An Introduction to Message Passing and Parallel Programming with MPI

Alireza Ghasemi and Georg Hager

Erlangen National High Performance Computing Center

Volker Weinberg

Leibniz Supercomputing Centre (LRZ)

A collaborative course of NHR@FAU and LRZ Garching

Introduction to MPI: Agenda

- **Message-passing paradigm**
- **Pint-to-point communication: Blocking**
- **Point-to-point communication: Nonblocking**
- **Helper functions**
- **Collectives**
- **Datatypes**

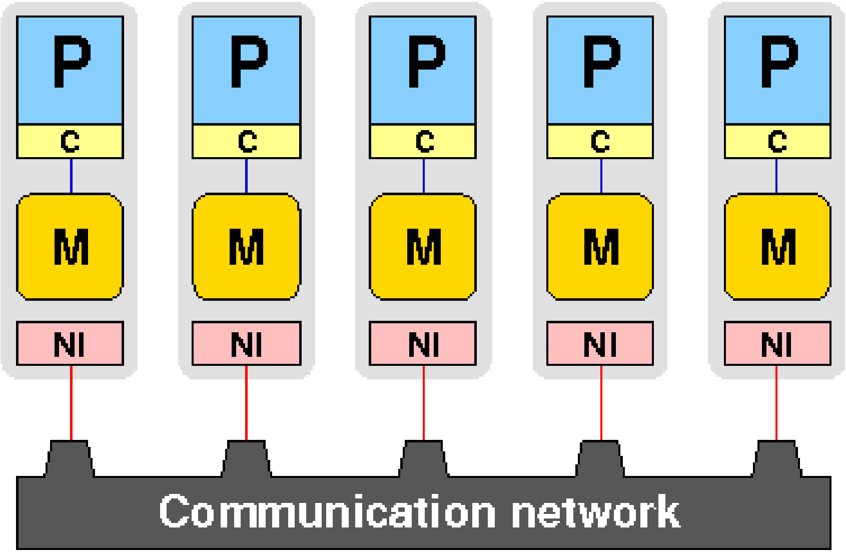
Message-Passing Paradigm

The message passing paradigm

Distributed-memory architecture:

Each process(or) can only access its dedicated address space.

No global shared address space



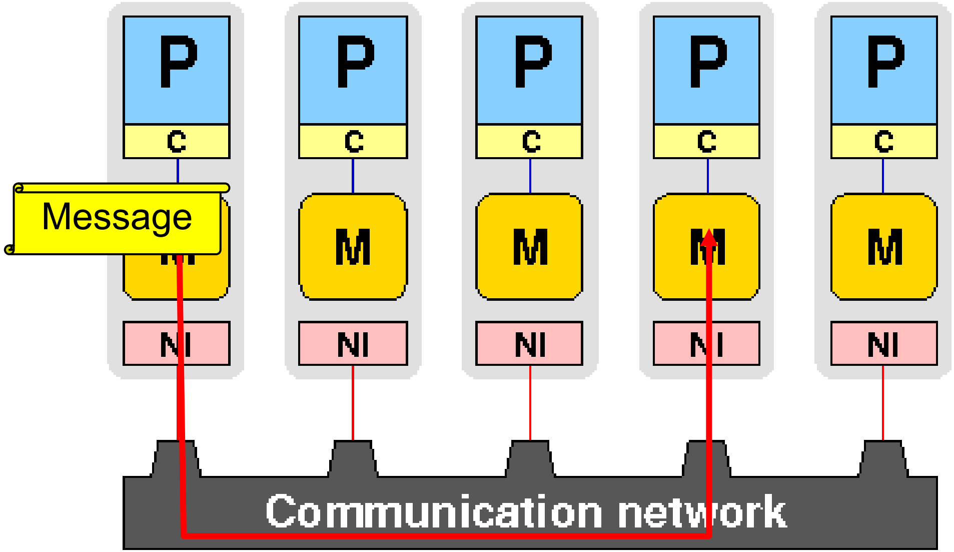
Data exchange and communication between processes is done by explicitly passing messages through a communication network

The message passing paradigm

Distributed-memory architecture:

Each process(or) can only access its dedicated address space.

No global shared address space



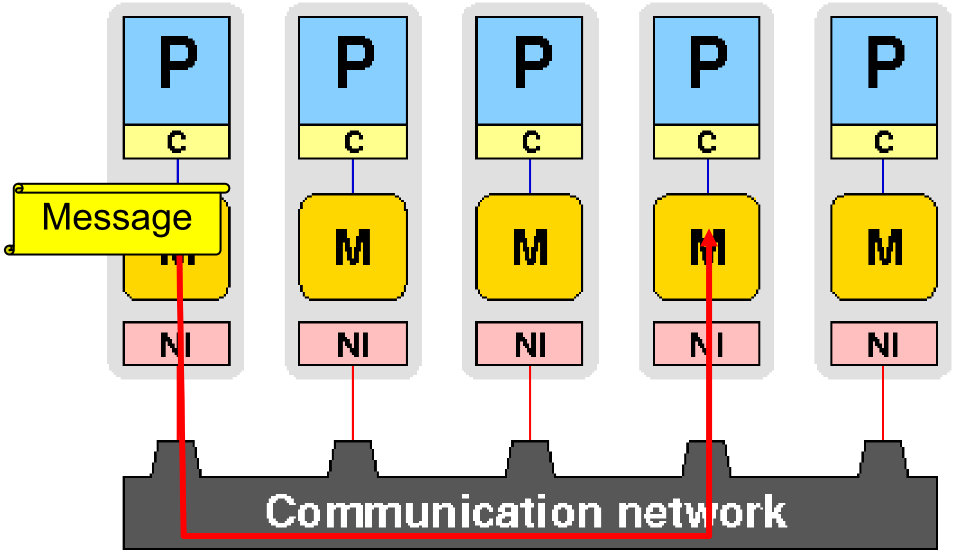
Data exchange and communication between processes is done by explicitly passing messages through a communication network

The message passing paradigm

Distributed-memory architecture:

Each process(or) can only access its dedicated address space.

No global shared address space



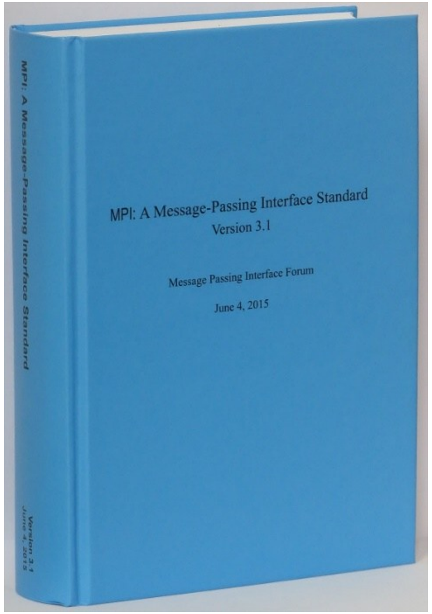
Data exchange and communication between processes is done by explicitly passing messages through a communication network

Message passing library:

- Should be flexible, efficient and portable
- Hide communication hardware and software layers from application developer

The MPI standard

- MPI forum – defines MPI standard / library subroutine interfaces
- Latest standard in use: MPI 3.1 (2015), 868 pages
  - MPI-4.1 was approved by the MPI Forum on 02.11.2023
- Members (approx. 60) of MPI standard forum
  - Application developers
  - Research institutes & computing centers
  - Manufacturers of supercomputers & software designers

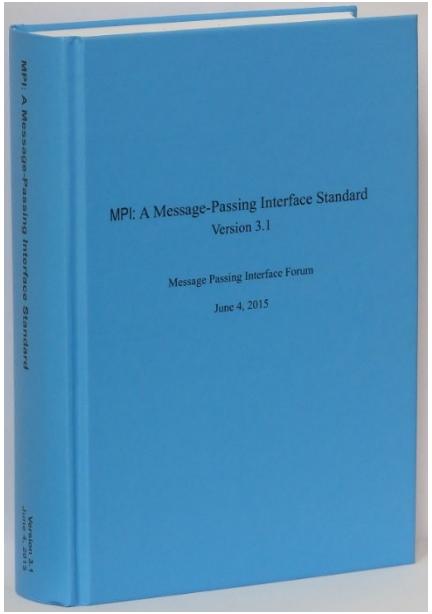


The MPI standard

- MPI forum – defines MPI standard / library subroutine interfaces

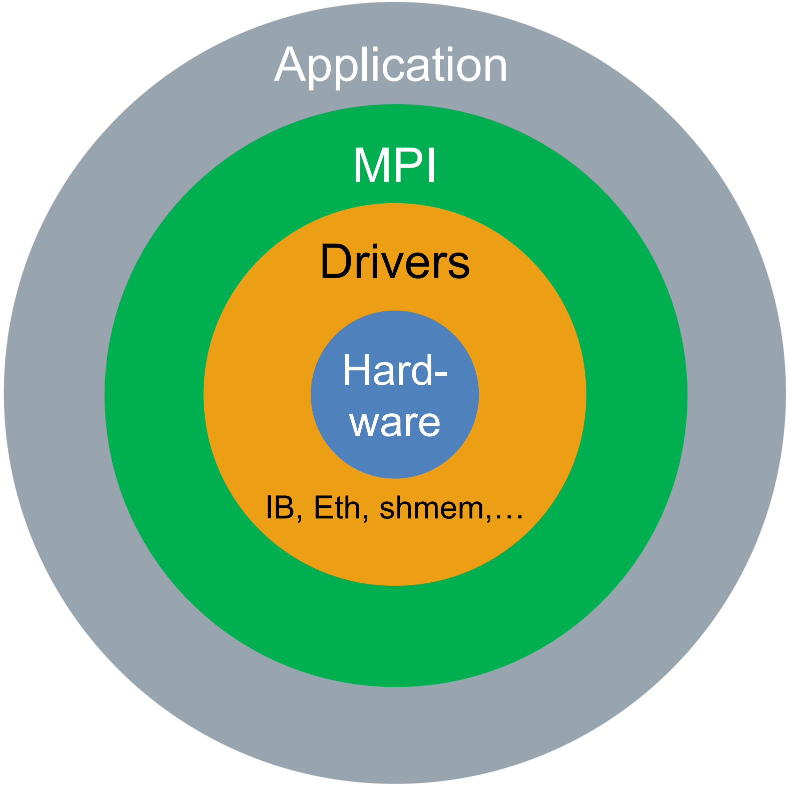
- Latest standard in use: MPI 3.1 (2015), 868 pages
  - MPI-4.1 was approved by the MPI Forum on 02.11.2023

- Members (approx. 60) of MPI standard forum
  - Application developers
  - Research institutes & computing centers
  - Manufacturers of supercomputers & software designers
- Successful free implementations (MPICH, mvapich, OpenMPI) and vendor libraries (Intel, Cray, HP,…)
- Documents: http://www.mpi-forum.org/

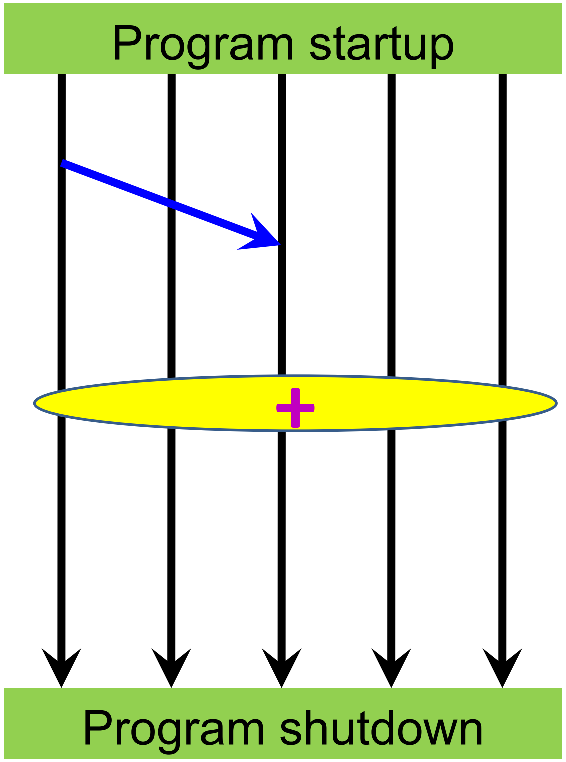


MPI goals and scope

- Portability is main goal: architecture- and hardware-independent code
- Fortran and C interfaces (C++ deprecated)
- Features for supporting parallel libraries
- Support for heterogeneous environments (e.g., clusters with compute nodes of different architectures)

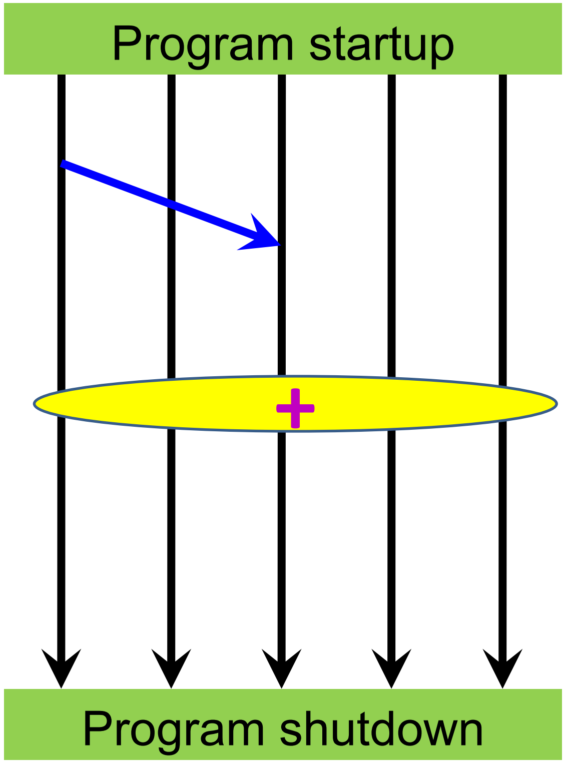


Parallel execution in MPI



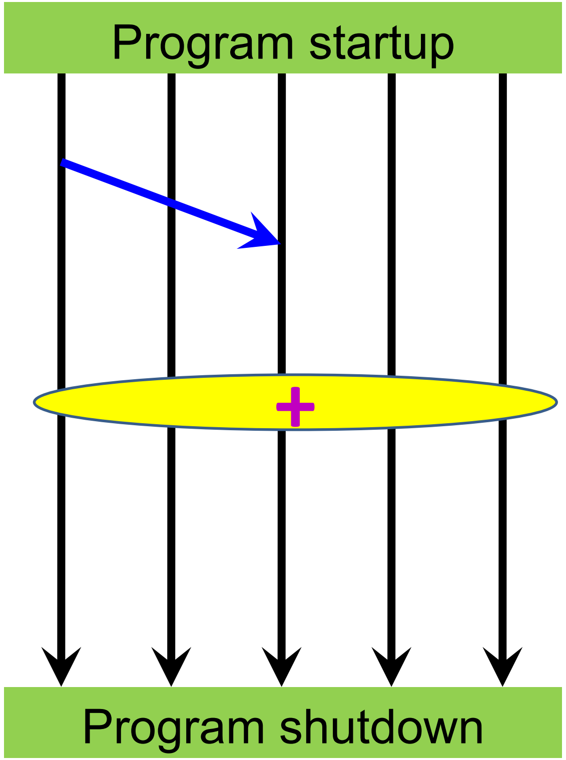
- Processes run throughout program execution

Parallel execution in MPI



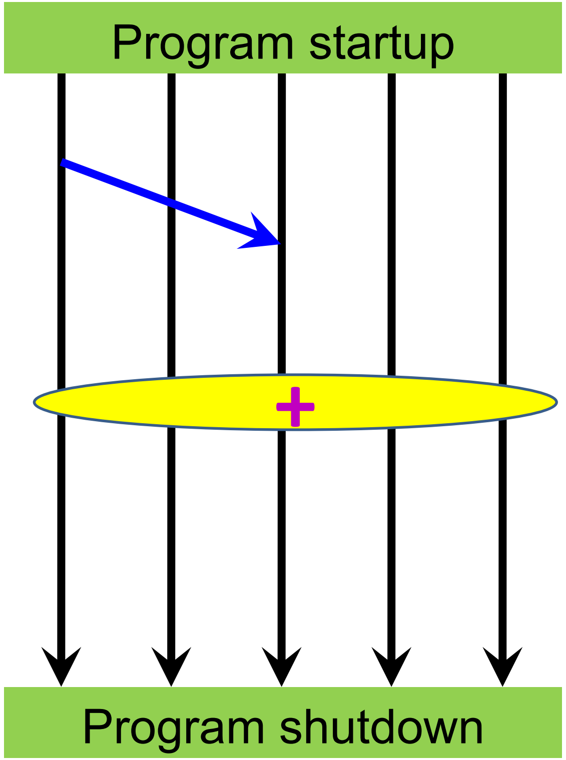
- Processes run throughout program execution
- MPI startup mechanism:
  - launches tasks/processes
    - think of executing multiple copies of a program
  - establishes communication context ("communicator")

Parallel execution in MPI



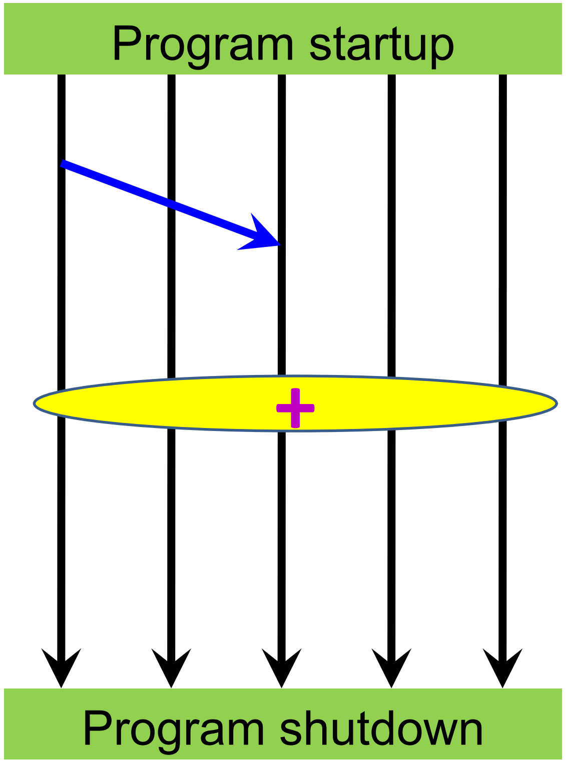
- Processes run throughout program execution
- MPI startup mechanism:
  - launches tasks/processes
    - think of executing multiple copies of a program
  - establishes communication context ("communicator")
- MPI Point-to-point communication:
  - between pairs of tasks/processes

Parallel execution in MPI



- Processes run throughout program execution
- MPI startup mechanism:
  - launches tasks/processes
    - think of executing multiple copies of a program
  - establishes communication context ("communicator")
- MPI Point-to-point communication:
  - between pairs of tasks/processes
- MPI Collective communication:
  - between all processes or a subgroup
  - barrier, reductions, scatter/gather

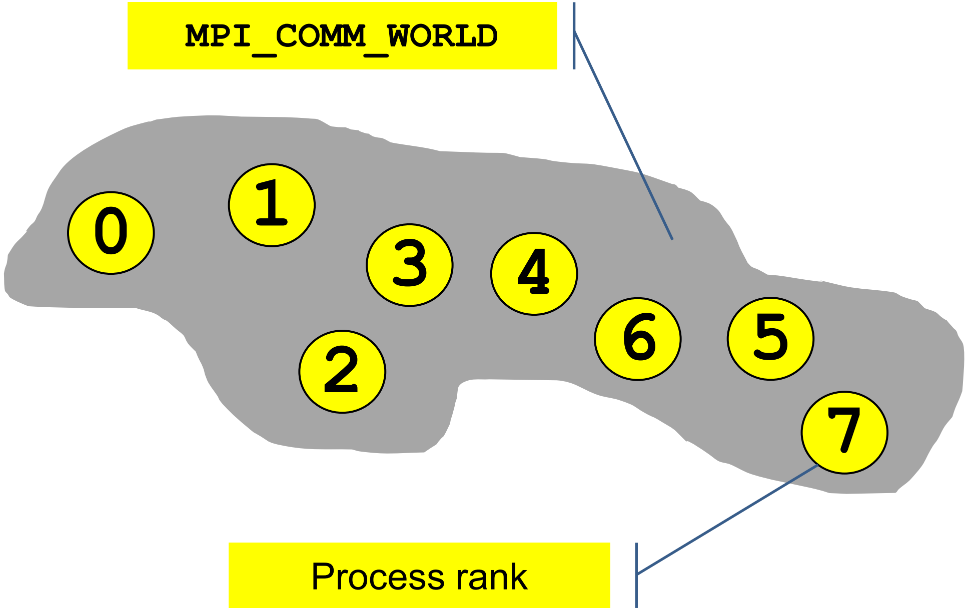
Parallel execution in MPI



- Processes run throughout program execution
- MPI startup mechanism:
  - launches tasks/processes
    - think of executing multiple copies of a program
  - establishes communication context ("communicator")
- MPI Point-to-point communication:
  - between pairs of tasks/processes
- MPI Collective communication:
  - between all processes or a subgroup
  - barrier, reductions, scatter/gather
- Clean shutdown by MPI

World communicator and rank

- Entities must be in a group/community to be able to communicate.
- Communicator is a handle
- `MPI_Init():`
  - MPI_COMM_WORLD
  - all processes
- **MPI_COMM_WORLD**
  - Fortran and C[++]



Initialization and finalization

- Startup command of an MPI application is implementation dependent

Initialization and finalization

- Startup command of an MPI application is implementation dependent

- First call in an MPI program: initialization of parallel machine

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

Initialization and finalization

- Startup command of an MPI application is implementation dependent

- First call in an MPI program: initialization of parallel machine

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

- Last call: clean shutdown of parallel machine

```
int MPI_Finalize();
```

  Only “master” process is guaranteed to continue after finalize

Initialization and finalization

- Startup command of an MPI application is implementation dependent

- First call in an MPI program: initialization of parallel machine

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

- Last call: clean shutdown of parallel machine

```
int MPI_Finalize();
```

  Only “master” process is guaranteed to continue after finalize
- Stdout/stderr of each MPI process
  - usually redirected to console where program was started
  - many options possible, depending on implementation

Communicator and rank

- Communicator defines a set of processes (`MPI_COMM_WORLD`: all)

Communicator and rank

- Communicator defines a set of processes (`MPI_COMM_WORLD`: all)

- rank: an integer identifying each process within a communicator
  - Obtain rank:
    ```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ```
  - `rank =` 0,1,2,…, (number of processes in communicator – 1)
  - Not unique: one process may have distinct ranks in different communicators

Communicator and rank

- Communicator defines a set of processes (`MPI_COMM_WORLD`: all)

- rank: an integer identifying each process within a communicator
  - Obtain rank:
    ```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ```
  - `rank =` 0,1,2,…, (number of processes in communicator – 1)
  - Not unique: one process may have distinct ranks in different communicators

- Obtain number of processes in communicator:
  ```
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  ```

MPI “Hello World!” in C

```c
#include <mpi.h>

int main(char 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();
}
```

MPI “Hello World!” in C

```
#include <mpi.h>

int main(char 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();
}
```

Never forget that these are pointers to the original variables!

MPI “Hello World!” in C

```c
#include <mpi.h>

int main(char 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();
}
```

Never forget that these are pointers to the original variables!

Communicator required for (almost) all MPI calls

MPI “Hello World!” in Fortran

```
program hello
  use mpi
  implicit none
  integer:: rank, size, ierr
  !include "mpif.h"
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  write(*,'(2(a,i))') &
    "Hello World! I am ",rank," of ",size

  call MPI_FINALIZE(ierr)
end program hello
```

MPI “Hello World!” in Fortran

```
program hello
  use mpi
  implicit none
  integer:: rank, size, ierr
  !include "mpif.h"
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  write(*,'(2(a,i))') &
    "Hello World! I am ",rank," of ",size

  call MPI_FINALIZE(ierr)
end program hello
```

By default, Fortran arguments are passed by reference!

MPI "Hello World!" in Fortran

```
program hello
  use mpi
  implicit none
  integer:: rank, size, ierr
  !include "mpif.h"
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  write(*,'(2(a,i))') &
    "Hello World! I am ",rank," of ",size

  call MPI_FINALIZE(ierr)
end program hello
```

By default, Fortran arguments are passed by reference!

Communicator required for (almost) all MPI calls

Compiling and running the code

- Compiling/linking
  - Headers and libs must be found by compiler
  - Most implementations provide wrapper scripts, e.g.,
    - **`mpif77`** / **`mpif90`**
    - **`mpicc`** / **`mpiCC`**
  - Behave like normal compilers/linkers

Compiling and running the code

- Compiling/linking
  - Headers and libs must be found by compiler
  - Most implementations provide wrapper scripts, e.g.,
    - `mpif77` / `mpif90`
    - `mpicc` / `mpiCC`
  - Behave like normal compilers/linkers
- Running
  - Details are implementation specific
  - Startup wrappers: `mpirun`, `mpiexec`, `aprun`, `poe`
    - `Job scheduler wrappers: srun`

Compiling and running the code

- Compiling/linking
  - Headers and libs must be found by compiler
  - Most implementations provide wrapper scripts, e.g.,
    - `mpif77` / `mpif90`
    - `mpicc` / `mpiCC`
  - Behave like normal compilers/linkers
- Running
  - Details are implementation specific
  - Startup wrappers: `mpirun`, `mpiexec`, `aprun`, `poe`
    - `Job scheduler wrappers: srun`

```
$ mpiCC -o hello hello.cc
$ mpirun -np 3 ./hello
Hello World! I am 2 of 3
Hello World! I am 1 of 3
Hello World! I am 0 of 3
```

Compiling and running the code

- Compiling/linking
  - Headers and libs must be found by compiler
  - Most implementations provide wrapper scripts, e.g.,
    - `mpif77` / `mpif90`
    - `mpicc` / `mpiCC`
  - Behave like normal compilers/linkers
- Running
  - Details are implementation specific
  - Startup wrappers: `mpirun`, `mpiexec`, `aprun`, `poe`
    - `Job scheduler wrappers: srun`

```
$ mpiCC -o hello hello.cc
$ mpirun -np 3 ./hello
Hello World! I am 2 of 3
Hello World! I am 1 of 3
Hello World! I am 0 of 3
```

```
$ mpirun -np 1 ./hello :
-np 1 ./hello : -np 1 ./hello
Hello World! I am 1 of 3
Hello World! I am 0 of 3
Hello World! I am 2 of 3
```

Point-to-Point Communication

It is a communication between two processes where a sender (source process) sends message to a receiver (destination process).

- Procedure (C/C++ binding, Fortran binding, Fortran 2008 binding)
- Message data
  - Buffer (address)
  - Datatype (basic or derived?)
  - Count (number of elements, not bytes)
- Message envelope
  - Source
  - Destination
  - Tag

Basic Datatypes (C/C++)

MPI datatype	C datatype
MPI_INT	int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_BOOL	_Bool
MPI_CHAR	char
MPI_BYTE	-----
MPI_PACKED	-----
and many more -> https://www.mpi-forum.org/docs/	

Basic Datatypes (Fortran)

MPI datatype	Fortran datatype
MPI_INTEGER	integer
MPI_REAL	real(kind=4)
MPI_DOUBLE_PRECISION	real(kind=8)
MPI_COMPLEX	complex(kind=4)
MPI_DOUBLE_COMPLEX	complex(kind=8)
MPI_LOGICAL	logical
MPI_CHARACTER	character(len=1)
MPI_BYTE	-----
MPI_PACKED	-----

MPI Data Types Cont'd

- **`MPI_BYTE`: Eight binary digits**
  - hack value, do not use!
- **`MPI_PACKED`**: can implement new data types → however, it is more flexible to use …
- **Derived data types**: Built at run time from basic data types or previously defined derived data types
- **Data type matching**: Same MPI data type in SEND and RECEIVE call
  - type must match on both ends in order for the communication to take place
- **Support for heterogeneous systems/clusters**
  - implementation-dependent
  - automatic data type conversion between systems of differing architecture may be needed

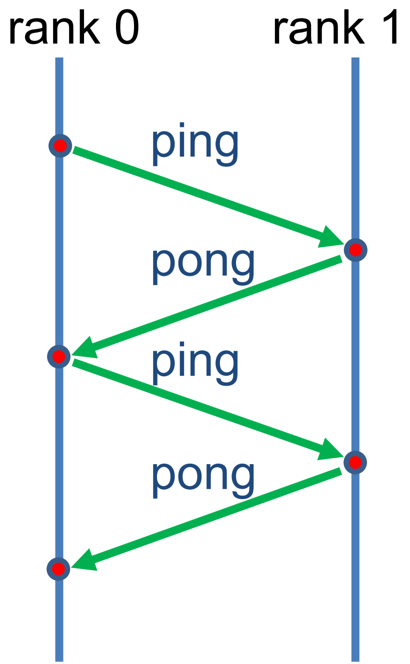
Point-to-Point Communication
Blocking

Blocking communication

- Definition: a blocking communication does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer after return.
- The term blocking may be confusing. Indeed based on the definition above, one can infer:
  - The call to a send procedure does not obstruct the flow of the program at that line of the code up to the completion of the communication. Therefore, a blocking sender may return when the transmission of the message may be:
    - not yet started
    - ongoing
    - completed (less likely)

Single-round ping-pong

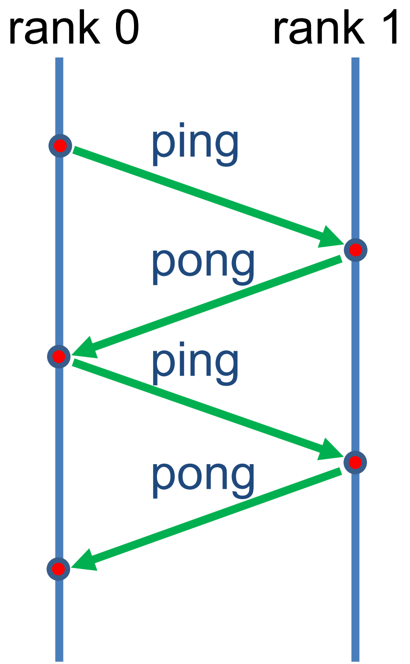
Single-round ping-pong



- First rank 0 sends and rank 1 receives, then the opposite:
  - Final value of d at rank 0?

Single-round ping-pong

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
      int ierr, irank, nrank, COUNT=1000;
      MPI_Status status;
      double *d=malloc(COUNT * sizeof(double));
      ierr=MPI_Init(&argc,&argv);
      ierr=MPI_Comm_rank(MPI_COMM_WORLD,&irank);
      ierr=MPI_Comm_size(MPI_COMM_WORLD,&nrank);
      if(irank==0) for(int i=0;i<COUNT;i++) d[i]=100.0;
      if(irank==1) for(int i=0;i<COUNT;i++) d[i]=200.0;
      printf("BEFORE: nrank,irank,d = %5d%5d%8.1f\n",nrank,irank,d[0]);
      if(irank==0) {
          MPI_Send(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
          MPI_Recv(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&status);
      }
      else if(irank==1) {
          MPI_Recv(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
          MPI_Send(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
      }
      printf("AFTER: nrank,irank,d = %5d%5d%8.1f\n",nrank,irank,d[0]);
      ierr=MPI_Finalize();
}
```



- First rank 0 sends and rank 1 receives, then the opposite:
  - Final value of d at rank 0?

DEADLOCK

DEADLOCK

Let’s consider changing the order of send and receive in rank 1, i.e. both ranks call first MPI_SEND and then MPI_RECV:

```
if(irank==0) {
    MPI_Send(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
    MPI_Recv(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&status);
}
else if(irank==1) {
    MPI_Send(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
    MPI_Recv(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

DEADLOCK

Let's consider changing the order of send and receive in rank 1, i.e. both ranks call first MPI_SEND and then MPI_RECV:

```
if(irank==0) {
    MPI_Send(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
    MPI_Recv(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&status);
}
else if(irank==1) {
    MPI_Send(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
    MPI_Recv(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

- Is DEADLOCK expected?
    - Final value of d at rank 0?

DEADLOCK

Let's consider changing the order of send and receive in rank 1, i.e. both ranks call first MPI_SEND and then MPI_RECV:

```
if(irank==0) {
    MPI_Send(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
    MPI_Recv(d,COUNT,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&status);
}
else if(irank==1) {
    MPI_Send(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
    MPI_Recv(d,COUNT,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

- Is DEADLOCK expected?
  - Final value of d at rank 0?

Executing with different values of COUNT:

```
mpirun -n 2 ./a.out 10          # OK
mpirun -n 2 ./a.out 100         # OK
mpirun -n 2 ./a.out 1000        # OK
mpirun -n 2 ./a.out 10000       # OK
mpirun -n 2 ./a.out ????????    # at some array length DEADLOCK occurs
```

Communication modes

- There are four send communication modes:

Mode	Binding
Synchronous	MPI_Ssend
Buffered (asynchronous)	MPI_Bsend
Standard	MPI_Send
Ready	MPI_Rsend

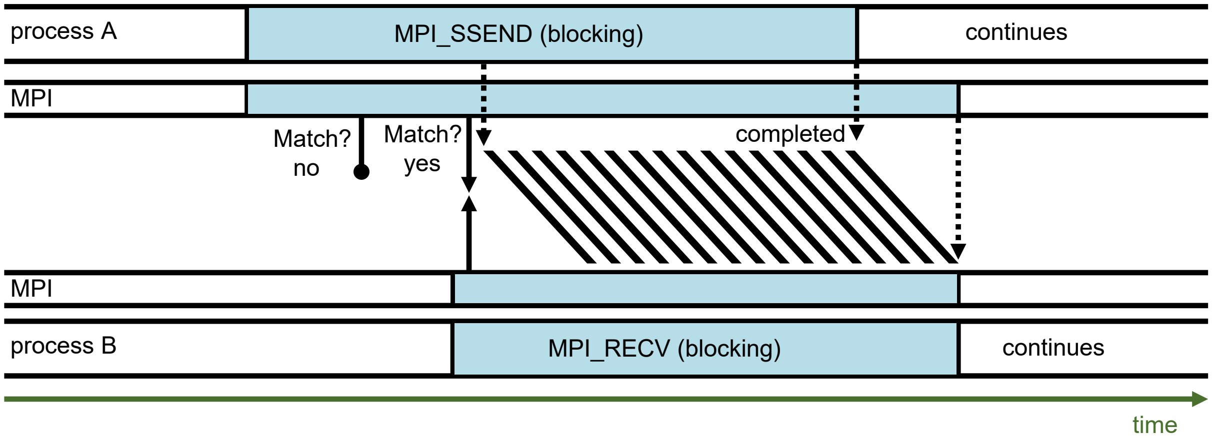
- There is only one receive communication mode:
  - Standard: MPI_Recv

Synchronous send: MPI_Ssend

- It can be started whether or not a matching receive was posted
- It will complete successfully only if a matching receive is posted
  - Send buffer can be reused
  - Receiver has reached a certain point in its execution

Tips

- Useful for debugging
- Serialization
- High latency
  (synchronization overhead)
- Best bandwidth



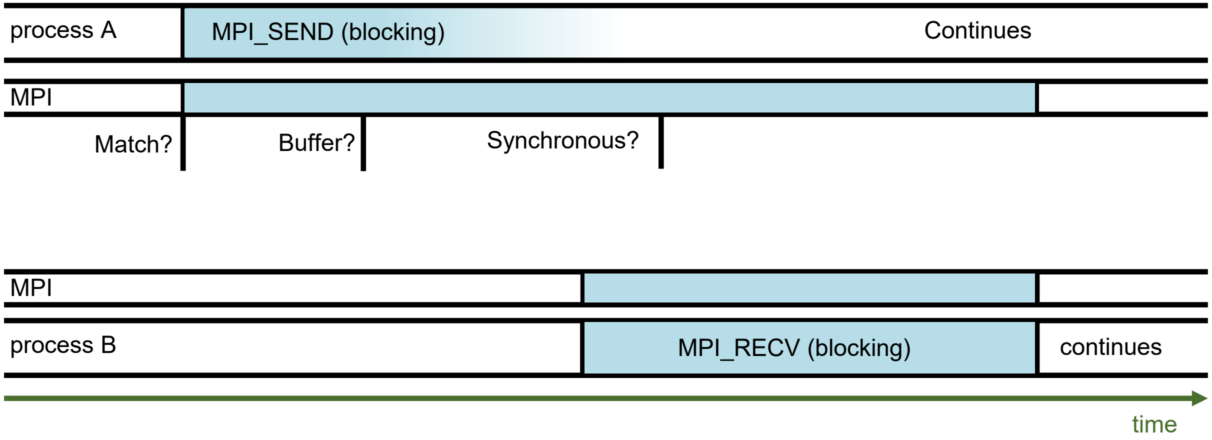
Standard send: MPI_Send

- It can be started whether or not a matching receive was posted
- It may complete before a matching receive is posted
  - Send buffer can be reused
  - The operation is local or nonlocal
  - It buffers or sends synchronously: message size, MPI implementation, etc.

Tips

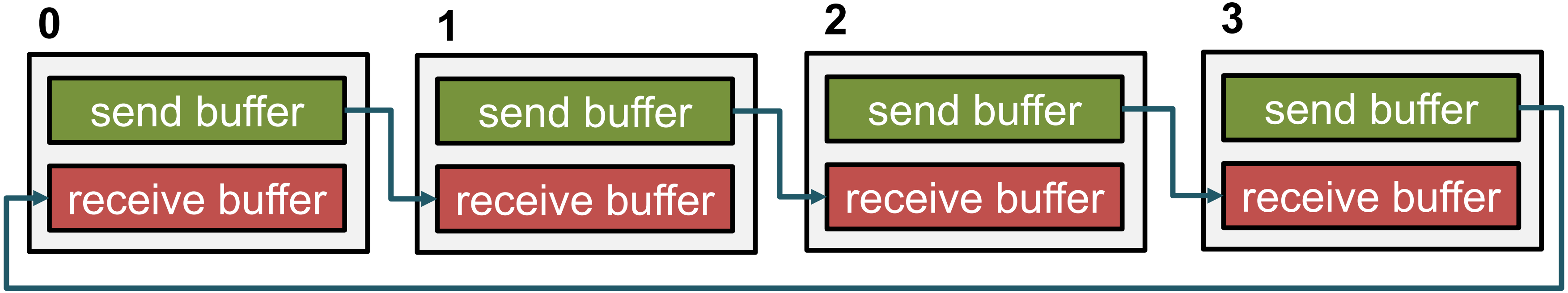
- Deadlock may occur
- Minimal transfer time

The standard send is the standard choice for you!



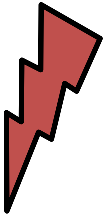
Point-to-Point Communication MPI_SEND/MPI_RECV

- Sending/Receiving at the same time is a common use case
- e.g.: shift messages, ring topologies, ghost cell exchange



- MPI_Send/MPI_Recv: pairs are not reliable!

```
//my left neighbor
left=(rank-1)%size;
//my right neighbor
right=(rank+1)%size;
MPI_Send(sendbuf,n,type,right,tag,comm);
MPI_Recv(recvbuf,n,type,left,tag,comm,status);
```



- How to avoid potential deadlock?

Point-to-Point Communication MPI_SENDRECV

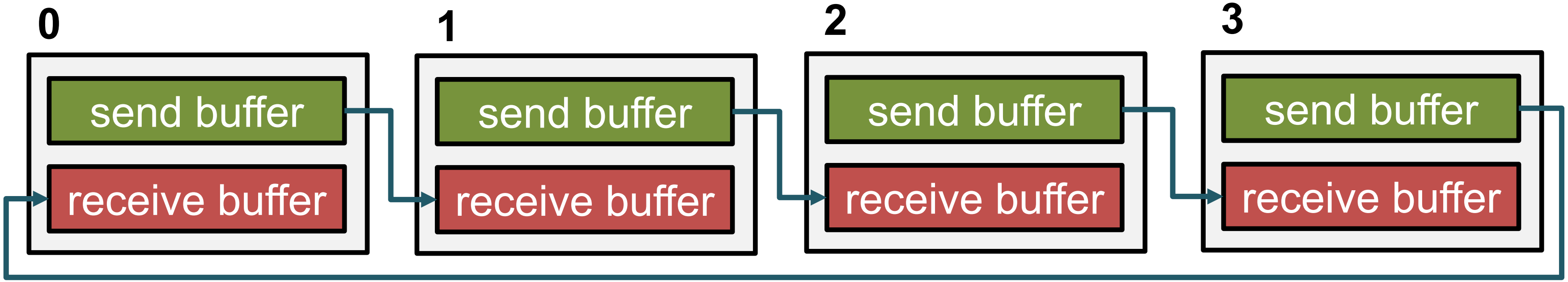
- Syntax: simple combination of send and receive arguments:

```
MPI_Sendrecv(buffer_send, sendcount, sendtype, dest,   sendtag,
             buffer_recv, recvcount, recvtype, source, recvtag,
             comm, MPI_Status * status)
```

- disjoint send/receive buffers
- can have different count & data type
- blocking call

- MPI takes care, thereby no deadlocks occur:

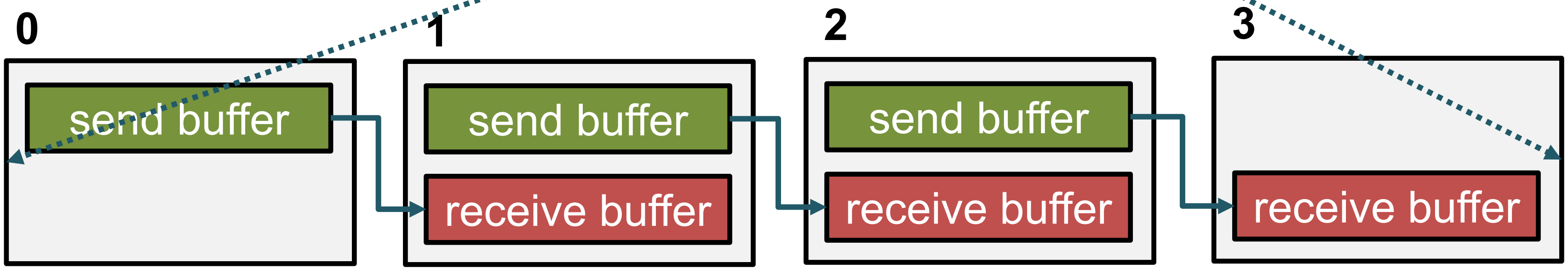
```
// Rank left from myself
left = (rank - 1 + size) % size;
// Rank right from myself
right = (rank + 1) % size;
MPI_Sendrecv(buffer_send, n, MPI_INT, right, 0,
             buffer_recv, n, MPI_INT, left,  0,
             MPI_COMM_WORLD, status);
```



Point-to-Point Communication MPI_SENDRECV

- useful for open chains/non-circular shifts:

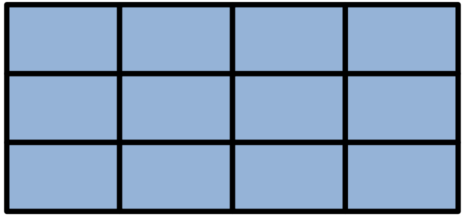
```
// Rank left from myself.
left = rank – 1; if (left < 0) { left = MPI_PROC_NULL; }
// Rank right from myself.
right = rank + 1; if (right >= size) {right = MPI_PROC_NULL; }
MPI_Sendrecv(buffer_send, n, MPI_INT, right, 0,
             buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, &status);
```



- MPI_PROC_NULL as source/destination acts as no-op
  - send/recv with MPI_PROC_NULL return immediately, buffers are not altered

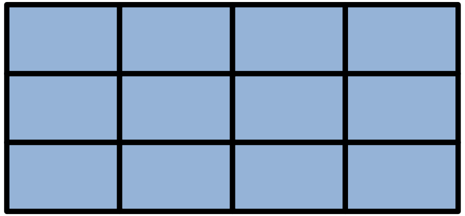
Ghost Cell Exchange with MPI_Sendrecv

Domain distributed to ranks here 4 x 3 ranks each rank gets one tile

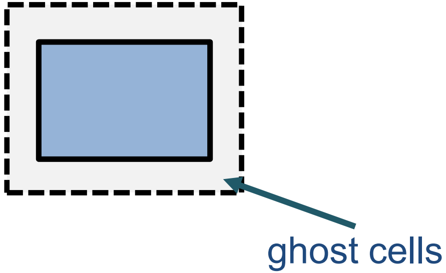


Ghost Cell Exchange with MPI_Sendrecv

Domain distributed to ranks here 4 x 3 ranks each rank gets one tile

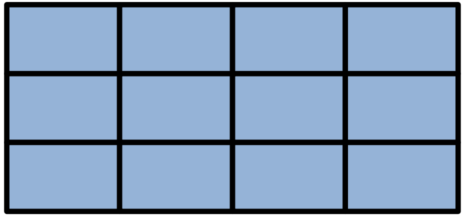


Each ranks tile is surrounded by ghost cells, representing the cells of the neighbors

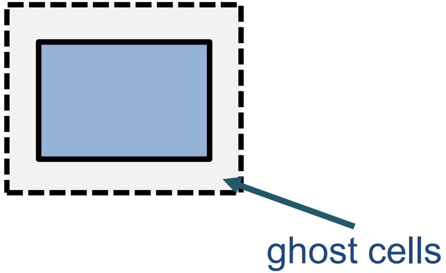


Ghost Cell Exchange with MPI_Sendrecv

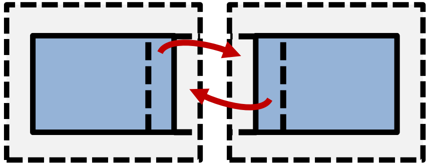
Domain distributed to ranks here 4 x 3 ranks each rank gets one tile



Each ranks tile is surrounded by ghost cells, representing the cells of the neighbors

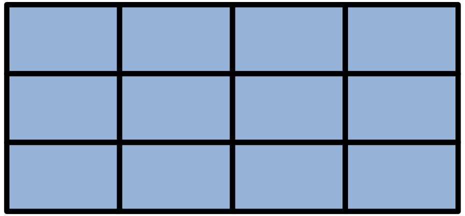


After each sweep over a tile perform ghost cell exchange, i.e. update ghost cells with new values of neighbor cells

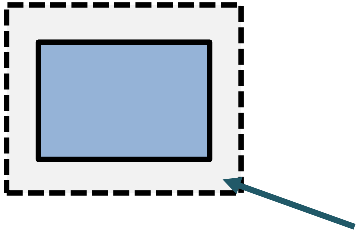


Ghost Cell Exchange with MPI_Sendrecv

Domain distributed to ranks here 4 x 3 ranks each rank gets one tile

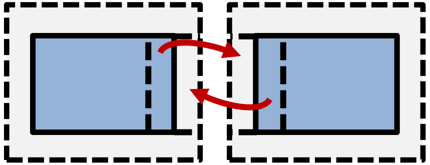


Each ranks tile is surrounded by ghost cells, representing the cells of the neighbors



ghost cells

After each sweep over a tile perform ghost cell exchange, i.e. update ghost cells with new values of neighbor cells

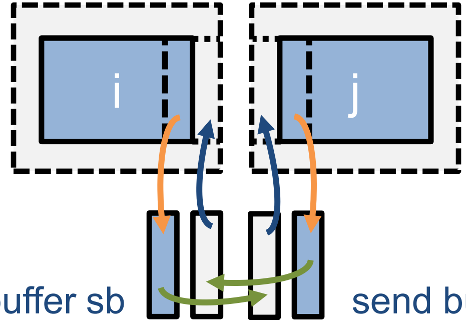


Possible implementation:

1. copy new data into contiguous send buffer
2. send/receive new data to/from the neighbor
3. copy new data into ghost cells

```
MPI_Sendrecv(
sb, …, j,
rb, …, j, …)
```

step 2




```
MPI_Sendrecv(
sb, …, i,
rb, …, i, …)
```

step 2

send buffer sb
receive buffer rb

send buffer sb
receive buffer rb

MPI Error Handling

- **Fortran MPI routines**
  - `ierror` argument — can/cannot be omitted!
- **C MPI routines**
  - return an `int` — may be ignored
- **Return value `MPI_SUCCESS`**
  - indicates that all went ok
- **Default:**
  - abort parallel computation in case of other return values
  - but can also define error handlers

Handling Status Information

- **MPI status provides additional information about the message**
  - size, source, tag, error code – may not be otherwise known if wildcards are used
  - can also use MPI_STATUS_IGNORE in some contexts

- **MPI_status in Fortran**

```
integer :: status(MPI_STATUS_SIZE)
```

  - Array of integers of size MPI_STATUS_SIZE
  - index values for query: MPI_SOURCE, MPI_TAG, MPI_ERROR

- **MPI_status in C/C++**

```
MPI_Status status;
```

  - Structure of type MPI_Status
  - hand a reference to MPI_Recv
  - component names for query: status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR

- Inquiring message length needs an additional MPI call:
  - Fortran: `call MPI_GET_COUNT(status, datatype, count, ierror)`
  - C: `MPI_Get_count(&status, datatype, &count);`
    - count is output argument
    - datatype must be the same datatype used in the MPI call that produced the status variable

MPI_Recv Example: C/C++

- **Example: receive array of floats from any source**

```
int count, countrecv;

MPI_Status status;

field = (float *)malloc(count*sizeof(float));

…

MPI_Recv(field, count, MPI_FLOAT, MPI_ANY_SOURCE,
 MPI_ANY_TAG, MPI_COMM_WORLD, &status);

printf("Received from %i with tag %i count: %i \n",
 status.MPI_SOURCE, status.MPI_TAG)
```

- **Obtain number of actually received items:**
  - `MPI_Get_count(&status, MPI_FLOAT, &countrecv);`

Blocking Point-to-Point Communication: Summary

- Blocking MPI communication calls:
  - send/receive buffer can safely be reused when a blocking call returns
  - Blocking send has 4 communication modes:
    1. Synchronous 2. Buffered 3. Standard 4. Ready
  - Blocking Receive has only one communication mode: MPI_Recv
  - Blocking calls can lead to deadlocks
- Shift operations: keep eye on deadlocks and serialization
- MPI_Sendrecv: combined send and receive
  - MPI ensures no deadlocks occur
  - MPI_Sendrecv_replace: useful when only one single buffer is required

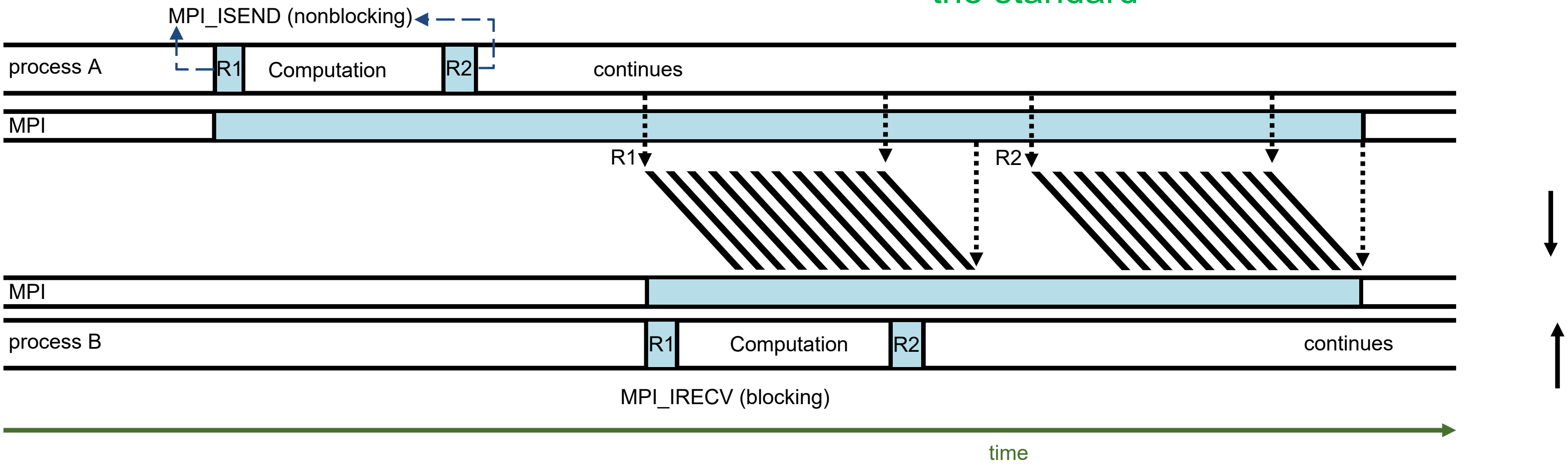
Point-to-Point Communication Nonblocking

Nonblocking point-to-point communication

- Call to a nonblocking send/recv procedure returns straight away. It avoids synchronization so that the following opportunities can be exploited:
  - Avoiding certain deadlocks
  - Truly bidirectional commun.
  - Avoid idle time:
    - Overlapping communication and computation but not guaranteed by the standard

Nonblocking point-to-point communication

- Call to a nonblocking send/recv procedure returns straight away. It avoids synchronization so that the following opportunities can be exploited:
  - Avoiding certain deadlocks
  - Truly bidirectional commun.
  - Avoid idle time:
    - Overlapping communication and computation but not guaranteed by the standard



Standard nonblocking send/receive

- `MPI_Isend(sendbuf, count, datatype, dest,   tag,`
  `               comm, MPI_Request * request);`

  `MPI_Irecv(recvbuf, count, datatype, source, tag,`
  `               comm, MPI_Request * request);`

  `request`: pointer to variable of type **`MPI_Request`**,
  will be associated with the corresponding operation
- **Do not reuse `sendbuf/recvbuf` before `MPI_Isend/MPI_Irecv` has been completed!**
  - Return of a nonblocking call does not imply completion ← Be careful!
- **`MPI_Irecv`** has no status argument
  - obtained later during completion via **`MPI_Wait*/MPI_Test*`**

Nonblocking send and receive variants

- Completion
  - Return of `MPI_I*` call does not imply completion
  - Check for completion via `MPI_Wait* / MPI_Test*`
  - Semantics identical to blocking call combined with a "wait"

nonblocking MPI function	blocking MPI function	type	completes when
`MPI_Isend`	`MPI_Send`	synchronous or buffered	depends on type
`MPI_Ibsend`	`MPI_Bsend`	buffered	buffer has been copied
`MPI_Issend`	`MPI_Ssend`	synchronous	remote starts receive
`MPI_Irecv`	`MPI_Recv`	--	message was received

Test for completion

Two test modes:

- Blocking
  - **`MPI_Wait*`**: Wait until the communication has been completed and buffer can safely be reused
- Nonblocking
  - **`MPI_Test*`**: Return true (false) if the communication has (not) completed

Despite the naming, the modes both pertain to nonblocking point-to-point communication!

Test for completion – single request

- Test one communication handle for completion:

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

request: request handle of type **MPI_Request**

status: status object of type **MPI_Status** (cf. **MPI_Recv**)

flag: variable of type **int** to test for success

- MPI_Wait waits until the communication has been completed and buffer can safely be reused: Blocking
- MPI_Test returns TRUE (FALSE) if the communication has (not) completed: Nonblocking

Use of wait/test

MPI_Wait

```
MPI_Request request;
MPI_Status status;
MPI_Isend(send_buffer,count,MPI_CHAR,
    dst,0,MPI_COMM_WORLD,&request);

// do some work…
// do not use send_buffer
MPI_Wait(&request, &status);

// send_buffer can now be used safely
```

Nonblocking communication:
- Return from function != completion
- Each initiated operation must have a matching wait/test!

MPI_Test

```
MPI_Request request;
MPI_Status status;
int flag;
MPI_Isend(send_buffer,count,MPI_CHAR,
    dst,0,MPI_COMM_WORLD,&request);

do {
    // do some work…
    // do not use send_buffer
    MPI_Test(&request, &flag, &status);
} while (!flag);

// send_buffer can now be used safely
```

loop

Wait for completion – all requests in a list

- MPI can handle multiple communication requests
- Wait/Test for completion of multiple requests:

```
MPI_Waitall(int count, MPI_Request requests[],
                       MPI_Status statuses[]);

MPI_Testall(int count, MPI_Request requests[],
            int *flag, MPI_Status statuses[]);
```

- Waits for/Tests if all provided requests have been completed

Use of MPI_Waitall

```
MPI_Request requests[2];
MPI_Status  statuses[2];

MPI_Isend(send_buffer, …, &(requests[0]));
MPI_Irecv(recv_buffer, …, &(requests[1]));

// do some work…

MPI_Waitall(2, requests, statuses)
// Isend & Irecv have been completed
```

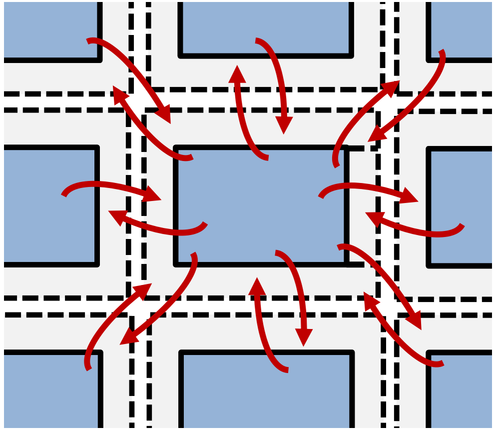
Arrays of requests and statuses

number of elements in the arrays

Requests can be from one or multiple send/receive operations or combination of them!

Ghost Cell Exchange: nonblocking PtP Communication

- Ghost cell exchange: communication using nonblocking send/recv can be initiated with all neighbors at once.



Possible implementation:

1. Copy new data into contiguous send buffers
2. Start nonblocking receives/sends from/to corresponding neighbors
3. Wait with MPI_Waitall for all obtained requests to complete
4. Copy new data into ghost cells

Other Ways of Testing for Completion

- Examine the completion of multiple requests:
  - MPI_Waitall
  - MPI_Testall
  - MPI_Waitany
  - MPI_Testany
  - MPI_Waitsome
  - MPI_Testsome

- Completed requests are automatically set to MPI_REQUEST_NULL

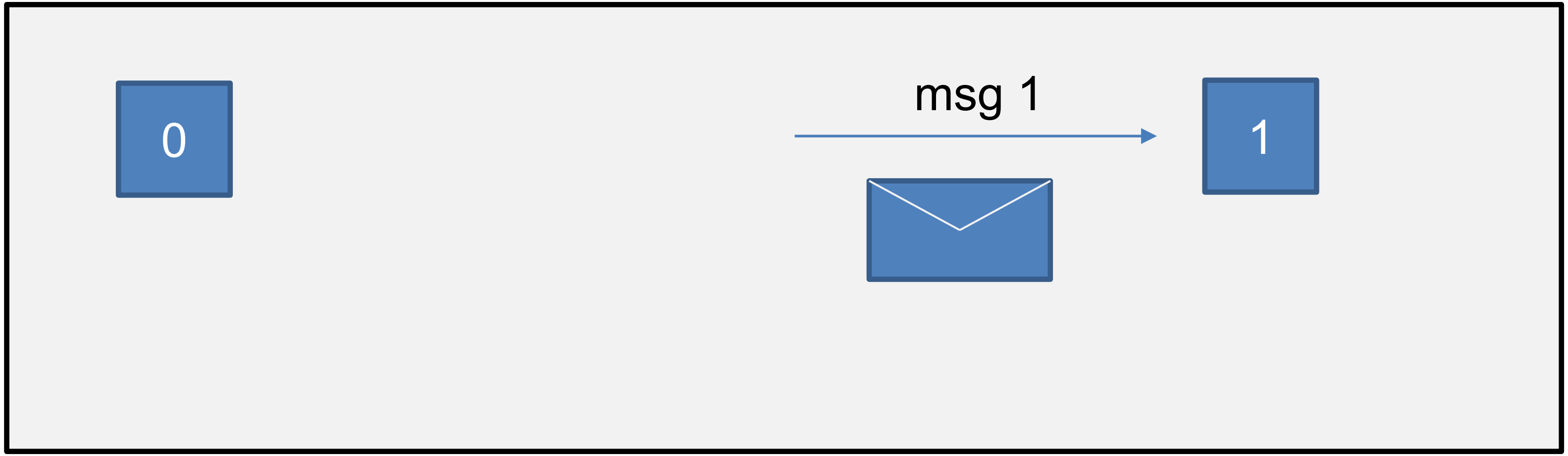
Helper functions and Semantics

Semantics

- non-overtaking rule: message order preservation is guaranteed within a communicator

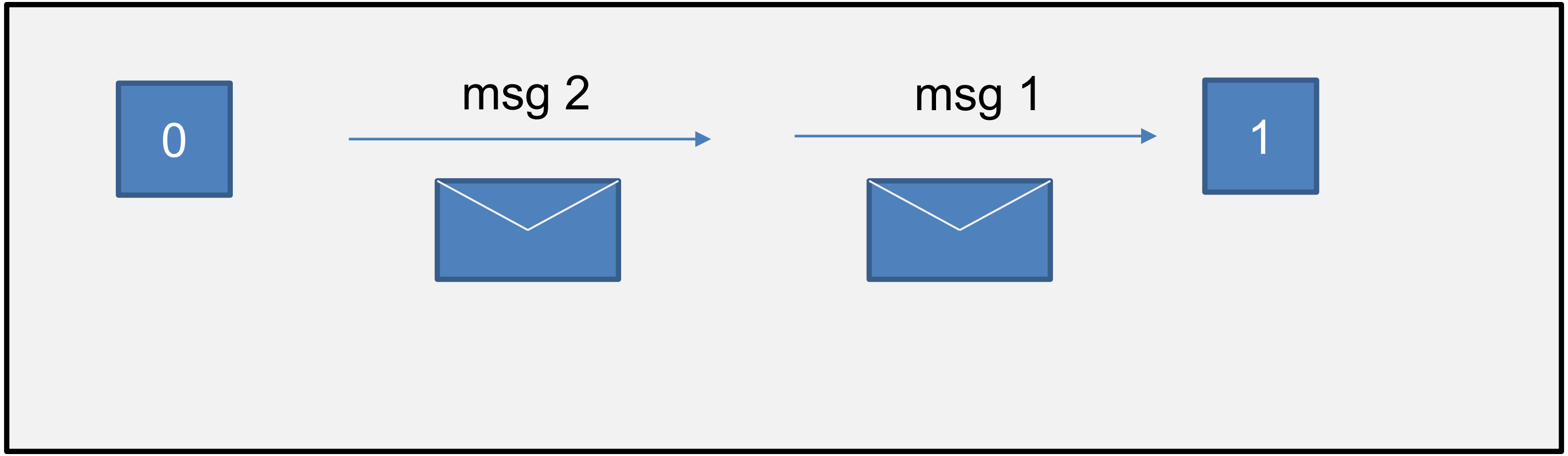
Semantics

- non-overtaking rule: message order preservation is guaranteed within a communicator



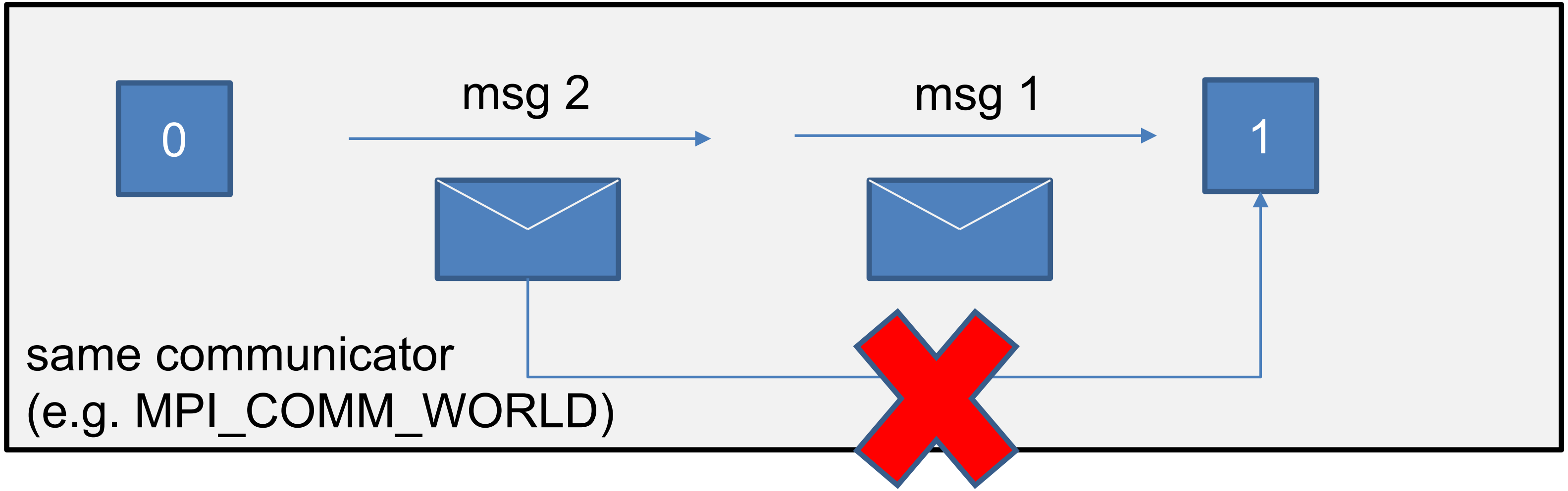
Semantics

- non-overtaking rule: message order preservation is guaranteed within a communicator



Semantics

- non-overtaking rule: message order preservation is guaranteed within a communicator



Useful MPI Calls: MPI_WTIME

- Returns seconds since one point in past time

```
double MPI_Wtime()
```

- Use only for computation of time differences

```
time_start = MPI_Wtime()
// … working …
duration = MPI_Wtime() - time_start
```

- Returns time resolution in seconds,

```
double MPI_Wtick()
```

  - e.g. if resolution is 1ms `MPI_Wtick()` returns `1e-3`
- No `ierror` argument in Fortran: both modules mpi and mpi _f08
- Typically clocks from different ranks are not synchronized

Useful MPI Calls: MPI_ABORT

- MPI_Abort forces an MPI program to terminate:

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

- Aborts all processes in communicator
- `errorcode` will be handed as exit value to calling environment
- Safe and well-defined way of terminating an MPI program (if implemented correctly)
- In general, if something unexpected happens, try to shut down your MPI program the standard way (MPI_Finalize())

Questions?

Collective Communication in MPI

Collective Communication Introduction

- Operations including all ranks of a communicator
  - All ranks must call the function!
- Blocking calls: buffer can be reused after return
- Nonblocking calls with MPI-3.0
- Cannot interfere with point-to-point communication
  - Completely separate modes of operation!
- Data type matching
- No tags
- Sent message must fill receive buffer (count is exact)
- Typically MPI libraries provide optimized implementations for operations
  - Do not write your own collectives using PtP calls!

Collective Communication Introduction

- May or may not synchronize the processes
- Types of collective calls:
  - Synchronization (barrier)
  - Data movement (broadcast, scatter, gather, all to all)
  - Collective computation/operations (reduction)
- MPI_*v bindings: allow for unequal data size across ranks

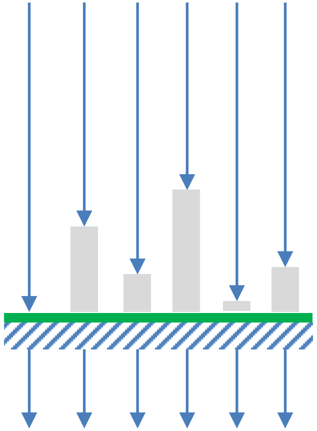
Collective Communication Synchronization

- Explicit synchronization of all ranks from specified communicator

  `MPI_Barrier(comm);`

- Ranks only return from call after every rank has called the function

- `MPI_Barrier:` rarely needed
  - Debugging

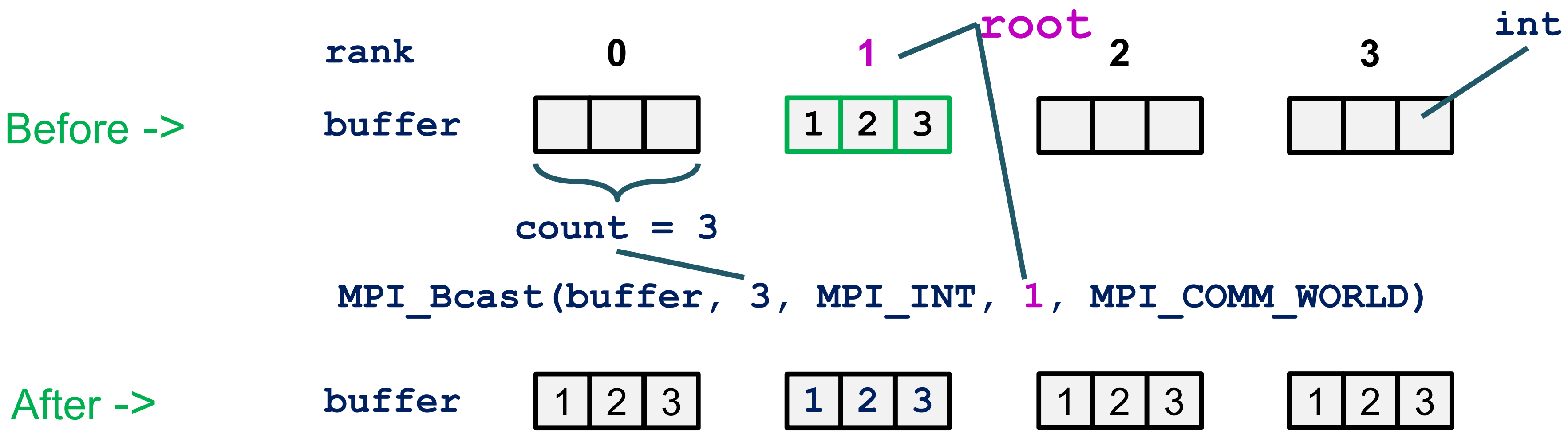


Collective Communication Broadcast

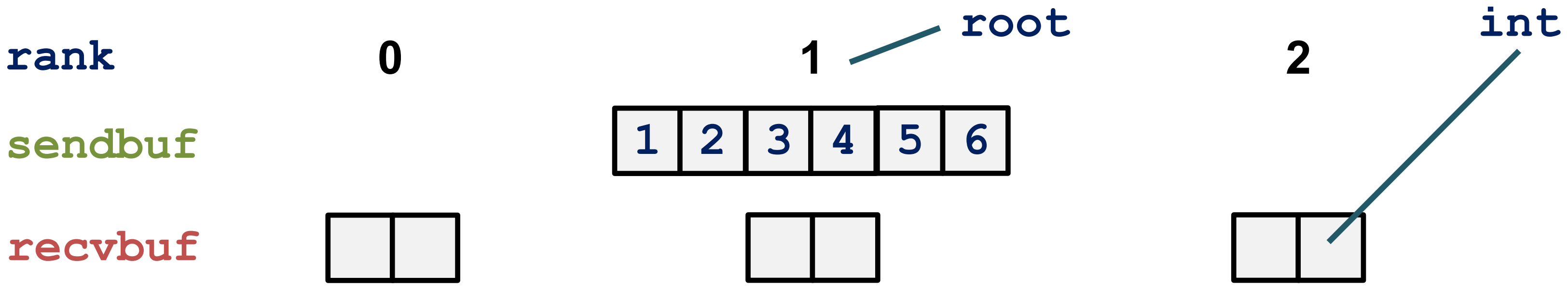
- Send buffer contents from one rank (“`root`”) to all ranks

```
MPI_Bcast(buf, count, datatype, int root, comm);
```

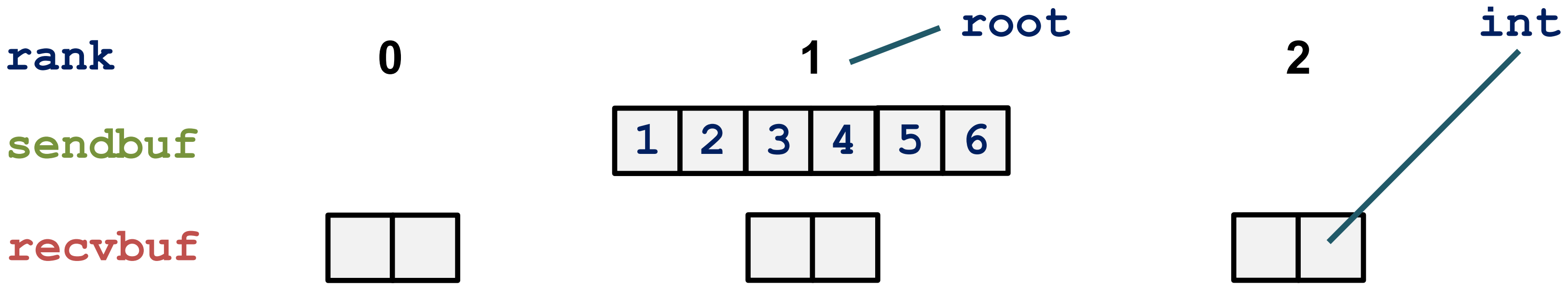
- No restrictions on which rank is root – often rank 0



Scattering to other Processes

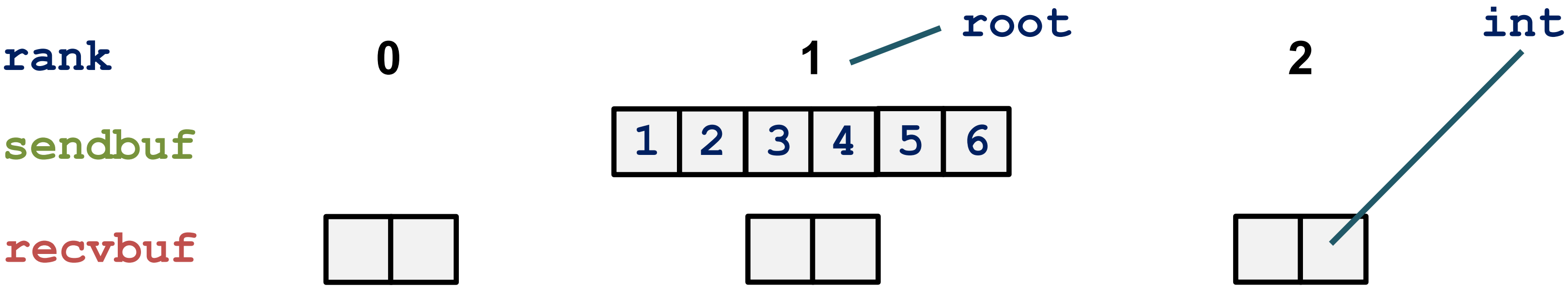


Scattering to other Processes

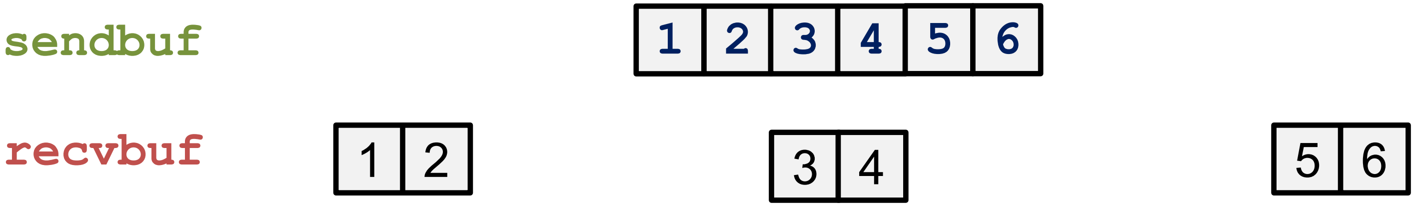



```
MPI_Scatter(sendbuf, 2, MPI_INT, recvbuf, 2, MPI_INT,
            root, MPI_COMM_WORLD)
```

Scattering to other Processes




```
MPI_Scatter(sendbuf, 2, MPI_INT, recvbuf, 2, MPI_INT,
            root, MPI_COMM_WORLD)
```



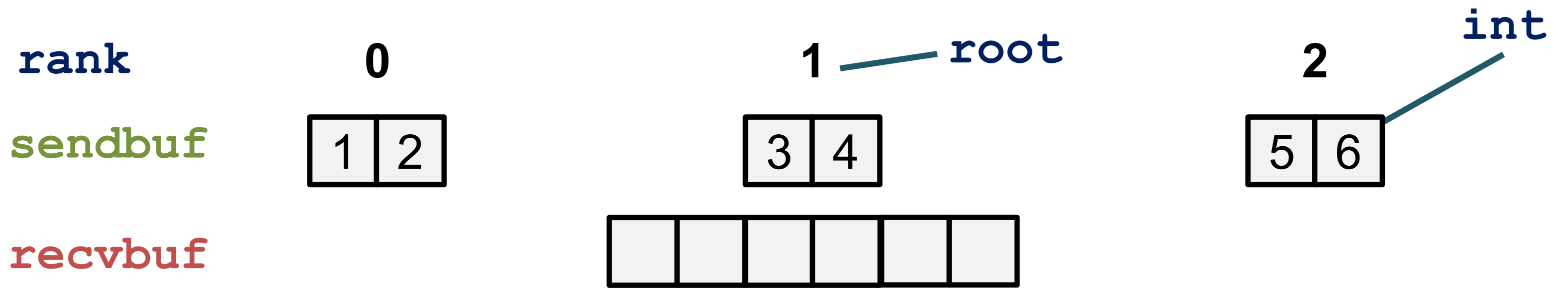
MPI_Scatter

- Send every i-th chunk of an array to the i-th rank

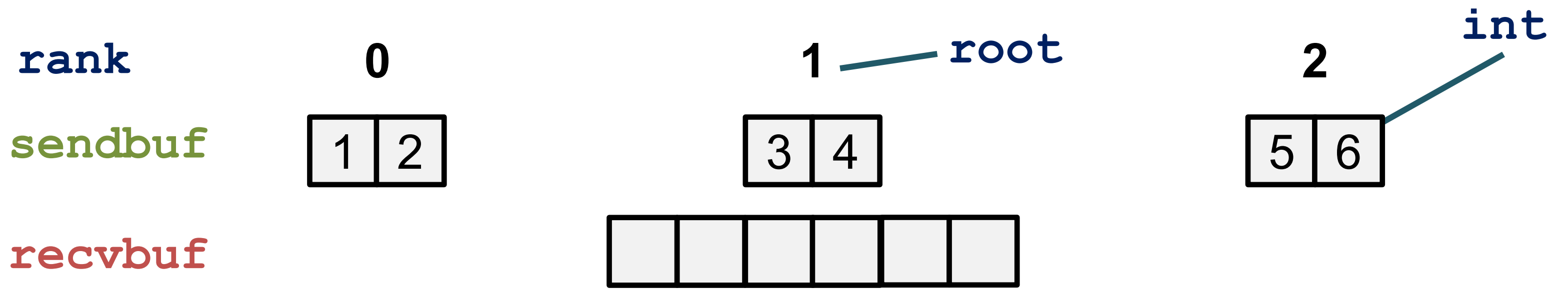
```
MPI_Scatter(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype,
            root, comm);
```

- Root and comm must be the same on all processes
- Type signature of send and receive variables must match
- Usually, **`sendcount`** = **`recvcount`** because **`sendtype = recvtype`**
  - This is the length of the chunk
- **`sendbuf`** is ignored on non-root ranks because there is nothing to send

Gathering Data from Other Processes

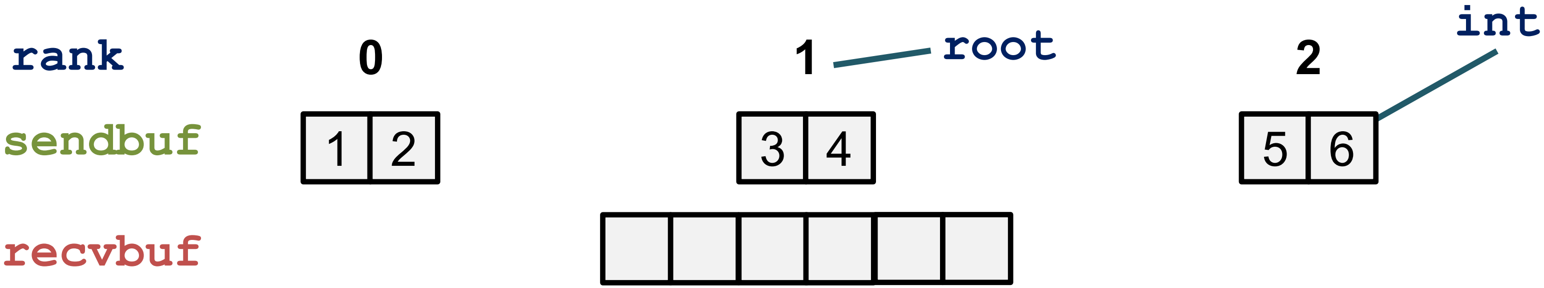


Gathering Data from Other Processes

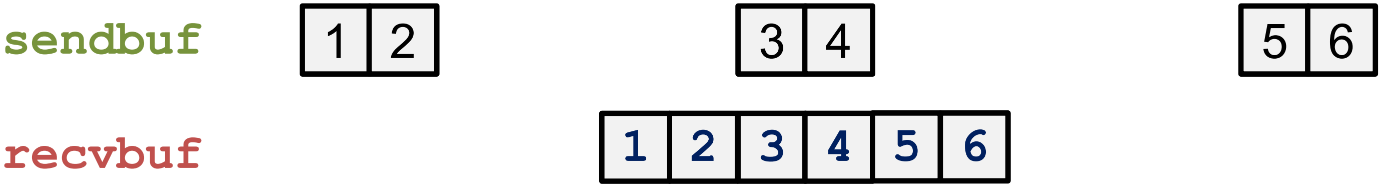



```
MPI_Gather(sendbuf, 2, MPI_INT, recvbuf, 2, MPI_INT,
           root, MPI_COMM_WORLD)
```

Gathering Data from Other Processes




```
MPI_Gather(sendbuf, 2, MPI_INT, recvbuf, 2, MPI_INT,
           root, MPI_COMM_WORLD)
```



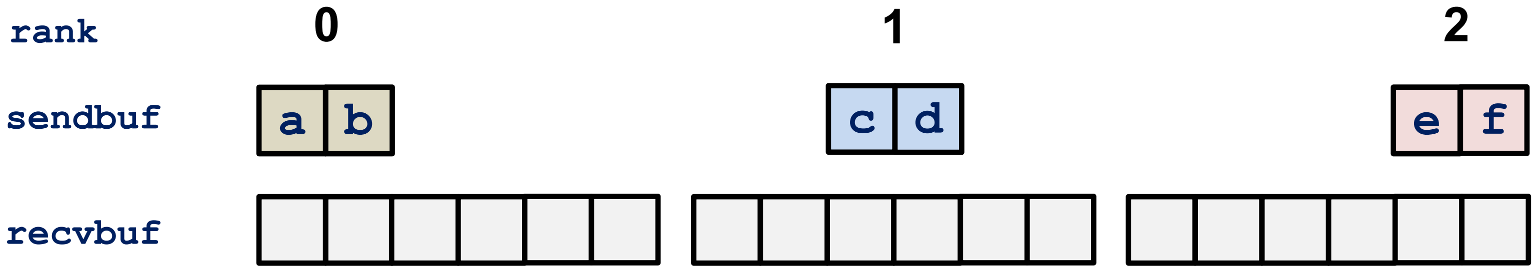
MPI_Gather

- Receive a message from each rank and place i-th rank's message at i-th position in receive buffer

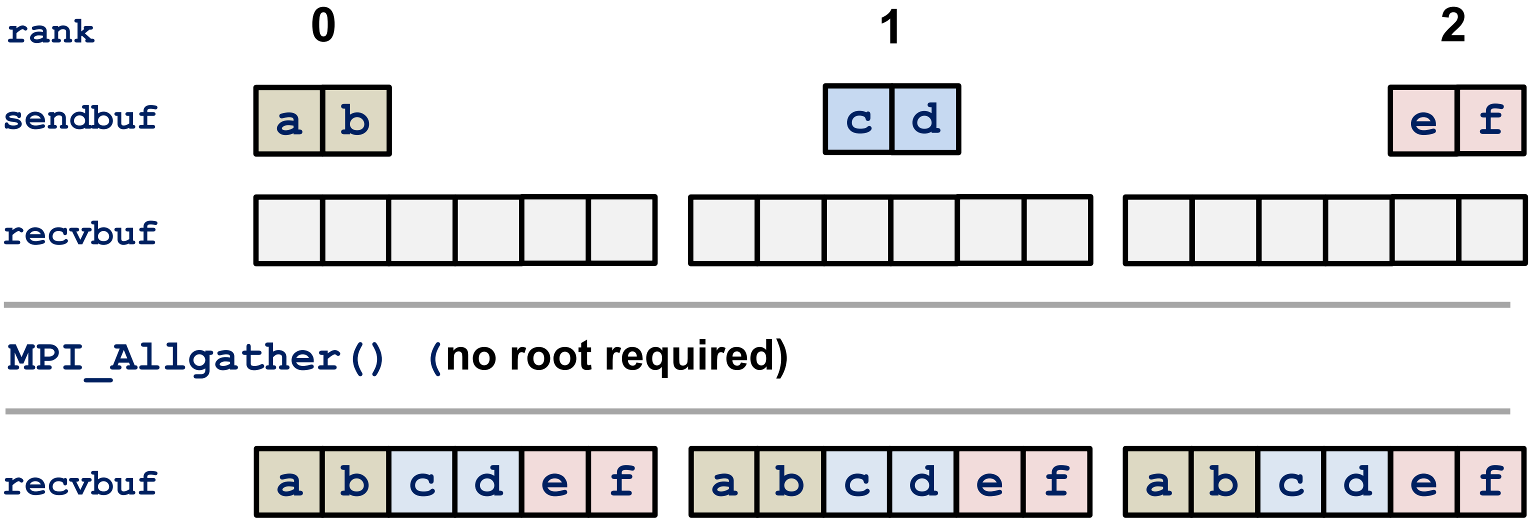
```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype,
           root, comm)
```

- Root and comm must be the same on all processes
- Type signature of send and receive variables must match
- Usually, `sendcount` = `recvcount` because `sendtype = recvtype`
- `recvbuf` is ignored on non-root ranks because there is nothing to receive

Gathering on All Processes



Gathering on All Processes



In this example: `sendcount=recvcount=2`

MPI_ALLGATHER

- Combination of gather and broadcast

```
MPI_Allgather(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype,
              comm);
```

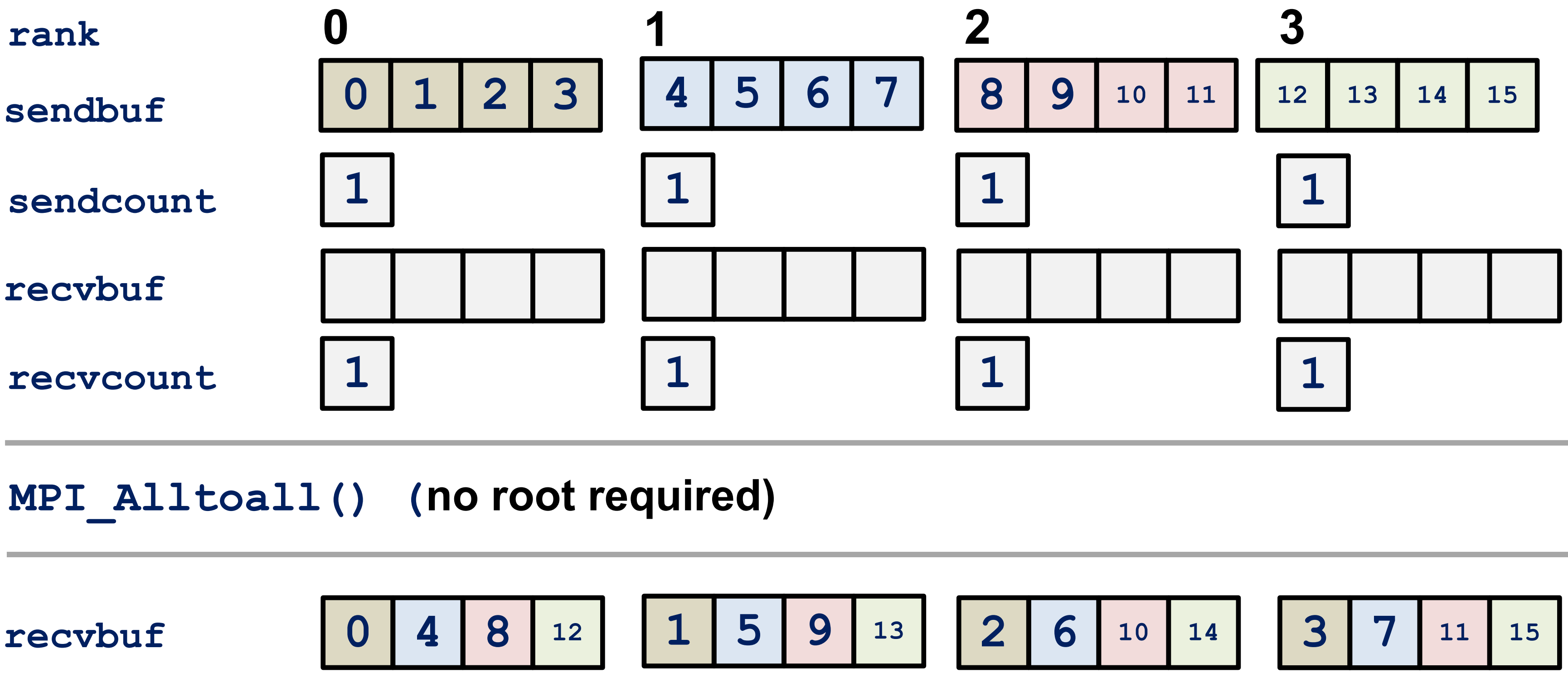
- Why not just use gather followed by a broadcast instead?
  - `MPI_Gather(…,root=i,…)` then `MPI_Bcast(…,root=i,…)`
  - MPI library has more options for optimization
  - General assumption: `MPI_Allgather` is faster than using separate `MPI_Gather` followed by `MPI_Bcast`

There is no MPI_Allscatter!!!

MPI_Alltoall

rank	0	1	2	3
sendbuf	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15
sendcount	1	1	1	1
recvbuf				
recvcount	1	1	1	1

MPI_Alltoall



Collective Communication MPI_ALLTOALL

- `MPI_Alltoall`: For all ranks, send i-th chunk to i-th rank

```
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype,
             comm);
```

Summary of MPI Collective Communications

- MPI (blocking) collectives
  - All ranks in communicator must call the function
- Communication and synchronization
  - Barrier, broadcast, scatter, gather, and combinations thereof
- In-place buffer specification **`MPI_IN_PLACE`**
  - Save some space if need be

Global Operations Syntax

- Compute results over distributed data

```
MPI_Reduce(sendbuf, recvbuf, count,
           datatype, MPI_Op op,
           root, comm);
```

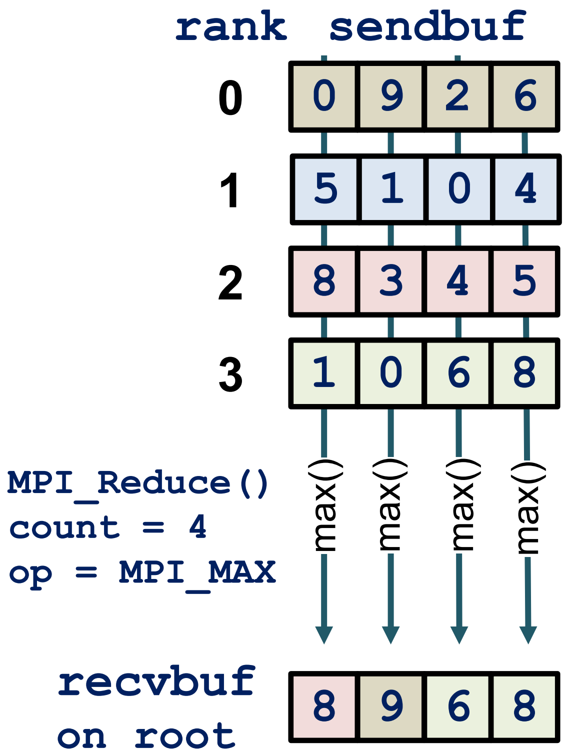
- Result in `recvbuf` only available on root process
- Perform operation on all `count` elements of an array
- If all ranks need the result, then use `MPI_Allreduce()`
- There are 12 predefined operations
- If the predefined operations are not enough, then use `MPI_Op_create`/`MPI_Op_free` to create own ops

Global Operations Syntax

- Compute results over distributed data

```
MPI_Reduce(sendbuf, recvbuf, count,
           datatype, MPI_Op op,
           root, comm);
```

- Result in `recvbuf` only available on root process
- Perform operation on all `count` elements of an array
- If all ranks need the result, then use `MPI_Allreduce()`
- There are 12 predefined operations
- If the predefined operations are not enough, then use `MPI_Op_create`/`MPI_Op_free` to create own ops



Global operations – predefined operators

Name	Operation	Name	Operation
`MPI_SUM`	Sum	`MPI_PROD`	Product
`MPI_MAX`	Maximum	`MPI_MIN`	Minimum
`MPI_LAND`	Logical AND	`MPI_BAND`	Bit-AND
`MPI_LOR`	Logical OR	`MPI_BOR`	Bit-OR
`MPI_LXOR`	Logical XOR	`MPI_BXOR`	Bit-XOR
`MPI_MAXLOC`	Maximum+Position	`MPI_MINLOC`	Minimum+Position

- Define own operations with `MPI_Op_create`/`MPI_Op_free`
- MPI assumes that the operations are associative
  → be careful with floating-point operations, as floating-point arithmetic is not associative due to rounding

“In-place” buffer specification

- Override local input buffer with a result

MPI_Reduce

```
// out-of-place
int partial_sum, total_sum;
MPI_Reduce(&partial_sum, &total_sum,
           1, MPI_INT, MPI_SUM, root, comm);

// in-place
int partial_sum, total_sum;
if (rank == root) {
  total_sum = partial_sum;
  MPI_Reduce(MPI_IN_PLACE, &total_sum,
             1, MPI_INT, MPI_SUM,
             root, comm);
}
else {
  MPI_Reduce(&partial_sum, &total_sum,
             1, MPI_INT, MPI_SUM,
             root, comm);
}
```

MPI_Allreduce

```
// out-of-place
int partial_sum, total_sum;
MPI_AllReduce(&partial_sum, &total_sum,
              1, MPI_INT, MPI_SUM, comm);


// in-place
int partial_sum, total_sum;

total_sum = partial_sum;
MPI_AllReduce(MPI_IN_PLACE, &total_sum,
              1, MPI_INT, MPI_SUM, comm);
```

MPI_IN_PLACE Cheat Sheet

Function	MPI_IN_PLACE argument	At which rank(s)	Comment [MPI 3.0]
MPI_GATHER	send buffer	root	Recv value at root already in the correct place in receive buffer.
MPI_GATHERV	send buffer	root	Recv value at root already in the correct place in receive buffer.
MPI_SCATTER	receive buffer	root	Root-th segment of send buffer is not moved.
MPI_SCATTERV	receive buffer	root	Root-th segment of send buffer is not moved.
MPI_ALLGATHER	send buffer	all	Input data at the correct place were process would receive its own contribution.
MPI_ALLGATHERV	send buffer	all	Input data at the correct place were process would receive its own contribution.
MPI_ALLTOALL	send buffer	all	Data to be send is taken from receive buffer and replaced by received data, data send/received must be of the same type map specified in receive count/receive type.
MPI_ALLTOALLV	send buffer	all	Data to be send is taken from receive buffer and replaced by received data. Data send/received must be of the same type map specified in receive count/receive type. The same amount of data and data type is exchanged between two processes.
MPI_REDUCE	send buffer	root	Data taken from receive buffer, replaced with output data.
MPI_ALLREDUCE	send buffer	all	Data taken from receive buffer, replaced with output data.

Derived Data Types in MPI

Why do we need data types in MPI?

- Example: Root reads configuration and broadcasts it to all others

```
// root: read configuration from
// file into struct config
MPI_Bcast(&cfg.nx, 1, MPI_INT, …);
MPI_Bcast(&cfg.ny, 1, MPI_INT, …);
MPI_Bcast(&cfg.du, 1, MPI_DOUBLE,…);
MPI_Bcast(&cfg.it, 1, MPI_INT, …);
```

Want to do something like:

```
MPI_Bcast(
   &cfg, 1, <type cfg>,…);
```

```
MPI_Bcast(&cfg, sizeof(cfg),
   MPI_BYTE, ..)
```

is **not** a solution. Its not portable as no data conversion can take place

MPI is supposed to support parallel computations across heterogeneous environments and communication in such environments may require data conversions.

Why do we need data types in MPI?

- Example: Send column of matrix (noncontiguous in C):
  - Send each element alone?
  - Manually copy elements out into a contiguous buffer and send it?

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

Creating an MPI data type

Three steps:

1. Construct with

   ```
   MPI_Type_*(...);
   ```

2. Commit new data type with

   ```
   MPI_Type_commit(MPI_Datatype * nt);
   ```

3. After use, deallocate the data type with

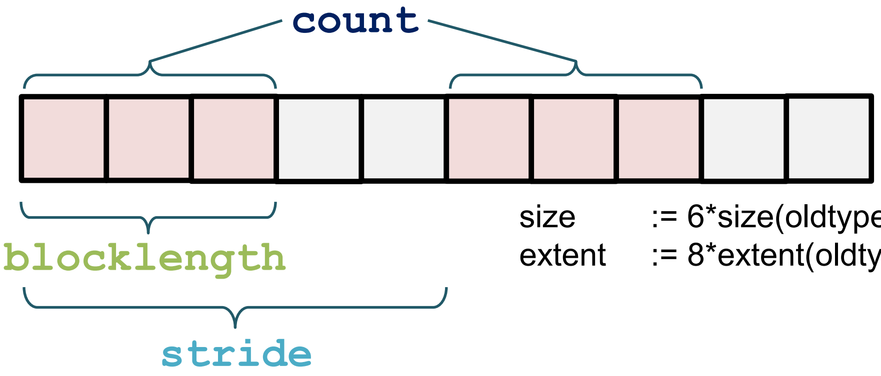
   ```
   MPI_Type_free(MPI_Datatype * nt);
   ```

All local, non-collective calls

A flexible, vector-like type: MPI_Type_vector

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

`count`	`2 (no. of blocks)`
`blocklength`	`3 (no. of elements in each block)`
`stride`	`5 (no. of elements b/w start of each block)`
`oldtype`	`MPI_INT`

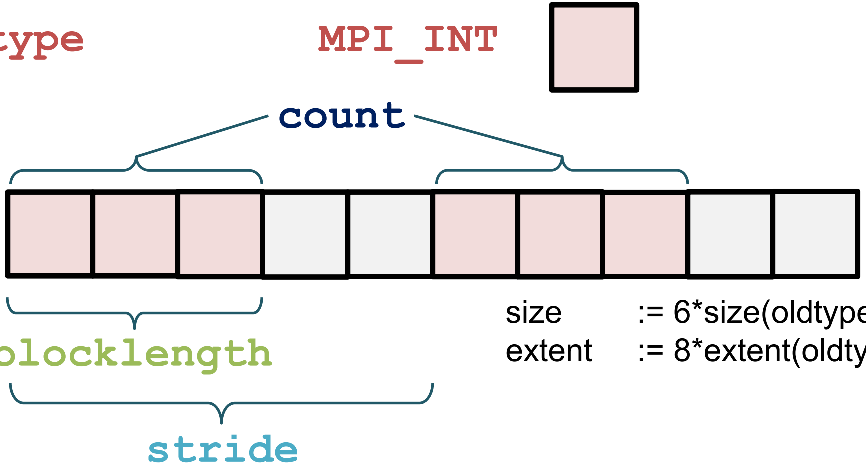


size := 6*size(oldtype)
extent := 8*extent(oldtype)

A flexible, vector-like type: MPI_Type_vector

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

```
count          2 (no. of blocks)
blocklength    3 (no. of elements in each block)
stride         5 (no. of elements b/w start of each block)
oldtype        MPI_INT
```



size := 6*size(oldtype)
extent := 8*extent(oldtype)

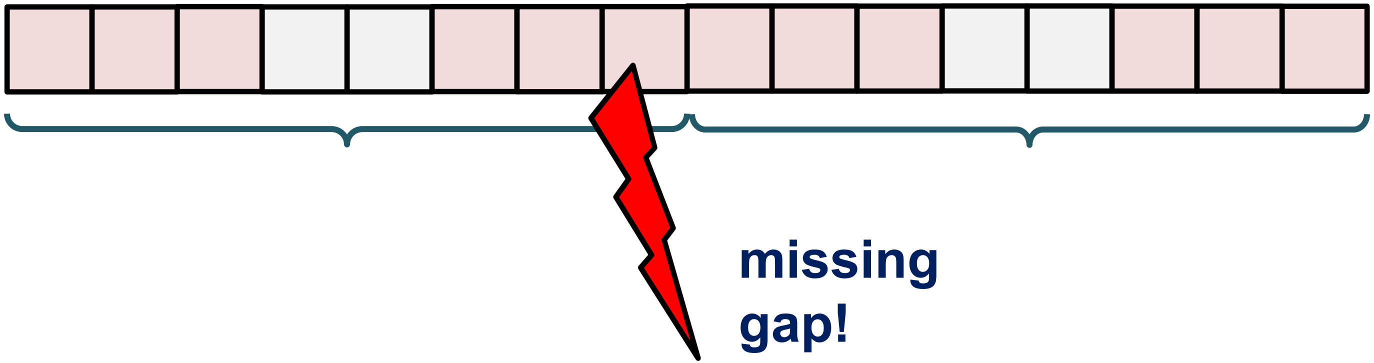
```
MPI_Datatype nt;
MPI_Type_vector(
2, 3, 5, MPI_INT, &nt);

MPI_Type_commit(&nt);
// use nt…
MPI_Type_free(&nt);
```

Caveat when using a type

- Caution: Concatenating such types in a send operation can lead to unexpected results!
- `count` argument to `send` and others must be handled with care:

`MPI_Send(buf, 2, nt,...)` with `nt` (newtype from prev. slide)



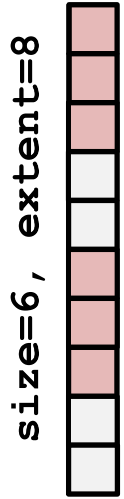
Data type size and extent

- Get the total **size** (in bytes) of datatype in a message

```
int MPI_type_size(MPI_Datatype newtype, int *size);
```

- Get the lower bound and the **extent** (span from the first byte to the last byte) of datatype

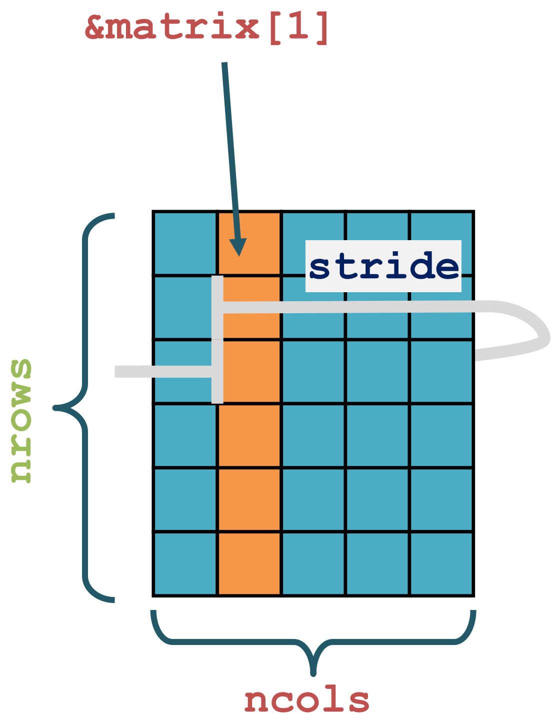
```
int MPI_type_get_extent(MPI_Datatype newtype,
                        MPI_Aint *lb,
                        MPI_Aint *extent);
```



- MPI allows to change the extent of a datatype using
  - `MPI_Type_create_resized`
    - `Sizeof`
    - `MPI_Get_address/MPI_Aint_diff`

Sending a column of a matrix in C

Row-major data layout in C → cannot use plain array




```
double matrix[30];
MPI_Datatype nt;

// count = nrows, blocklength = 1,
// stride = ncols
MPI_Type_vector(nrows, 1, ncols,
                MPI_DOUBLE, &nt);
MPI_Type_commit(&nt);

// send column
MPI_Send(&matrix[1], 1, nt, …);

MPI_Type_free(&nt);
```

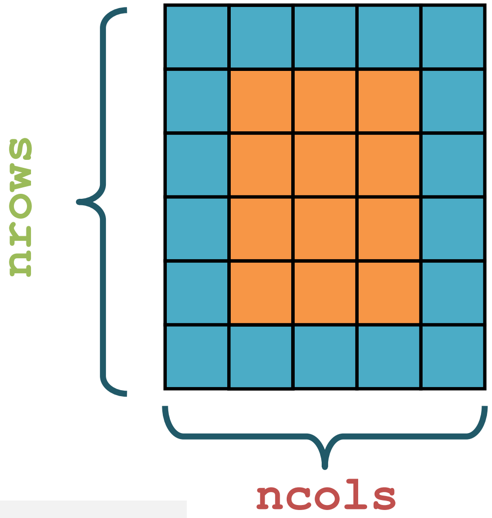
Sub-array Data Type

```
MPI_Type_create_subarray(int dims,
   int ar_sizes[], int ar_subsizes[], int ar_starts[],
   int order, MPI_Datatype oldtype, MPI_Datatype * newtype);
```

- `dims`: dimension of the array
- `ar_sizes`: array with sizes of array (dims entries)
- `ar_subsizes`: array with sizes of subarray (dims entries)
- `ar_starts`: start indices of the subarray inside array (dims entries), start at 0 (also in Fortran)
- `order`
  - row-major: `MPI_ORDER_C`
  - column-major: `MPI_ORDER_FORTRAN`

Example for a sub-array type: “bulk” of a matrix

dims	2
ar_sizes	{ncols, nrows}
ar_subsizes	{ncols-2, nrows-2}
ar_starts	{1, 1}
order	MPI_ORDER_C
oldtype	MPI_INT




```
MPI_Type_create_subarray(dims, ar_sizes, ar_subsizes,
                         ar_starts, order, oldtype, &nt);
MPI_Type_commit(&nt);
// use nt...
MPI_Send(&buf[0], 1, nt, …); // etc.
MPI_Type_free(&nt);
```

A Short List of MPI Bindings to Create Data Type

Function	Description
MPI_Type_contiguous	Creates a new data type that is a concatenation of a number of elements of an existing data type.
MPI_Type_vector	Creates a vector consisting of a number of elements of the same datatype repeated with a certain stride.
MPI_Type_indexed	Creates a new data type that consists of a specified number of blocks of arbitrary size.
MPI_Type_create_subarray	Creates a new data type that consists of an n-dimensional subarray of an n-dimensional array.
MPI_Type_create_darray	Creates a data type corresponding to a distributed, multidimensional array. It supports block, cyclic and no distribution for each dimension.
MPI_Type_create_struct	Creates an MPI datatype from a general set of datatypes, displacements, and block sizes.

There exist more bindings, not all listed here!

- Remarks to Fortran programmers:
  - Arrays in Fortran are stored in column-major order
  - It requires special care for derived datatypes (`MPI_Type_create_struct`) because of some sort of optimizations such as reordering elements of a derived datatype

Questions?