

# Collective Communication

Pedro Ojeda, Xin Li, Joachim Hein

HPC2N, Umeå University

LUNARC & Centre of Mathematical Sciences

Lund University

PDC, KTH

# Overview

- Introduces the most important collective communication operations within MPI
- Discusses their MPI application interfaces

# Collective calls I

- So far discussed: point-to-point communication
  - One sending process
  - One receiving process
- Often required: Communication in a group of processes
- Examples:
  - Distribution of simulation parameters
  - Averages of distributed data structures

# Collective calls II

- All collective calls can be built up of point-to-point calls
- On a well tuned system you should not be able to beat the performance of a collective using MPI point-to-point
  - Some systems offer dedicated hardware for collectives (e.g. IBM BlueGene L and P)<sup>1</sup>
- Whenever there is a collective: Use it
- There are no non-blocking collective calls in MPI 2.x
  - Introduced in MPI 3.0

<sup>1</sup>The Int. Jour. of High Perf. Comp., S. Kumar (2014)

# Collective and point-to-point calls differences

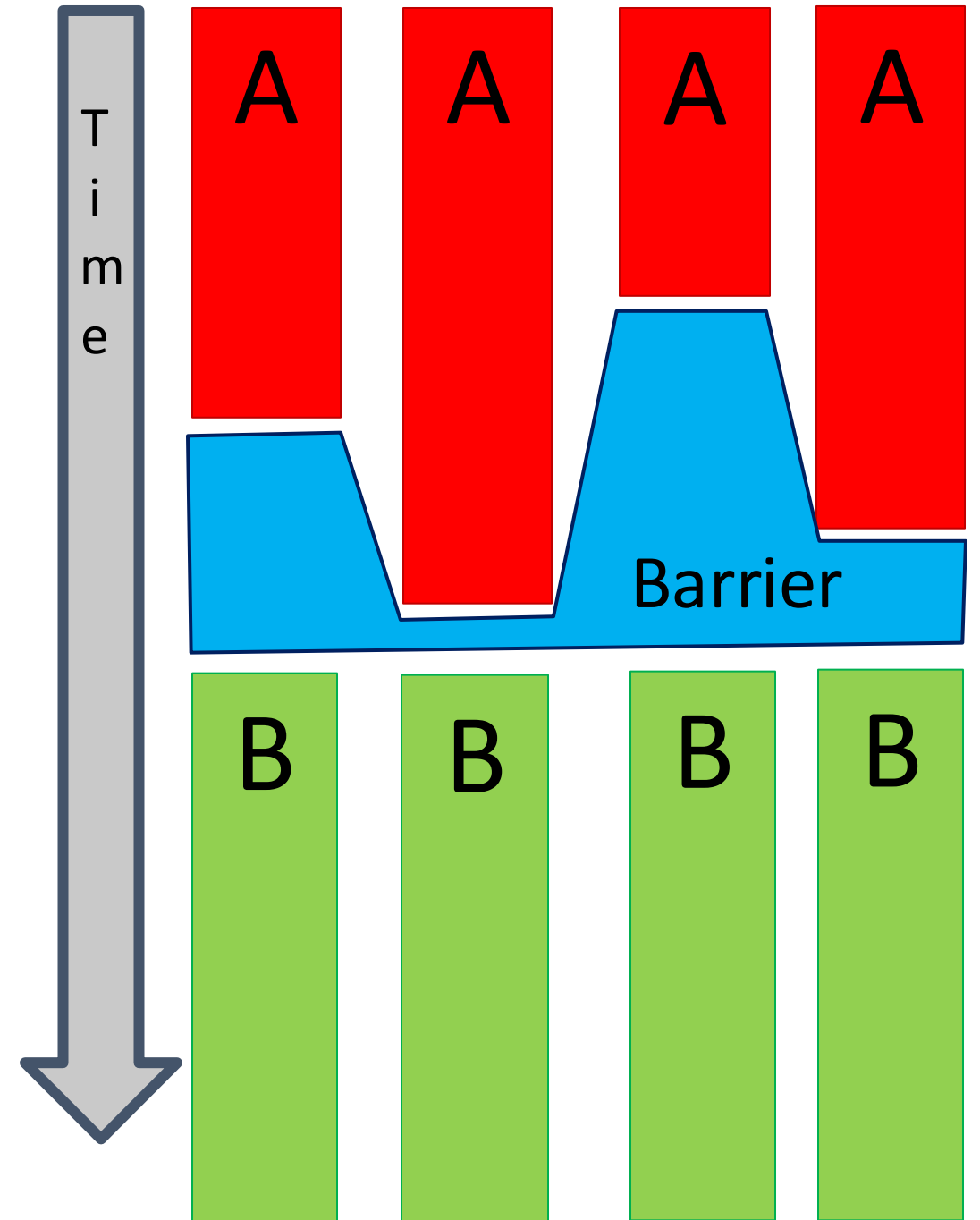
- The collective function must be called by all the processes
- Arguments passed to the collective functions by the processes need to match their counterparts (for instance, the root rank)
- Point-to-point calls use tags and communicators to establish message transfer. Collective calls are matched by the order in which they appear
- Don't use the same buffer for input and output

# Barrier

- Program (each Proc:)

```
Call calc_A()  
Call barrier(comm)  
Call calc_B()
```

- Tasks wait in barrier until last finished `calc_A`



# MPI\_Barrier

In C:

```
int MPI_Barrier(MPI_Comm comm)
```

In Fortran 90:

```
MPI_BARRIER(COMM, IERROR)  
INTEGER:: COMM, IERROR
```

In Python:

```
comm.barrier()
```

- Typically no good reason to use, except
  - Performance measurement/Benchmarking
  - Single sided communication – not in this course

In C	In Fortran
<pre>#include "mpi.h" #include &lt;stdio.h&gt; int main(int argc, char *argv[]) { int myrank, numprocs, ierr; MPI_Comm my_comm; MPI_Init(&amp;argc,&amp;argv); my_comm = MPI_COMM_WORLD; MPI_Comm_size(my_comm,&amp;numprocs); MPI_Comm_rank(my_comm,&amp;myrank);  <b>ierr</b> = MPI_Barrier(my_comm);  MPI_Finalize(); return 0; }</pre>	<pre>program main use mpi <b>implicit none</b>  integer myrank, numprocs, ierr, my_comm call MPI_INIT(ierr) my_comm = MPI_COMM_WORLD call MPI_COMM_RANK(my_comm, myrank, ierr) call MPI_COMM_SIZE(my_comm, numprocs, ierr)  call MPI_Barrier(my_comm, <b>ierr</b>)  call MPI_FINALIZE(ierr)  end</pre>



## In Python

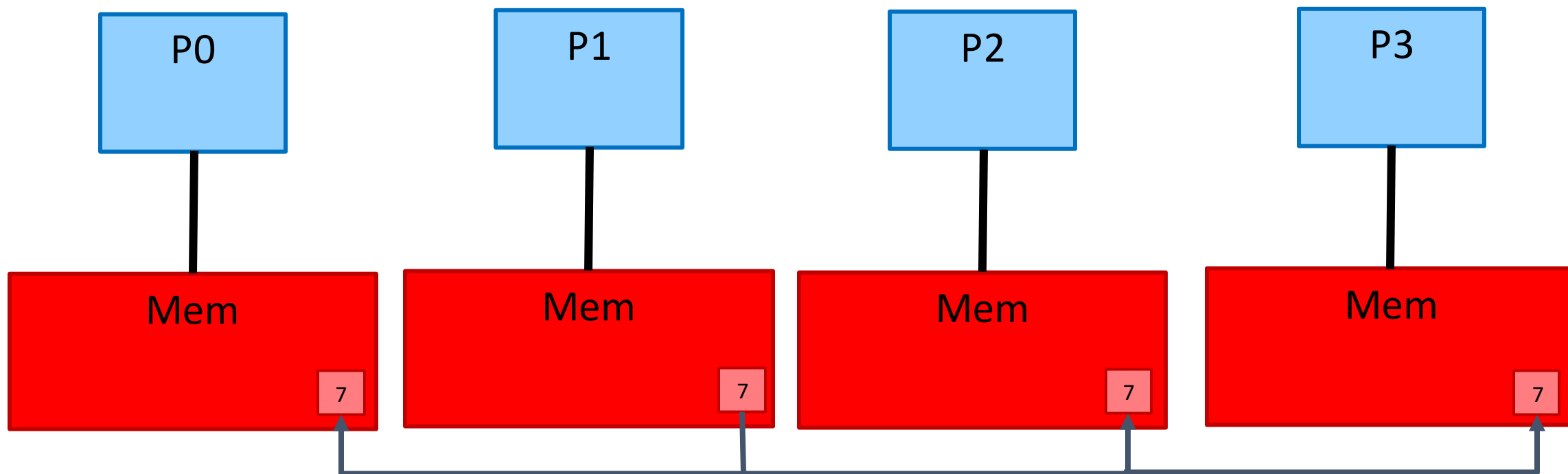
```
from mpi4py import MPI

comm = MPI.COMM_WORLD

numprocs = comm.Get_size()
myrank = comm.Get_rank()

comm.Barrier()
```

# Broadcast



# MPI\_Bcast in C

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

- `buf`: address of buffer (send on root, receive else)
- `count`: number of data
- `datatype`: type of data
- `root`: root rank – rank of task sending data
- `comm`: communicator – every task in comm gets data

**Remark:** Depending on your rank, this is a send **or** receive

# MPI\_Bcast in Fortran 90

```
MPI_BCAST (BUF, COUNT, DATATYPE, ROOT, COMM, & IERROR)  
  <type>::  BUF  
  INTEGER::  COUNT, DATATYPE, ROOT, COMM, IERROR
```

- buf: buffer (send on root, receive else)
- count: number of data
- datatype: type of data
- root: root rank – rank of task sending data
- comm: communicator – every task in comm gets data

**Remark:** Depending on your rank, this is a send **or** receive

# bcast in Python

```
comm.bcast(obj, root=root)
```

- `obj` : the Python object to broadcast
- `root` : root rank – rank of task sending data

**Remark:** Depending on your rank, this is a send **or** receive

The Python object will be returned by `bcast`. Example:

```
data = comm.bcast(data, root=0)
```

In C	In Fortran
<pre> #include "mpi.h" #include &lt;stdio.h&gt;  int main(int argc, char *argv[]) { int myrank, numprocs, ierr, alpha; MPI_Comm my_comm; MPI_Init(&amp;argc,&amp;argv); my_comm = MPI_COMM_WORLD; MPI_Comm_size(my_comm,&amp;numprocs); MPI_Comm_rank(my_comm,&amp;myrank);  if (myrank == 0) {     printf("Type some integer\n");     scanf("%d", &amp;alpha); }  <b>MPI_Bcast</b>(&amp;alpha, 1, MPI_INT, 0, my_comm); printf("Value of alpha on each rank %d\n", alpha);  MPI_Finalize(); return 0; } </pre>	<pre> program main use mpi implicit none  integer myrank, numprocs, ierr, alpha, my_comm call MPI_INIT(ierr) my_comm = MPI_COMM_WORLD call MPI_COMM_RANK(my_comm, myrank, ierr) call MPI_COMM_SIZE(my_comm, numprocs, ierr)  if (myrank .eq. 0) then     print *, 'Type some integer'     read(*,*) alpha endif  <b>call MPI_BCAST</b>(alpha, 1, MPI_INTEGER, 0, my_comm, ierr) print *, 'Value of alpha on each rank', alpha  call MPI_FINALIZE(ierr) end </pre>

## In Python

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

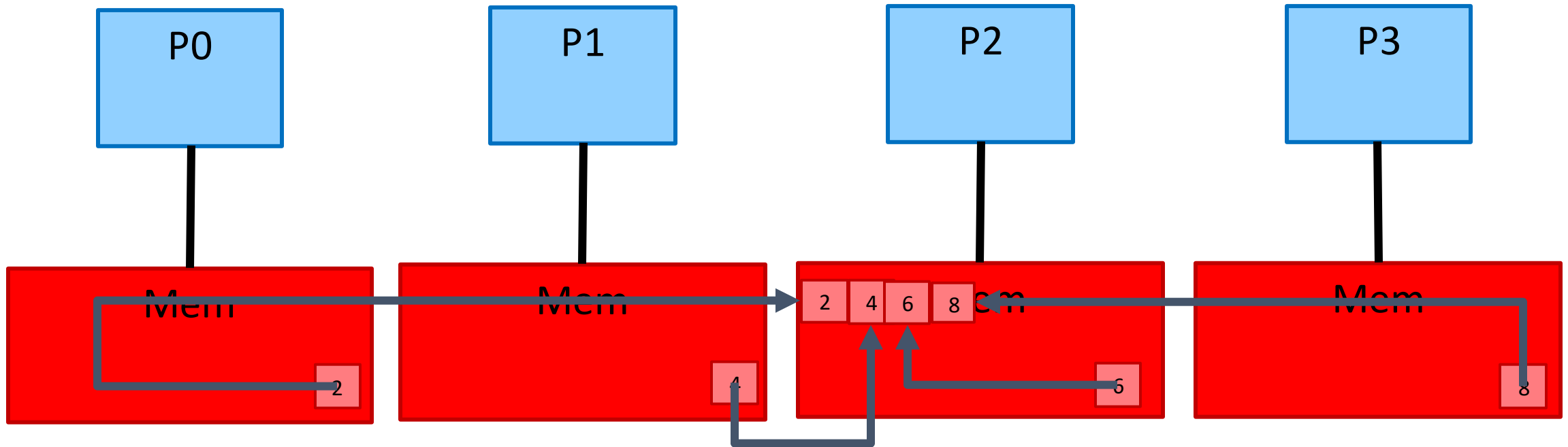
numprocs = comm.Get_size()
myrank = comm.Get_rank()

if myrank == 0:
    alpha = int(input("Type some integer:\n"))
else:
    alpha = None

alpha = comm.bcast(alpha, root=0)

print("Value of alpha on rank {:d} is: {:d} ".format(myrank,
alpha))
```

# Gather



- Collects data from all processors into a large array on root
- Order: 1<sup>st</sup> all data from rank 0, followed by all data from rank 1, followed by all data from rank 2, ...
- This is not a scalable call - think again if you want to use
  - On 10000 cores you easily run out of memory



# MPI\_Gather in C

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype  
sendtype, void* recvbuf, int recvcount, MPI_Datatype  
recvtype, int root, MPI_Comm comm)
```

- sendbuf: address of send buffer
- sendcount: number of elements in send buffer
- sendtype: type of data
- recvbuf: address of receive buffer (only root)
- recvcount: number of data received from **each** task
- recvtype: type of data
- root: root rank – rank collecting the data
- comm: communicator – every task has to send

# MPI\_Gather in Fortran 90

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,  
           RECVTYPE, ROOT, COMM, IERROR)
```

```
<type>:: SENDBUF, RECVBUF
```

```
INTEGER:: SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,  
          COMM, IERROR
```

- SENDBUF: send buffer
- SENDCOUNT: number of elements in send buffer
- SENDTYPE: type of data
- RECVBUF: receive buffer (significant only on root)
- RECVCOUNT: number of data received from **each** task
- RECVTYPE: type of data
- ROOT: root rank – rank collecting the data
- COMM: communicator – every task has to send

# gather in Python

```
comm.gather(obj, root=root)
```

- `obj` : the Python object to gather
- `root` : root rank – rank collecting the data

A `list` will be returned by `gather`.

Example:

```
a = comm.Get_rank()
```

```
b = comm.gather(a, root=0)
```

`b` on the root rank is a `list` containing `a` from all MPI processes.

In C	In Fortran
<pre> root = 0; counts = 3; //nr. of elements to be sent/received size_recvbuf = counts * numprocs; //size receiving buffer //allocating receiving buffer: counts elements per rank if( myrank == 0) recvbuf = malloc(size_recvbuf * sizeof(float));  //initializing sending buffer float sendbuf[4]={1.0*myrank,2.0*myrank,3.0*myrank,4.0*myrank}; <b>MPI_Gather</b>(sendbuf,counts,MPI_FLOAT,recvbuf,counts,MPI_FLOAT,root,my_comm);  if (myrank == 0) {     for( i = 0; i &lt; size_recvbuf; i++)         printf("Array %.3f \n", recvbuf[i]);      free(recvbuf); } </pre>	<pre> root = 0 counts =3 !nr. of elements to be sent/received size_recvbuf = counts * numprocs !size receiving buffer !allocating receiving buffer: 2 elements per rank if (myrank == 0 ) allocate( recvbuf(size_recvbuf) )  !initializing sending buffer sendbuf = (/1.0*myrank,2.0*myrank,3.0*myrank,4.0*myrank /) call <b>MPI_Gather</b>(sendbuf,counts,MPI_REAL,recvbuf,counts,MPI_REAL,root,my_comm, ierr)  if(myrank == 0) then     do i=1,size_recvbuf         print *, "Array", recvbuf(i)     enddo     deallocate ( recvbuf ) endif </pre>

## In Python

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

numprocs = comm.Get_size()
myrank = comm.Get_rank()

send_data = [1.0*myrank, 2.0*myrank, 3.0*myrank, 4.0*myrank]
send_count = 3

recv_data = comm.gather(send_data[:send_count], root=0)

if myrank == 0:
    for entry in recv_data:
        for number in entry:
            print("Array: {:.3f} ".format(number))
```

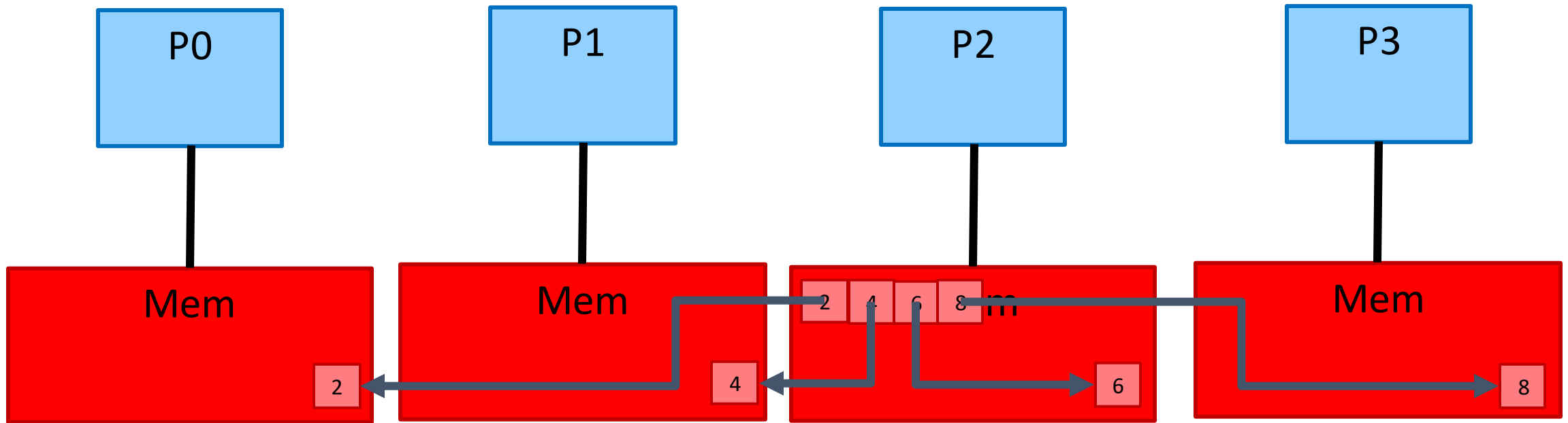
# Deadlock in collectives

- Consider the following code snippet:
- At least on rank 6, the gather can not finish, it needs input from rank 5
- Rank 6 will wait forever

```
if (rank /= 5) then
  call CalcA(a)
  call MPI_GATHER(a, 7, &
    MPI_REAL, b, 7, MPI_REAL, &
    6, mcomm, merror)
else
  call CalcB(a)
endif

if (rank == 6) then
  call Vcalc(b)
endif
```

# Scatter



- Distributes data from a large array on root
- Order: 1<sup>st</sup> lot of data go to rank 0, followed by all data for rank 1, followed by all data for rank 2, ...
- “Inverse” of gather
- This is not a scalable call - think again if you want to use
  - On 10000 cores you easily run out of memory

# MPI\_Scatter in C

```
int MPI_Scatter(void* sendbuf, int sendcount,  
MPI_Datatype sendtype, void* recvbuf, int recvcount,  
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- `sendbuf`: address of send buffer (only root significant)
- `sendcount`: number of elements send to **each** task
- `sendtype`: type of data
- `recvbuf`: address of receive buffer
- `recvcount`: number of data received from root
- `recvtype`: type of data
- `root`: root rank – rank sending the data
- `comm`: communicator – every task receives



# MPI\_Scatter in Fortran 90

```
MPI_Scatter(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,  
            RECVTYPE, ROOT, COMM, IERROR)
```

```
<type>:: SENDBUF, RECVBUF
```

```
INTEGER:: SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,  
            COMM, IERROR
```

- **SENDBUF**: send buffer (significant only on root)
- **SENDCOUNT**: number of elements send to **each** task
- **SENDTYPE**: type of data
- **RECVBUF**: receive buffer
- **RECVCOUNT**: number of data received from root
- **RECVTYPE**: type of data
- **ROOT**: root rank – rank sending the data
- **COMM**: communicator – every task receives

# scatter in Python

```
comm.scatter(obj, root=root)
```

- `obj` : The Python object (should be a list) to scatter
- `root` : root rank – rank sending the data

`obj` should be a list that contains the objects to be sent to each process.

`len(obj)` should be equal to the number of processes.

Example:

```
b = comm.scatter(a, root=0)
```

`a` on the root rank should be a `list` containing Python objects.

After scatter, `b` is the object on each process.

In C	In Fortran
<pre>float *sendbuf=NULL; float recvbuf[4]; MPI_Comm my_comm; MPI_Init(&amp;argc,&amp;argv); my_comm = MPI_COMM_WORLD; MPI_Comm_size(my_comm,&amp;numprocs); MPI_Comm_rank(my_comm,&amp;myrank);  root = 0; counts = 3; //nr. of elements to be sent/received size_sendbuf = counts * numprocs; //size receiving buffer if( myrank == 0) {     sendbuf = malloc(size_sendbuf * sizeof(float));     for( i = 0; i &lt; size_sendbuf; i++)         sendbuf[i] = 1.0*i; }  <b>MPI_Scatter</b>(sendbuf,counts,MPI_FLOAT,recvbuf,counts,MPI_FLOAT,root,my_comm);  if( myrank == 0) {     free(sendbuf); } MPI_Finalize();</pre>	<pre>real, pointer :: sendbuf(:) real recvbuf(4) Integer my_comm call MPI_INIT(ierr) my_comm = MPI_COMM_WORLD call MPI_COMM_RANK(my_comm, myrank, ierr) call MPI_COMM_SIZE(my_comm, numprocs, ierr)  root = 0 counts =3 !nr. of elements to be sent/received size_sendbuf = counts * numprocs !size receiving buffer if (myrank == 0 ) then     allocate( sendbuf(size_sendbuf) )     do i=1,size_sendbuf         sendbuf(i) = 1.0*i-1.0     enddo endif <b>call MPI_Scatter</b> (sendbuf, counts,MPI_REAL,recvbuf,counts, MPI_REAL,root,my_comm, ierr)  if(myrank == 0) then     deallocate ( sendbuf ) endif Call MPI_FINALIZE(ierr)</pre>

## In Python

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
```

```
numprocs = comm.Get_size()
```

```
myrank = comm.Get_rank()
```

```
send_count = 3
```

```
send_size = send_count * numprocs
```

```
if myrank == 0:
```

```
    send_data = []
```

```
    for i in range(numprocs):
```

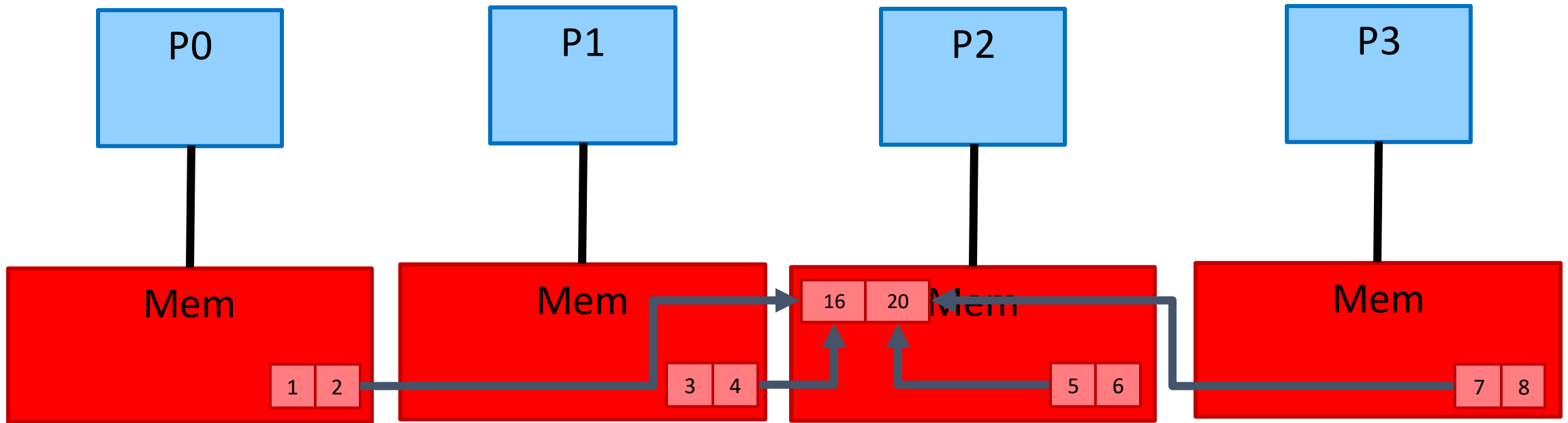
```
        send_data.append([i * send_count + j for j in range(send_count)])
```

```
else:
```

```
    send_data = None
```

```
comm.scatter(send_data, root=0)
```

# Reduce



- Example: Vector addition for count of 2
- Combines data from all processors into data structure on root
- This is a widely used scalable call
  - Structures on each processor task count independent

# MPI\_Reduce in C

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

- `sendbuf`: send buffer
- `recvbuf`: receive buffer (significant only on root)
- `count`: length of send and receive buffer
- `datatype`: data type of data – required for correct op
- `op`: handle of operation (more later)
- `root`: rank of root process
- `comm`: communicator, every rank contributes

**Option:** Constant **MPI\_IN\_PLACE** as `sendbuf` on root

# MPI\_Reduce in Fortran 90

```
MPI_REDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, &  
            COMM, IERROR)
```

```
<type>::  SENDBUF, RECVBUF
```

```
INTEGER::  COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

- `sendbuf`: send buffer
- `recvbuf`: receive buffer (significant only on root)
- `count`: length of send and receive buffer
- `datatype`: data type of data – required for correct op
- `op`: handle of operation (more later)
- `root`: rank of root process
- `comm`: communicator, every rank contributes

**Option:** Constant `MPI_IN_PLACE` as `sendbuf` on root

# Predefined reduction operations

Name	Function	MPI data types
<b>MPI_MAX</b>	Maximum	C integer, Fortran integer, Floating point
<b>MPI_MIN</b>	Minimum	C integer, Fortran integer, Floating point
<b>MPI_SUM</b>	Sum	C integer, Fortran integer, Floating point, Complex
<b>MPI_PROD</b>	Product	C integer, Fortran integer, Floating point, Complex
<b>MPI_LAND</b>	Logical and	C integer, Fortran logical
<b>MPI_BAND</b>	Bit-wise and	C integer, Fortran logical, Byte
<b>MPI_LOR</b>	Logical or	C integer, Fortran logical
<b>MPI_BOR</b>	Bit-wise or	C integer, Fortran logical, Byte
<b>MPI_LXOR</b>	Logical xor	C integer, Fortran logical
<b>MPI_BXOR</b>	Bit-wise xor	C integer, Fortran logical, Byte



# reduce in Python

```
comm.reduce(obj, op=op, root=root)
```

- `obj` : The Python object to reduce
- `op` : handle of operation (more later)
- `root` : rank of root process

Example:

```
pi = comm.reduce(partial_pi, op=MPI.SUM, root=0)
```

# Predefined reduction operations in Python

Name	Function	MPI data types
<b>MPI . MAX</b>	Maximum	C integer, Fortran integer, Floating point
<b>MPI . MIN</b>	Minimum	C integer, Fortran integer, Floating point
<b>MPI . SUM</b>	Sum	C integer, Fortran integer, Floating point, Complex
<b>MPI . PROD</b>	Product	C integer, Fortran integer, Floating point, Complex
<b>MPI . LAND</b>	Logical and	C integer, Fortran logical
<b>MPI . BAND</b>	Bit-wise and	C integer, Fortran logical, Byte
<b>MPI . LOR</b>	Logical or	C integer, Fortran logical
<b>MPI . BOR</b>	Bit-wise or	C integer, Fortran logical, Byte
<b>MPI . LXOR</b>	Logical xor	C integer, Fortran logical
<b>MPI . BXOR</b>	Bit-wise xor	C integer, Fortran logical, Byte

# Quiz

Supposing that the destination rank is 0, what are the values of “b” and “d” after running the following MPI calls?

Time	Rank 0	Rank 1
0	a=0; c=2	a=1; c=3
1	MPI_Reduce(&a,&b,...)	MPI_Reduce(&c,&d,...)
2	MPI_Reduce(&c,&d,...)	MPI_Reduce(&a,&b,...)

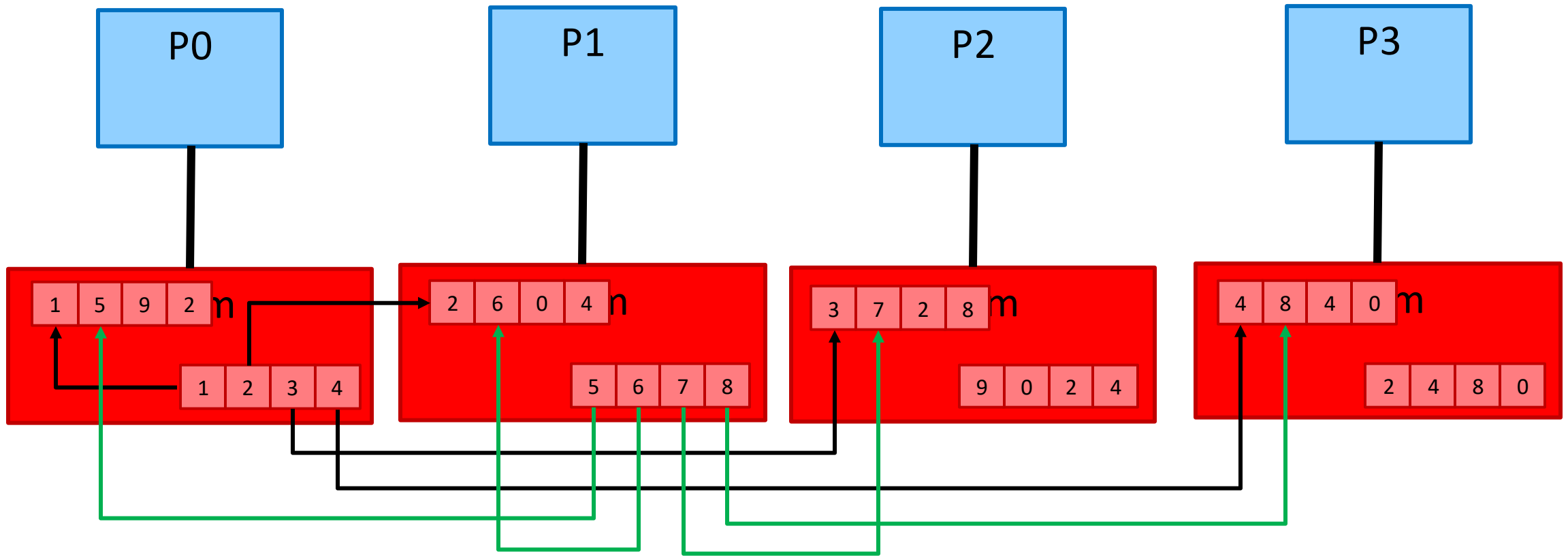
# Quiz

Supposing that the destination rank is 0, what are the values of “b” and “d” after running the following MPI calls?

Time	Rank 0	Rank 1
0	a=0; c=2	a=1; c=3
1	MPI_Reduce(&a,&b,...)	MPI_Reduce(&c,&d,...)
2	MPI_Reduce(&c,&d,...)	MPI_Reduce(&a,&b,...)

**Solution:**  $b=0+3=3$   
 $d=2+1=3$

# All-to-all (the worse of the worst)



- Every processor sends to every other processor
- 1<sup>st</sup> portion of send buffer → rank 0, 2<sup>nd</sup> portion → rank 1, etc.
- 1<sup>st</sup> portion in recv buffer ← rank 0, 2<sup>nd</sup> portion ← rank 1, etc.
- *Extremely important in spectral codes, e.g. parallel FFT*

# MPI\_Alltoall in C

```
int MPI_Alltoall(void* sendbuf, int sendcount,  
    MPI_Datatype sendtype, void* recvbuf, int recvcount,  
    MPI_Datatype recvtype, MPI_Comm comm)
```

- sendbuf: Address of send buffer
- sendcount: Number of elements send from each task
- sendtype: Data type of send buffer
- recvbuf: Address of receive buffer
- recvcount: Number of elements received f. each task
- recvtype: Data type of receive buffer
- comm: Communicator, every task sends and recvs

**Rem:** The counts are not the buffer size!

# MPI\_Alltoall in Fortran 90

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
  RECVCOUNT, RECVMODE, COMM, IERROR)  
<type>:: SENDBUF, RECVBUF  
INTEGER:: SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMODE, COMM,  
IERROR
```

- `sendbuf`: Address of send buffer
- `sendcount`: Number of elements send from each task
- `sendtype`: Data type of send buffer
- `recvbuf`: Address of receive buffer
- `recvcount`: Number of elements received f. each task
- `recvtype`: Data type of receive buffer
- `comm`: Communicator, every task sends and recvs

**Rem:** The counts are not the buffer size!

# alltoall in Python

```
comm.alltoall(obj)
```

- `obj` : The Python object (should be a list)

`obj` should be a list that contains the objects to be sent to each process.

`len(obj)` should be equal to the number of processes.

Example:

```
b = comm.alltoall(a)
```

`a` should be a `list` containing Python objects.

After scatter, `b` is a `list` containing the result of `alltoall` operation.



# Variations: Allgather and Allreduce

- They are “**All**” versions for calls which receive only on root:
  - `MPI_Allgather` (or `comm.allgather` in Python)
  - `MPI_Allreduce` (or `comm.allreduce` in Python)
- Every task has a receive buffer – the result is known on every task
- These calls can be thought of as
  - `MPI_Gather` followed by `MPI_Bcast`
  - `MPI_Reduce` followed by `MPI_Bcast`
- The `root` argument is omitted from the interface
- “**All**”-communications can take longer to complete
  - Only use them if you need them

# Advanced topic: Vector collectives

- The calls of this lecture: Same count on all tasks
- Vector collectives relax this condition:
  - `MPI_Gatherv`
  - `MPI_Scatterv`
  - `MPI_Allgatherv`
  - `MPI_Alltoallv`
- These calls go beyond the scope of this course

# Non blocking collectives in MPI 3.x

- Similar to non-blocking point-to-point communication:
  - Non-blocking call (e.g. **MPI\_Ibcast**) initiates communication
  - A completion call (e.g. **MPI\_Wait**) ensures that local part of communication is finalised
    - Send buffers can be overwritten
    - Receive buffers contain data
- Allows for
  - Overlapping communication and calculation
  - Avoiding synchronisation if MPI library avoids sync.
    - The call **MPI\_Ibarrier** has to avoid synchronisation
  - Avoiding dead locks (e.g. overlapping communicators)

# Summary

- Discussed collective communications:
  - Barrier
  - Broadcast
  - Gather/Scatter
  - Reduction
  - Alltoall
- Variations of the above (all-version, vector-version)
- Non-blocking collective communication in MPI 3.0

# Exercise 1

Create a version of your  $\pi^2$ -code using collective calls

- Time the communication times
- Compare performance of the versions using
  - Point-to-point
  - Collectives

## Remarks:

- You might need a barrier in the beginning of your code to absorb differences in "task wake up"
- You might need to run repeatedly

## Exercise 2

Modify your *messages around a ring* code to use a collective to add the send-buffers onto rank 0

Compare the performance with original code

## Exercise 3. 2D Integration

In this example, you will calculate the double integral:

$$\int_0^{\pi} \int_0^{\pi} \sin(x + y) dx dy = 0$$

One way to parallelize this calculation is to divide the integration range of one variable, let's say "x", evenly among N processes and perform the regular integration over the other variable, "y", in this case.

This problem is useful for two reasons: first, we know the exact value of the integral (0), and second, because it includes a double integral, the computation is sufficiently heavy to detect the effects of a parallel implementation.