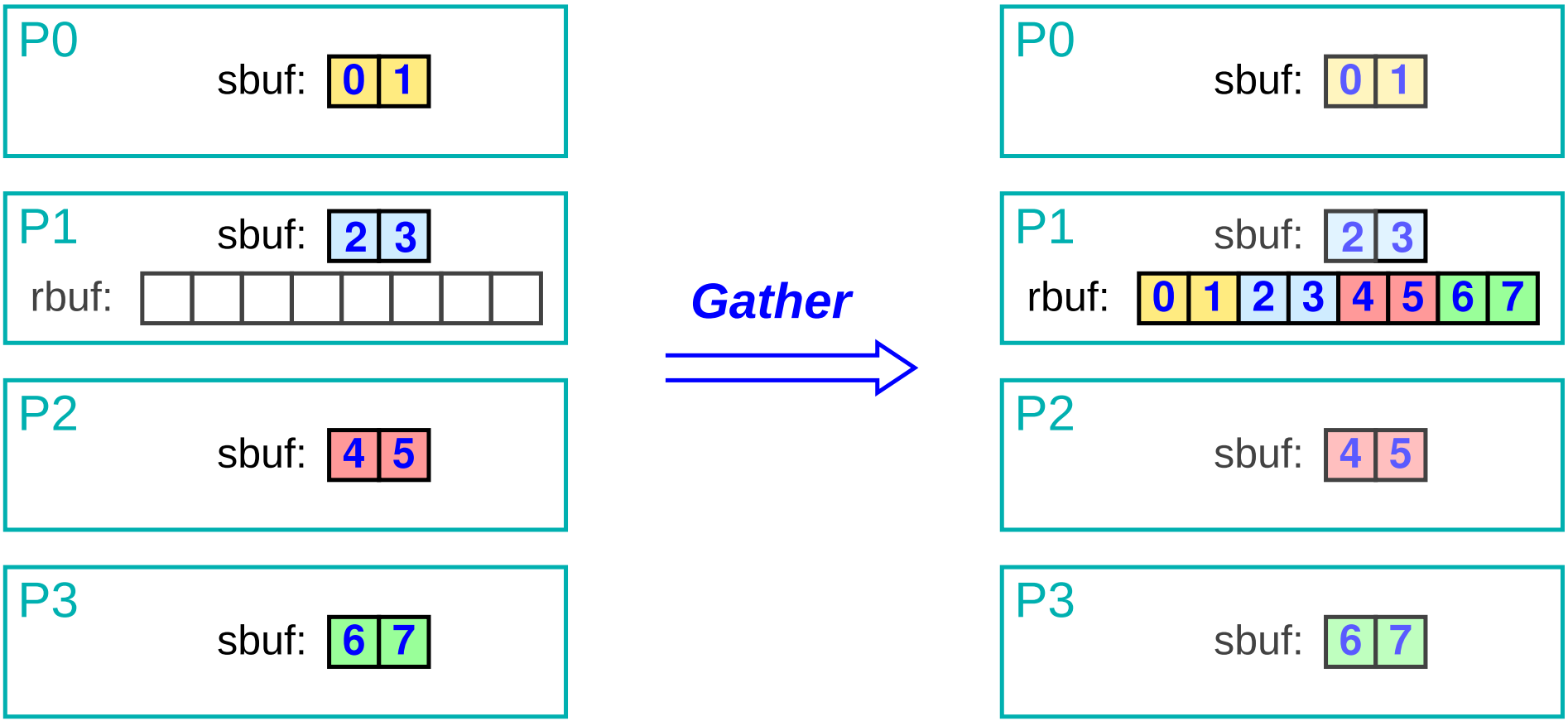


MPI_Scatter

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

- Process `root` sends a part of the data to each process
  - including itself
- `sendcount`: data length for each process (not the total length!)
- Process `i` receives `sendcount` elements of `sendbuf` starting from position `i * sendcount`
- Alternative `MPI_Scatterv`: length and position can be specified individually for each receiver

Collective communication: *Gather*



MPI_Gather

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

- All processes send `sendcount` elements to process `root`
  - even `root` itself
- Important: each process must sent the same amount of data
- `root` stores the data from process `i` starting at position `i` * `recvcount` in `recvbuf`
- `recvcount`: data length for each process (not the total length!)
- Alternative `MPI_Gatherv`: analogous to `MPI_Scatterv`

Example: multiplication of vector and scalar (☞ 03/vecmult.cpp)

```
double a[N], factor, local_a[LOCAL_N];
... // Process 0 reads a and factor from file
MPI_Bcast(&factor, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(a, LOCAL_N, MPI_DOUBLE, local_a, LOCAL_N,
            MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0; i<LOCAL_N; i++)
   local_a[i] *= factor;
MPI_Gather(local_a, LOCAL_N, MPI_DOUBLE, a, LOCAL_N,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
... // Process 0 writes a into file
```

- **Caution:** LOCAL_N must have the same value in all processes!
  - otherwise: use MPI_Scatterv / MPI_Gatherv (☞ 03/vecmult3.cpp)

Reduction: MPI_Reduce

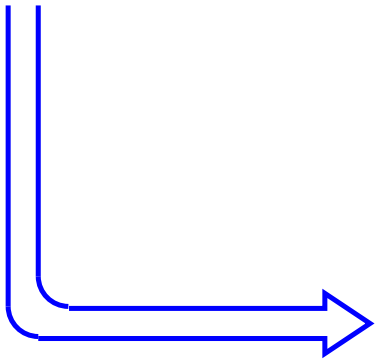
```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype dtype,
               MPI_Op op, int root,
               MPI_Comm comm)
```

- Each element in the receive buffer is the result of a reduction operation (e.g., the sum) of the corresponding elements in the send buffer
- `op` defines the operation
  - predefined: minimum, maximum, sum, product, AND, OR, XOR, ...
  - in addition, user defined operations are possible, too

Example: summing up an array

Sequential

```
s = 0;
for (i=0;i<size;i++)
  s += a[i];
```



Parallel

```
local_s = 0;
for (i=0;i<local_size;i++)
   local_s += a[i];

MPI_Reduce(&local_s, &s,
           1, MPI_INT,
           MPI_SUM,
           0, MPI_COMM_WORLD);
```

More collective communication operations

- `MPI_Alltoall`: all-to-all broadcast (☞ **1.9.5**)
- `MPI_Allgather` and `MPI_Allgatherv`: at the end, all processes have the gathered data
  - corresponds to a gather with subsequent broadcast
- `MPI_Allreduce`: at the end, all processes have the result of the reduction
  - corresponds to a reduce with subsequent broadcast
- `MPI_Scan`: prefix reduction
  - e.g., using the sum: process i receives the sum of the data from processes 0 up to and including i

3.2.7 Further concepts

- Topologies
  - the application's communication structure is stored in a communicator
    - e.g., cartesian grid
  - allows to simplify and optimize the communication
    - e,g,. "send to the left neighbor"
    - the communicating processes can be placed on neighboring nodes
- Dynamic process creation (since MPI-2)
  - new processes can be created at run-time
  - process creation is a collective operation
  - the newly created process group gets its own `MPI_COMM_WORLD`
    - communication between process groups uses an *intercommunicator*

3.2.7 Further concepts ...

- One-sided communication (since MPI-2)
  - access to the address space of other processes
  - operations: read, write, atomic update
  - weak consistency model
    - explicit *fence* and *lock*/*unlock* operations for synchronisation
  - useful for applications with irregular communication
    - one process alone can execute the communication
- Parallel I/O (since MPI-2)
  - processes have individual views to a file
    - specifiec by an MPI data type
  - file operations: individual / collective, private / shared file pointer, blocking / non-blocking

3.2.8 Summary

- Basic routines:
  - `Init`, `Finalize`, `Comm_size`, `Comm_rank`, `Send`, `Recv`
- Complex data types in messages
  - `Pack` and `Unpack`
  - user defined data types
    - also for non-contiguous data (e.g., column of a matrix)
- Communicators: process group + communication context
- Non-blocking communication: `Isend`, `Irecv`, `Test`, `Wait`
- Collective operations
  - `Barrier`, `Bcast`, `Scatter(v)`, `Gather(v)`, `Reduce`, ...