Introduction to High-Performance Computing

Session 05

Introduction to MPI

PARALLEL COMPUTING WITH MPI

Overview MPI

- Introduction to the Message Passing Interface
- Point-to-Point Communication
- Collective Communication
- Other and New Features of MPI
  - Derived Datatypes
  - Virtual Topologies
  - Process Creation and Management
  - One-sided Communication and Shared Memory
  - MPI and Threads
  - Parallel File I/O

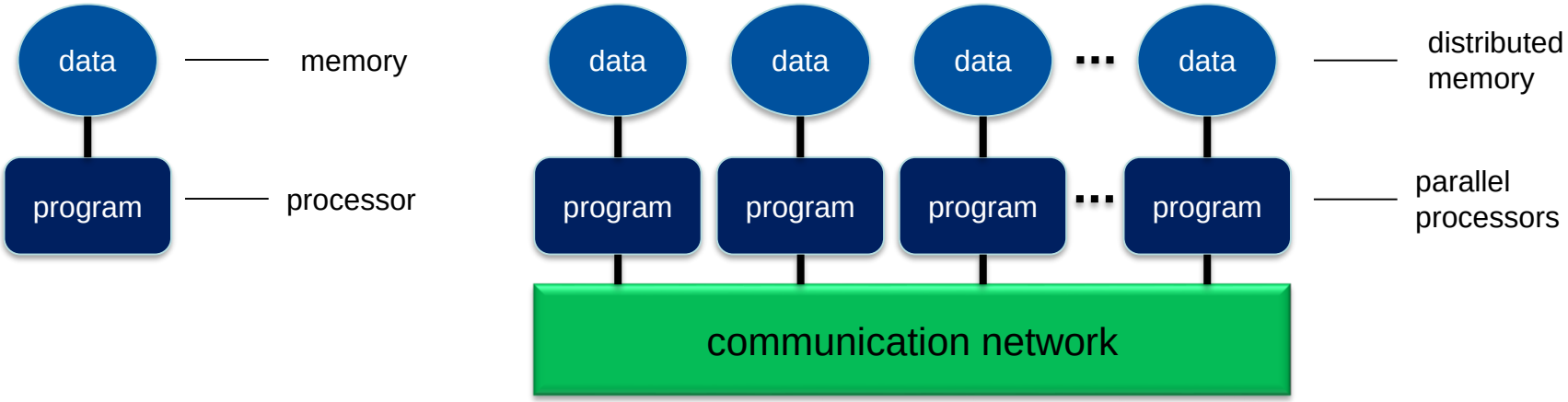
History of MPI

- MPI is a standard with the prime goals
  - to provide a message-passing interface
  - to provide source-code portability
  - to allow efficient implementations
- MPI exists for more than 20 years
  - MPI-1.0 was released in June, 1994
  - MPI-2.0 was released in July, 1997 and provided additional functionality
  - MPI-3.0 (current standard MPI-3.1) was released in October, 2012 and was developed for better platform and application support (in particular clusters of SMP nodes)

http://mpi-forum.org/docs/

A Message-Passing Interface

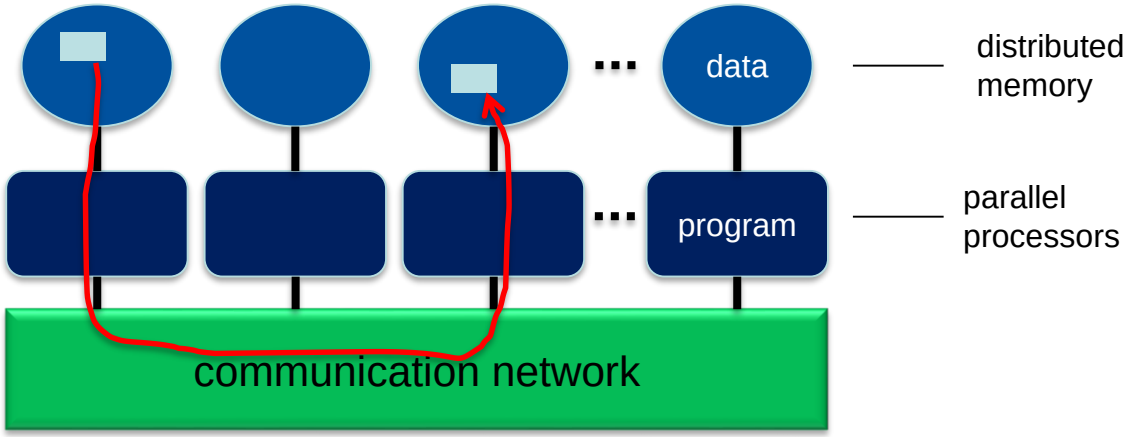
- sequential program vs. message-passing program



- message-passing programming paradigm:
  - each processor runs a (sub)program, typically the same (SPMD)
  - variables of subprograms have the same name but different (distributed) data
  - communication by special library routines ➔ message passing

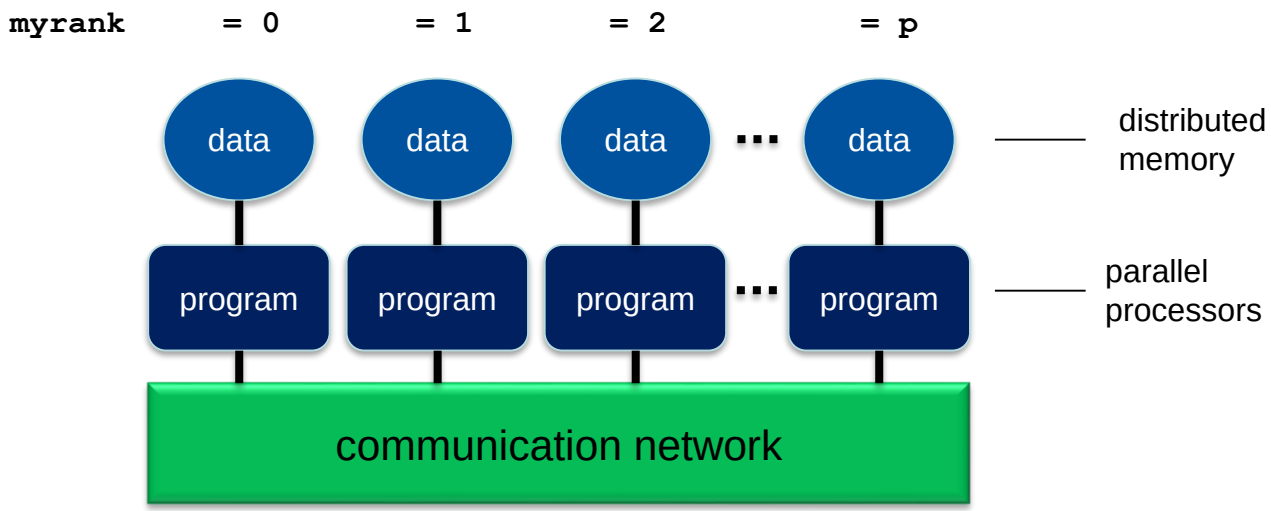
Message Passing

- messages are passed through the communication network
- messages require the following information:
  - sending and receiving process
  - data location
  - data type
  - data size



- in order to use the message-passing interface the program must be
  - connected to the MPI library (at compile time)
  - started with the MPI startup tool (mpirun or mpiexec)
  - at runtime MPI is initialized with special library calls (MPI_Init())

Process Identification



- processes in MPI are identified by their rank
  - determined by calling a library function
  - rank is used for addressing when sending messages
  - rank is used for making decisions, e.g. when distributing the data and work

Example: MPI_HelloWorld

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

using namespace std;

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);

   cout << "Hello world!" << endl;

   MPI_Finalize();

}
```

MPI Header and Module Files

MPI standard defines language bindings for C and Fortran

- C/C++: `#include <mpi.h>`
- Fortran: `include "mpif.h"`
  or `use mpi`
  or `use mpi_f08`
  - the use of the old style include-statement is strongly discouraged as no compile-time argument checking can be done
  - highly recommended is the use of mpi_f08

MPI with other Languages

- C++ is supported through the C bindings
  - special C++ bindings are no longer part of the standard although many MPI implementation may still support them
  - the C++ Boost library includes an MPI implementation
- Python
  - MPI is supported through the mpi4py package
- R
  - the package Rmpi provides MPI functionality
- Matlab
  - parallel computing uses MPI in the background
  - includes low-level functions for message passing

MPI Library Calls

- in general an MPI library call has the form

C/C++:
```
error = MPI_Xxxxx(parameter, …);
MPI_Xxxxx(parameter, …);
```

Fortran:
```
CALL MPI_Xxxxx(parameter, …, ierror)
```

  - in Fortran the use of ierror has changed with MPI-3.0:
    if (and only if!) you are using the module file mpi_f08, ierror is an optional argument. In any other case ierror cannot be omitted otherwise terrible unforeseen things may happen.
  - refer to the MPI-3.0 standard document to look up the definitions and argument list of available MPI functions
    http://www.mpi-forum.org/docs/

MPI_Init() and MPI_Finalize()

- MPI is initialized with
  - C/C++: `MPI_Init(&argc, &argv);`
  - Fortran: `CALL MPI_Init(ierror)`

  - must be the first MPI-routine that is called (few exceptions)
  - call as early as possible in your program
  - in C/C++ argv and argc are passed by reference (possibly cleans argv from unwanted MPI arguments)

- MPI is finalized with
  - C/C++: `MPI_Finalize();`
  - Fortran: `CALL MPI_Finalize(ierror)`

  - must be the last MPI-routine that is called (few exceptions)

Compiling an MPI Program

- programs are compiled using a wrapper command:

C: `$ mpicc [options] <source.c> -o <executable>`

C++: `$ mpicxx [options] <source.cpp> -o <executable>`

Fortran: `$ mpifort [options] <source.f90> -o <executable>`

  - uses the standard compiler (GCC, Intel) with some extra options

- example on CARL:

```
[abcd1234@carl ~]$ # module load gompi/5.2.01 # if you want GCC/OpenMPI
[abcd1234@carl ~]$ module load intel/2016b
[abcd1234@carl ~]$ mpicxx MPI_HelloWorld.cpp -o MPI_HelloWorld
```

Running an MPI Program

- programs are executed using the MPI startup tool

  $ mpirun -np <N> [options] <executable>

- example on CARL:
  - note: do not normally run programs on the head nodes

```
[abcd1234@hpc1002 ~]$ mpirun -np 4 MPI_HelloWorld
Hello world!
Hello world!
Hello world!
Hello world!
```

Running an MPI Program

- running an MPI program on the compute nodes
  - using sbatch with a job script (see next slide)
  - using srun interactively

```
$ export I_MPI_PMI_LIBRARY=/cm/shared/apps/slurm/current/lib64/libpmi.so
$ srun -p carl.p -n 4 MPI_HelloWorld
Hello world!
Hello world!
Hello world!
Hello world!
```

  - only starts when resources are available
  - srun can be used a replacement for mpirun (recommended) but it requires additional setting of environment variable for Intel MPI

Job Script for an MPI Program

- minimal example batch script

```
#!/bin/bash

#### SLURM settings
#SBATCH --partition=carl.p
#SBATCH --job-name=MPI_HelloWorld
#SBATCH --ntasks=4

module load intel/2016b

export I_MPI_PMI_LIBRARY=/cm/shared/apps/slurm/current/lib64/libpmi.so
srun MPI_HelloWorld
```

  - note that the executable will only work with the MPI used for compilation
  - srun and mpirun are SLURM aware and know the number of processes to start

Setting the Number of Processes

- typically the number of processes is set by requesting the resources
  - can be changed with the -n or -np option to srun or mpirun
- number of tasks is the number of processes spawned
- number of tasks can be requested in different ways
  - simple: --ntasks=<number> or -n <number>
  - restricted: --nodes=<min>-<max> and --ntasks=<number>
  - control: --nodes=<min>-<max> and --tasks-per-node=<number>
  - user can decide how the job can be distributed

EXERCISE

MPI_HelloWorld v2.0

- modify the MPI_HelloWorld example so that
  - only one process (the root process) prints „Hello World!“
  - all processes print a message
    „I am process %i of %n processes running on %host“
  - try out the different SLURM-options and see how the process distribution is changed

- look up how to use the following MPI library calls
  - MPI_Comm_rank(…)
  - MPI_Comm_size(…)
  - MPI_Get_processor_name(…)

HelloWorld v2.0

- see code on Stud.IP
- solution requires the following MPI library calls

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

int MPI_Comm_size(MPI_Comm comm, int *size)

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

  - first two functions use argument of type **`MPI_Comm`**

MPI Communicators

- all MPI processes (subprograms) are combined in the communicator MPI_COMM_WORLD
  - MPI_COMM_WORLD is a handle predefined in the header files
  - each process in a communicator has its own rank starting from 0 until (size-1)
  - the size of a communicator and the rank of a process within the communicator can be determined with special library calls
  - it is possible to define your own communicators (e.g. for a subset of processes) and handles

MPI_Comm_size() and MPI_Comm_rank()

- to determine the size of a communicator use

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

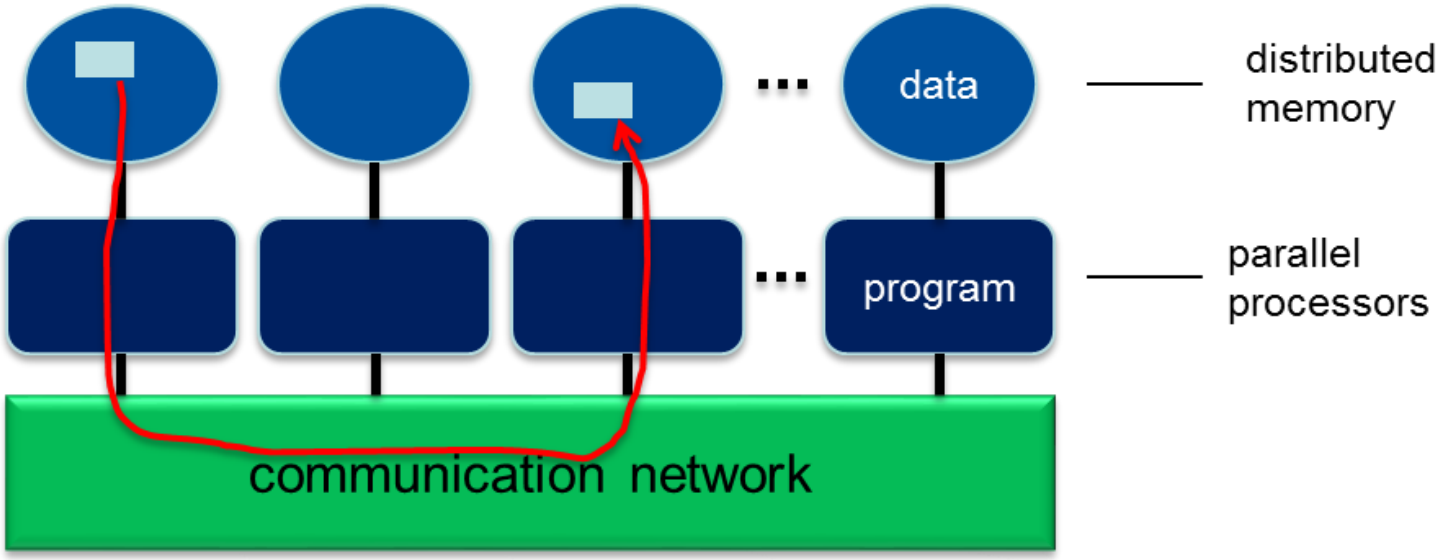
- to determine the rank of a process within a communicator use

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

  - note that **`size`** is the same on every process whereas **`rank`** is different

MPI Point-to-Point Communication

MPI Point-to-Point Communication



communication by sending messages within an MPI communicator

Sending and Receiving Messages

- the MPI library provides functions to send and receive messages:
  - sending: MPI_Send(…)
  - receiving: MPI_Recv(…)

  - any message sent must be received, otherwise → deadlock
  - function prototypes (here C/C++, Fortran is analogous)

```
int MPI_Send(const void *data, int count, MPI_Datatype datatype,
             int destination, int tag, MPI_Comm comm)

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

MPI Send/Recv Data

- the data send or received by a message is passed as a void pointer
  - a void pointer can be cast on a pointer of any data type
  - MPI does not care about the data type, the message is just a collection of bits (continueous in memory)
  - variables require a reference, arrays are already a pointer
- the integer count argument gives the number of data values
- the MPI_Datatype argument gives the data type and allows MPI to interpret the data correctly
  - using the wrong data type can produce interesting errors

MPI Data Types

- MPI needs to know the type of data that is send
- predefined handles are provided for standard data types, e.g.:
  - **`MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR,`** …
- you can also define handles for your own data types

Other MPI Send/Recv Arguments

- source and destination give the rank of the sending and receiving process
- the tag is an integer identifier for each message sent
  - useful if more than one message is sent at the same time from one source to the same destination
- the MPI_Status object contains information about the received message
  - required for non-blocking communication
- MPI::Comm:Recv allows the use of wildcards
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

MPI Send and Receive Example

```
...
int number;
if (rank == 0) {
    number = 42;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    cout << "Process 1 received number" << number << endl;
}
...
```

Sending and Receiving Messages

- message can be sent in different ways
  - synchronous vs. asynchronous:
    sender receives a confirmation receiving of the message is initiated
  - unbuffered vs. buffered:
    the message can be buffered so the sender can continue using the sent variable, requires additional memory
  - blocking vs. non-blocking:
    send or receive functions return immediately allowing to overlap communication and computation

for details refer to the MPI-3.0 standard

Communication Modes

- the standard function MPI_Send and MPI_Recv are blocking operations
  - MPI_Send may use a buffer and thus can return before the message was received
  - the use of the buffer depends on the MPI implementation and situation
- for optimal performance you can control the communication mode by using
  - MPI_Isend/IRecv for non-blocking communication
  - MPI_Bsend for buffered sending
  - …

Deadlocks

- every MPI_Send() must be matched by a corresponding MPI_Recv() and vice versa
  - otherwise the program hangs waiting forever for a communication to finish → deadlock

typical pitfalls:

- every process is sending data to a neighbour process
  - only one process must send data before receiving
  - use non-blocking send or receive
  - use MPI_Sendrecv(…)
- a condition prevents one or more processes to initiate communication

Deadlock Example

consider two processes:

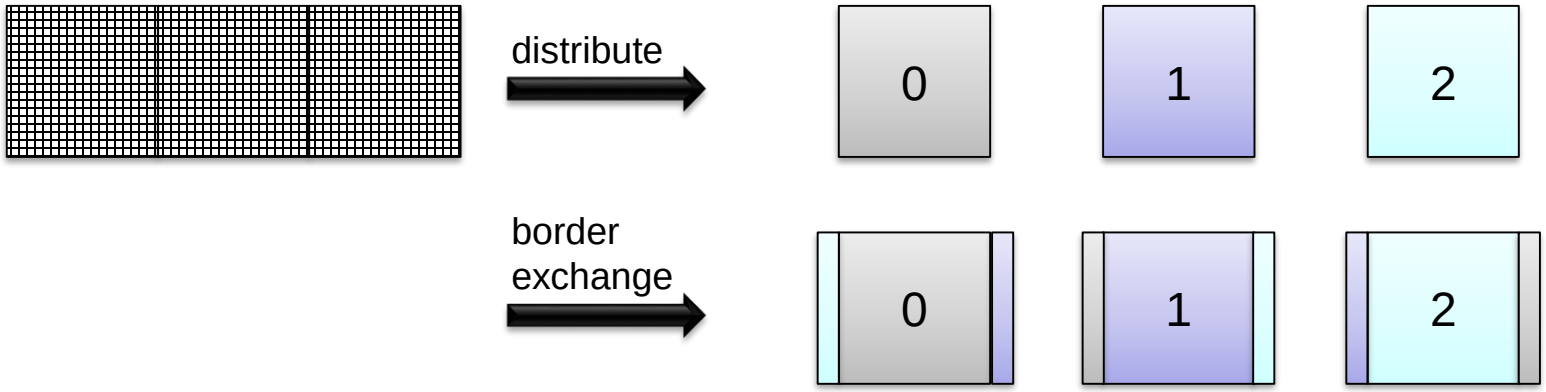
```
...
int number = 42 + rank;
int other  = 1 - rank;
MPI_Recv(&number, 1, MPI_INT, other, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
MPI_Send(&number, 1, MPI_INT, other, 0, MPI_COMM_WORLD);
cout << "Received number" << number << endl;
}
...
```

MPI_Sendrecv

- in some situations MPI_Sendrecv can be used for effective and dead-lock free P2P communication
  - syntax

```
int MPI_Sendrecv(const void *sbuf, int scount, MPI_Datatype stype, int dest, int stag,
                 void *rbuf, int rcount, MPI_Datatype rtype, int source, int rtag,
                 MPI_Comm comm, MPI_Status *status)
```

  - note that sendbuf and recvbuf have to be different variables
  - typical situation is exchange of borders



EXERCISE

MPI_PingPong

- an MPI_PingPong program to run with two processes doing the following:
  - initialize a counter
  - one process increments the counter and sends it to the other
  - the other process receives the message and then increments the counter and sends it back
  - repeat until n messages have been sent

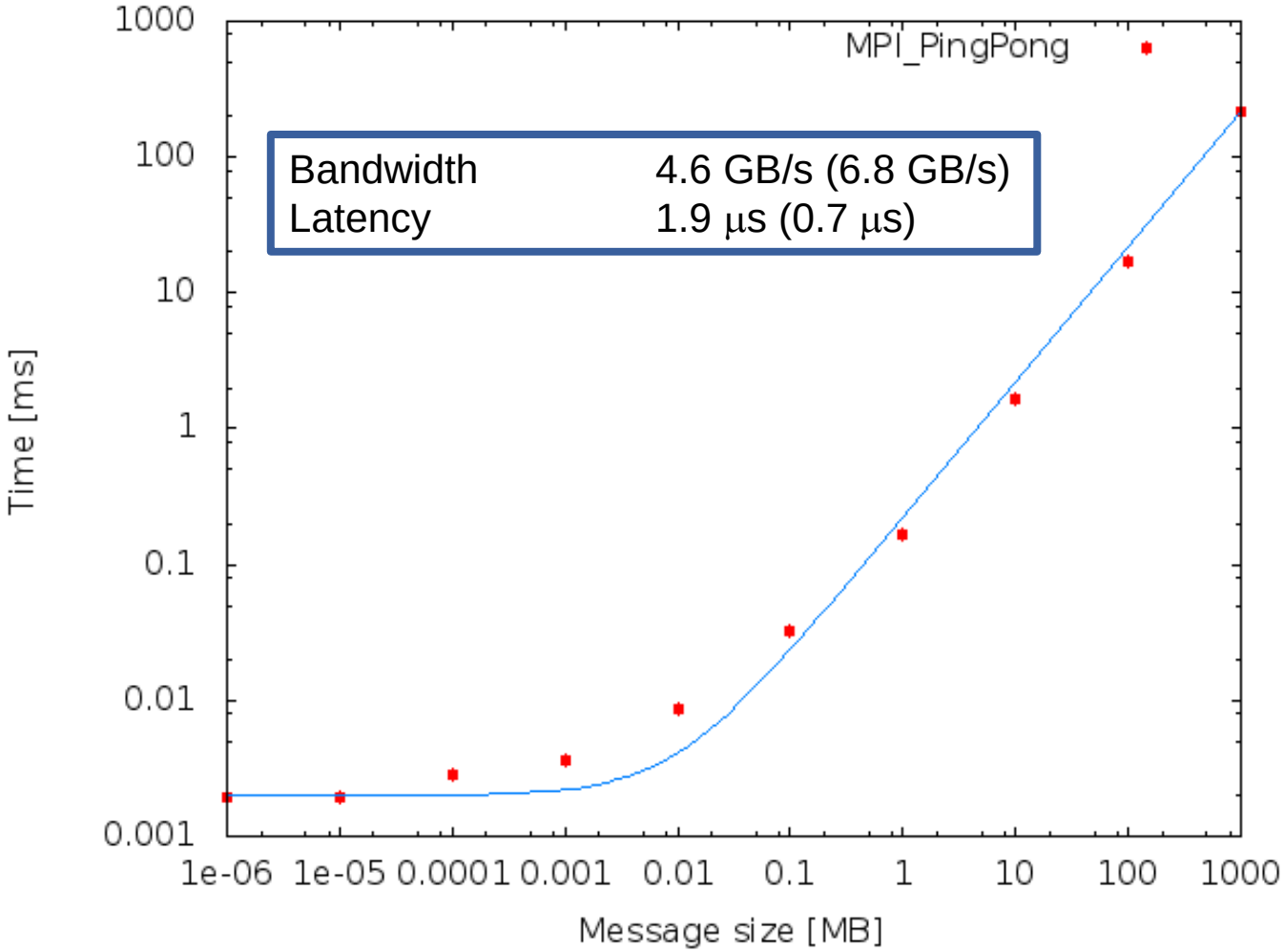
MPI_RingSend

- Complete the MPI_RingSend program to run with exactly 12 processes:
  - each process, beginning with root (rank==0), should send text to the right neighbor and receive text from the left neighbor
  - after receiving but before sending text each process should modify text as follows: `text[rank] -= rank;`
  - the ring is terminated after one round when text reaches root again (the final output should tell you if you code is correct)
  - note: the left/right neighbor for rank 0/(size-1) is (size-1)/0

MPI_Summation

- Complete the MPI_Summation program so that the root process calculates the sum of all my_val values
  - my_val is rank+1 so the sum is n*(n+1)/2
  - only use MPI_Send and MPI_Recv (or variants with different communications modes)

MPI PingPong



MPI Collective Communication

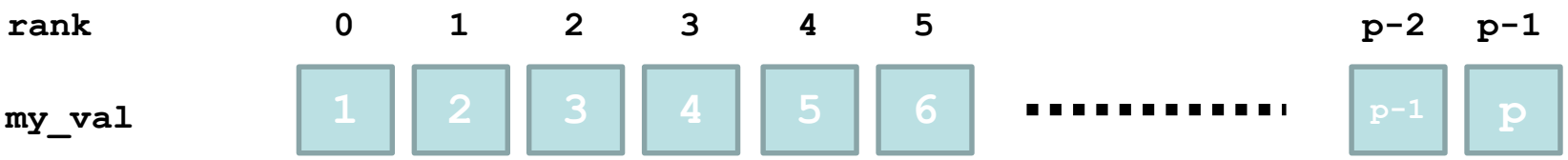
Collective Communication

- so far we have looked at point-to-point communication
- MPI allows also knows collective communications
  - one-to-all
  - all-to-one
  - all-to-all

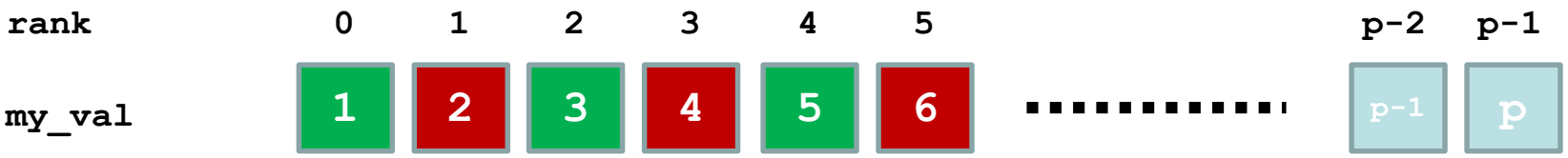
  } communication

- example: calculate the sum of the elements of an array

Calculate Sum in Parallel

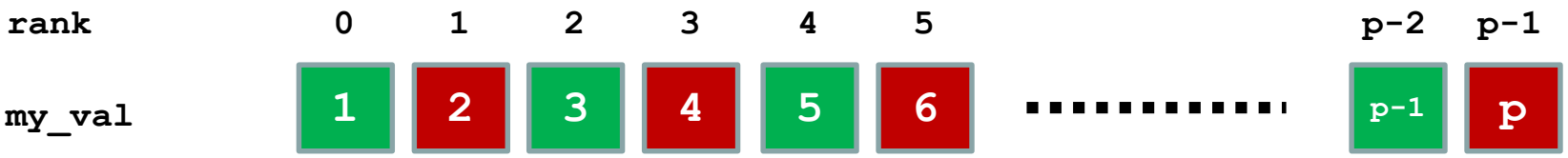


Calculate Sum in Parallel



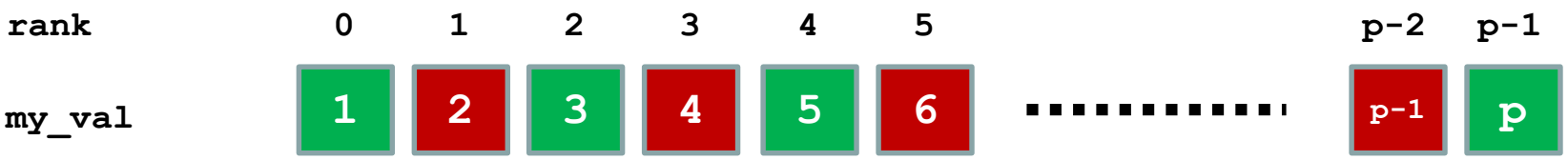
- even processes become receiver, odd process sender

Calculate Sum in Parallel



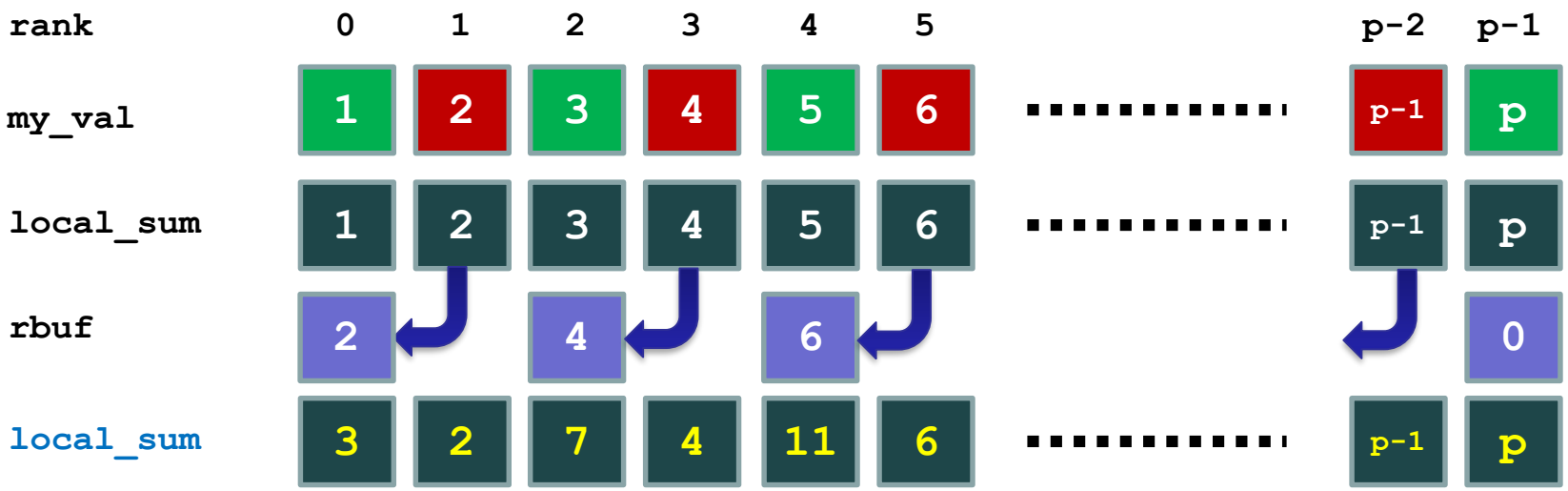
- even processes become receiver, odd process sender
- the last process can be
  - a sender (size `p` even) with matching receiver

Calculate Sum in Parallel



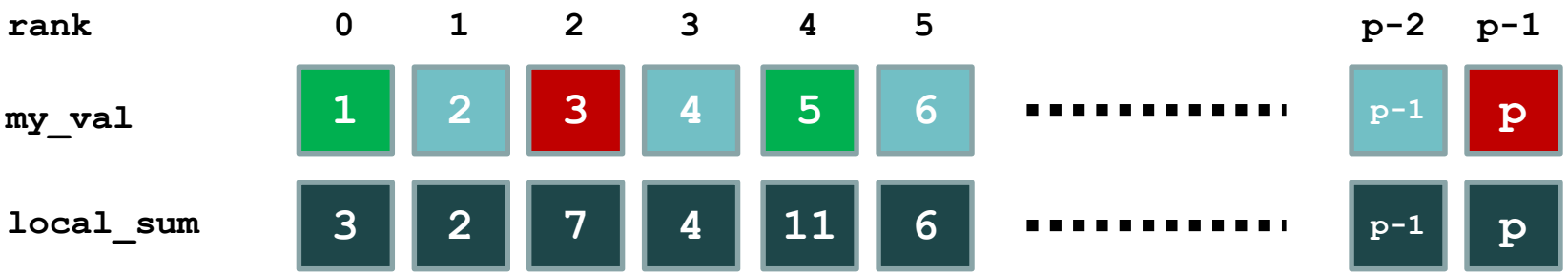
- even processes become receiver, odd process sender
- the last process can be
  - a sender (size **p** even) with matching receiver
  - a receiver (size **p** odd) with no matching sender (**from >= p**)

Calculate Sum in Parallel



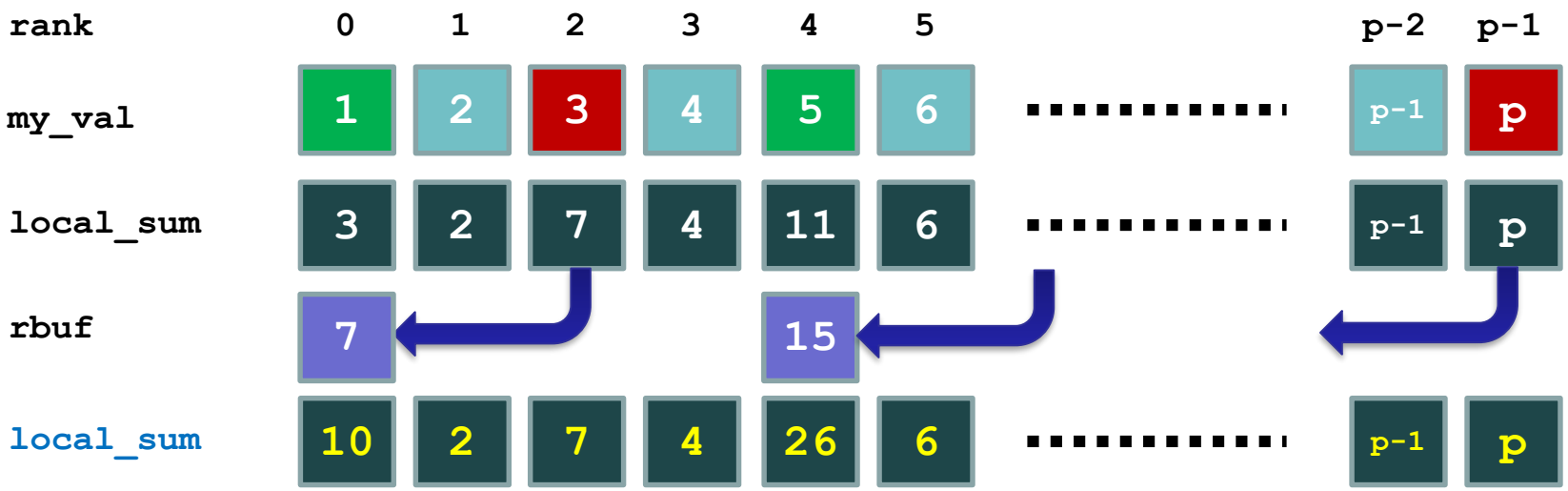
- every process copies **`my_val`** to **`local_sum`**
- every sender sends **`local_sum`** to the left to receiver's **`rbuf`**
- every receiver adds **`rbuf`** to its **`local_sum`**
  - rightmost receiver may not receive value (and adds `0`)
  - all sender also add `0`

Calculate Sum in Parallel



- every sender becomes inactive (value was added to sum)
- every other receiver becomes a sender

Calculate Sum in Parallel



- previous steps of sending, receiving and adding to local sum are repeated
- after each send more processes become inactive
- final result is obtained on root when all other processes are inactive

Collective Communication

- so far we have looked at point-to-point communication
- MPI allows knows
  - one-to-all
  - all-to-one
  - all-to-all

  } communication

- example: calculate the sum of the elements of an array
- MPI collective communication is very efficient due to tree-based communication
- collective communication can still be very expensive, in particular all-to-all

Collective Communication

- a selection of collective MPI communications:
  - MPI_Bcast(…) sending data from one process to all others
  - MPI_Scatter(…) distributing an array of data from one to all
  - MPI_Gather(…) collecting an array of data from all to one
  - MPI_Reduce(…) reduction operation defined by a handle, e.g. MPI_SUM
  - MPI_Barrier(…) used to synchronize all processes
  - …
- some also have all-to-all variant, e.g. MPI_Allreduce
- since MPI-3.0 also non-blocking calls

MPI_Reduce

- MPI function to reduce values from all processes
  - syntax

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

  - the reduce operation is defined by op
  - can be selected from pre-defined list or user-defined
  - reduce operation is applied for every element in sendbuf separately
  - result is only obtained on root (unless MPI_Allreduce is used)
  - see example MPI_Reduce_Sum.cpp

MPI_Reduce Operators

- pre-defined operators for MPI_Reduce

MPI_MAX	**maximum**
MPI_MIN	**minimum**
MPI_SUM	**sum**
MPI_PROD	**product**
MPI_LAND	**logical and**
MPI_BAND	**bit-wise and**
MPI_LOR	**logical or**
MPI_BOR	**bit-wise or**
MPI_LXOR	**logical xor**
MPI_BXOR	**bit-wise xor**
MPI_MAXLOC	**max value and location**
MPI_MINLOC	**min value and location**

Synchronization of MPI Processes

- in MPI this can be achieved with a barrier
  - syntax
    `int MPI_Barrier(MPI_Comm comm)`
  - every process must reach barrier call before proceeding
- barriers are normally not needed in MPI
  - synchronization is done by data communication automatically
  - maybe used for debugging purposes (make sure all processes write debug message in order)
  - for profiling to measure communication times and/or load imbalances

MPI_Bcast

- one process sending data to all other processes is achieved with a broadcast
  - syntax

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

  - typical example

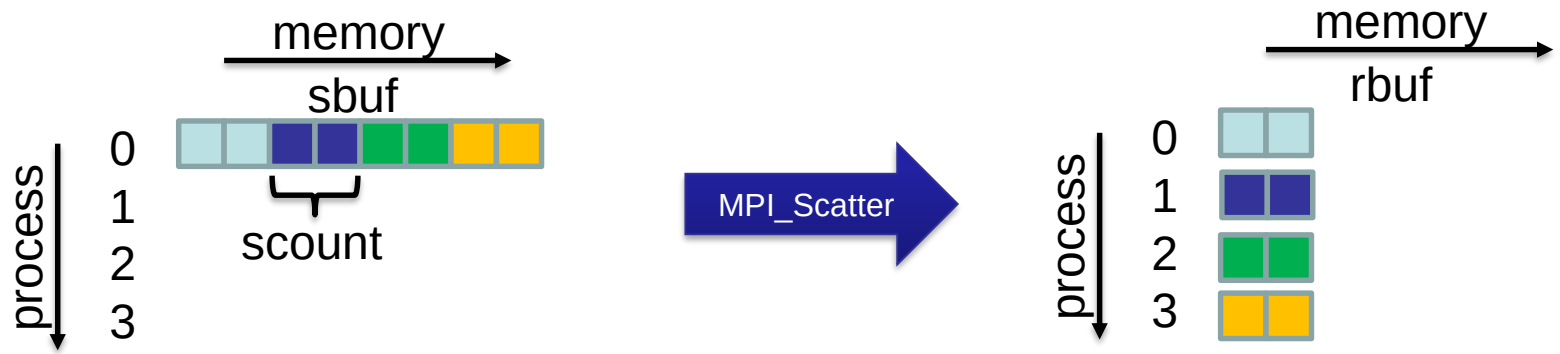
```
if (rank==root) cin >> value;
MPI_Bcast(&value, 1, MPI_INT, root,
          MPI_COMM_WORLD);
```

Distributing Arrays

- MPI provides a function to scatter arrays across processes
  - syntax:

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype,
                void *rbuf, int rcount, MPI_Datatype rtype,
                int root, MPI_Comm comm)
```

  - sends a continuous number of elements from an array on the root process to every other process including the root process



MPI_Scatter

- the value of **`scount`** is given by **`sizeof(sbuf)/size`**
- what happens if division is not even?
  - if **`scount*size < sizeof(sbuf)`** only part of the array is scattered → incorrect result
  - with **`(scount+1)*size > sizeof(sbuf)`** out-of-bounds elements are scattered → anything can happen
  - possible solution is padding of global vector but then one process has (much) less work to do → load imbalancing

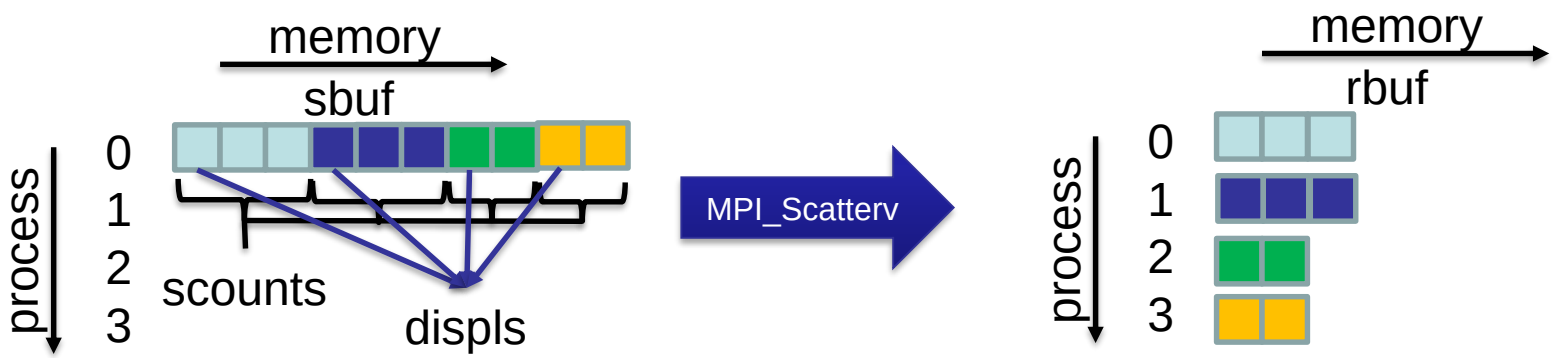
- better solution
  - scatter global vector so that **`scount`** differs by 1 at most for all processes
  - can be achieved with **`MPI_Send/Recv`** or **`MPI_Scatterv`**

MPI_Scatterv

- MPI_Scatterv gives additional control for data distribution
  - syntax:

```
int MPI_Scatterv(void *sbuf, int *scounts, int *displs
                 MPI_Datatype stype,
                 void *rbuf, int rcount, MPI_Datatype rtype,
                 int root, MPI_Comm comm)
```

  - arrays **`scounts`** and **`displs`** to define data distribution

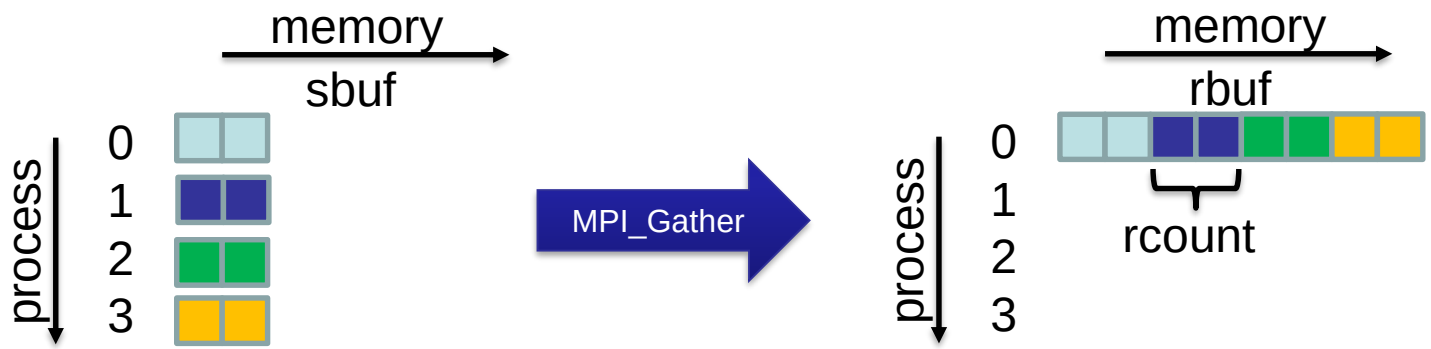


Gathering Data

- the opposite of MPI_Scatter is called MPI_Gather
  - syntax:

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype,
               void *rbuf, int rcount, MPI_Datatype rtype,
               int root, MPI_Comm comm)
```

  - each process (including root) sends a block of data to the root process where all data blocks are collected in continuous array



Gathering Data

- variants of MPI_Gather
  - MPI_Allgather
  - MPI_Gatherv
  - MPI_AllGatherv