UNIL | Université de Lausanne

Basics on the Message Passing Interface
MPI

Simon Scheidegger
simon.scheidegger@gmail.com
July 23th, 2019
Open Source Economics Laboratory – BFI/UChicago

Including adapted teaching material from books, lectures and presentations by
B. Barney, W. Gropp, G. Hager, M. Martinasso, R. Rabenseifner, O. Schenk, G. Wellein

Outline

1. What is MPI (Message Passing Interface)?
   → Hello World in MPI

2. Messages and point-to-point communication
   → basics on how to send and receive messages

3. Collective Communication
   → Max, Sum, ...

Before we start...

On MIDWAY
>cd ~
>vi .bashrc

→ add the following lines to the .bashrc

module load openmpi

Some literature & other resources

Full standard/API specification (do not look at this – its a specification):
- http://www.mpi-forum.org/

Tutorials:
- https://computing.llnl.gov/tutorials/mpi/

Books:
- “Introduction to High Performance Computing for Scientists and Engineers” Georg Hager, Gerhard Wellein
- "Using MPI", Gropp, Lusk and Skjellum. MIT Press, 1994.

MPI availability

- MPI is standard defined in a set of documents compiled by a consortium of organizations: http://www.mpi-forum.org/

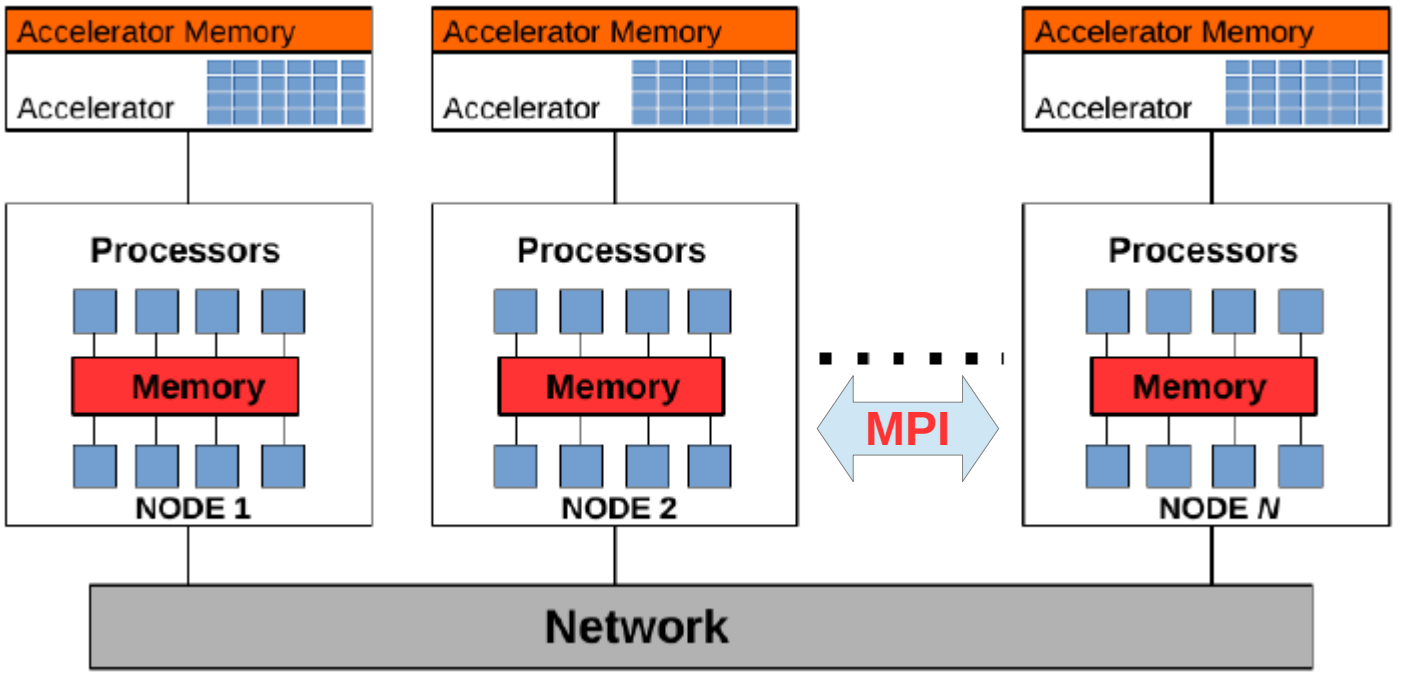
- In particular the MPI documents define the APIs (application programming interfaces) for C, C++, FORTRAN77 and FORTRAN 90.

- Bindings available for Perl, Python, Java...

- In all systems MPI is implemented as a library of subroutines/functions over the network drivers and primitives.

1. What is MPI?

Recall: today's HPC systems



- Ever since parallel computers hit the HPC market, there was an intense discussion about what should be an appropriate programming model for them.

- Message passing is required if a parallel computer is of the distributed memory type, i.e. if there is no way for one processor to directly access the address space of another.

- **The use of explicit message passing (MP), i.e., communication between processes, is surely the most tedious and complicated but also the most flexible parallelization method.**

Message Passing

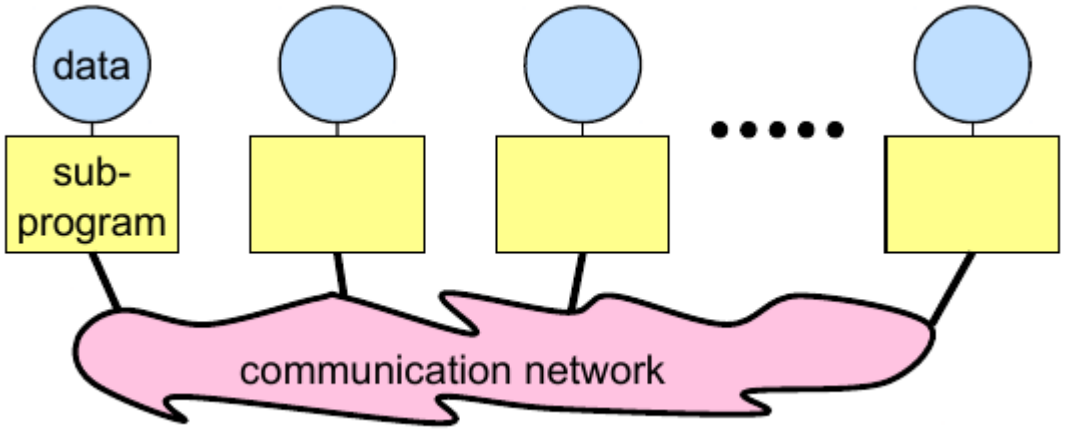
Each processor in a message passing program runs a sub-program:

→ written in a conventional sequential language, e.g., C or C++

→ typically the same on each processor (**single program multiple data - SPMD**)

→ the variables of each sub-program have
- **the same name**
- **but different locations** (distributed memory) and different data!
- **i.e., all variables are private/local (no concept of shared memory)**

→ communicate via special send & receive routines (message passing)



Parallel programming with MPI

The Message Passing Interface Standard (**MPI**) is a **message passing library standard** based on the consensus of the **MPI Forum**, which has over 40 participating organizations, including vendors, researchers, software library developers, and users.

The goal of the MPI is to establish a **portable, efficient, and flexible standard** for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library.

Several MPI implementations exist (Mpich, OpenMPI, IntelMPI, CrayMPI, …).

The current MPI standard currently defines over 500 functions, and it is beyond the scope of this introduction even try to cover them all. Here, I will concentrate on the important concepts of message passing and MPI in particular, **and provide some knowledge that will enable you to consult more advanced textbooks and tutorials.**

MPI history

- Early 80's many communication libraries existed: PVM, LAM, P4,. . .

- **92: agreement to develop one generic library, MPI was born**.

- Many companies helped finance the standard: IBM, Cray,. . .

- **94: First version of the standard was released, MPI-1**

- 95: Mpich and Lam-MPI were the first implementations

- 98: Second version of the standard was released, MPI-2

- 02: MPI implementations were MPI-2 compliant

- 08: Third version of the standard was raised, MPI-3

- 12: MPI-3 released

...**MPI-4.0 Ongoing technical effort: extensions to better support hybrid programming models and fault tolerance**.

Message Passing Interface (MPI)

- Resources are LOCAL (different from shared memory).

- Each process runs in an “isolated” environment. Interactions requires **Messages** to be exchanged.

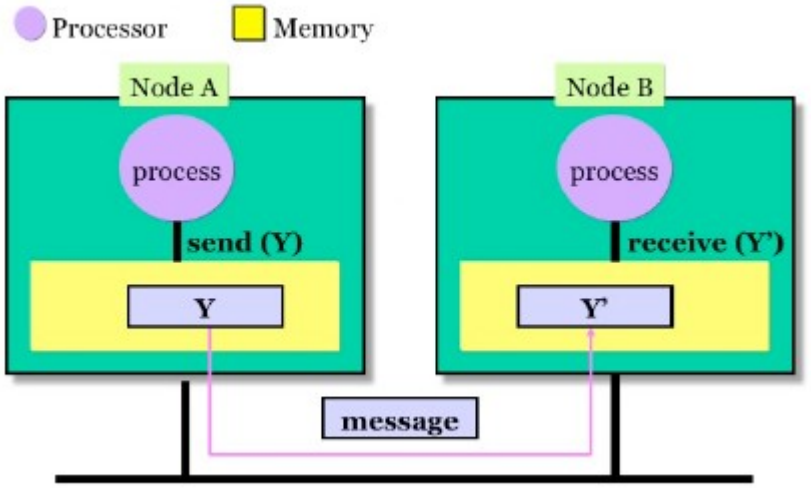
- Messages can be: **instructions, data, synchronization.**

- **MPI works also on Shared Memory systems**.

- Time to exchange messages is much larger than accessing local memory.

→ **Massage Passing is a COOPERATIVE Approach, based on 3 operations:**

- **SEND** (a message)
- **RECEIVE** (a message)
- **SYNCHRONIZE**



Messages

- A message can be as simple as a **single item** (like a number) or even a **complicated structure**, perhaps scattered all over the address space.

- For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance, such as:

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process is going to receive the message?
- Where should the data be left on the receiving process?
- What amount of data is the receiving process prepared to accept?

→ **All MPI calls that actually transfer data have to specify those parameters in some way.**

MPI: Pro's & Con's

Pro's

- **Distribute Memory → use more nodes and cores.**
- Communications is the most important part of HPC.
  → It can be (and is) highly optimized.
- MPI is portable (runs on almost any platform).
- Many current applications/libraries use MPI.
- MPI de-facto standard for distributed memory processing.

Con's

- **Error-prone**.
- Discourages frequent communications (overhead).
- **Technically "hard" to implement**.

General program structure

Header file

→ required for all programs that make MPI calls.
→ contains definitions of MPI constants, types and functions.

Fortran 90
```
USE MPI
```

C/C++
```
#include <mpi.h>
```

MPI initialize/finalize

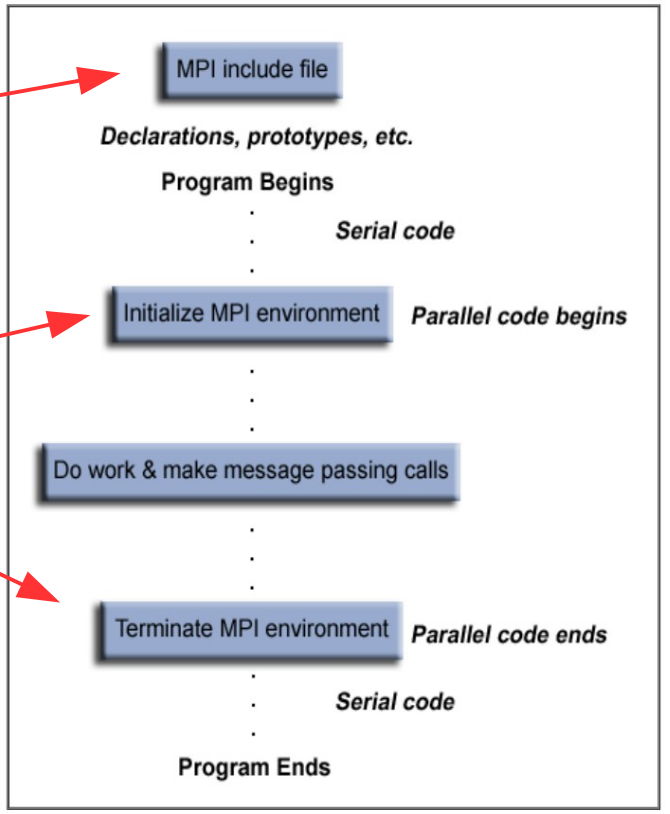
→ Every MPI program starts by calling **MPI_Init**
→ Every MPI program ends by calling **MPI_Finalize**

Fortran
```
INTEGER IERR
MPI_INIT(IERR)
```

Fortran
```
INTEGER IERR
MPI_FINALIZE(IERR)
```

C/C++
```
int MPI_Init(int*argc, char***argv)
```

C/C++
```
int MPI_Finalize()
```



Format of MPI calls

→ C names: case sensitive; Fortran not.
→ Programs must not declare variables or functions with the prefix MPI_ …

MPI communicators and ranks

Communicator

- MPI uses objects called **communicators** and groups to **define which collection of processes may communicate with each other**.
- Most MPI routines require you to specify a communicator as an argument.

- Communicators and groups will be covered in more detail later. For now, simply use **MPI_COMM_WORLD** whenever a communicator is required.

- **It is the predefined communicator that includes all of your MPI processes**.

Rank

- Within a communicator, every process has its own unique, **integer identifier** assigned by the system when the process initializes (**=Rank**)

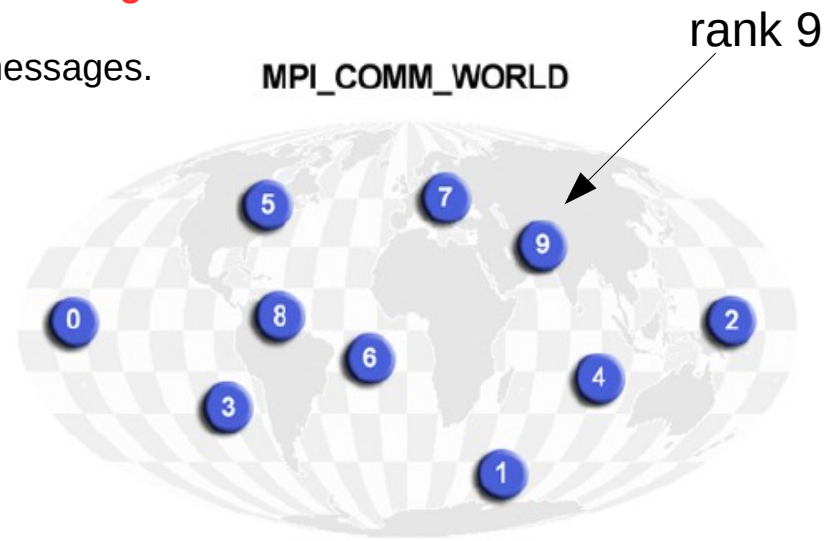
- A rank is sometimes also called a **"task ID"**. Ranks are contiguous and **begin at zero**.

- Used by the programmer to specify the source and destination of messages.

- Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

Error Handling

Most MPI routines include a return/error code parameter.



MPI communicators and ranks (II)

- Every process has a MPI rank and belongs to an MPI communicator.

- An MPI rank is an identification number.

- An MPI communicator is a set of MPI rank.

- Ranks are numbered locally to communicator.

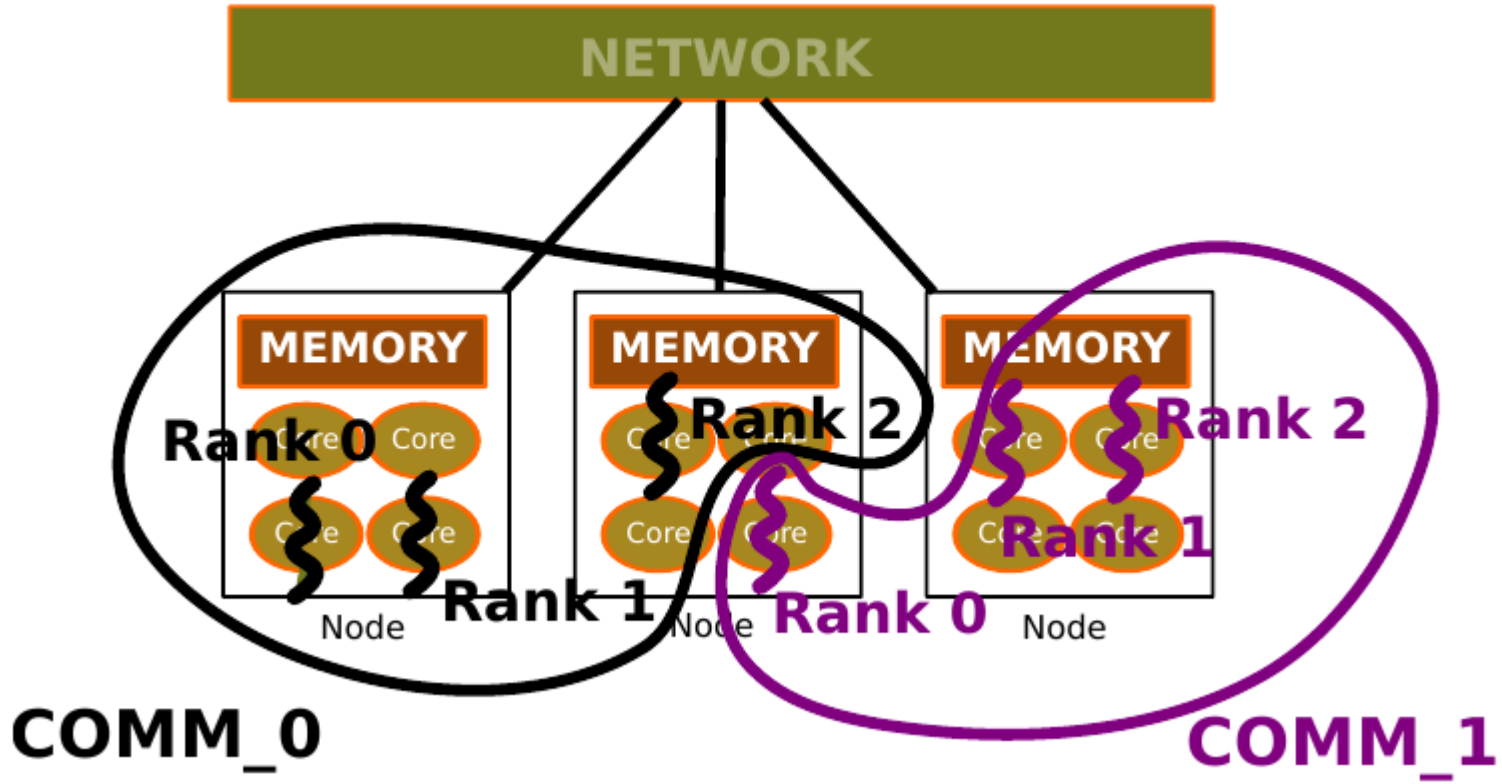


Fig. from M. Martinasso

Process Identification

- How many processes are associated with a communicator?

Fortran
```
INTEGER COMM, SIZE, IERR
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

C/C++
```
MPI_Comm_size(MPI_Comm comm, int *size)
```

- How to get the rank of a process?

Fortran
```
INTEGER COMM, RANK, IERR
CALL MPI_COMM_RANK(COMM, RANK, IERR)
OUTPUT: RANK
```

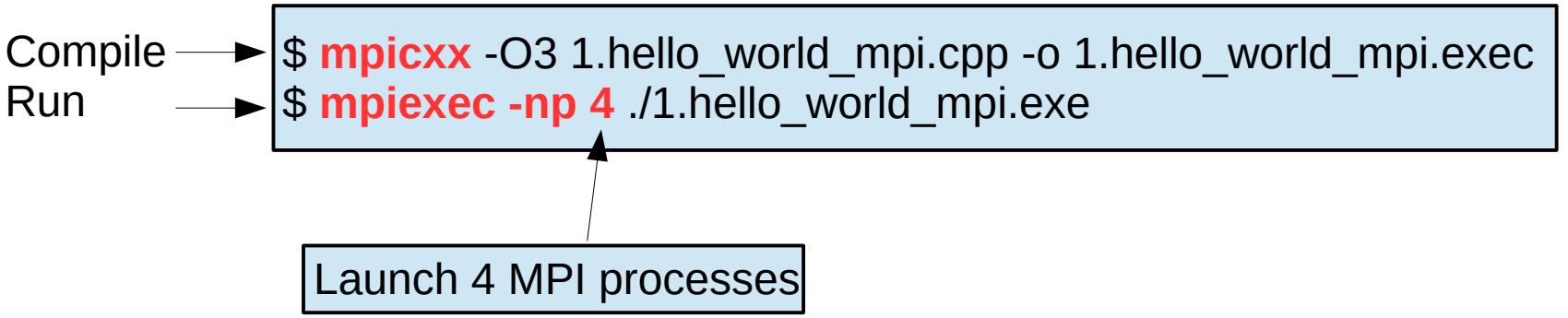
C/C++
```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Compiling and running MPI

MPI is always available as a library. In order to **compile and link** an MPI program, compilers and linkers need options that specify where include files (i.e., C, C++ headers and Fortran modules) and libraries can be found.

As there is considerable variation in those locations among installations, most MPI implementations provide compiler wrapper scripts (often called **mpicxx, mpif90, etc.**), which supply the required options automatically but otherwise behave like “normal” compilers.

Note that the way that MPI programs should be compiled and started is not fixed by the standard, so please consult the system documentation by all means.



Hello World in MPI (Fortran/CPP)

1. go to OSE2019/day3/code_day3/MPI:

> cd OSE2019/day3/code_day3/MPI

2. Have a look at the code

>vi 1.hello_world_mpi.f90 / 1a.hello_world_mpi.cpp

3. compile by typing:

> make

4. Experiment with different numbers of MPI processes

>mpiexec -np 4 ./1.hello_world_mpi.exec (-np #processes)

>mpiexec -np 4 ./1a.hello_world_mpi_cpp.exec (-np #processes)

Hello World in C++

The MPI bindings for the C language follow the case-sensitive name pattern MPI_Xxxxx..., while Fortran is case-insensitive!

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

int main(int argc, char** argv) {
int rank, size;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Hello World, I am %d of %d\n", rank, size);
MPI_Finalize();
return 0;
}
```

This initializes the parallel environment

The **MPI_COMM_WORLD** handle describes all processes that have been started as part of the parallel program.
If required, other communicators can be defined as subsets of **MPI_COMM_WORLD**.

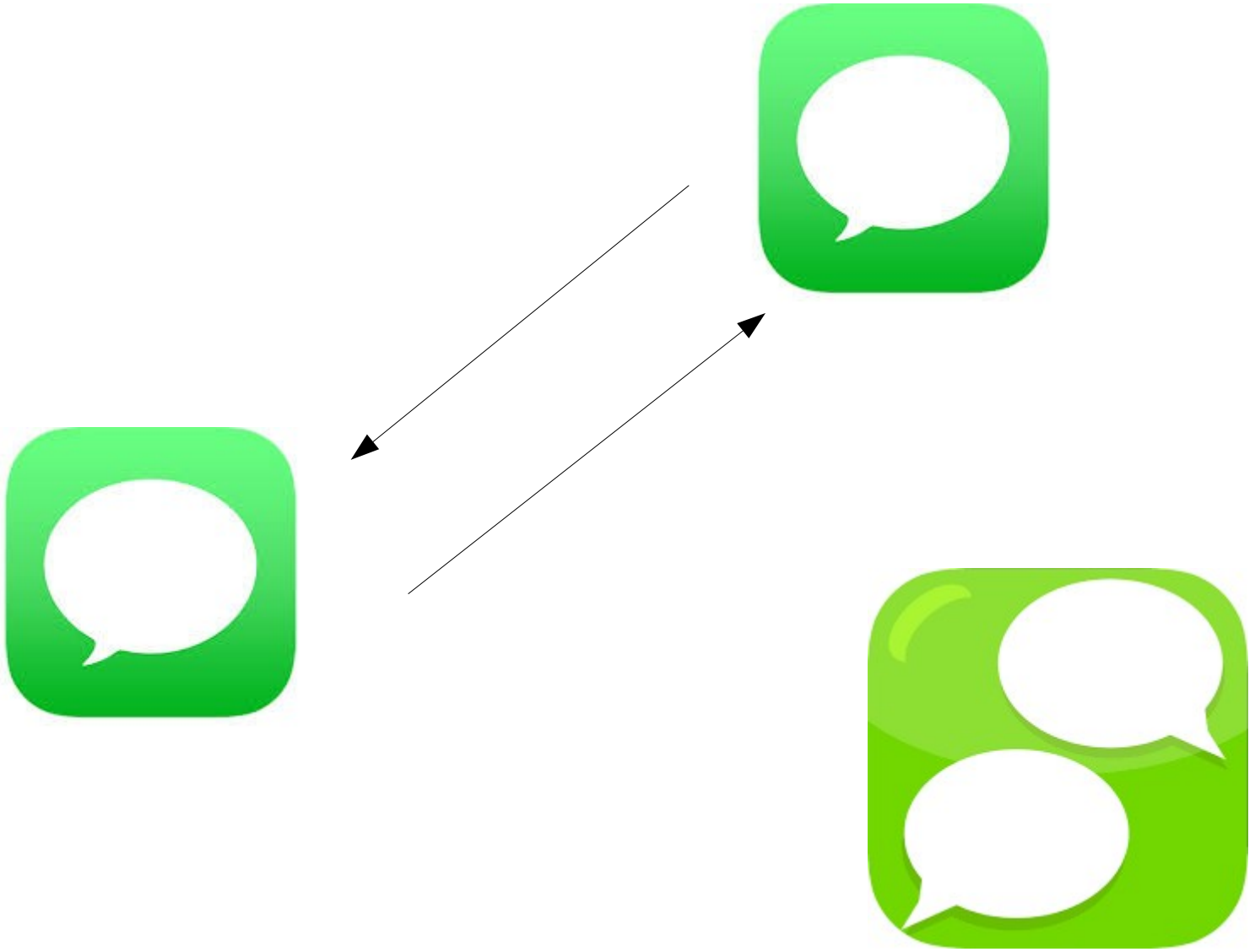
Note that the C bindings require output arguments (like rank and size) to be specified as pointers.

```
mpiexec -np 4 ./1a.hello_world_mpi_cpp.exec
```

```
Hello World, I am 2 of 4
Hello World, I am 0 of 4
Hello World, I am 3 of 4
Hello World, I am 1 of 4
```

Note that no MPI process except rank 0 is guaranteed to

2. Messages and point-to-point communication



Message Envelope

- Communication is performed by explicitly **sending** and **receiving** messages.

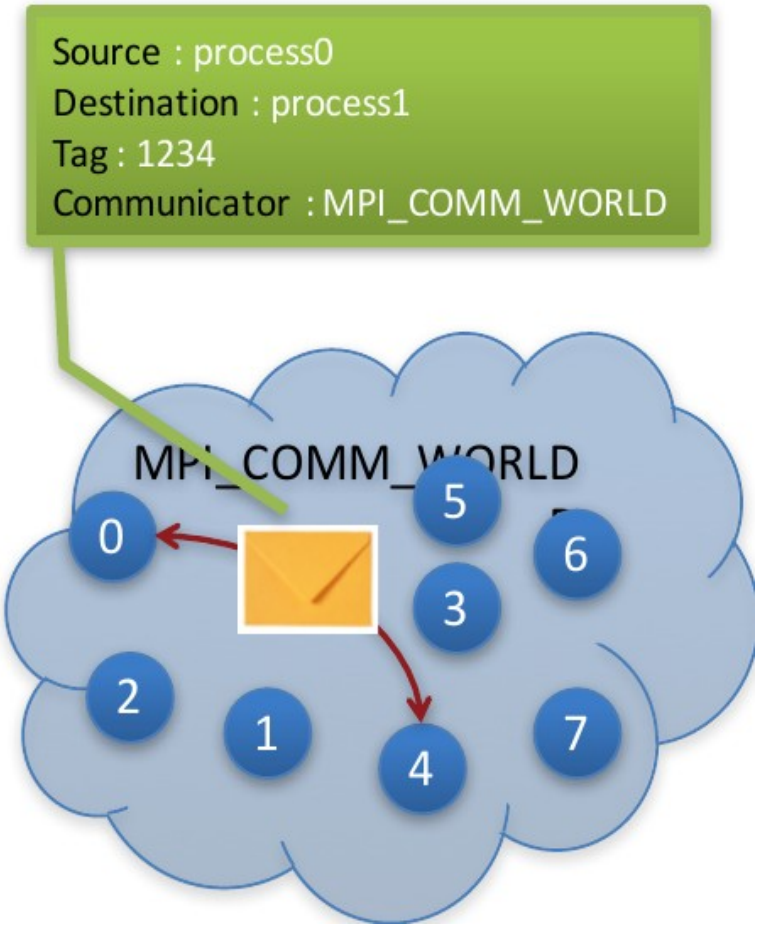
- Messages need **meta data** to describe sender and receiver as well as message contents.

- Sender and receiver are described by ranks within a group.

- Message contents given by starting address, data type, and count, where data type is:

  → Elementary (all C and Fortran datatypes)
  → Contiguous array of datatypes
  → Strided blocks of datatypes
  → Indexed array of blocks of datatypes
  → General structure

MPI also allows messages to be tagged so that programs can deal with the arrival of messages in an orderly way.



Point-to-Point communication

It is the fundamental communication facility provided by a MPI library.

Communication between 2 processes.

It is conceptually simple:

→ **source process A sends a message to destination process B.**
→ **B receives the message from A.**

Communication takes place within a communicator.

Source and Destination are identified by their rank in the Communicator.

Six Functions in MPI

MPI is in principle very simple.

These six functions allow you to write many programs:

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV
- MPI_FINALIZE

Simple MPI communication model

Process with rank 1 sends to process with rank 2 (pseudo code)

```
Rank 1: MPI_Send(<send data buffer>, 2, MyCommunicator)
Rank 2: MPI_Recv(<recv data buffer>, 1, MyCommunicator)
```

- same communicator: MyCommunicator

- send and recv buffer should be compatible:
  - → **receive buffer should be large enough**.
  - → **data type should match**.

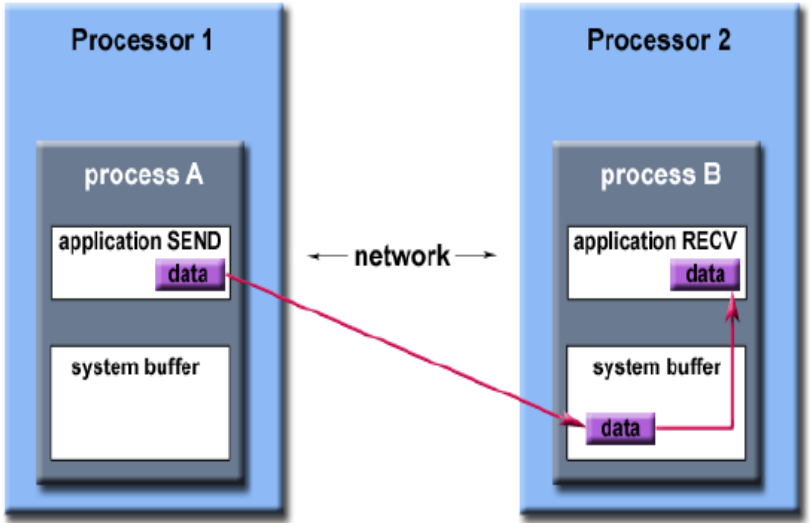
- Rank 2 is prepare to receive data from Rank 1
  - → MPI_Recv is called in ”the right order” (deadlock).
  - → Rank 2 knows the maximum bound on the buffer size.

Message Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive.

- This is rarely the case.

- The MPI implementation must be able to deal with storing the data **when the two tasks are out of sync.**



Consider the following two cases:

→ A send operation occurs 5 seconds before the receive is ready where is the message while the receive is pending?

→ Multiple sends arrive at the same receiving task which can only accept one send at a time
what happens to the messages that are "backing up"?

- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit.

Point-to-point communication

- “Hello World” example did not contain any real communication apart from starting and stopping processes.

- **Point-to-Point** communication is the fundamental communication facility provided by MPI.
→ Communication between two separate processes.

→ **Source process A** sends a message to **destination process B**.
→ B receives the message from A.
→ Communication takes place within a communicator.
→ Source and destination are identified by their rank in the communicator.

MPI_Send - Perform a standard-mode blocking send

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

Parameters

buf	Initial address of send buffer
count	Number of elements to send
datatype	Datatype of each send element
dest	Rank of destination
tag	Message tag
comm	Communicator

MPI_Send - Perform a standard-mode blocking receive

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

Parameters

buf	Initial address of send buffer
count	Number of elements to send
datatype	Datatype of each send element
dest	Rank of destination
tag	Message tag
comm	Communicator
status	Status object

Messages – terminology

Data is exchanged in the **buffer, an array of count elements of some particular MPI data type**.

→ One argument that usually must be given to MPI routines is the **type of the data being passed**.

→ This allows MPI programs to run automatically in heterogeneous environments.

Messages are identified by their envelopes.

A message could be exchanged only if the sender and receiver specify the correct envelope.

body			envelope			
buffer	count	datatype	source	destination	communicator	tag

Send/receive revisited

```
Fortran
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, source, tag, comm, status, ierr)
```

buf	BODY
count	
type	
source/dest	ENVELOPE
tag	
comm	
status	
ierr	(INTEGER) error code (if ierr=0 no error occurs)

MPI status

MPI_Status structures are used by the message receiving functions to return data about a message.

It is an INTEGER array of MPI_STATUS_SIZE elements in Fortran.

The array contains the following info:

→ **MPI_SOURCE -** ID of processor sending the message
→ **MPI_TAG -** the message tag
→ **MPI_ERROR -** error status

There may also be other fields in the structure, but these are reserved for the implementation.

Wildcards

Both in Fortran and C/C++, MPI_Recv accepts wildcards:

→ To receive from any source: **MPI_ANY_SOURCE**

→ To receive with any tag: **MPI_ANY_TAG**

→ Actual source and tag are returned in the receivers status parameter.

Message ordering

- Each process has a **FIFO (first in, first out)** receipt (queue), incoming messages never overtake each other.

- **If process A does multiple sends to process B those messages arrive in the same order.**

Data types

- MPI provides its own **data type** for **send and recv buffers**.

- Handle type conversion in a heterogeneous collection of Machines.

General rule:
MPI data type must match data types among pairs of send and recv.

- MPI defines ”handles” to allow programmers to refer to data types.

Fortran - MPI Intrinsic Data types

MPI Data type	Fortran Data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_PACKED	-
MPI_BYTE	-

C - MPI Intrinsic Data types

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

Example Ping - Pong

1. go to OSE2019/day3/code_day3/MPI:

> cd OSE2019/day3/code_day3/MPI

2. Have a look at the code

>vi 2a.ping_poing.cpp

3. compile by typing:

> make

4. run the code

>mpiexec -np 2 ./2a.ping_pong.exec

5. What happens if you do not run the executable with 2 processes?

Ping - Pong

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

int main(int argc,char **argv)
{
  int rank,size;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int count;
  if ( rank == 0 ) {
    // initialize count on process 0
    count = 0;
  }
  for (int i=0;i<42;i++) {
    if ( rank == 0 ) {
      MPI_Send(&count,1,MPI_INT,1,0,MPI_COMM_WORLD); // send "count" to rank 1
      MPI_Recv(&count,1,MPI_INT,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE); // receive it back
      count++;
    } else {
      MPI_Recv(&count,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      MPI_Send(&count,1,MPI_INT,0,0,MPI_COMM_WORLD);
    }
  }

  if ( rank == 0 ) printf(" Round trip count = %d\n",count);

  MPI_Finalize();
}
```

Run with

>mpiexec -np 2 ./2.ping_pong.exec

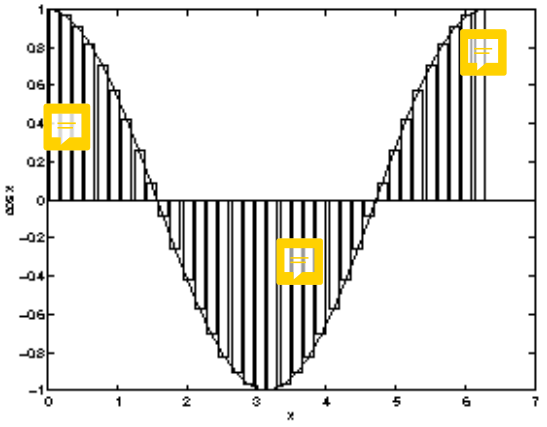
Example – Integration

$$\int_a^b \cos(x)dx = \sum_{i=0}^{p-1}\sum_{j=0}^{n-1}\int_{a_{ij}}^{a_{ij}+h}\cos(x)dx$$

$$\simeq \sum_{i=0}^{p-1}\left[\sum_{j=0}^{n-1}\cos(a_{ij}+\frac{h}{2})h\right]$$

where

$$\begin{aligned} p &= \text{number of processes} \\ n &= \text{number of intervals per process} \\ a &= \text{lower limit of integration} \\ b &= \text{upper limit of integration} \\ h &= \frac{b-a}{np} \\ a_{ij} &= a+[i*n+j]h \end{aligned}$$



Program fragment: parallel Integration

```
pi = acos(-1.0);  /* = 3.14159... */
a = 0.;           /* lower limit of integration */
b = pi*1./2.;     /* upper limit of integration */
n = 500;          /* number of increment within each process */
tag = 123;        /* set the tag to identify this particular job */
h = (b-a)/n/p;    /* length of increment */

ai = a + myid*n*h;  /* lower limit of integration for partition myid */
my_int = integral(ai, h, n);   /* 0<=myid<=p-1 */

cout << "Process " << myid <<" has the partial integral of " << my_int << endl;

MPI_Send(        /* Send my_int from myid to master */
        &my_int, 1, MPI_FLOAT,
        master,        /* message destination */
        tag,          /* message tag */
        comm);

if(myid == master) {
  integral_sum = 0.0;
  for (proc=0;proc< p; proc++)

  {   /* for-loop serializes receives */
    MPI_Recv(         /* Receive my_int from proc to master  */
          &my_int, 1, MPI_FLOAT,
          proc,        /* message source */
          tag,         /* message tag */
          comm, & status);     /* status reports source, tag */
    integral_sum += my_int;
  }
  cout << "The Integral = " << integral_sum << endl; /* sum of my_int */
}
MPI_Finalize();                       /* let MPI finish up ... */
```

Split work/domain wrt. all ranks available

Where is partial sum from?

Results on rank 0

Example – Integration

1. go to OSE2019/day3/code_day3/MPI:

> cd OSE2019/day3/code_day3/MPI

2. Have a look at the code

>vi 2c.integration.cpp

3. compile by typing:

> make

4. run the code

>mpiexec -np 2 ./2c.integrate.exec

→ **Play with different number of procs.**
→ **Change f(x), and also the integration bounds.**

More communication synchronous send

- Different types of point-to-point communication:

– **synchronous send**

→ **The sender gets an information that the message is received.**
→ **Analogue to the beep or okay-sheet of a fax.**

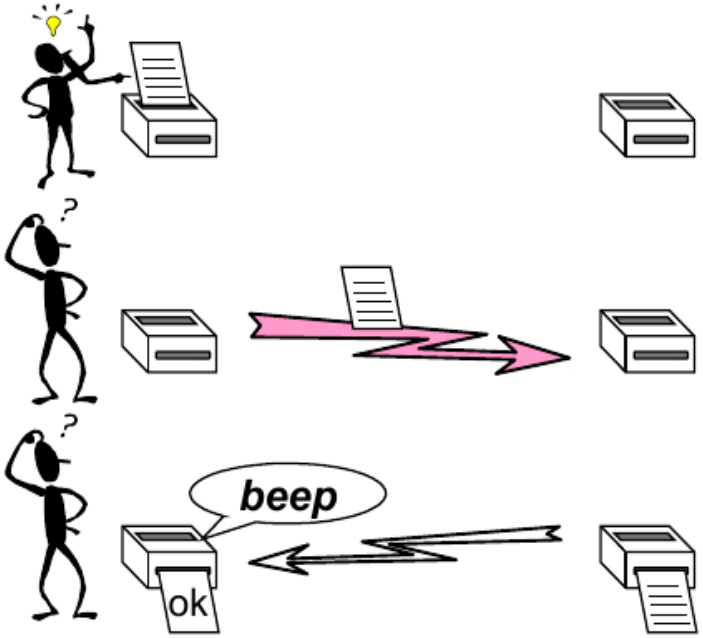


Fig. from R. Rabenseifner

Buffered = Asynchronous Sends

Only know when the message has left.

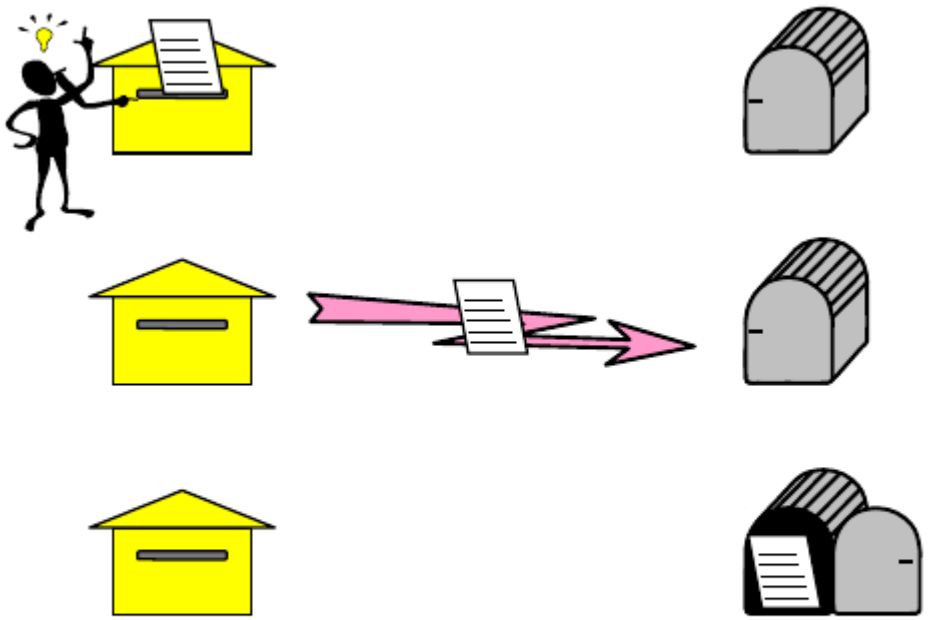


Fig. from R. Rabenseifner

Blocking Operations

Operations are local activities, e.g.,
→ sending (a message)
→ receiving (a message)

Some operations may block until another process acts:
→ synchronous send operation blocks until receive is posted.
→ receive operation blocks until message was sent.

Relates to the completion of an operation.

Blocking subroutine returns only when the operation has completed.

A blocking communication means. . .

That the buffer can be re-used after the function as returned.

Non-Blocking Operations

- Non-blocking operation: returns immediately and allow the sub-program to perform other work.
- At some later time the sub-program must test or wait for the completion of the non-blocking operation.

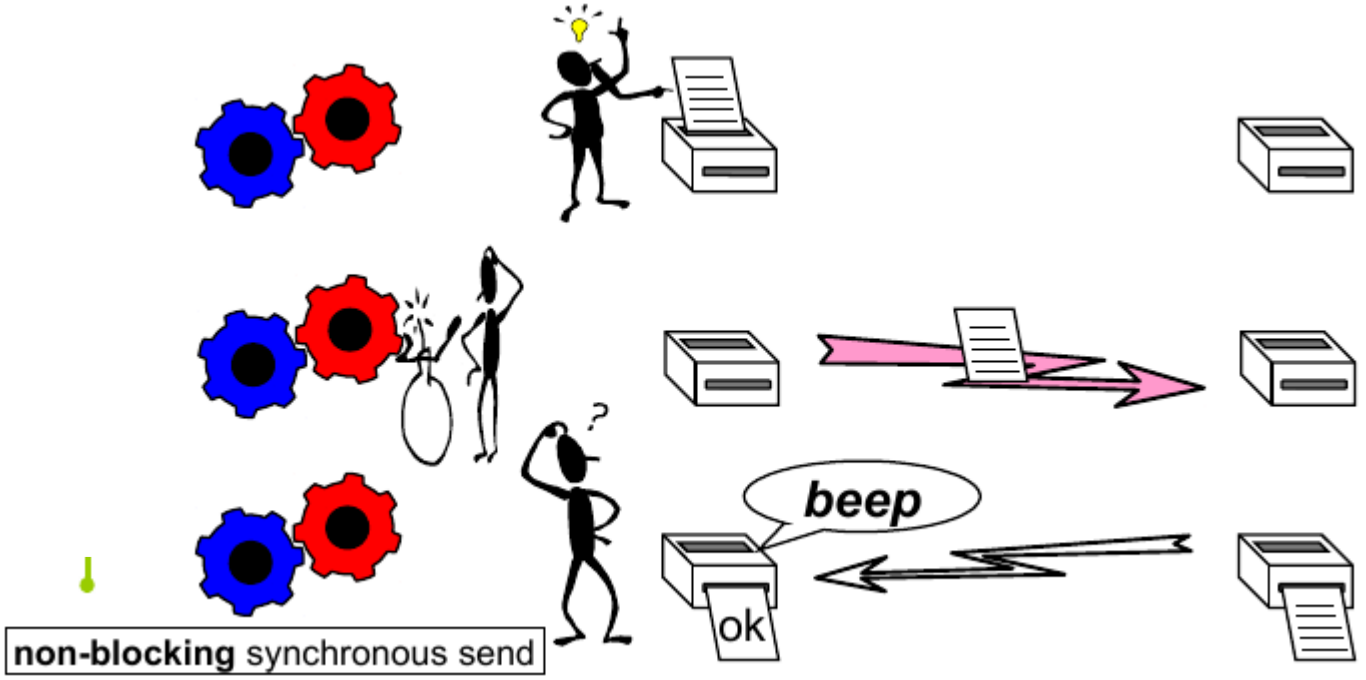


Fig. from R. Rabenseifner

Non-Blocking Operations II

- All non-blocking operations must have matching wait (or test) operations (some system or application resources can be freed only when the non blocking operation is completed.);

- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.

- Non-blocking operations are not the same as sequential subroutine calls:

  → the operation may continue while the application executes the next statements!

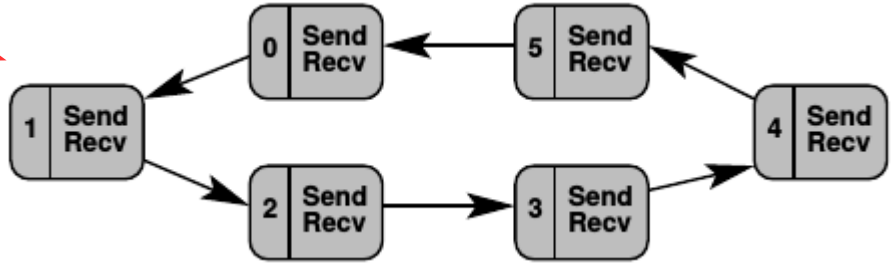
A non-blocking communication means. . .

That the buffer can only be re-used after a wait function as returned.

Until then, the buffer must not be overwritten (and even read).

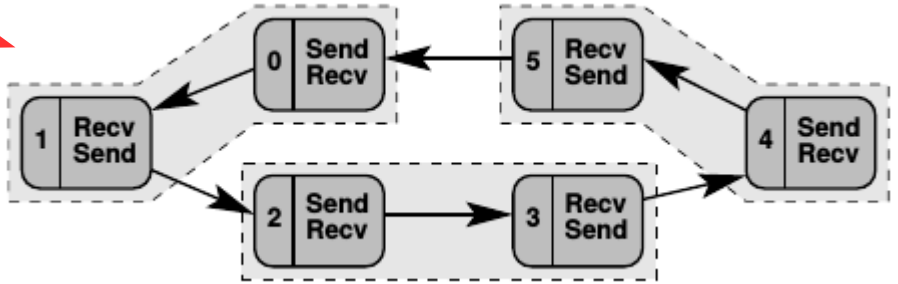
Deadlock – an example

A ring shift communication pattern. If sends and receives are performed in the order shown, a **deadlock** can occur because MPI_Send() may be synchronous.

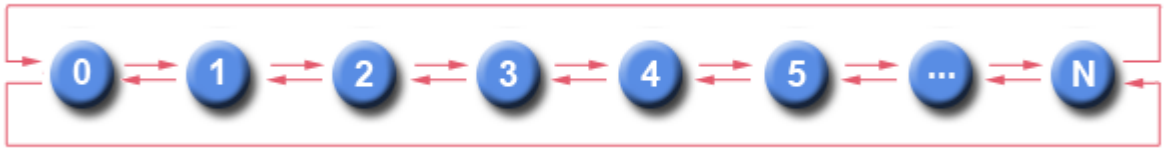


A possible **solution for the deadlock** problem with the ring shift:

→ By changing the order of MPI_Send() and MPI_Recv() on all odd-numbered ranks, pairs of processes can communicate without deadlocks because there is now a matching receive for every send operation (dashed boxes).



Better: Non-blocking routines



Nearest neighbor exchange in a ring topology

Go to OSE2019/day3/code_day3/MPI
→ vi 2d.ringtopo.cpp

mpiexec -np 4 2d.ringtopo.exec

```
using namespace std;

main(int argc, char *argv[])  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];   // required variable for non-blocking calls
MPI_Status stats[4];   // required variable for Waitall routine

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

// determine left and right neighbors
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

// post non-blocking receives and sends for neighbors
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

cout << "my rank is " << rank << "  and my neighbors are rank " << prev << " and " << next << endl;


// wait for all non-blocking operations to complete
MPI_Waitall(4, reqs, stats);

   // continue - do more work

MPI_Finalize();
```

```
my rank is          0 and my neighbors are rank          3 and          1
my rank is          1 and my neighbors are rank          0 and          2
my rank is          2 and my neighbors are rank          1 and          3
my rank is          3 and my neighbors are rank          2 and          0
```

Non-blocking send and receive

Waiting for completion

Transfer modes overview

Mode	Completion Condition	Blocking	Non-blocking
Standard send	Message sent (receive state unknown)	`MPI_Send`	`MPI_Isend`
Receive	Completes when a matching message has arrived	`MPI_Recv`	`MPI_Irecv`
Synchronous send	Only completes after a matching recv() is posted and the receive operation is at some stages.	`MPI_Ssend`	`MPI_Issend`
Buffered send	Always completes, irrespective of receiver. Guarantees the message to be buffered.	`MPI_Bsend`	`MPI_Ibsend`
Ready send	Always completes, irrespective of whether the receive has completed.	`MPI_Rsend`	`MPI_Irsend`

Communication – summary*

Blocking send and recv

→ Does not mean that the process is stopped during communication.

→ It means that, at return, it is safe to use the variables Involved in communication.

Non Blocking send and recv

→ Cannot use variables involved in communication until ”wait” completion functions are called.

Transfer modes

→ Define the behaviour of the various function for point to point communication. The behaviour can be Implementation dependent.

*Many more operations such as send-receive,...

3. Collective Communication

Collective communication in MPI

→ Coordinated communication among a group of processes, as specified by a communicator.

→ **All collective operations are blocking**.

→ All processes in the communicator group must call the collective operation.

→ **Message tags are not used**.

→ Three classes of collective operations:
- Data movement
- Collective computation
- Synchronization

Collective communication in MPI (II)

Collective communication routines must involve all processes within the scope of a Communicator (e.g. MPI_COMM_WORLD).

Types of Collective Operations:

Synchronization:

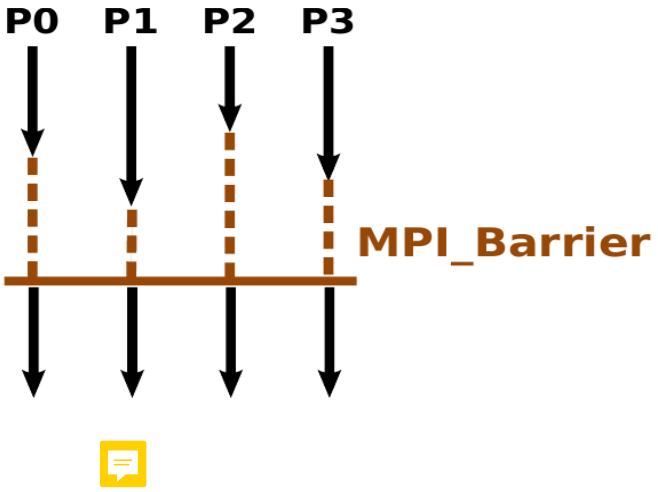
→ processes wait until all members of the group have reached the synchronization point (e.g. a barrier).

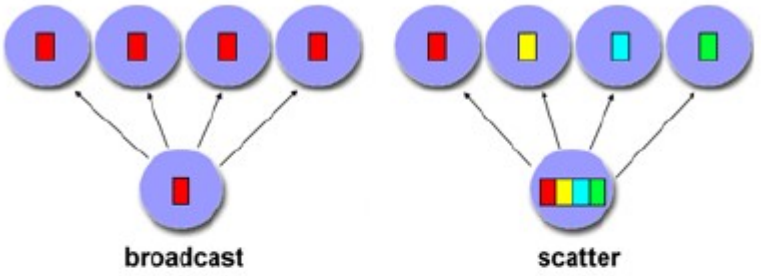
Data Movement:

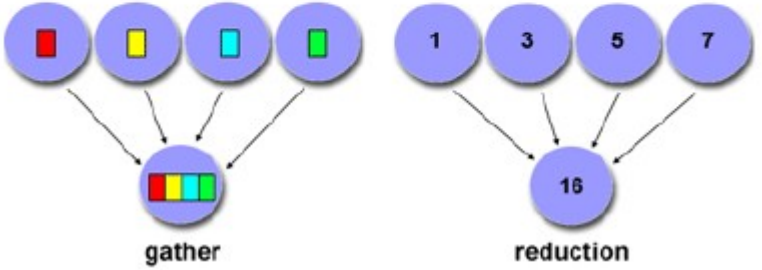
→ broadcast, scatter/gather, all to all.

Collective Computation (reductions):

→ one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.







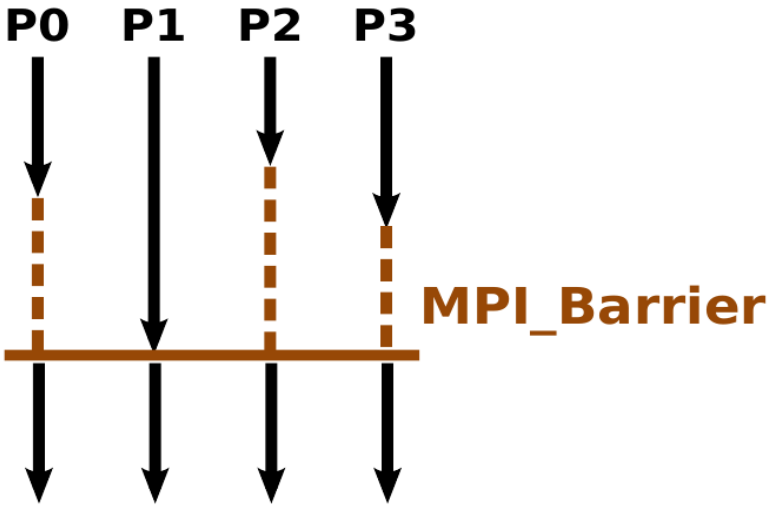
MPI Barrier

The simplest collective in MPI, and one that does actually not perform any real data transfer.

The barrier synchronizes the members of the communicator, i.e., all processes must call it before they are allowed to return to the user code.

Can be useful e.g. for **debugging** or profiling.

MPI_Barrier – Blocks until all processes in communicator have called this routine	
`int MPI_Barrier(MPI_Comm comm)`	
Parameters	
comm	Communicator



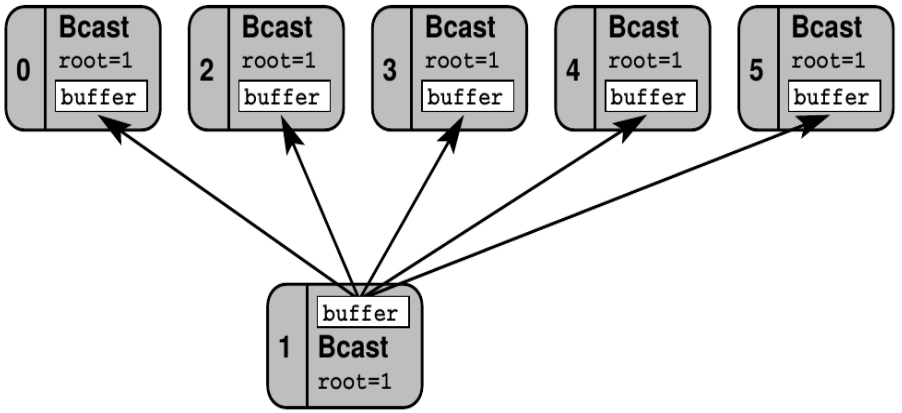
MPI Broadcast

One-to-all communication: same data sent from root process to all other processes in the communicator.

The concept of a “root” rank, at which some general data source or sink is located, is common to many collective routines.

Although rank 0 is a natural choice for “root,” it is in no way different from other ranks.

The buffer argument to MPI_Bcast() is a send buffer on the root and a receive buffer on any other process.



MPI_Bcast - Broadcast a message from the process with rank *root* to all other processes of the group

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

Parameters

buffer	Starting address of buffer
count	Number of entries in buffer
datatype	Datatype of each send element
root	Rank of broadcast root
comm	Communicator

Example – Bcast

1. go to OSM2019/day3/code_day3/MPI:

> cd OSM2019/day3/code_day3/MPI

2. Have a look at the code

>vi 3a.bcast.cpp

3. compile by typing:

> make

4. run the code

>mpiexec -np 2 ./3a.cpp.bcast.exec

→ **Play with the root process.**

Example - Bcast

```
/*============================================================================
 *
 *     example program to demonstrate Bcast
 *
 * ============================================================================*/

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

   using namespace std;

   int main(int argc, char *argv[]) {
   int rank, data;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int root_process = 0;

   if (rank==0){
     data =5;
   }
   else{
     data = 10;
   }


   /* broadcast the value of data of rank 0 to all ranks */
   MPI_Bcast(&data, 1, MPI_INT, root_process, MPI_COMM_WORLD);

   cout << "I am rank " << rank << " and the value is " <<  data << endl;
   MPI_Finalize();
   return 0;
}
```

Scatter

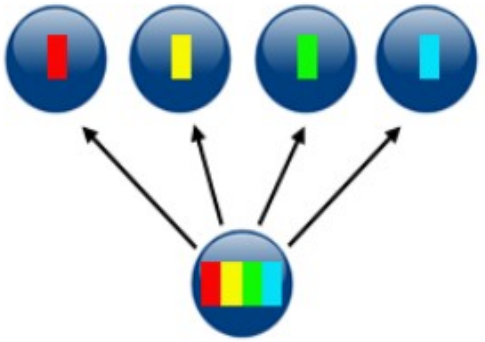
One-to-all communication: different data sent from the root process to all other processes in the communicator.

```
CALL MPI_SCATTER(sndbuf, sndcount, sndtype,
                 rcvbuf, rcvcount, rcvtype,
                 root, comm, ierr)
```

sndcount (INTEGER) number of elements sent to each process, not the size of sndbuf, that should be sndcount times the number of process in the communicator

rcvcount (INTEGER) number of element in the receive buffer

Note: buffer size can be different!



scatter

Scatter with identical buffer size

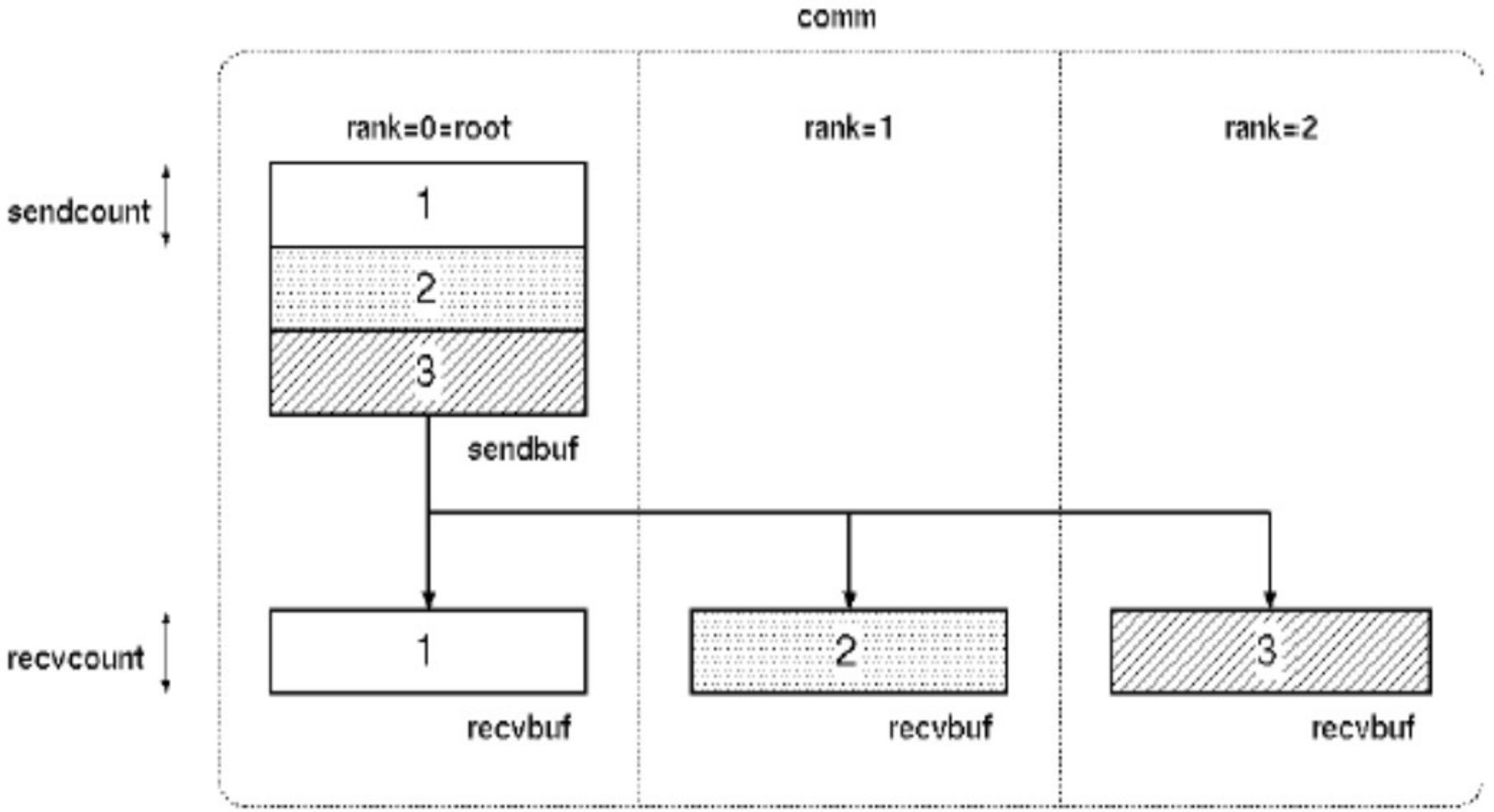


Fig. from M. Martinasso

Scatter with varying buffer size

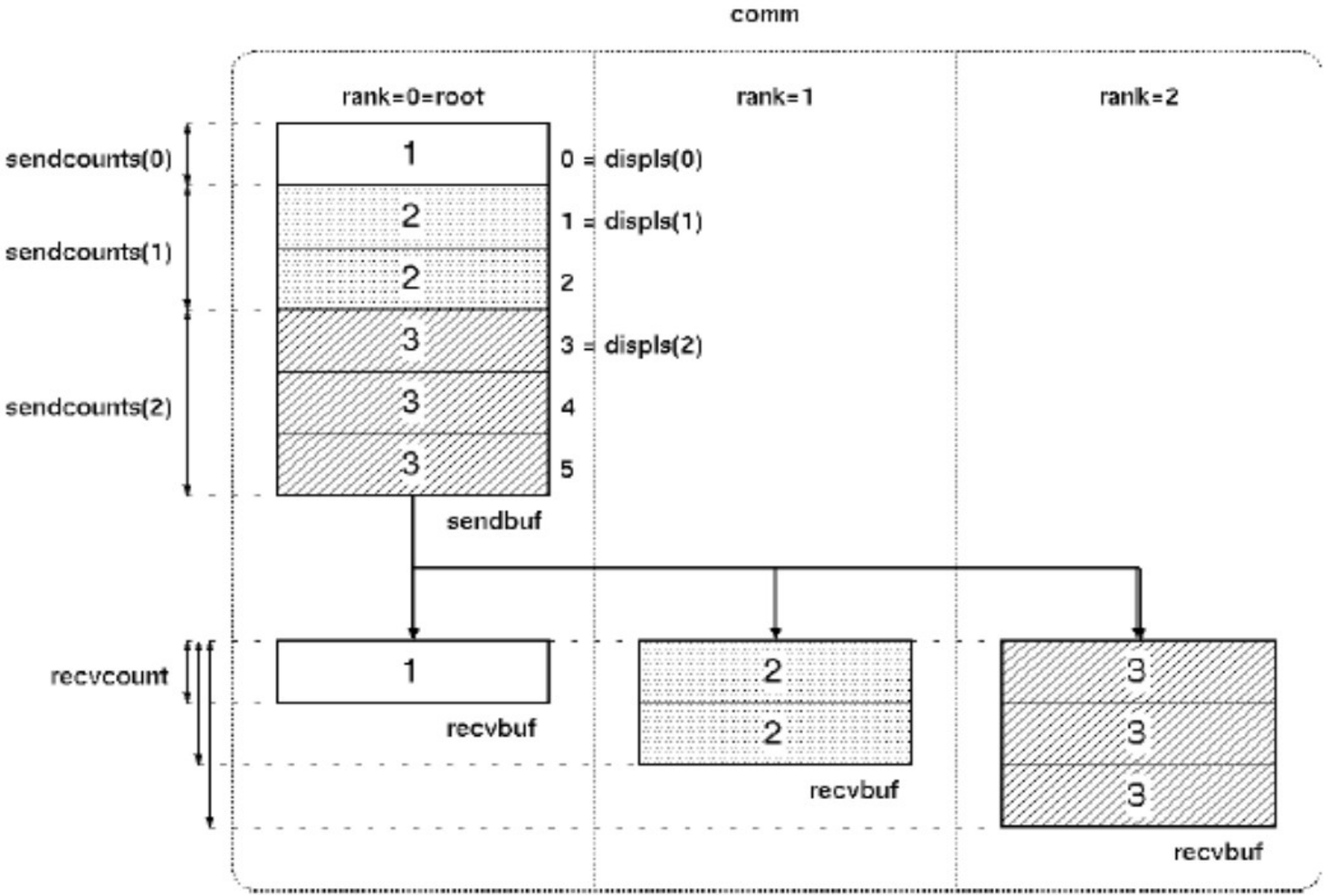


Fig. from M. Martinasso

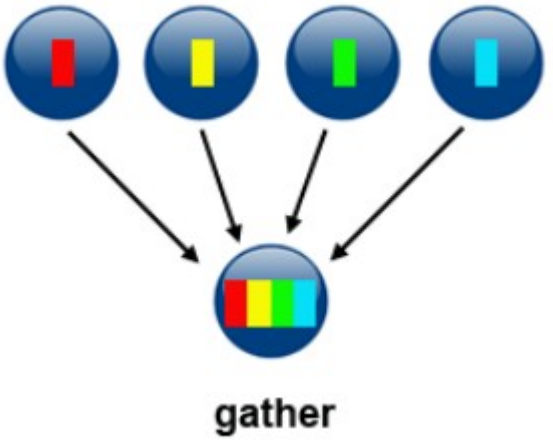
Gather

One-to-all communication: different data collected by the root process, from all others processes in the communicator.

```
CALL MPI_GATHER(sndbuf, sndcount, sndtype,
                rcvbuf, rcvcount, rcvtype,
                root, comm, ierr)
```

rcvcount (INTEGER) the number of elements collected from each process, not the size of rcvbuf, that should be rcvcount times the number of process in the communicator

sndcount (INTEGER) number of element in the send buffer



Gather with identical buffer size

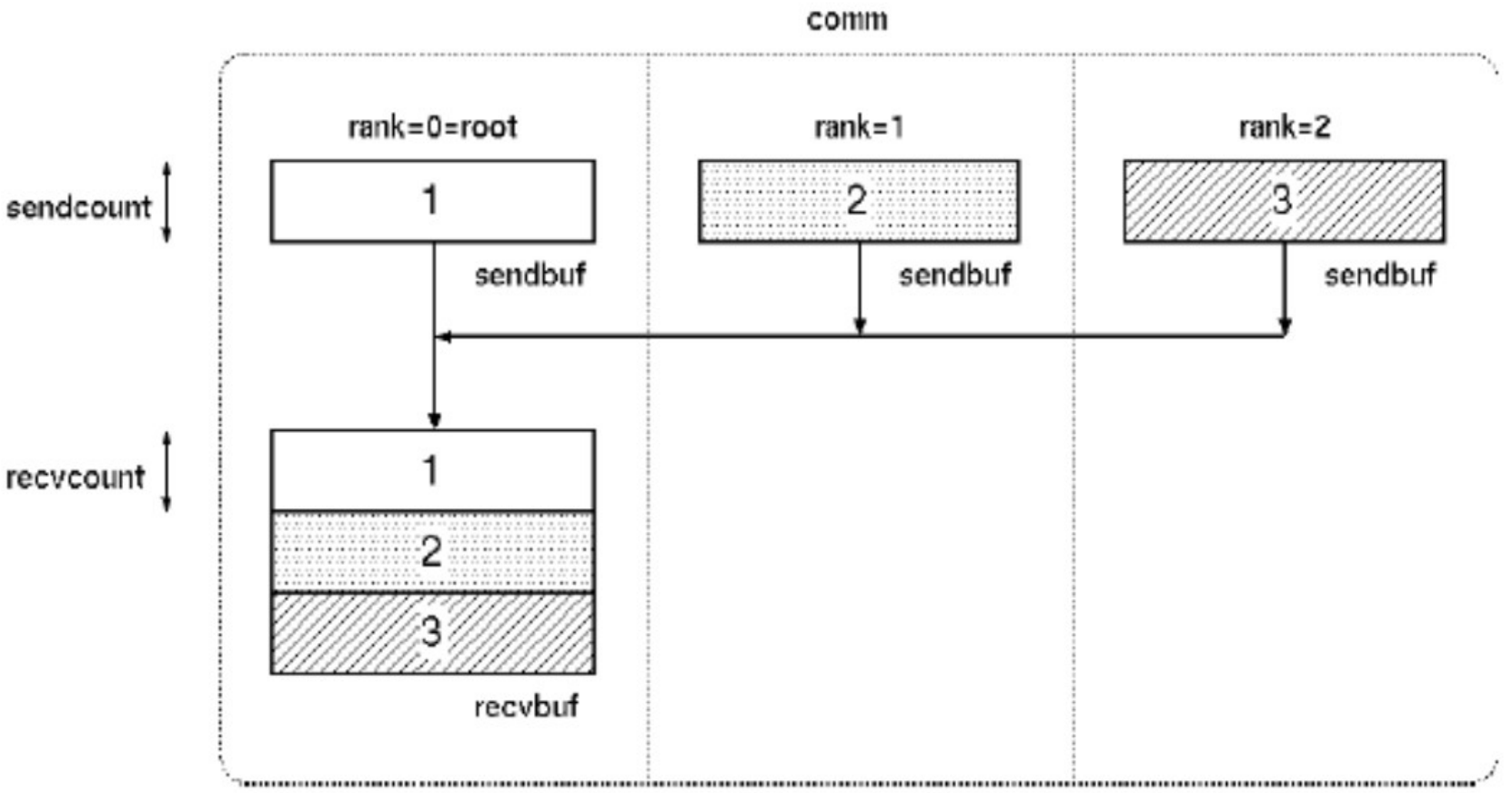


Fig. from M. Martinasso

Gather with varying buffer size

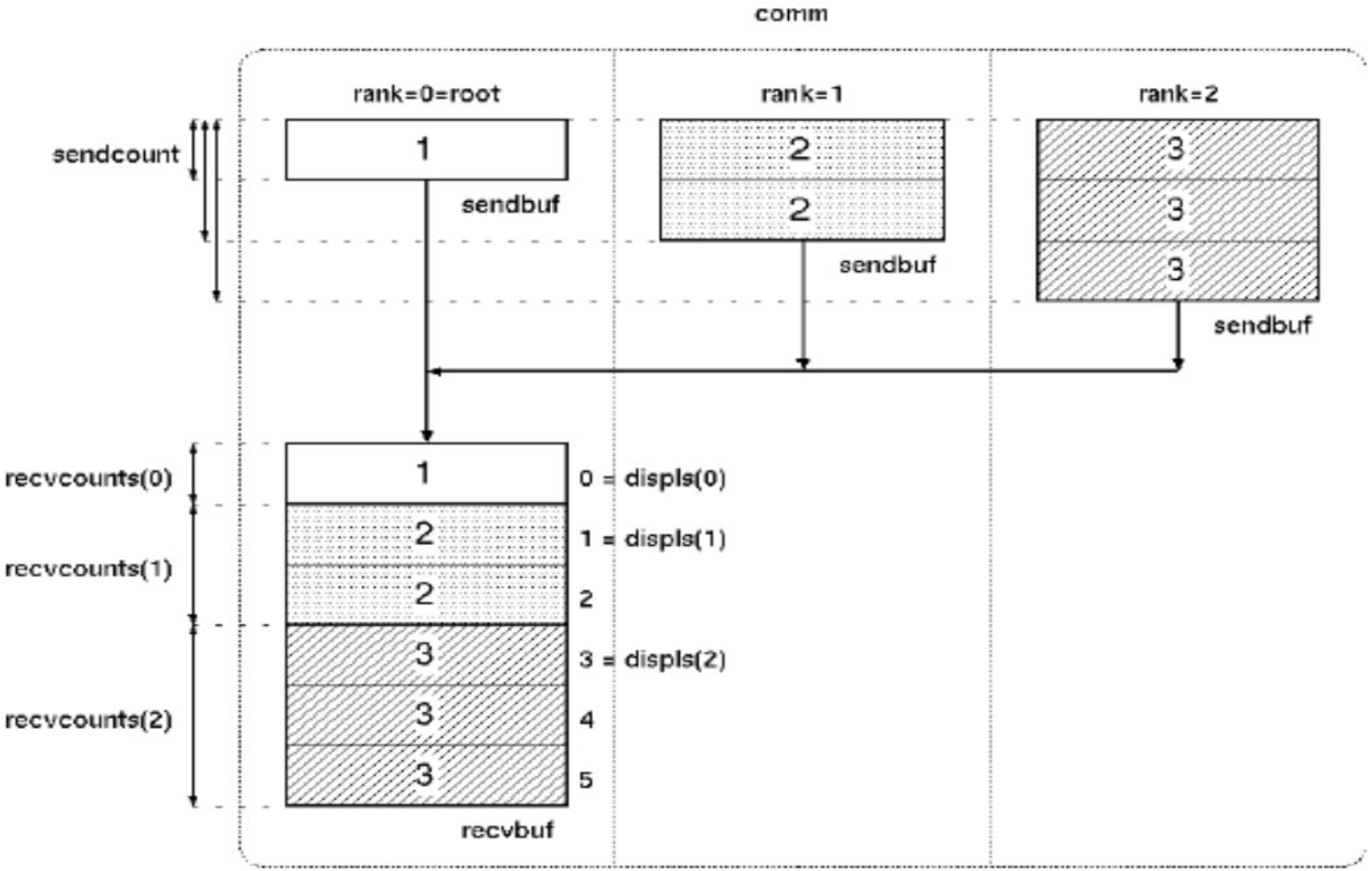


Fig. from M. Martinasso

Global exchange

All-to-all communication: global exchange, all processes exchange their data. Useful for data transposition.

```
CALL MPI_ALLTOALL(sndbuf, sndcount, sndtype,
                  rcvbuf, rcvcount, rcvtype, comm, ierr)
```

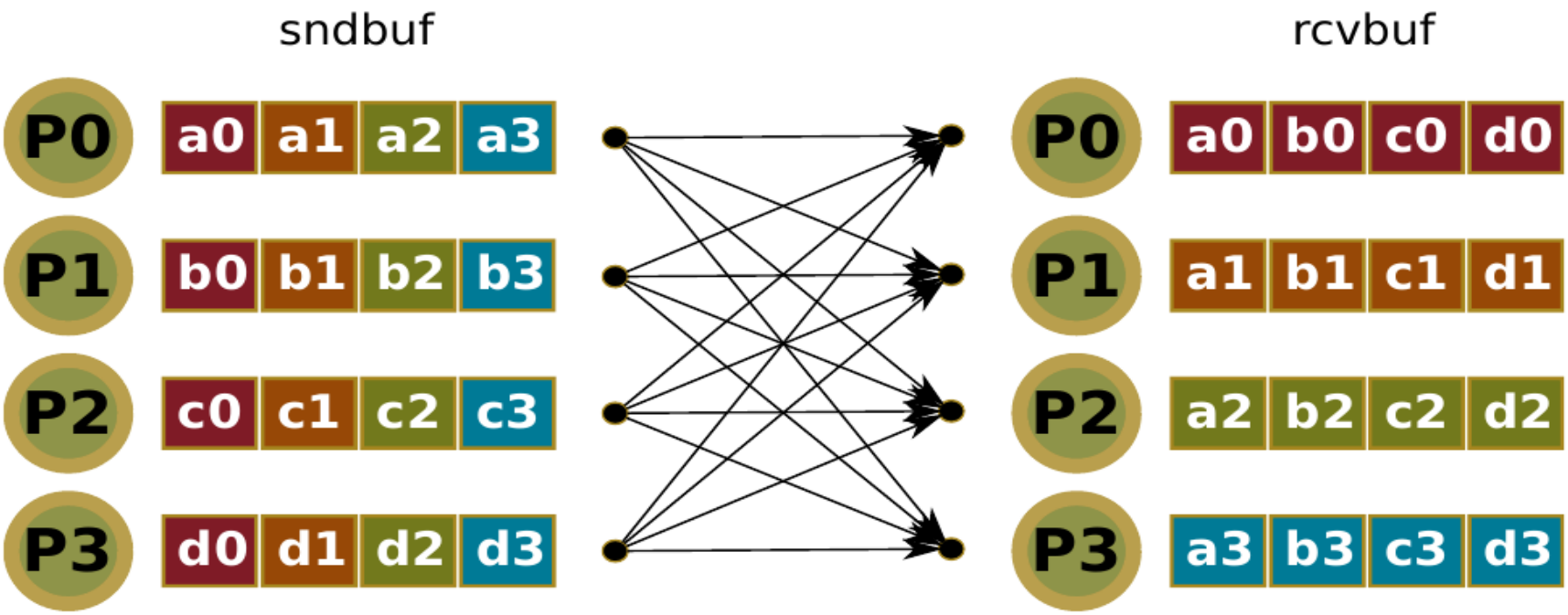


Fig. from M. Martinasso

Reductions

The reduction operation allows to:

- Collect data from each process.

- Reduce the data to a single value.

- Store the result on the root processes.

- Store the result on all processes.

- Overlap communication and computation.

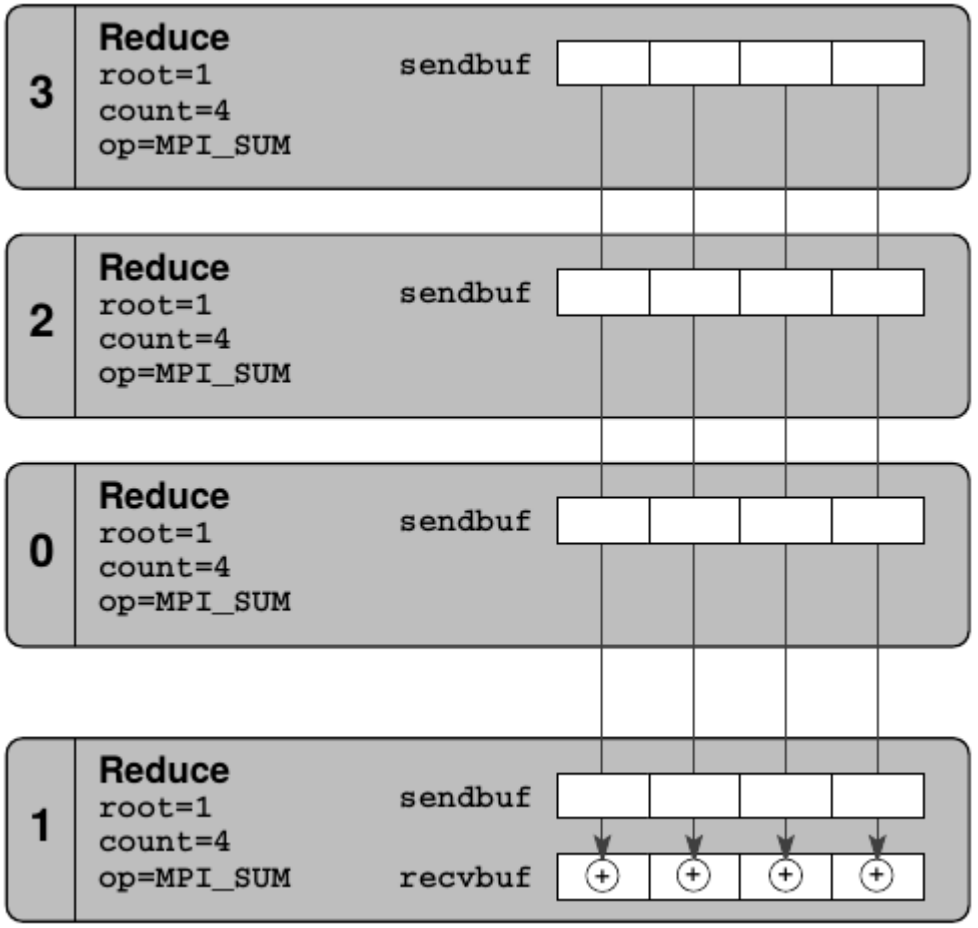
Reduction graphically

A reduction on an array of length count (a sum in this example) is performed by MPI_Reduce().

Every process must provide a send buffer.

The receive buffer argument is only used on the root process.

The local copy on root can be prevented by specifying MPI_IN_PLACE instead of a send buffer address.



Example: Reductions → MPI_SUM

- MPI, has mechanisms that make reductions much simpler (and in most cases) more efficient than looping over all ranks/collecting results.

- There are at the moment 12 predefined operators.

MPI op	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

MPI_Reduce - Reduces values on all processes within a group.

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

Parameters

sendbuf	Address of send buffer
recvbuf	Address of receive buffer
count	Number of elements in send buffer
dtype	Datatype of elements in send buffer
op	Reduction operation to be applied
root	Rank of root process
comm	Communicator

Specific call of MPI_Reduce()
→ **MPI_SUM**

Example – Reduction

1. go to OSM2019/day3/code_day3/MPI:

> cd OSM2019/day3/code_day3/MPI:

2. Have a look at the code

>vi 3b.MPI_reduce.cpp

3. compile by typing:

> make

4. run the code

>mpiexec -np 2 ./3b.MPI_reduce.exec (experimpent with # processes)

5. change the “root” from 0 to 1. What happens?

Example – Reduction

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

using namespace std;

int main(int argc, char *argv[]) {
    int rank, input, result;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    input=rank+1;

    MPI_Reduce(&input, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank==0){
      cout << "Rank 0 says: result is "<<  result << endl;
    }

    MPI_Finalize();
    return 0;
}
```