

# Tutorial on MPI programming

Victor Eijkhout

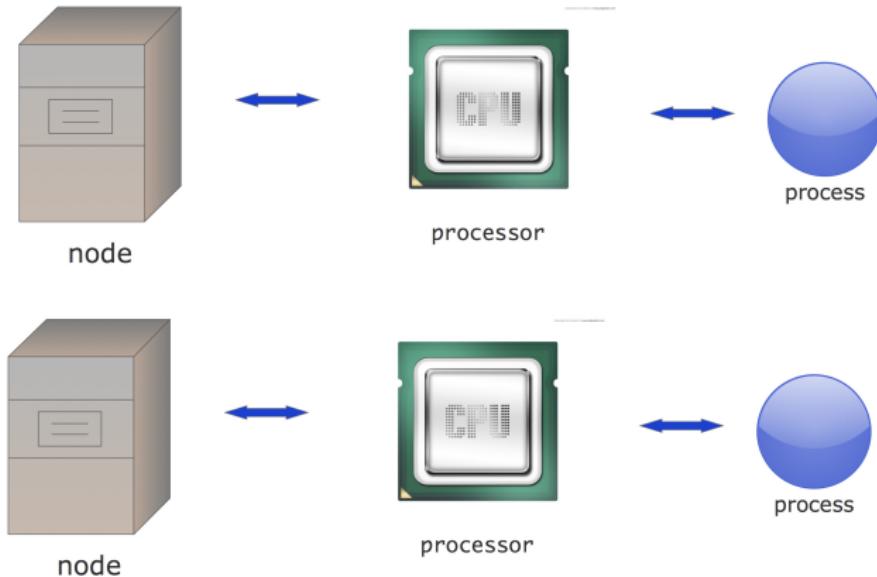
2015

# Justification

The MPI library is the main tool for parallel programming on a large scale. This course introduces the main concepts through lecturing and exercises.

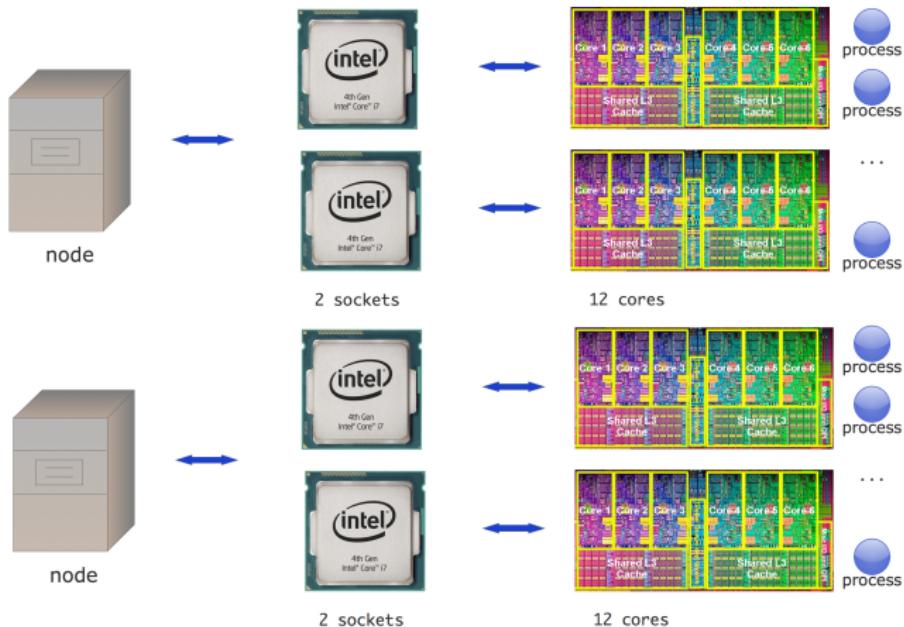
# The SPMD model

# Computers when MPI was designed



One process per node; all communication goes through the network.

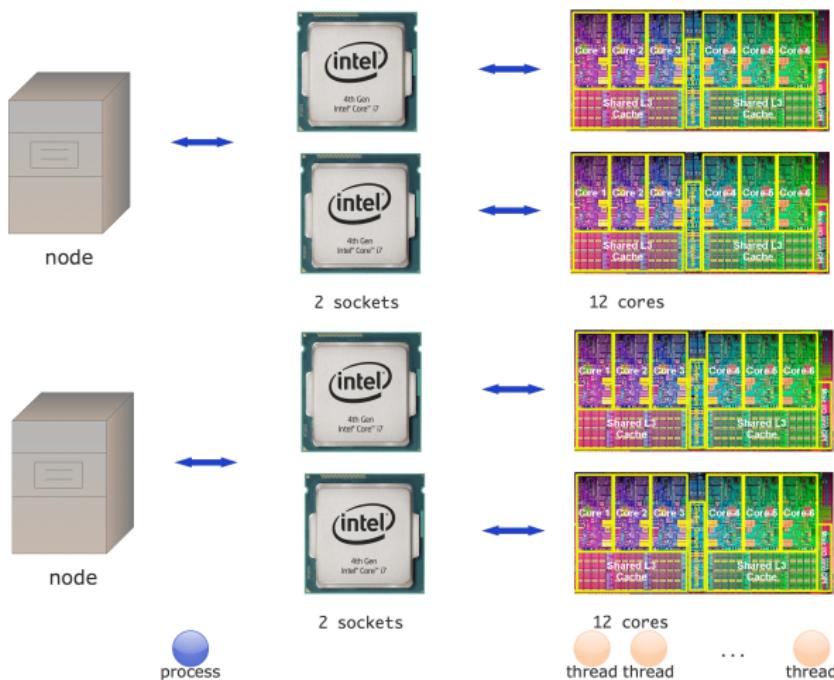
# Pure MPI



A node has multiple sockets, each with multiple cores.

Pure MPI puts a process on each core: pretend shared memory doesn't exist.

# Hybrid programming



Hybrid programming puts a process per node or per socket; further parallelism comes from threading.

## Lab setup

Open two windows on stampede. In one window you will be editing and compiling, in the other, type `idev -N 2 -n 32 -t 4:0:0` which gives you an interactive session of 2 nodes, 32 cores, for the next 4 hours.

The C compiler is `mpicc`, Fortran is `mpif90`. To run (on a compute node!) type  
`ibrun yourprog`

## Exercise 1

Write a ‘hello world’ program, without any MPI in it, and run it in parallel with `mpieexec` or your local equivalent. Explain the output.

## MPI Init / Finalize

You need an include file:

```
#include "mpi.h" // for C  
#include "mpif.h" ! for Fortran
```

Then put these calls around your code:

```
MPI_Init(&argc,&argv);  
// your code  
MPI_Finalize();
```

and for Fortran:

```
call MPI_Init(ierr)  
! your code  
call MPI_Finalize(ierr)
```

# Python bindings

```
module load python-mpi
```

```
from mpi4py import MPI
```

No initialization needed.

## Exercise 2

Now add the commands `MPI_Init` and `MPI_Finalize` to your code. Put three different print statements in your code: one before the init, one between init and finalize, and one after the finalize. Again explain the output.

## Exercise 3

Now use the command `MPI_Get_processor_name` in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run a program that uses two different nodes.

TACC nodes have a hostname `cRRR-CNN`, where RRR is the rack number, C is the chassis number in the rack, and NN is the node number within the chassis. Communication is faster inside a rack than between racks!

C:

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

Fortran:

```
MPI_Get_processor_name(name, resultlen, ierror)
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
INTEGER, INTENT(OUT) :: resultlen
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
MPI.Get_processor_name()
```

# Processor identification

Every processor has a number (with respect to a communicator)

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

For now, the communicator will be `MPI_COMM_WORLD`.

Note: mapping of ranks to actual processors and cores is not predictable!

## Exercise 4

Write a program where each process prints out message reporting its number, and how many processes there are.

Write a second version of this program, where each process opens a unique file and writes to it. *On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.*

## Exercise 5

Write a program where only the process with number zero reports on how many processes there are in total.

# Functional Parallelism

Parallelism by letting each processor do a different thing.

Example: divide up a search space.

Each processor knows its rank, so it can find its part of the search space.

## Exercise 6

Is the number  $N = 2,000,000,111$  prime? Let each process test a range of integers, and print out if they find a factor. You don't have to test all integers  $< N$ : any factor is at most  $\sqrt{N} \approx 45,200$ .

# Collectives

# Table of Contents

1 Introduction

2 Simple collectives

3 Advanced collectives

# Collectives

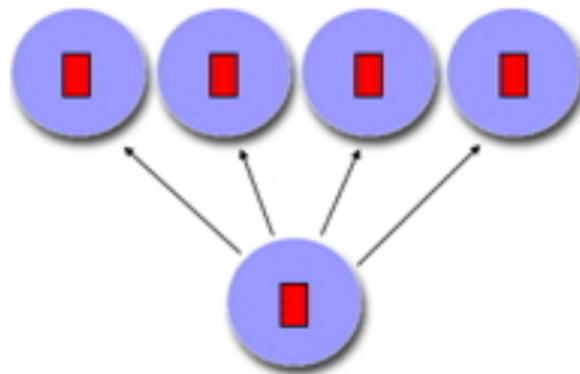
Gathering and spreading information:

- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

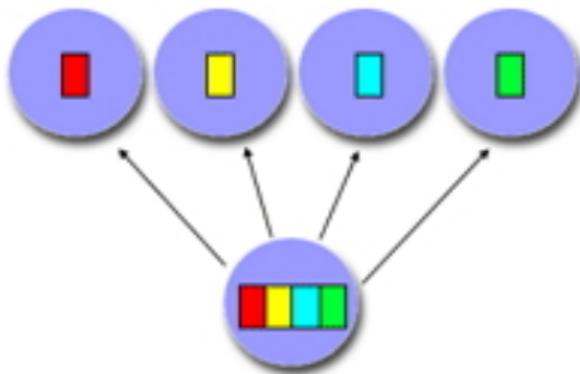
Root process: the one doing the collecting or disseminating.

Basic cases:

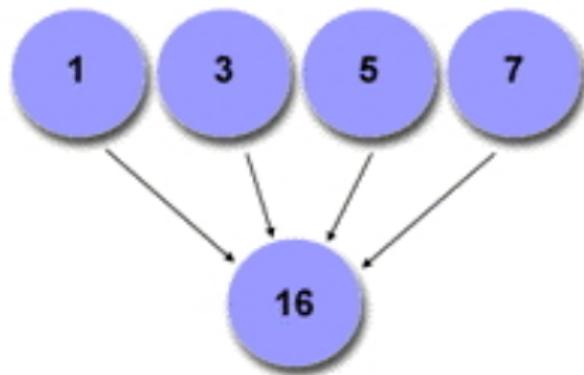
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



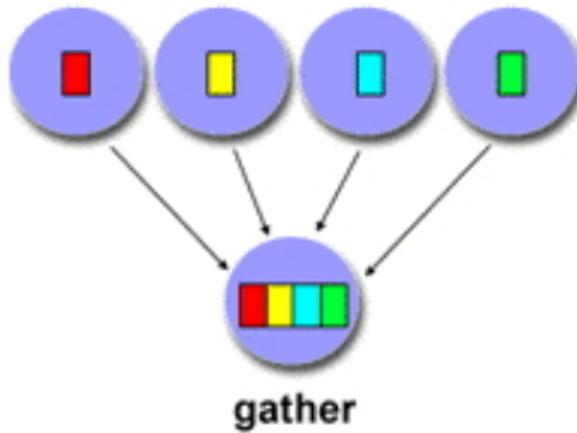
broadcast



scatter



reduction



gather

## Exercise 7

How would you realize the following scenarios with MPI collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

## More collectives

- Instead of a root, collect to all: `MPI_All...`
- Scatter individual data, but also individual size: `MPI_Scatterv`
- Everyone broadcasts: all-to-all
- Scan: like a reduction, but with partial results

...and more

# Table of Contents

1 Introduction

2 Simple collectives

3 Advanced collectives

# Broadcast

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

- **root** is the rank of the process doing the broadcast
- Each process allocates buffer space;  
root explicitly fills in values,  
all others receive values through broadcast call.
- Datatype is MPI\_FLOAT, MPI\_INT et cetera, different between C/Fortran.
- comm is usually MPI\_COMM\_WORLD

# Buffers in C

General principle: buffer argument is address in memory of the data.

- Buffer is void pointer:
- write `&x` or `(void*)&x` for scalar
- write `x` or `(void*)x` for array

# Buffers in Fortran

General principle: buffer argument is address in memory of the data.

- Fortran always passes by reference:
- write  $x$  for scalar
- write  $x$  for array

# Buffers in Python

For many routines there are two variants:

- lowercase: can send Python objects;  
output is return result  
this uses pickle: slow.
- uppercase: communicates numpy objects;  
input and output are function argument.

## Exercise 8

If you give a commandline argument to a program, that argument is available as a character string as part of the `argv`, `argc` pair that you typically use as the arguments to your main program. You can use the function `atoi` to convert such a string to integer.

Write a program where process 0 looks for an integer on the commandline, and broadcasts it to the other processes. Initialize the buffer on all processes, and let all processes print out the broadcast number, just to check that you solved the problem correctly.

# Reduction

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

- Compare buffers to ▶ bcast
- recvbuf is ignored on non-root processes
- MPI\_Op is MPI\_SUM, MPI\_MAX et cetera.

## Exercise 9

Write a program where each process computes a random number, and process 0 finds and prints the maximum generated value. Let each process print its value, just to check the correctness of your program.

Here is how you initialize the random number generator uniquely on each process:

```
// Initialize the random number generator
srand((int)(mytid*(double)RAND_MAX/ntids));
// compute a random number
randomfraction = (rand() / (double)RAND_MAX);

integer :: randsize
integer,allocatable,dimension(:) :: randseed
real :: random_value

call random_seed(size=randsize)
allocate(randseed(randsize))
do i=1,randsize
    randseed(i) = 1023*mytid
```

# Random numbers

C:

```
// Initialize the random number generator  
srand((int)(mytid*(double)RAND_MAX/ntids));  
// compute a random number  
randomfraction = (rand() / (double)RAND_MAX);
```

Fortran:

```
integer :: randsize  
integer,allocatable,dimension(:) :: randseed  
real :: random_value  
  
call random_seed(size=randsize)  
allocate(randseed(randsize))  
do i=1,randsize  
    randseed(i) = 1023*mytid  
end do
```

## Exercise 10

Create on each process an array of length 2 integers, and put the values 1,2 in it on each process. Do a sum reduction on that array. Can you predict what the result should be? Code it. Was your prediction right?

# Gather/Scatter

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

- Compare buffers to ▶ reduce
- Scatter: the sendcount / Gather: the recvcount:  
this is not, as you might expect, the total length of the buffer; instead, it is  
the amount of data to/from each process.

## Exercise 11

Let each process compute a random number. You want to print on what processor the maximum value is computed. What collective do you use? Write a short program.

# Table of Contents

1 Introduction

2 Simple collectives

3 Advanced collectives

# Scan

Scan or ‘parallel prefix’: reduction with partial results

- Useful for indexing operations:
- Each processor has an array of  $n_p$  elements;
- My first element has global number  $\sum_{q < p} n_q$ .

C:

```
int MPI_Scan(const void* sendbuf, void* recvbuf,
             int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
IN sendbuf: starting address of send buffer (choice)
OUT recvbuf: starting address of receive buffer (choice)
IN count: number of elements in input buffer (non-negative integer)
IN datatype: data type of elements of input buffer (handle)
IN op: operation (handle)
IN comm: communicator (handle)
```

Fortran:

```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
res = intracomm.scan( sendobj=None, recvobj=None, op=MPI.SUM)
```

# All

- MPI\_Allreduce: do a reduction, but leave the result everywhere.
- MPI\_AllGather: gather, and leave the result everywhere.

C:

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

Semantics:

IN sendbuf: starting address of send buffer (choice)

OUT recvbuf: starting address of receive buffer (choice)

IN count: number of elements in send buffer (non-negative integer)

IN datatype: data type of elements of send buffer (handle)

IN op: operation (handle)

IN comm: communicator (handle)

Fortran:

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
```

TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf

TYPE(\*), DIMENSION(..) :: recvbuf

INTEGER, INTENT(IN) :: count

TYPE(MPI\_Datatype), INTENT(IN) :: datatype

TYPE(MPI\_Op), INTENT(IN) :: op

TYPE(MPI\_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Exercise 12

How can you simulate an allreduce with routines you already know? What is the point of having this one routine?

## V-type collectives

- Gather/scatter but with individual sizes
- Requires displacement in the gather/scatter buffer

C:

```
int MPI_Gatherv(
    const void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, const int recvcounts[], const int displs[],
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Semantics:

IN sendbuf: starting address of send buffer (choice)  
IN sendcount: number of elements in send buffer (non-negative integer)  
IN sendtype: data type of send buffer elements (handle)  
OUT recvbuf: address of receive buffer (choice, significant only at rank 0)  
IN recvcounts: non-negative integer array (of length group size) containing the number of elements to receive  
IN displs: integer array (of length group size). Entry i specifies the displacement of the i-th receive buffer  
IN recvtype: data type of recv buffer elements (significant only at rank 0)  
IN root: rank of receiving process (integer)  
IN comm: communicator (handle)

Fortran:

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
            TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
            TYPE(*), DIMENSION(..) :: recvbuf
            INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root)
```

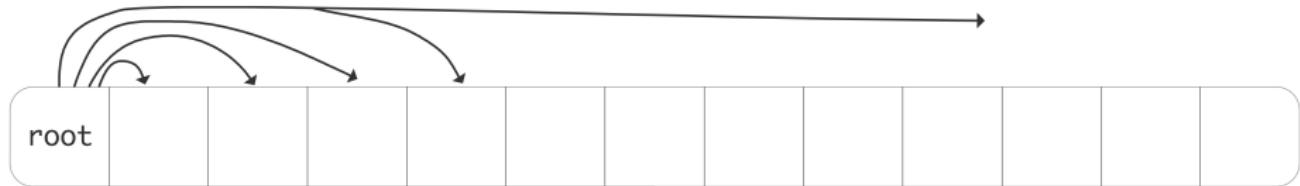
## All-to-all

- Every process does a scatter;
- each individual data
- Very rarely needed.

## Barrier

- Synchronize processors:
- each process waits at the barrier until all processes have reached the barrier
- **This routine is almost never needed**
- One conceivable use: timing

## Naive realization of collectives

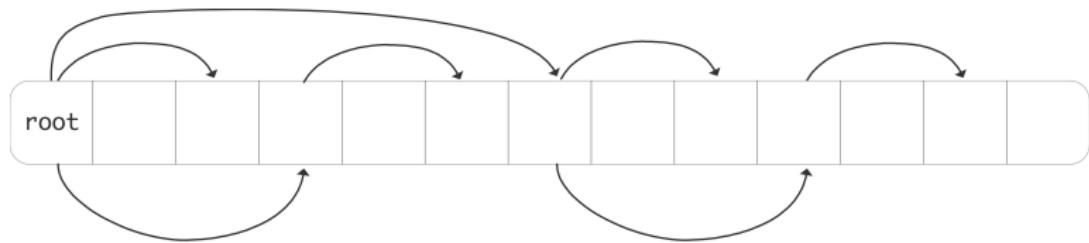


Message time is modeled as

$$\alpha + \beta n$$

Time for collective? Can you improve on that?

## Better implementation of collective



What is the running time now?

# **Point-to-point communication**

# Table of Contents

4 Distributed data

5 Local information exchange

6 Blocking communication

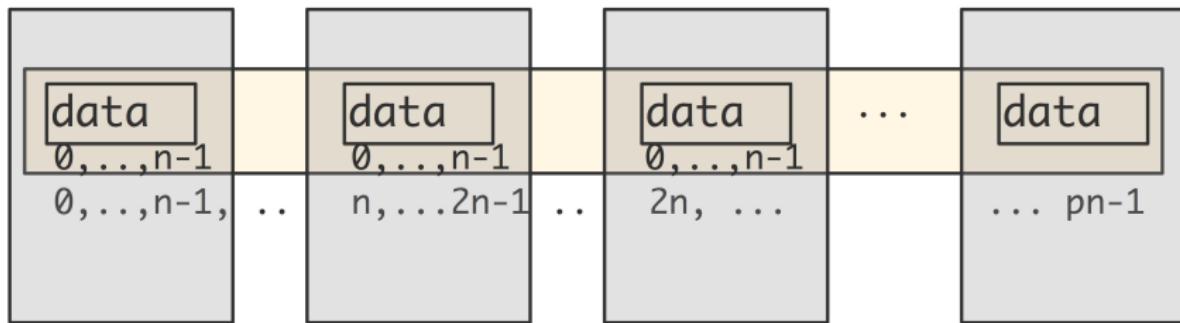
7 Pairwise exchange

8 Irregular exchanges: non-blocking communication

# Distributed data

Distributed array: each process stores disjoint local part

```
int n;  
double data[n];
```



Local numbering  $0, \dots, n_{\text{local}}$ ;  
global numbering is ‘in your mind’.

## Local and global indexing

Every local array starts at 0 (Fortran: 1);  
you have to translate that yourself to global numbering:

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
    int iglobal = myfirst+ilocal;
    array[ilocal] = f(iglobal);
}
```

## Exercise 13

We want to compute  $\sum_{n=1}^N n^2$ , and we do that as follows. Read in the global  $N$  parameter, and make sure that it is a multiple of the number  $P$  of processors. Your code should produce an error message and exit immediately if it doesn't.

- Now allocate the local parts: each processor should allocate only  $N/P$  elements.
- On each processor, initialize the local array so that the  $i$ -th location of the distributed array (for  $i = 0, \dots, N - 1$ ) contains  $(i + 1)^2$ .
- Now use a collective operation to compute the sum of the array values. The right value is  $(2N^3 + 3N^2 + N)/6$ . Is that what you get?

To debug your program, first start with  $N = P$ .

(Did you allocate your array as real numbers? Why are integers not a good idea?)

# Load balancing

If the distributed array is not perfectly divisible:

```
int Nglobal, // is something large
    Nlocal = Nglobal/ntids,
    excess = Nglobal%ntids;
if (mytid==ntids-1)
    Nlocal += excess;
```

This gives a load balancing problem. Better solution?

# Inner product calculation

Given vectors  $x, y$ :

$$x^t y = \sum_{i=0}^{N-1} x_i y_i$$

Start out with a distributed vector.

- Wrong way: collect the vector on one processor and evaluate.
- Right way: compute local part, then collect local sums.

```
local_inprod = 0;  
for (i=0; i<localsize; i++)  
    local_inprod += x[i]*y[i];  
MPI_Reduce( &local_inprod, &global_inprod, 1,MPI_DOUBLE ... )
```

## Exercise 14

Implement an inner product routine: let  $x$  be a distributed vector of size  $N$  with elements  $x[i] = i$ , and compute  $x^t x$ . As before, the right value is  $(2N^3 + 3N^2 + N)/6$ .

Use the inner product value to scale to vector so that it has norm 1. Check that your computation is correct.

# Table of Contents

- 4 Distributed data
- 5 Local information exchange
- 6 Blocking communication
- 7 Pairwise exchange
- 8 Irregular exchanges: non-blocking communication

# Operating on distributed data

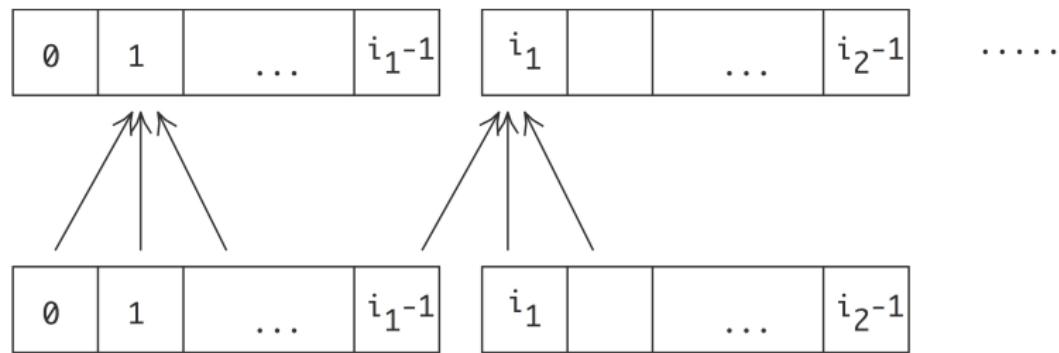
Array of numbers  $x_i$ :  $i = 0, \dots, N$

compute

$$y_i = (x_{i-1} + x_i + x_{i+1})/3: i = 1, \dots, N-1$$

'owner computes'

This leads to communication:



so we need a point-to-point mechanism.

## MPI point-to-point mechanism

- Matched send and receive calls
- One process sends to a specific other process
- Other process does a specific receive.

# Ping-pong

A sends to B, B sends back to A

What is the code for A? For B?

# Ping-pong

A sends to B, B sends back to A

Process A executes the code

```
MPI_Send( /* to: */ B .... );  
MPI_Recv( /* from: */ B ... );
```

Process B executes

```
MPI_Recv( /* from: */ A ... );  
MPI_Send( /* to: */ A .... );
```

# Ping-pong in MPI

Remember SPMD:

```
if ( /* I am process A */ ) {  
    MPI_Send( /* to: */ B ..... );  
    MPI_Recv( /* from: */ B ... );  
} else if ( /* I am process B */ ) {  
    MPI_Recv( /* from: */ A ... );  
    MPI_Send( /* to: */ A ..... );  
}
```

C:

```
int MPI_Send(  
    const void* buf, int count, MPI_Datatype datatype,  
    int dest, int tag, MPI_Comm comm)
```

Semantics:

IN buf: initial address of send buffer (choice)

IN count: number of elements in send buffer (non-negative integer)

IN datatype: datatype of each send buffer element (handle)

IN dest: rank of destination (integer)

IN tag: message tag (integer)

IN comm: communicator (handle)

Fortran:

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
```

TYPE(\*), DIMENSION(..), INTENT(IN) :: buf

INTEGER, INTENT(IN) :: count, dest, tag

TYPE(MPI\_Datatype), INTENT(IN) :: datatype

TYPE(MPI\_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python native:

C:

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

Semantics:

OUT buf: initial address of receive buffer (choice)

IN count: number of elements in receive buffer (non-negative integer)

IN datatype: datatype of each receive buffer element (handle)

IN source: rank of source or MPI\_ANY\_SOURCE (integer)

IN tag: message tag or MPI\_ANY\_TAG (integer)

IN comm: communicator (handle)

OUT status: status object (Status)

Fortran:

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
```

TYPE(\*), DIMENSION(..) :: buf

INTEGER, INTENT(IN) :: count, source, tag

TYPE(MPI\_Datatype), INTENT(IN) :: datatype

TYPE(MPI\_Comm), INTENT(IN) :: comm

TYPE(MPI\_Status) :: status

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Status object

- Receive call can have various wildcards
- MPI\_ANY\_SOURCE, MPI\_ANY\_TAG
- use status object to retrieve actual description of the message

## Exercise 15

Implement the ping-pong program. Add a timer using `MPI_Wtime`. For the status argument of the receive call, use `MPI_STATUS_IGNORE`.

Then modify the program to use longer messages. How does the timing increase with message size?

# Table of Contents

4 Distributed data

5 Local information exchange

6 Blocking communication

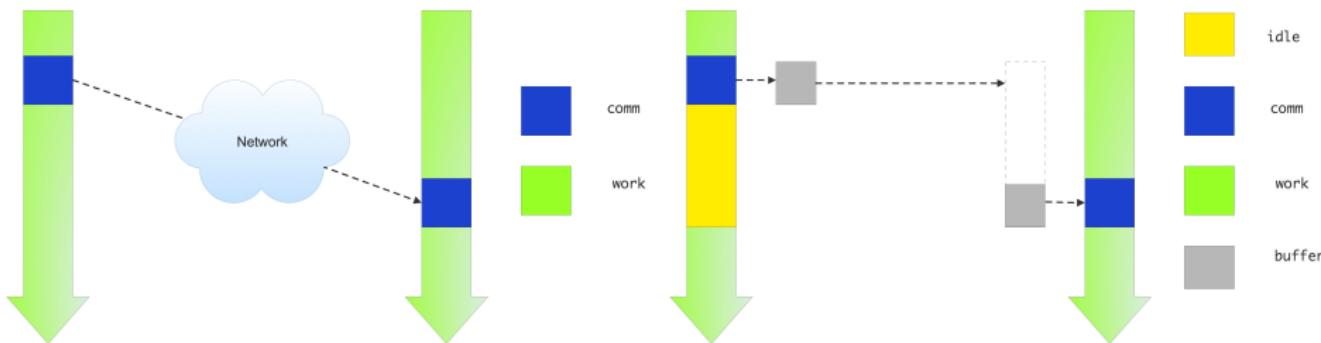
7 Pairwise exchange

8 Irregular exchanges: non-blocking communication

# Blocking send/recv

`MPI_Send` and `MPI_Recv` are *blocking* operations:

- The process waits ('blocks') until the operation is concluded.
- A send can not complete until the receive executes.



Ideal vs actual send/recv behaviour.

# Deadlock

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
receive(source=other);
send(target=other);
```

A subtlety.

This code may actually work:

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
send(target=other);
receive(source=other);
```

Small messages get sent even if there is no corresponding receive.

# Protocol

Communication is a ‘rendez-vous’ or ‘hand-shake’ protocol:

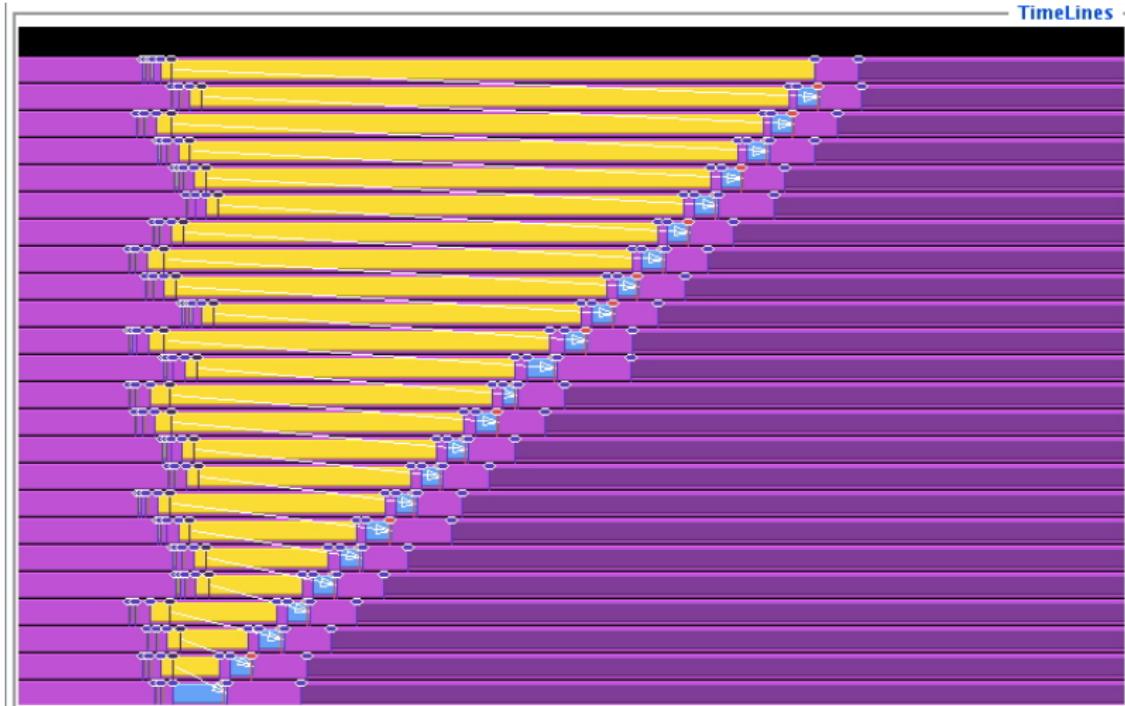
- Sender: ‘I have data for you’
- Receiver: ‘I have a buffer ready, send it over’
- Sender: ‘Ok, here it comes’
- Receiver: ‘Got it.’

## Exercise 16

(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and execute the following program:

- ① If you are not the rightmost student, turn to the right and give the paper to your right neighbour.
- ② If you are not the leftmost student, turn to your left and accept the paper from your left neighbour.

# TAU trace: serialization



## The problem here...

Here you have a case of a program that computes the right output, just way too slow.

Beware! Blocking sends/receives can be trouble.  
(How would you solve this particular case?)

# Table of Contents

4 Distributed data

5 Local information exchange

6 Blocking communication

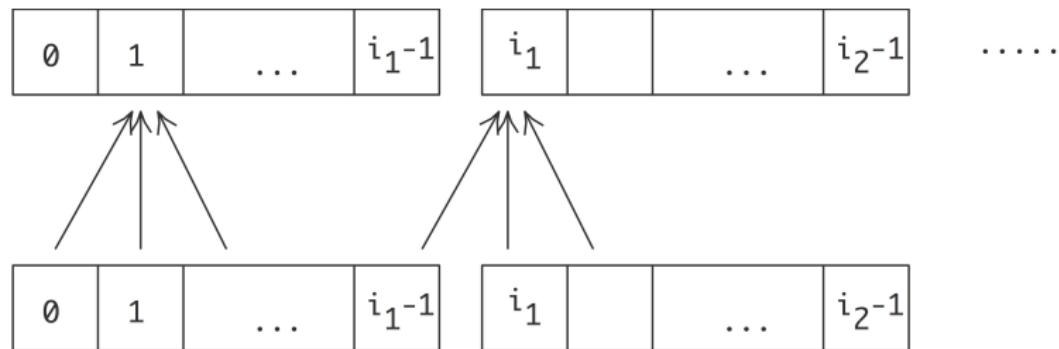
7 Pairwise exchange

8 Irregular exchanges: non-blocking communication

# Operating on distributed data

Take another look:

$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \dots, N-1$$



- One-dimensional data and linear processor numbering;
- Operation between neighbouring indices: communication between neighbouring processors.

## Sendrecv

Instead of separate send and receive: use

```
int MPI_Sendrecv(
    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)
```

## Sendrecv with incomplete pairs

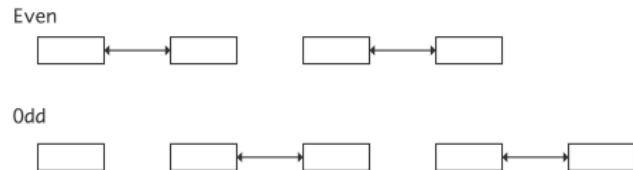
```
MPI_Comm_rank( .... &mytid );
if ( /* I am not the first processor */ )
    predecessor = mytid-1;
else
    predecessor = MPI_PROC_NULL;
if ( /* I am not the last processor */ )
    successor = mytid+1;
else
    successor = MPI_PROC_NULL;
sendrecv(from=predecessor,to=successor);
```

## Exercise 17

Implement the above right-shift scheme using MPI\_Sendrecv. If you have TAU installed, make a trace. Does it look different from the serialized send/recv code?

## Exercise 18

A very simple sorting algorithm is *exchange sort*: pairs of processors compare data, and if necessary exchange. The elementary step is called a *compare-and-swap*: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the maximum. For simplicity, in this exercise we give each processor just a single number.



The exchange sort algorithm is split in even and odd stages:

- In the even stage, processors  $2i$  and  $2i + 1$  compare and swap data;
- In the odd stage, processors  $2i + 1$  and  $2i + 2$  compare and swap.

You need to repeat this  $P/2$  times, where  $P$  is the number of processors.

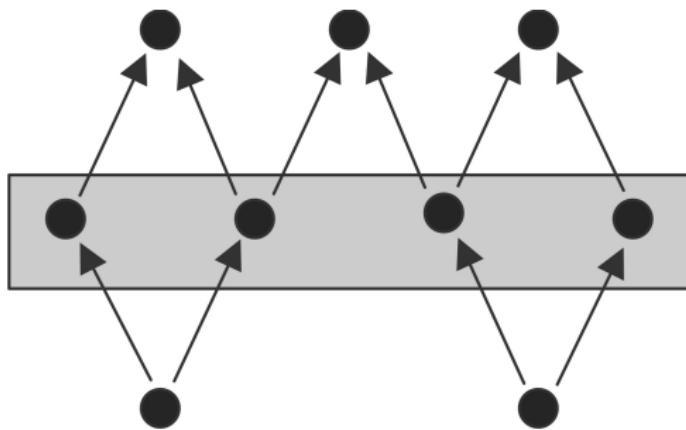
Use `MPI_PROC_NULL` for the edge cases.

# Table of Contents

- 4 Distributed data
- 5 Local information exchange
- 6 Blocking communication
- 7 Pairwise exchange
- 8 Irregular exchanges: non-blocking communication

# Sending with irregular connections

Graph operations:



## How do you approach this?

- It is very hard to figure out a send/receive sequence that does not deadlock or serialize
- Even if you manage that, you may have processor idle time.

Instead:

- Declare 'this data needs to be sent' or 'these messages are expected', and
- then wait for them collectively.

## Non-blocking send/recv

```
// start non-blocking communication  
MPI_Isend( ... ); MPI_Irecv( ... );  
// wait for the Isend/Irecv calls to finish in any order  
MPI_Wait( ... );
```

# Syntax

Very much like blocking  and  :

```
int MPI_Isend(void *buf,  
    int count, MPI_Datatype datatype, int dest, int tag,  
    MPI_Comm comm, MPI_Request *request)  
int MPI_Irecv(void *buf,  
    int count, MPI_Datatype datatype, int source, int tag,  
    MPI_Comm comm, MPI_Request *request)
```

The `MPI_Request` can be tested:

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],  
    MPI_Status array_of_statuses[])
```

(also `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome`)

## Exercise 19

Now use nonblocking send/receive routines to implement the averaging operation on a distributed array.

# Comparison

- Obvious: blocking vs non-blocking behaviour.
- Buffer reuse: when a blocking call returns, the buffer is safe for reuse;
- A buffer in a non-blocking call can only be reused after the wait call.

# Buffer use in blocking/non-blocking case

Blocking:

```
double *buffer;
for ( ... p ... ) {
    buffer = // fill in the data
    MPI_Send( buffer, ... /* to: */ p );
```

Non-blocking:

```
double **buffers;
for ( ... p ... ) {
    buffers[p] = // fill in the data
    MPI_Isend( buffers[p], ... /* to: */ p );
```

# Latency hiding

Other motivation for non-blocking calls:

overlap of computation and communication, provided hardware support.

Also known as 'latency hiding'.

Example: three-point combination operation (see above):

- ① Start communication for edge points,
- ② Do local operations while communication goes on,
- ③ Wait for edge points from neighbour processors
- ④ Incorporate incoming data.

## Test: non-blocking wait

- Post non-blocking receives
- test for incoming messages
- if nothing comes in, do local work

```
while (1) {  
    if (MPI_Test( /* from: */ ANY_SOURCE ) )  
        // do something with incoming message  
    else  
        // do local work  
}
```

## More sends and receive

- MPI\_Bsend, MPI\_Ibsend: **buffered send**
- MPI\_Ssend, MPI\_Issend: **synchronous send**
- MPI\_Rsend, MPI\_Irsend: **ready send**

too obscure to go into.

# Complicated data

# Table of Contents

9 Discussion

10 Datatypes

11 Packed data

# Non-contiguous data

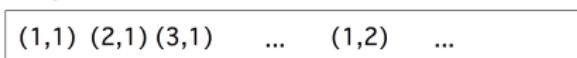
Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

Logical:

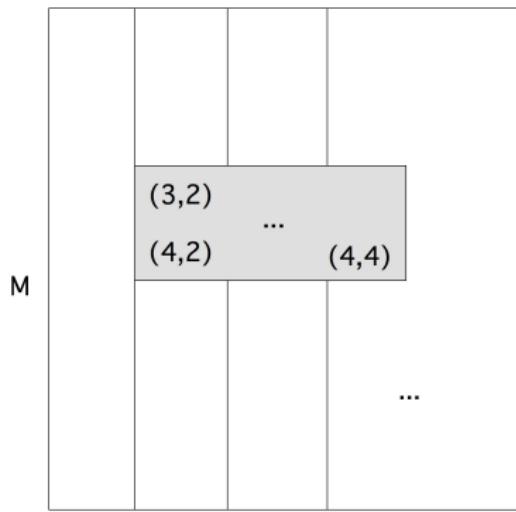


Physical:



# Submatrix storage

Logical:



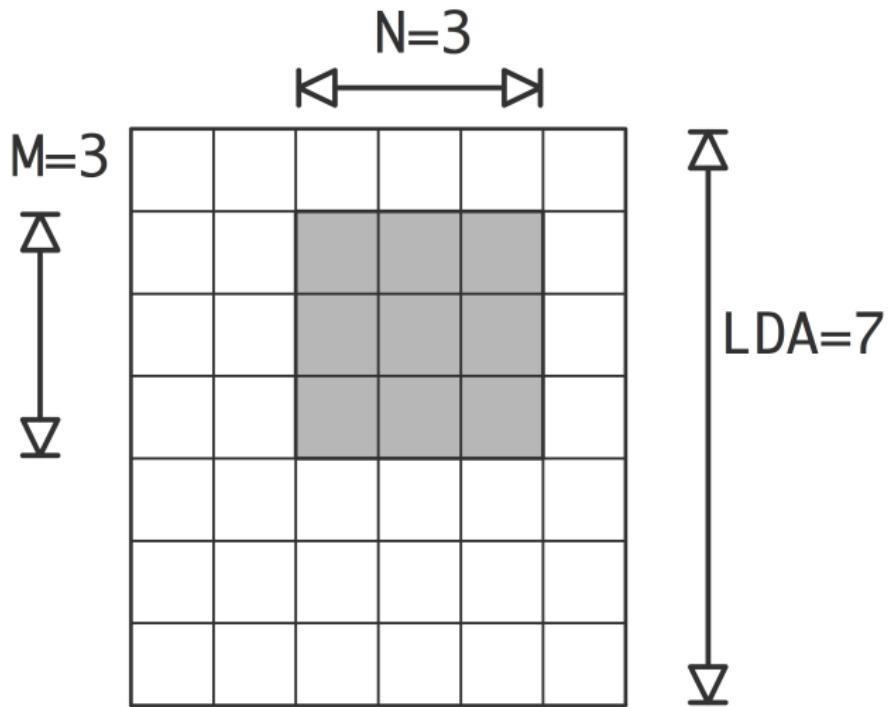
Physical:

	(3,2) (4,2)		(3,3) (4,3)	...
--	-------------	--	-------------	-----

- Location of first element
- Stride, blocksize

# BLAS/Lapack storage

Three parameter description:



## Exercise 20

How would you describe the memory layout of a submatrix, if the whole matrix has size  $M \times N$  and the submatrix  $m \times n$ ?

# Datatypes in MPI

We need data structures with gaps, or heterogeneous types.  
MPI allows for recursive construction of data types.

- Elementary types
- Derived types
- Packed data

# Table of Contents

9 Discussion

10 Datatypes

11 Packed data

# Elementary datatypes

C/C++	Fortran
MPI_CHAR	MPI_CHARACTER
MPI_UNSIGNED_CHAR	
MPI_SIGNED_CHAR	
MPI_SHORT	MPI_LOGICAL
MPI_UNSIGNED_SHORT	
MPI_INT	MPI_INTEGER
MPI_UNSIGNED	
MPI_LONG	
MPI_UNSIGNED_LONG	
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
MPI_LONG_DOUBLE	
	MPI_COMPLEX
	MPI_DOUBLE_COMPLEX

# How to use derived types

Create, commit, use, free:

```
MPI_datatype type;  
MPI_Type_xxx( .... &type);  
int MPI_Type_commit ( &type );  
  
// code using the type  
  
int MPI_Type_free ( &type );
```

# Contiguous type

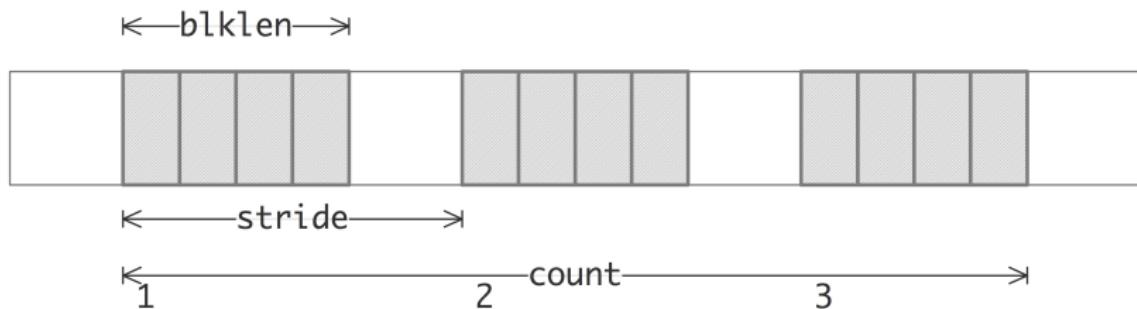
```
int MPI_Type_contiguous(  
    int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)
```



This one is indistinguishable from just sending `count` instances of the `old_type`.

## Vector type

```
int MPI_Type_vector(  
    int count, int blocklength, int stride,  
    MPI_Datatype old_type, MPI_Datatype *newtype_p  
) ;
```



Used to pick a regular subset of elements from an array.

```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (mytid==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (mytid==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}
```

Note: send and receive type not the same!

## Exercise 21

Let processor 0 have an array  $x$  of length  $10P$ , where  $P$  is the number of processors. Elements  $0, P, 2P, \dots, 9P$  should go to processor zero,  $1, P+1, 2P+1, \dots$  to processor 1, et cetera. Code this as a sequence of send/recv calls, using a vector datatype for the send, and a contiguous buffer for the receive.

For testing, define the array as  $x[i] = i$ .

## Exercise 22

Allocate a matrix on processor zero, using Fortran column-major storage. Using  $P$  sendrecv calls, distribute the rows of this matrix among the processors.

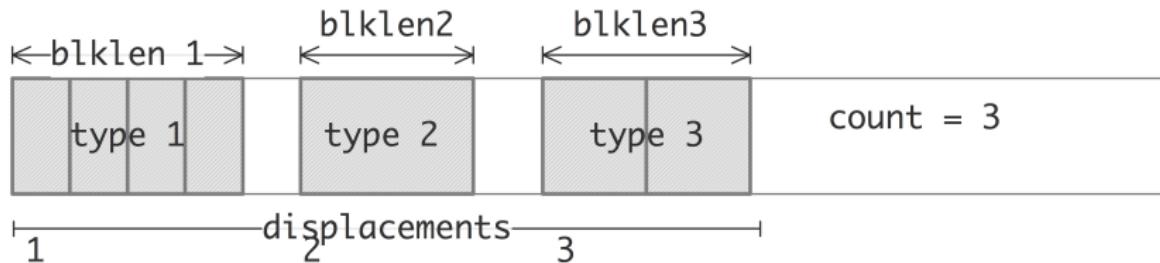
## Indexed type

```
int MPI_Type_indexed(  
    int count, int blocklens[], int indices[],  
    MPI_Datatype old_type, MPI_Datatype *newtype);
```

Also hindexed with byte offsets.

## Heterogeneous: Structure type

```
int MPI_Type_create_struct(  
    int count, int blocklengths[], MPI_Aint displacements[],  
    MPI_Datatype types[], MPI_Datatype *newtype);
```



This gets very tedious...

# Table of Contents

9 Discussion

10 Datatypes

11 Packed data

# Packing into buffer

```
int MPI_Pack(  
    void *inbuf, int incount, MPI_Datatype datatype,  
    void *outbuf, int outcount, int *position,  
    MPI_Comm comm);  
  
int MPI_Unpack(  
    void *inbuf, int insize, int *position,  
    void *outbuf, int outcount, MPI_Datatype datatype,  
    MPI_Comm comm);
```

## Example

```
// pack.c
if (mytid==sender) {
    MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
    for (i=0; i<nsends; i++) {
        double value = rand()/(double)RAND_MAX;
        MPI_Pack(&value,1,MPI_DOUBLE,buffer,buflen,&position,comm);
    }
    MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
    MPI_Send(buffer,position,MPI_PACKED,other,0,comm);
} else if (mytid==receiver) {
    int irecv_value;
    double xrecv_value;
    MPI_Recv(buffer,buflen,MPI_PACKED,other,0,comm,MPI_STATUS_IGNORE);
    MPI_Unpack(buffer,buflen,&position,&nsends,1,MPI_INT,comm);
    for (i=0; i<nsends; i++) {
        MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,MPI_DOUBLE);
    }
}
```

# Sub-computations

# Sub-computations

Simultaneous groups of processes, doing different tasks, but loosely interacting:

- Simulation pipeline: produce input data, run simulation, post-process.
- Climate model: separate groups for air, ocean, land, ice.
- Quicksort: split data in two, run quicksort independently on the halves.
- Processor grid: do broadcast in each column.

New communicators are formed recursively from `MPI_COMM_WORLD`.

# Communicator duplication

Simplest new communicator: identical to a previous one.

```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *newcomm)
```

This is useful for library writers:

```
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

# Use of a library

```
library my_library(comm);
MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
my_library.communication_start();
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));
MPI_Waitall(2,request,status);
my_library.communication_end();
```

# Use of a library

```
int library::communication_start() {  
    int sdata=6,rdata;  
    MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));  
    MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));  
    return 0;  
}  
  
int library::communication_end() {  
    MPI_Status status[2];  
    MPI_Waitall(2,request,status);  
    return 0;  
}
```

## Wrong way

```
// commdup_wrong.cxx
class library {
private:
    MPI_Comm comm;
    int mytid, ntid, other;
    MPI_Request *request;
public:
    library(MPI_Comm incomm)  {
        comm = incomm;
        MPI_Comm_rank(comm, &mytid);
        other = 1-mytid;
        request = new MPI_Request[2];
    };
    int communication_start();
    int communication_end();
};
```

## Right way

```
// commdup_right.cxx
class library {
private:
    MPI_Comm comm;
    int mytid, ntid, other;
    MPI_Request *request;
public:
    library(MPI_Comm incomm) {
        MPI_Comm_dup(incomm, &comm);
        MPI_Comm_rank(comm, &mytid);
        other = 1-mytid;
        request = new MPI_Request[2];
    };
    ~library() {
        MPI_Comm_free(&comm);
    }
    int communication_start();
}
```

# Disjoint splitting

Split a communicator in multiple disjoint others.

Give each process a ‘colour’, group processes by colour:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  
                    MPI_Comm *newcomm)
```

## Row/column example

```
MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;

MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, mytid, &column_comm );

MPI_Bcast( data, ... column_comm );
```

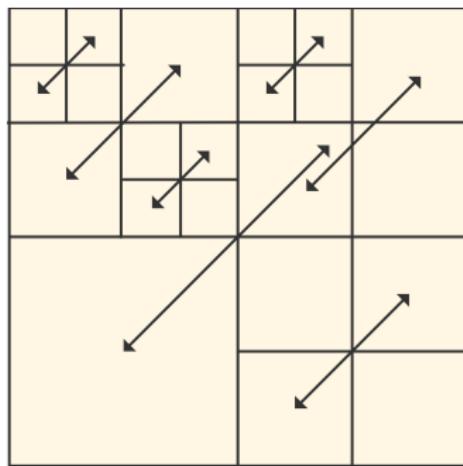
## Exercise 23

Organize your processors in a grid, and make subcommunicators for the rows and columns. Do a broadcast from the first row and column through the columns and rows respectively.

If you let the broadcast value be the column/row number, then processor  $(i, j)$  winds up with the numbers  $i$  and  $j$ . Test this.

## Exercise 24

Implement a recursive algorithm for matrix transposition:



- Swap blocks (1,2) and (2,1); then
- Divide the processors into four subcommunicators, and apply this algorithm recursively on each;
- If the communicator has only one process, transpose the matrix in place.

## More

- Non-disjoint subcommunicators through process groups.
- Intra-communicators and inter-communicators.
- Process topologies: cartesian and graph.

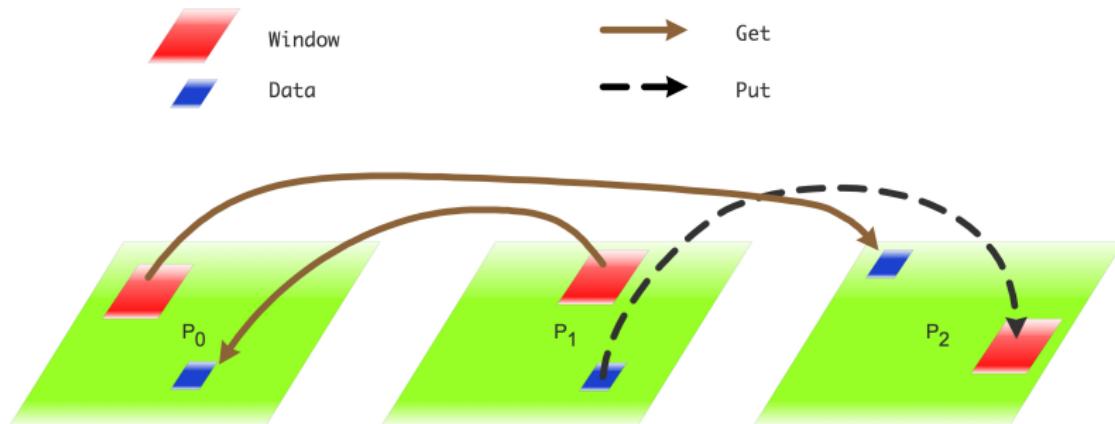
# One-sided communication

# Motivation

With two-sided messaging, you can not just put data on a different processor: the other has to expect it and receive it.

- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbours. Uncertain when this happens.
- Other irregular data structures: linked lists, hash tables.

# One-sided concepts



- A process has a *window* that other processes can access.
- Origin: process doing a one-sided call; target: process being accessed.
- One-sided calls: MPI\_Put, MPI\_Get, MPI\_Accumulate.
- Various synchronization mechanisms.

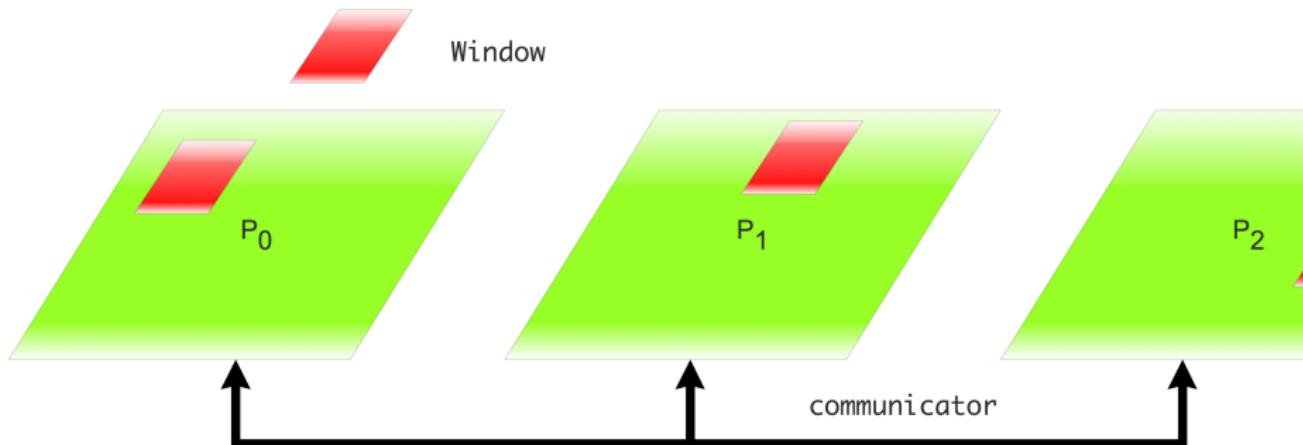
# Active target synchronization

All processes call MPI\_Win\_fence. Epoch is between fences:

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);  
if (mytid==producer)  
    MPI_Put( /* operands */, win);  
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Second fence indicates that one-sided communication is concluded:  
target knows that data has been put.

# Window creation



```
MPI_Win_create (void *base, MPI_Aint size,  
    int disp_unit, MPI_Info info,  
    MPI_Comm comm, MPI_Win *win)
```

- `size`: in bytes
- `disp_unit`: `sizeof(type)`
- Also: `MPI_Win_allocate`, can use dedicated fast memory.

C:

```
int MPI_Put(
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win)
```

Semantics:

IN origin\_addr: initial address of origin buffer (choice)  
IN origin\_count: number of entries in origin buffer (non-negative integer)  
IN origin\_datatype: datatype of each entry in origin buffer (handle)  
IN target\_rank: rank of target (non-negative integer)  
IN target\_disp: displacement from start of window to target buffer (non-negative integer)  
IN target\_count: number of entries in target buffer (non-negative integer)  
IN target\_datatype: datatype of each entry in target buffer (handle)  
IN win: window object used for communication (handle)

Fortran:

```
MPI_Put(origin_addr, origin_count, origin_datatype,
         target_rank, target_disp, target_count, target_datatype, win, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
```

## Exercise 25

Write code where process 0 randomly writes in the window on 1 or 2.

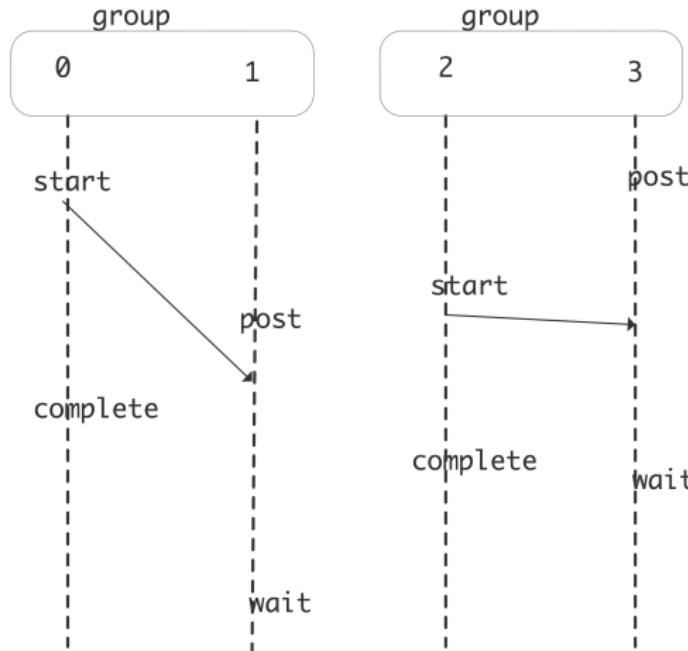
```
// randomput_skl.c
MPI_Win_create(&window_data, sizeof(int), sizeof(int),
                MPI_INFO_NULL, comm, &the_window);

for (int c=0; c<10; c++) {
    float randomfraction = (rand() / (double)RAND_MAX);
    if (randomfraction>.5)
        other = 2;
    else
        other = 1;
    window_data = 0;
    your_code_goes_here.....
    my_sum += window_data;
}

if (mytid>0 && mytid<3)
    Eijkhout MPI info f("Sum on %d: %d\n", mytid, my sum);
```

# A second active synchronization

Use Post, Wait, Start, Complete calls



More fine-grained than fences.

# Passive target synchronization

Lock a window on the target:

```
MPI_Win_lock (int locktype, int rank, int assert, MPI_Win win)  
MPI_Win_unlock (int rank, MPI_Win win)
```

Atomic operations:

```
int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,  
                     MPI_Datatype datatype, int target_rank, MPI_Aint target_  
                     MPI_Op op, MPI_Win win)
```

```
// passive.cxx
if (mytid==repository) {
    // Processor zero creates a table of inputs
    // and associates that with the window
}
if (mytid!=repository) {
    float contribution=(float)mytid,table_element;
    int loc=0;
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
    // read the table element by getting the result from adding
    err = MPI_Fetch_and_op(&contribution,&table_element,MPI_FLO
                           repository,loc,MPI_SUM,the_window); CHK(err);
    MPI_Win_unlock(repository,the_window);
}
```

# Index

- compare-and-swap, 79
- MPI\_Accumulate, 125
- MPI\_AllGather, 40
- MPI\_Allreduce, 40, **41**
- MPI\_ANY\_SOURCE, 65
- MPI\_ANY\_TAG, 65
- MPI\_CHAR, 99
- MPI\_COMM\_WORLD, 14, 112
- MPI\_DOUBLE, 99
- MPI\_DOUBLE\_PRECISION, 99
- MPI\_Finalize, 11
- MPI\_FLOAT, 99
- MPI\_Gatherv, **44**
- MPI\_Group, 125
- MPI\_INTEGER, 99
- MPI\_MAX, 31
- MPI\_Op, 31
- MPI\_Put, 125, **128**
- MPI\_REAL, 99
- MPI\_Recv, **64**, 68
- MPI\_Request, 84
- MPI\_Scan, **39**
- MPI\_Send, **63**, 68
- MPI\_STATUS\_IGNORE, 66
- MPI\_SUM, 31
- MPI\_Win\_allocate, 127
- MPI\_Win\_fence, 126

# Bibliography