

GWDG – Kurs
Parallel Programming with MPI

Point-to-Point Communication Exercises

Oswald Haan
oahan@gwdg.de

Outline

- Passing a message
 - importance of buffer size
- Message properties from **status** argument
 - how to inspect and how to probe
- Non-blocking message passing
 - wait and test for receiving a message with **irecv**
- Deadlocks
 - how to produce and 4 ways to avoid them
- The problem of serialization
- Collect data from all processes
 - collective communication with point-to-point messages
- Speed and latency of point-to-point communication

Exercise 1: Passing a Message



```
double precision vec(10); integer tag = 0, nsend = 3, nrecv = 3
```

```
MPI_RECV( vec(2), nrecv, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,  
MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
```

```
MPI_SEND( vec(5), nsend, MPI_DOUBLE_PRECISION, 0, tag,  
MPI_COMM_WORLD, ierr)
```

Exercise 1:

Source code in directory

`mpisexercises/[f,c,py]/MPI-p2p`

Message passing programs

`p2p_vector.f` `p2p_vector.c` `p2p_vector.py`

Exercise 1: Comments to `p2p_vector.f`

- types of send and receive buffers must coincide
- MPI datatypes must conform to types of these buffers

```
double precision vec(10)
```

```
MPI_SEND(vec(...), ..., MPI_DOUBLE_PRECISION, ...
```

```
MPI_RECV(vec(...), ..., MPI_DOUBLE_PRECISION, ...
```

Exercise 1: Comments to `p2p_vector.c`

- types of send and receive buffers must coincide
- MPI datatypes must conform to types of these buffers
- Address of buffers as first arguments : `&vec[...]`
(parameter passing by value in C)

```
double vec[10];
```

```
MPI_Send(&vec[...], ..., MPI_DOUBLE, ...
```

```
MPI_Recv(&vec[...], ..., MPI_DOUBLE, ...
```

Exercise 1: Comments to `p2p_vector.py`

Datatype information is attached to the numpy object instantiated by

```
vec = myid*np.arange(1,11,dtype = np.float64)
```

The array section object `vec[startsend:startsend+nsend]` includes information about datatype and number of elements, therefore no count and datatype arguments in mpi4py calls to send and receive:

```
comm.Send(vec[startsend:startsend+nsend],  
          dest=0, tag=11)
```

```
comm.Recv(vec[startrecv:startrecv+nrecv],  
          source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG)
```

Exercise 1:

What happens if $nrecv < nsend$?

What if $nrecv > nsend$?

Blocking Receive Operation: MPI_RECV

C:

```
int MPI_Recv(void *buf, int count,  
            MPI_Datatype datatype,  
            int src, int tag, MPI_Comm comm,  
            MPI_Status *stat)
```

Fortran:

```
MPI_RECV(buf, count, datatype, src, tag,  
         comm, stat, ierr)  
<type> buf(*)  
INTEGER count, datatype, src, tag, comm, ierr  
INTEGER stat(MPI_STATUS_SIZE)
```

Blocking Receive Operation: MPI_RECV

mpi4py:

```
obj = comm.recv(buf=None, source=MPI.ANY_SOURCE,  
                tag=MPI.ANY_TAG, status=stat)  
obj : python object, source, tag: integer  
stat: MPI.status()
```

```
comm.Recv(ar, source=MPI.ANY_SOURCE,  
          tag=MPI.ANY_TAG, status=stat)  
ar : numpy array; dest, tag: integer  
stat MPI.status()
```

Exercise 2: Retrieving Message Properties

Retrieving Message Properties from **stat**

C

Fortran

mpi4py

type	<code>MPI_Status stat</code>	<code>integer stat(MPI_STATUS_SIZE)</code>	<code>stat = MPI.Status()</code>
source	<code>stat.MPI_SOURCE</code>	<code>stat(MPI_SOURCE)</code>	<code>stat.Get_source()</code>
tag	<code>stat.MPI_TAG</code>	<code>stat(MPI_TAG)</code>	<code>stat.Get_tag()</code>
error	<code>stat.MPI_ERROR</code>	<code>stat(MPI_ERROR)</code>	<code>stat.Get_error()</code>
count	<code>MPI_Get_count (&stat, datatype, &count)</code>	<code>call MPI_GET_COUNT (stat, datatype, count, ierr)</code>	<code>stat.Get_elements (datatype)</code>
size	---	---	<code>stat.Get_size()</code>

Exercise 2: Retrieving Message Properties

Source code in directory

`mpisexercises/[f,c,py]/MPI-p2p`

Message passing programs

`p2p_status.f` `p2p_status.c` `p2p_status.py`

- Includes call to `MPI_GET_COUNT` to retrieve number of received elements .
- Add code to retrieve source and tag of received message

Exercise 3: Using MPI_PROBE

Retrieve number of elements in the pending message with MPI_PROBE:

Copy `p2p_status.<>` to `p2p_probe.<>`

Modify `p2p_probe.<>`:

Call MPI_PROBE before calling MPI_RECV:

```
C      MPI_Probe(MPI_ANY_SOURCE,  
               MPI_ANY_TAG, comm, &stat);  
Fortran call MPI_PROBE(MPI_ANY_SOURCE,  
                       MPI_ANY_TAG, comm, stat, ierr)  
mpi4py comm.Probe(MPI.ANY_SOURCE,  
                  MPI.ANY_TAG, stat)
```

Exercise 3: Using MPI_PROBE

Determine **nrecv** parameter to be used in the MPI_RECV call from the number of elements in the pending message by

```
C      MPI_Get_count(&stat, MPI_DOUBLE, &nrecv);
```

```
Fortran call MPI_GET_COUNT(stat,  
                        MPI_DOUBLE_PRECISION, nrecv, ierr)
```

```
mpi4py nrecv = stat.Get_elements(MPI.DOUBLE)
```

Exercise: Non-Blocking RECV

Source code in directory

`mpisexercises/[f,c,py]/MPI-p2p`

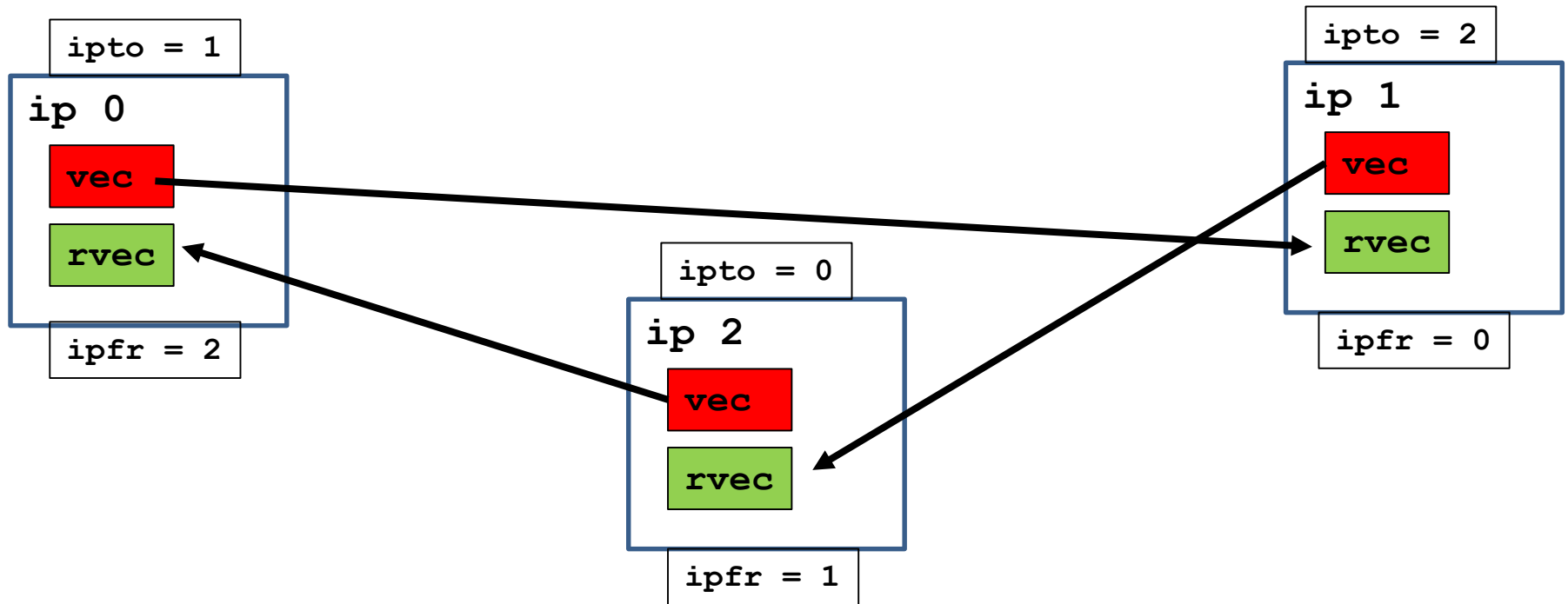
Non-blocking recv in programs

`p2p_irecv.f` `p2p_irecv.c` `p2p_irecv.py`

- Inspect and run the program.
- Modify the program by using the `MPI_WAIT` function instead of `MPI_TEST`
- What happens, if you omit the `MPI_WAIT` ?

Exercise 4:

Clockwise Ring Shift



```
call MPI_SEND( vec, n1, MPI_INTEGER, ipto, 0, MPI_COMM_WORLD, ierr )  
call MPI_RECV( rvec, n1, MPI_INTEGER, ipfr, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr )
```

Exercise 4: Deadlock

Ring shift of long messages leads to deadlock:

`p2p_deadlock.f`

`p2p_deadlock.c`

`p2p_deadlock.py`

Run the example program and determine, at which size of the message a deadlock occurs.

Compare openmpi and intel-mpi

Replace `MPI_SEND` by `MPI_SSEND`

Exercise 4: Avoiding the Deadlock – Change order

1st solution: Change order of send and receive for one process.

Copy `p2p_deadlock.f` to `p2p_no_deadlock_1.f`

Use the following modification:

```
if (myid.eq.0) then
  call MPI_RECV( rvec, n1, MPI_INTEGER, ipfr, 0,
:               MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr )
  call MPI_SEND( vec, n1, MPI_INTEGER, ipto, 0,
:               MPI_COMM_WORLD, ierr )
else
  call MPI_SEND( vec, n1, MPI_INTEGER, ipto, 0,
:               MPI_COMM_WORLD, ierr )
  call MPI_RECV( rvec, n1, MPI_INTEGER, ipfr, 0,
:               MPI_COMM_WORLD, stat, ierr )
end if
```

Exercise 4: Avoiding the Deadlock: BSEND

2nd solution: Using Buffered send:

Fortran:

Copy `p2p_deadlock.f` to `p2p_no_deadlock_2.f`
modify code, using:

```
integer temp(110000000)
integer bsize
bsize = 4*n1+MPI_BSEND_OVERHEAD
! Buffer Size in Byte including overhead
call MPI_BUFFER_ATTACH(temp,bsize,ierr)
...
call MPI_BSEND( vec, n1,...
...
call MPI_BUFFER_DETACH(temp,bsize,ierr)
```

Exercise 4: Avoiding the Deadlock: BSEND

2nd solution: Using Buffered send:

C:

Copy `p2p_deadlock.c` to `p2p_no_deadlock_2.c`
modify code, using:

```
int temp[110000000];
char *tempptr
int bsize = 4*n1 + MPI_BSEND_OVERHEAD
// Buffer Size in Byte including overhead
MPI_Buffer_attach(temp, bsize);
    ..
MPI_Bsend( vec, n1, ...;
    ...
MPI_Buffer_detach(&tempptr, &bsize);
```

Exercise 4: Avoiding the Deadlock: BSEND

2nd solution: Using Buffered send:

mpi4py:

Copy `p2p_deadlock.py` to `p2p_no_deadlock_2.py`
modify code, using:

```
    ntemp = n1 + MPI.BSEND_OVERHEAD
# Buffer length including overhead
    temp = np.empty(ntemp, dtype = np.int)
    MPI.Attach_buffer(temp)
    ...
    comm.Bsend(vec, dest=ipto)
    ...
    MPI.Detach_buffer()
```

Exercise 4: Avoiding the Deadlock: SENDRECV

3rd solution using **SENDRECV**

Copy `p2p_deadlock.f` to `p2p_no_deadlock_3.f`

Modify code:

FORTRAN:

```
call MPI_SENDRECV( vec,n1,MPI_DOUBLE_PRECISION,ipto,tag,
:                 rvec,n1,MPI_DOUBLE_PRECISION,ipfr,tag,
:                 MPI_COMM_WORLD,stat,ierr )
```

C:

```
MPI_Sendrecv( vec,n1,MPI_DOUBLE,ipto,tag,
              rvec,n1,MPI_DOUBLE,ipfr,tag,
              MPI_COMM_WORLD, &stat);
```

mpi4py:

```
comm.Sendrecv(sendbuf=vec,dest=ipto,
              recvbuf=rvec,source=ipfr)
```

Exercise 4: Avoiding the Deadlock: ISEND

4th solution: Using non-blocking ISEND:

Fortran:

Copy `p2p_deadlock.f` to `p2p_no_deadlock_4.f`
modify code, using:

```
integer req
...
call MPI_ISEND( sendvec, n1, MPI_DOUBLE_PRECISION,
ipto, tag, MPI_COMM_WORLD, req, ierr )
...
no need to modify receive call
...
call MPI_WAIT( req, stat, ierr )
```

Exercise 4: Avoiding the Deadlock: ISEND

4th solution: Using non-blocking Irecv:

C:

Copy `p2p_deadlock.c` to `p2p_no_deadlock_4.c`
modify code, using:

```
MPI_Request req;
...
MPI_Irecv( recvvec, n1, MPI_DOUBLE, ipfr, tag,
MPI_COMM_WORLD, &req);
...
no need to modify send call
...
MPI_Wait( &req, &stat );
```

Exercise 4: Avoiding the Deadlock: ISEND

4th solution: Using non-blocking Isend:

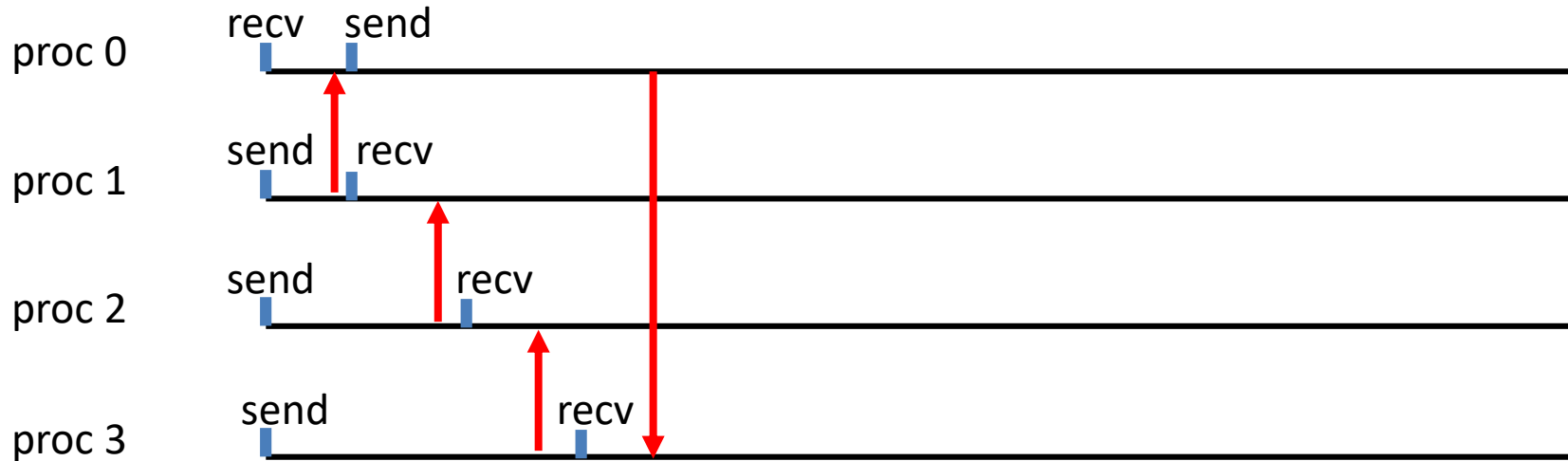
mpi4py:

Copy `p2p_deadlock.py` to `p2p_no_deadlock_4.py`
modify code, using:

```
req = comm.Isend(sendvec, dest=ipto)
...
no need to modify send call
...
MPI.Request.Wait(req)
```

Exercise 4: Comparison of Solutions

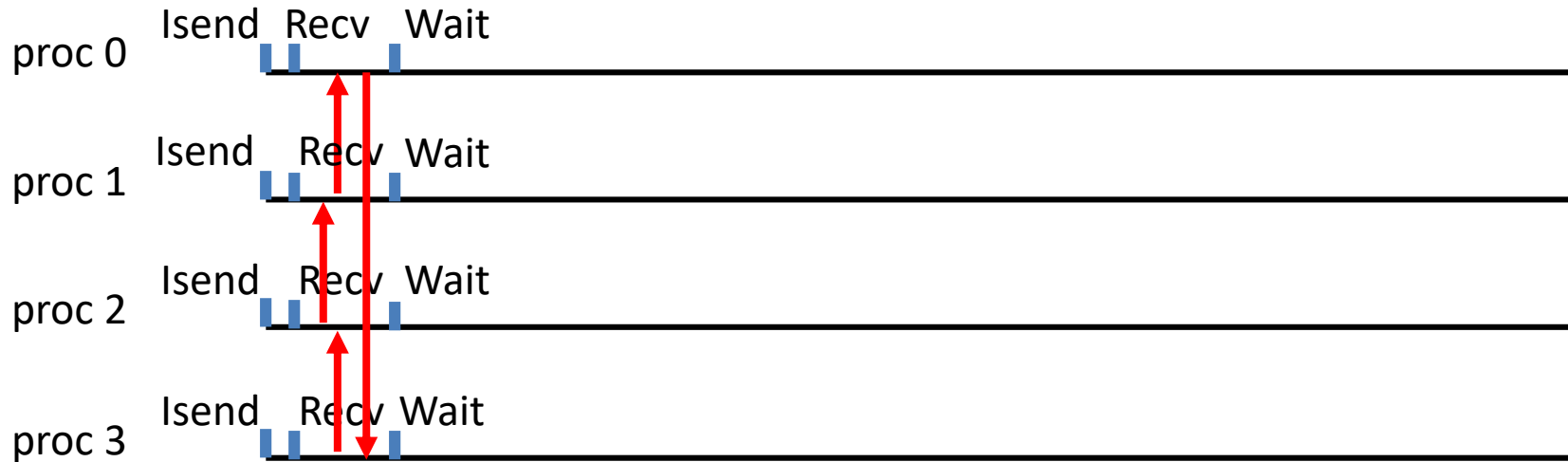
1th solution: Change order



serialized communication
time is linear in number of procs

Exercise 4: Comparison of Solutions

4th solution: Isend



simultaneous communication
time depends on network capabilities

Compare the performance of solutions 1 and 4
using the **time** command:

```
>time mpirun -n 4 ./a.out
```

Exercise 5: collect output in program `prime_clct`

In Program `prime_clct.<>` every process prints the number `nd` of divisors found in its set of tested integers.

Modify the program `prime_clct.<>`: let process 0 collect and print the numbers `nd` of divisors found in the different processes.

- define an integer array `numdiv`
- every process sends its `nd` to process 0:

```
call MPI_SEND(nd,1, MPI_INTEGER, 0,tag,  
             MPI_COMM_WORLD, ierr )
```
- process 0 stores `nd` from process `ip` in numbers in `numdiv(ip)`

```
do ip = 0 , nproc-1  
  call MPI_RECV(numdiv(ip),1,MPI_INTEGER,ip, tag,  
              MPI_COMM_WORLD,stat,ierr)  
end do
```
- process 0 prints `numdiv`

Exercise 5: collect output in program prime_clct

What happens, if instead of standard send **MPI_SEND** synchronous send **MPI_SSEND** is used ?

Solution for Exercises

If you have tried hard to perform the required exercises and the programs still don't work, you are allowed to look into the directories

`~ohaan/mpisolutions/f`

`~ohaan/mpisolutions/c`

`~ohaan/mpisolutions/py`

where you will find the completed programs for some exercises

Exercise 6: Speed and Latency of Communication

Point-to-Point Ping-Pong with [pp_mpi.f](#)

- Compile with `make pp`
- Communication parameters on local frontend node

```
mpirun -n 4 ./pp_mpi.exe
```

- Communication parameters between different cluster nodes

```
sbatch job.script
```

```
#!/bin/bash
#SBATCH -t 00:10:00
#SBATCH -N 2
#SBATCH --ntasks-per-node=2
#SBATCH -p medium
##SBATCH --reservation mpi-kurs

source ../modules_intel.x
mpirun ./pp_mpi.exe
```