NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Introduction to Parallel Programming with MPI

Dr. Alireza Ghasemi, Dr. Georg Hager

Erlangen National High Performance Computing Center

MPI 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)

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++)

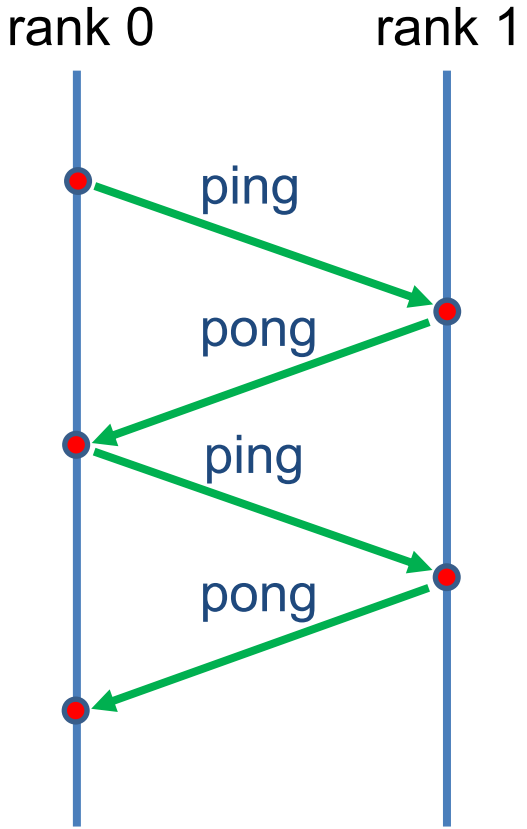
MPI datatype	C datatype
MPI_INT	int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_BOOL	_Bool
MPI_CHAR	char
MPI_BYTE	-----
MPI_PACKED	-----
and many more -> https://www.mpi-forum.org/docs/	

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	-----

The Ping-Pong example

- Consider two processes with ranks 0 and 1
- Rank 0 sends a message to rank 1
- Rank 1 receives it and sends it back to rank 0
- The operation can be repeated several times.



The Ping-Pong example

- In this example, the ping-pong is done only once.
- The code in C (or Fortran) will be shown to you. Then, think about the following questions:
  1. Does rank 0 receive the initial value of rank 1, if not, how can it be done?
  2. If we add a loop enclosing the data transfer lines of the program and repeat the operation, would the program run without any problem?
  3. Does the program have a deadlock problem? If not, can the problem be introduced to the program only by rearranging the orders of send/receive?

A deadlock is a scenario in which a process is trying to exchange data to another process but there is no match, e.g. it is ready to send a data but the other process is not and will not be prepared to accept or the opposite case, i.e. the process is waiting to receive but the other is not sending and will not send a matching message.

Single-round ping-pong in C

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
      int ierr, irank, nrank;
      MPI_Status status;
      double d=0.0;
      ierr=MPI_Init(&argc,&argv);
      ierr=MPI_Comm_rank(MPI_COMM_WORLD,&irank);
      ierr=MPI_Comm_size(MPI_COMM_WORLD,&nrank);
      if(irank==0) d=100.0;
      if(irank==1) d=200.0;
      printf("BEFORE: nrank,irank,d = %5d%5d%8.1f\n",nrank,irank,d);
      if(irank==0) {
          MPI_Send(&d,1,MPI_DOUBLE,1,11,MPI_COMM_WORLD);
          MPI_Recv(&d,1,MPI_DOUBLE,1,22,MPI_COMM_WORLD,&status);
      }
      else if(irank==1) {
          MPI_Recv(&d,1,MPI_DOUBLE,0,11,MPI_COMM_WORLD,&status);
          MPI_Send(&d,1,MPI_DOUBLE,0,22,MPI_COMM_WORLD);
      }
      printf("AFTER: nrank,irank,d = %5d%5d%8.1f\n",nrank,irank,d);
      ierr=MPI_Finalize();
}
```

Single-round ping-pong in Fortran

```
program pingpong
      use mpi_f08
      implicit none
      integer:: irank, nrank, ierr
      real(kind=8):: d=0.d0
      type(MPI_STATUS):: status
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,irank,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,nrank,ierr)
      if(irank==0) d=100.d0
      if(irank==1) d=200.d0
      write(*,'(a,2i5,f8.1)') 'BEFORE: nrank,irank,d = ',nrank,irank,d
      if(irank==0) then
          call MPI_SEND(d,1,MPI_DOUBLE_PRECISION,1,11,MPI_COMM_WORLD,ierr)
          call MPI_RECV(d,1,MPI_DOUBLE_PRECISION,1,22,MPI_COMM_WORLD,status,ierr)
      elseif(irank==1) then
          call MPI_RECV(d,1,MPI_DOUBLE_PRECISION,0,11,MPI_COMM_WORLD,status,ierr)
          call MPI_SEND(d,1,MPI_DOUBLE_PRECISION,0,22,MPI_COMM_WORLD,ierr)
      endif
      write(*,'(a,2i5,f8.1)') 'AFTER: nrank,irank,d = ',nrank,irank,d
      call MPI_FINALIZE(ierr)
end program pingpong
```

Inspection on question 3 of exercise 1

- Let's consider two changes in the solution code of exercise 1:
  1. Both processes call first MPI_SEND and then MPI_RECV
  2. The buffer is not a scalar but an array whose length is determined at run time as an argument:

```
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

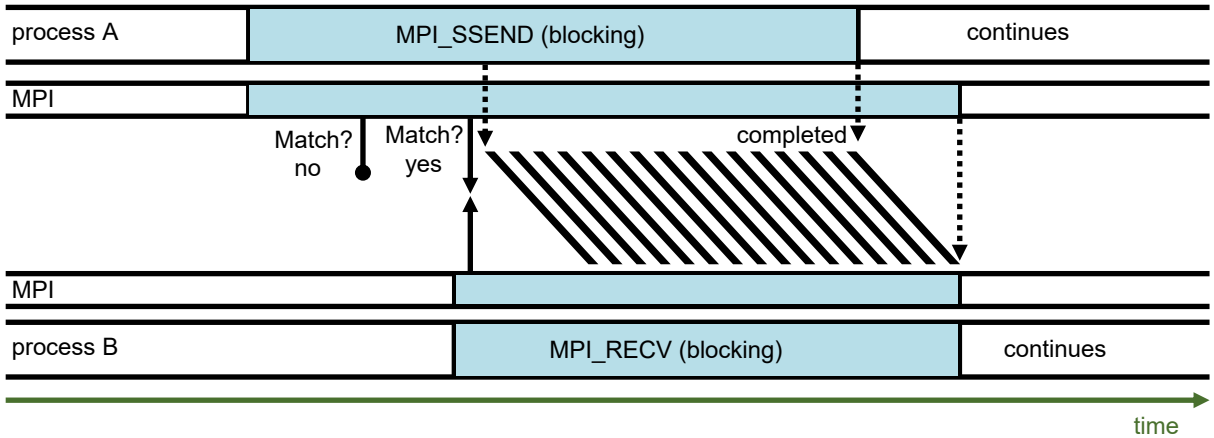
- There is only one receive communication mode:
  - Standard: MPI_Recv

Synchronous send

- 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



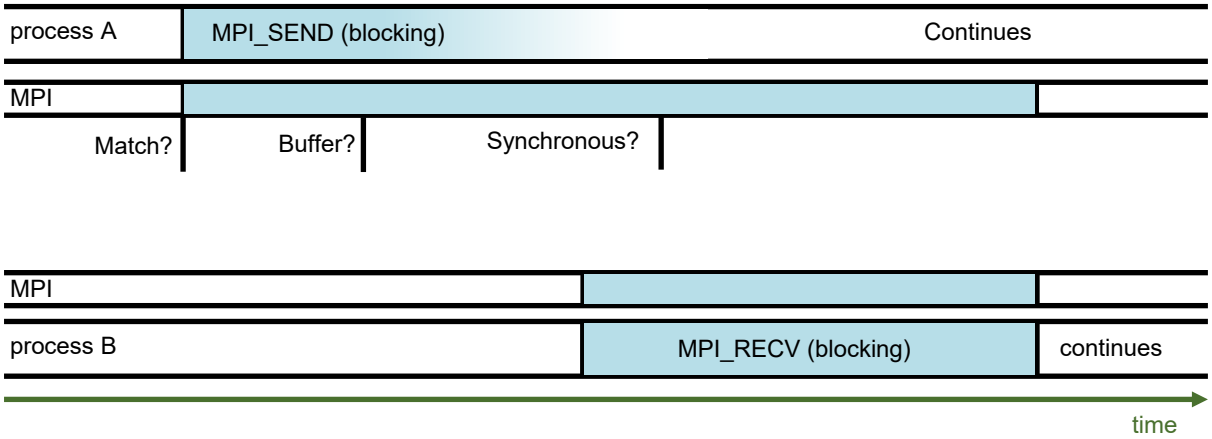
Standard send

- 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

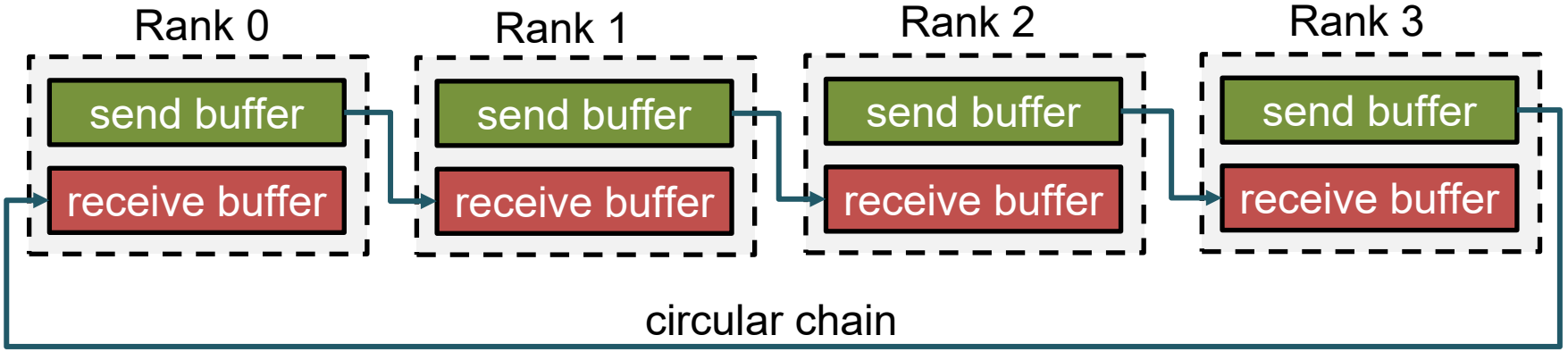
Tips

- Deadlock may occur
- Minimal transfer time

The standard send is the standard choice for you!



Example: Shift operation across a chain of processes



- Simplistic send/recv
  - pairing is 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);
```

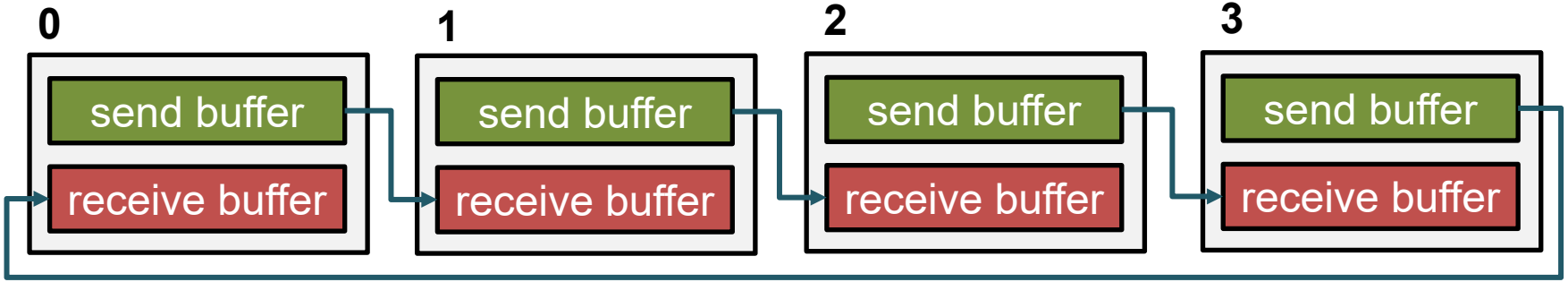
Point-to-Point Communication MPI_SENDRECV

- 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)
```

- 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

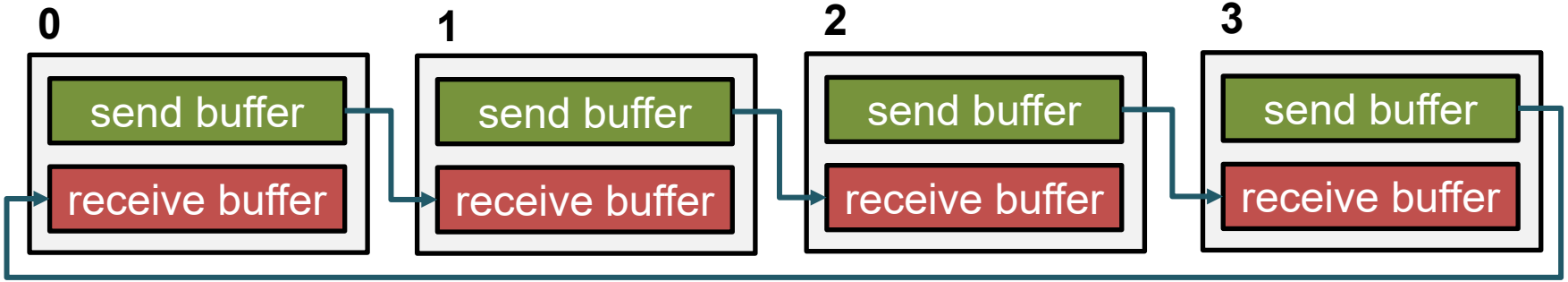
- Syntax: simple combination of send and receive arguments:

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

- 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);
```

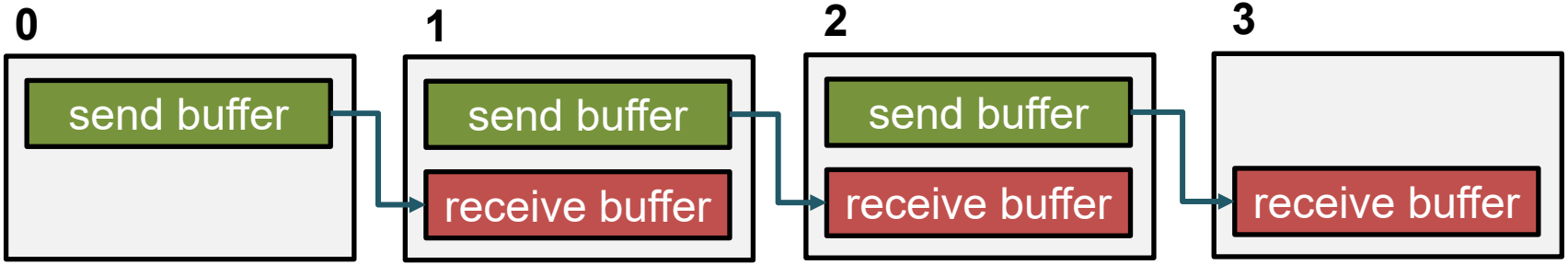
- disjoint send/receive buffers
- can have different count & data type
- blocking call



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);
```

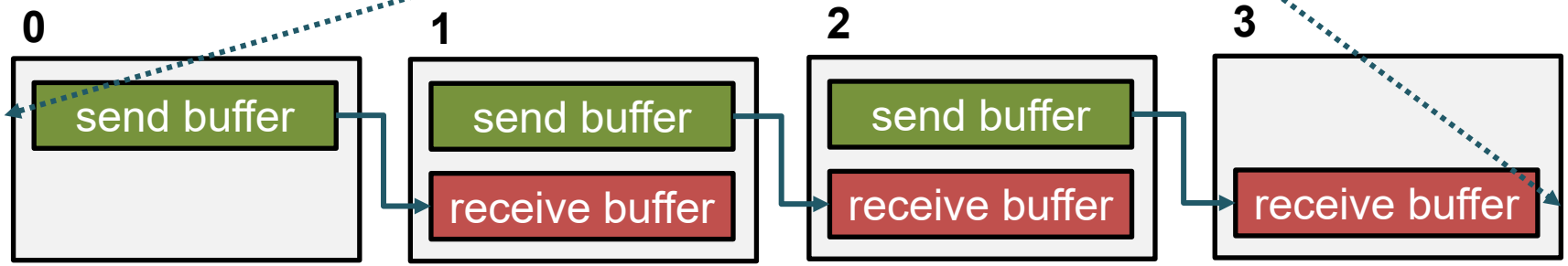


- MPI_PROC_NULL as source/destination acts as no-op
  - send/recv with MPI_PROC_NULL return immediately, buffers are not altered
- MPI_Sendrecv matches with simple *send/*recv point-to-point calls

Point-to-Point Communication MPI_SENDRECV

- useful for open chains/non-circular shifts:

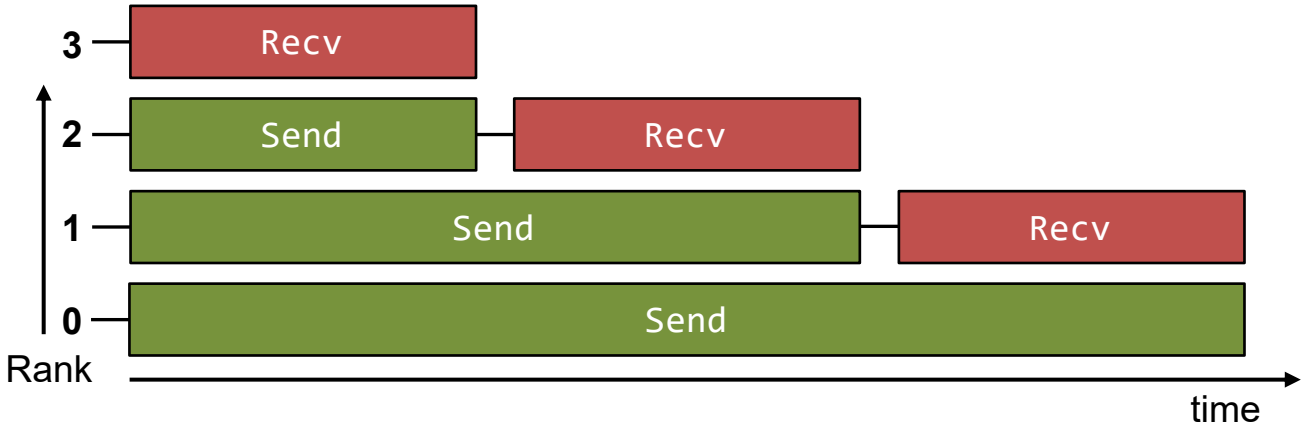
```
// Rank left from myself.
left = rank - 1; if (left < 0) { left = MPI_PROC_NULL; }
// Rank right from myself.
right = rank + 1; if (right >= size) {right = MPI_PROC_NULL; }
MPI_Sendrecv(buffer_send, n, MPI_INT, right, 0,
             buffer_recv, n, MPI_INT, left,  0, MPI_COMM_WORLD, &status);
```



- MPI_PROC_NULL as source/destination acts as no-op
  - send/recv with MPI_PROC_NULL return immediately, buffers are not altered
- MPI_Sendrecv matches with simple *send/*recv point-to-point calls

Serialization: Loss of efficiency

- Ring shift communication pattern: non-circular shifts
  - No concern over deadlock
  - Serialization
    - MPI_Send with rendezvous protocol
    - MPI_Ssend



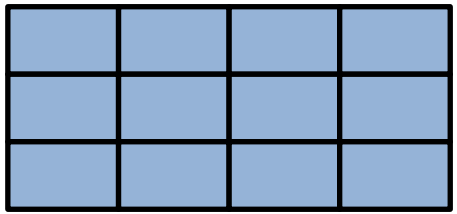
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

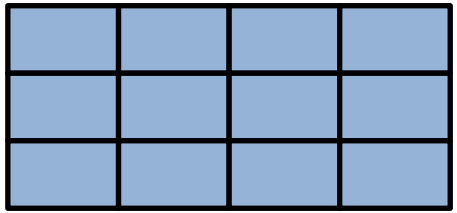
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



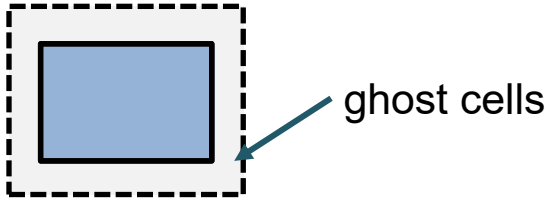
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

2D domain distributed to ranks (here 4 x 3), each rank gets one tile



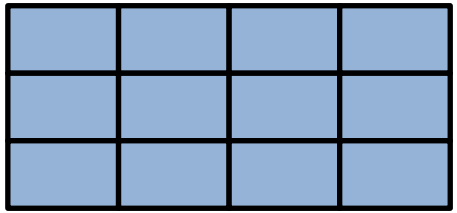
Each rank's tile is surrounded by ghost cells, representing the cells of the neighbors



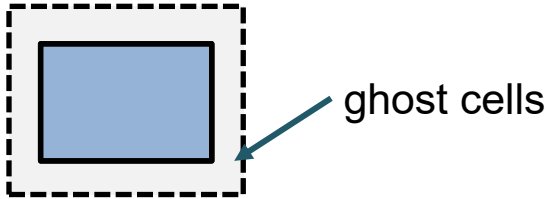
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

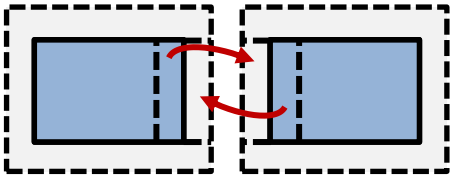
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's tile is surrounded by ghost cells, representing the cells of the neighbors



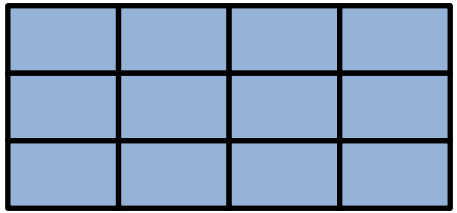
After each sweep over a tile, perform ghost cell exchange, i.e., update ghost cells with new values of neighbor cells



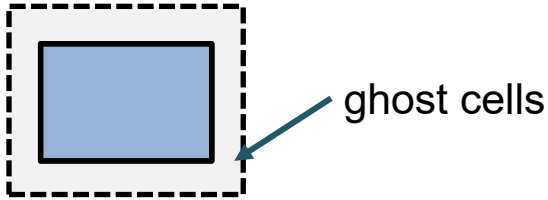
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

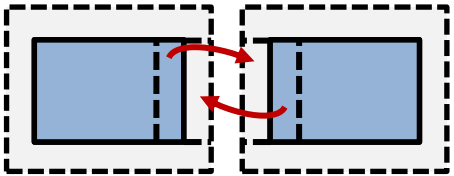
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's 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



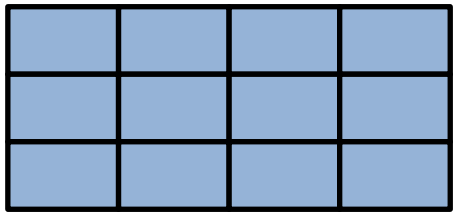
Possible implementation:

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

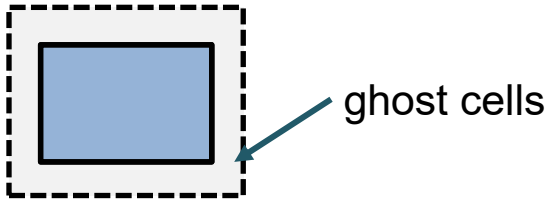
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

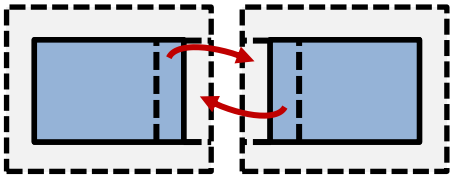
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's tile is surrounded by ghost cells, representing the cells of the neighbors

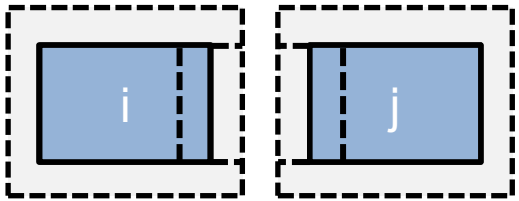


After each sweep over a tile, perform ghost cell exchange, i.e., update ghost cells with new values of neighbor cells



Possible implementation:

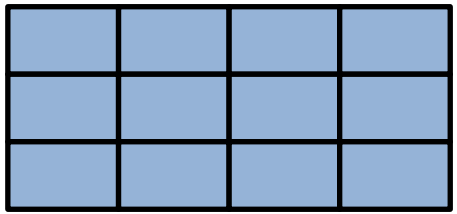
1. copy new data into contiguous send buffer
2. send to corresponding neighbor receive new data from same neighbor
3. copy new data into ghost cells



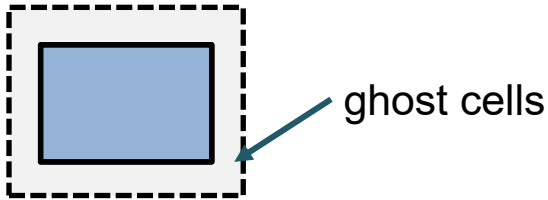
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

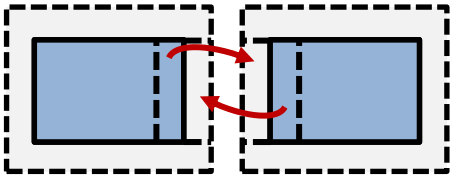
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's tile is surrounded by ghost cells, representing the cells of the neighbors

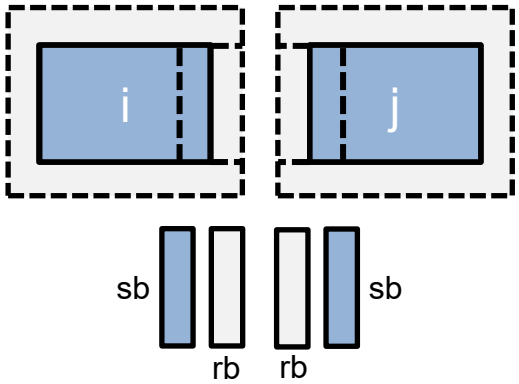


After each sweep over a tile, perform ghost cell exchange, i.e., update ghost cells with new values of neighbor cells



Possible implementation:

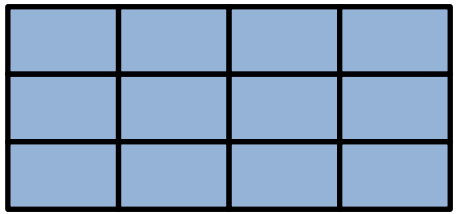
1. copy new data into contiguous send buffer
2. send to corresponding neighbor receive new data from same neighbor
3. copy new data into ghost cells



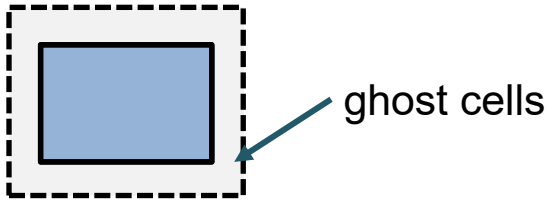
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

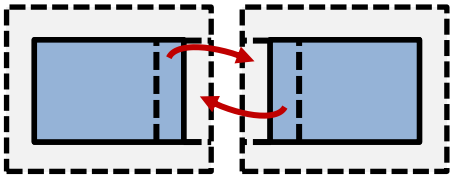
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's tile is surrounded by ghost cells, representing the cells of the neighbors

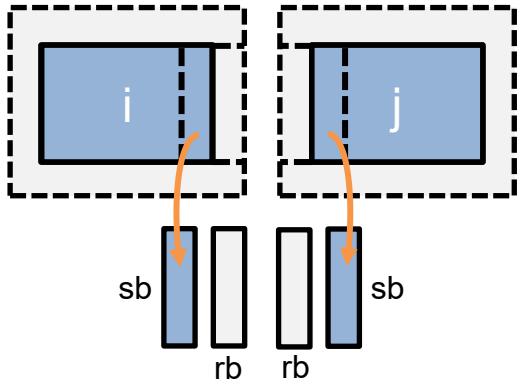


After each sweep over a tile, perform ghost cell exchange, i.e., update ghost cells with new values of neighbor cells



Possible implementation:

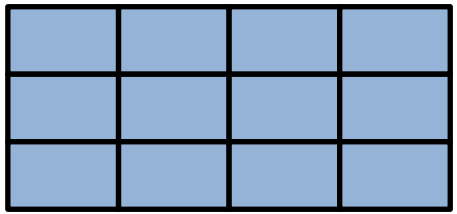
1. copy new data into contiguous send buffer
2. send to corresponding neighbor receive new data from same neighbor
3. copy new data into ghost cells



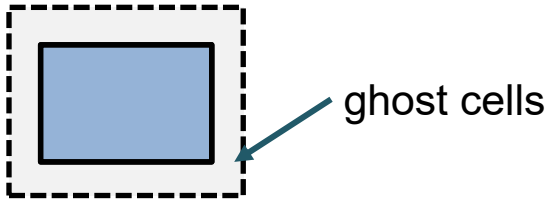
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

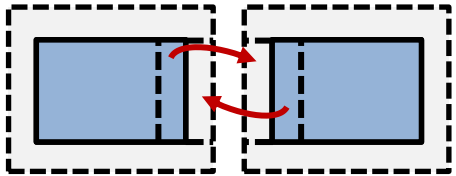
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's 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

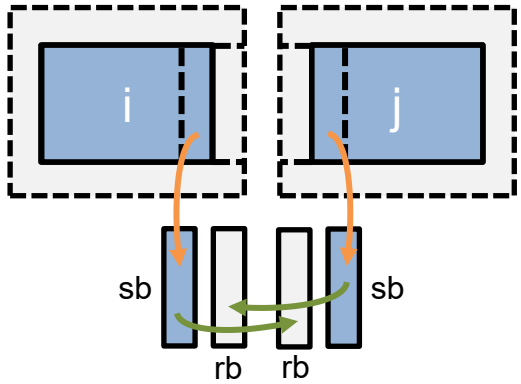


Possible implementation:

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

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

step 2



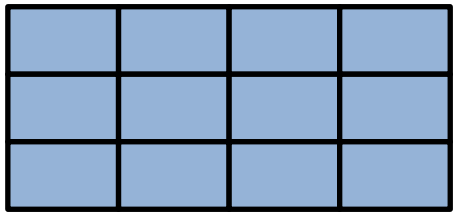

```
MPI_Sendrecv(
sb, …, i,
rb, …, i, …)
```

step 2

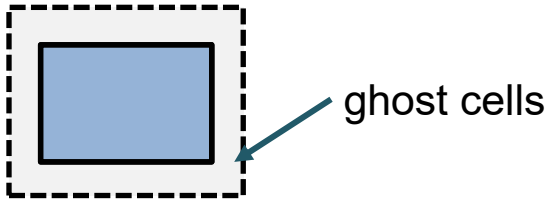
Pattern: ghost cell exchange

Many iterative algorithms require exchange of domain boundary layers

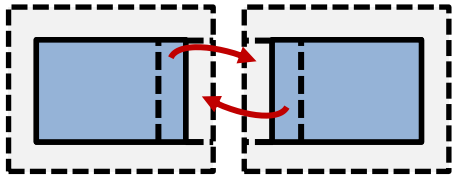
2D domain distributed to ranks (here 4 x 3), each rank gets one tile



Each rank's 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

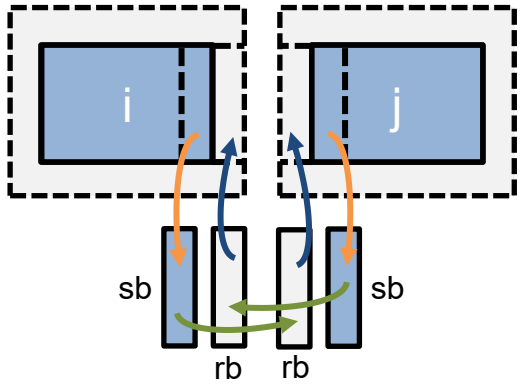


Possible implementation:

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

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

step 2




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

step 2

Combined send and recv

- MPI_Sendrecv combines a blocking send and receive into a single API call.
- Deadlocks are still possible if envelope does not match.
- Send and receive buffers must not overlap:
  - For specific cases: MPI_Sendrecv_replace

MPI_Sendrecv

- C/C++ binding:

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

- **sendbuf**: address of the first entry of the buffer to be sent
- **sendcount**: number of elements to be sent
- **sendtype**: type of the send data
- **recvbuf, recvcount, recvtype**: similarly for the receiving data
- **dest**: rank of the destination process within the communicator comm
- **sendtag** and **recvtag**: can have different values!

Exercise 3:

1. Laplace equation in 1D: $\frac{d^2V}{dx^2} = 0$ with Dirichlet BC $V(x)|_{x=0} = 0$, $V(x)|_{x=1} = 1$
   - Analytical solution is available, namely $V(x) = x$
2. Discretization and using the Jacobi method leads to stencil algorithm:

$$V_i = \frac{V_{i+1} + V_{i-1}}{2}$$

3. Domain decomposition: each domain needs the boundary values
   - should be supplied by the neighboring processes of both sides
   - leads to a double shift operation

Miscellaneous points

- Predefined macros:
  - Wild cards: MPI_ANY_SOURCE, MPI_ANY_TAG
  - MPI standard: MPI_VERSION, MPI_SUBVERSION
  - Others: MPI_SUCCESS, MPI_PROC_NULL, …
- `MPI_Status`: can be avoid MPI_STATUS_IGNORE
  - It is a structure in C/C++: MPI_SOURCE, MPI_TAG, MPI_ERROR, …
  - In Fortran:
    - It used to be an integer array of size MPI_STATUS_SIZE.
    - Now a derived-type is available as well.
- Some other C/C++ and Fortran bindings: `MPI_Probe`, `MPI_Get_count`, …

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

2. If you send two messages m_1 and m_2 from rank i to rank j using PtP bindings, is it possible that the transfer of m_2 overtakes, i.e., be received before m_1?

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

2. If you send two messages m_1 and m_2 from rank i to rank j using PtP bindings, is it possible that the transfer of m_2 overtakes, i.e., be received before m_1?

Answer: No, messages in PtP communication were in the past subject to nonovertaking rule.
In recent MPI standards, it is the default behavior but it can be overridden.

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

2. If you send two messages m_1 and m_2 from rank i to rank j using PtP bindings, is it possible that the transfer of m_2 overtakes, i.e., be received before m_1?

Answer: No, messages in PtP communication were in the past subject to nonovertaking rule.
In recent MPI standards, it is the default behavior but it can be overridden.

3. What are the major risks of synchronous send?
   Is any of the same risks a concern also for the standard send?

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

2. If you send two messages m_1 and m_2 from rank i to rank j using PtP bindings, is it possible that the transfer of m_2 overtakes, i.e., be received before m_1?

Answer: No, messages in PtP communication were in the past subject to nonovertaking rule. In recent MPI standards, it is the default behavior but it can be overridden.

3. What are the major risks of synchronous send? Is any of the same risks a concern also for the standard send?

Answer: (i) Deadlock, (ii) high latency. Both can be of concern for standard send.

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

2. If you send two messages m_1 and m_2 from rank i to rank j using PtP bindings, is it possible that the transfer of m_2 overtakes, i.e., be received before m_1?

Answer: No, messages in PtP communication were in the past subject to nonovertaking rule.
In recent MPI standards, it is the default behavior but it can be overridden.

3. What are the major risks of synchronous send?
   Is any of the same risks a concern also for the standard send?

Answer: (i) Deadlock, (ii) high latency. Both can be of concern for standard send.

4. Does the receive process have to know the **`rank`** of the send process and the **`tag`** of the message?

Quiz:

1. How many different MPI point-to-point send modes exist?

   a. 1 b. 2 c. 3 d. 4

Answer: d.

2. If you send two messages m_1 and m_2 from rank i to rank j using PtP bindings, is it possible that the transfer of m_2 overtakes, i.e., be received before m_1?

Answer: No, messages in PtP communication were in the past subject to nonovertaking rule. In recent MPI standards, it is the default behavior but it can be overridden.

3. What are the major risks of synchronous send?
   Is any of the same risks a concern also for the standard send?

Answer: (i) Deadlock, (ii) high latency. Both can be of concern for standard send.

4. Does the receive process have to know the **`rank`** of the send process and the **`tag`** of the message?

Answer: No, it is not necessary. One can use wild cards such as MPI_ANY_SOURCE and MPI_ANY_TAG.