TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Tutorial on MPI programming

Victor Eijkhout

SSiASC 2016

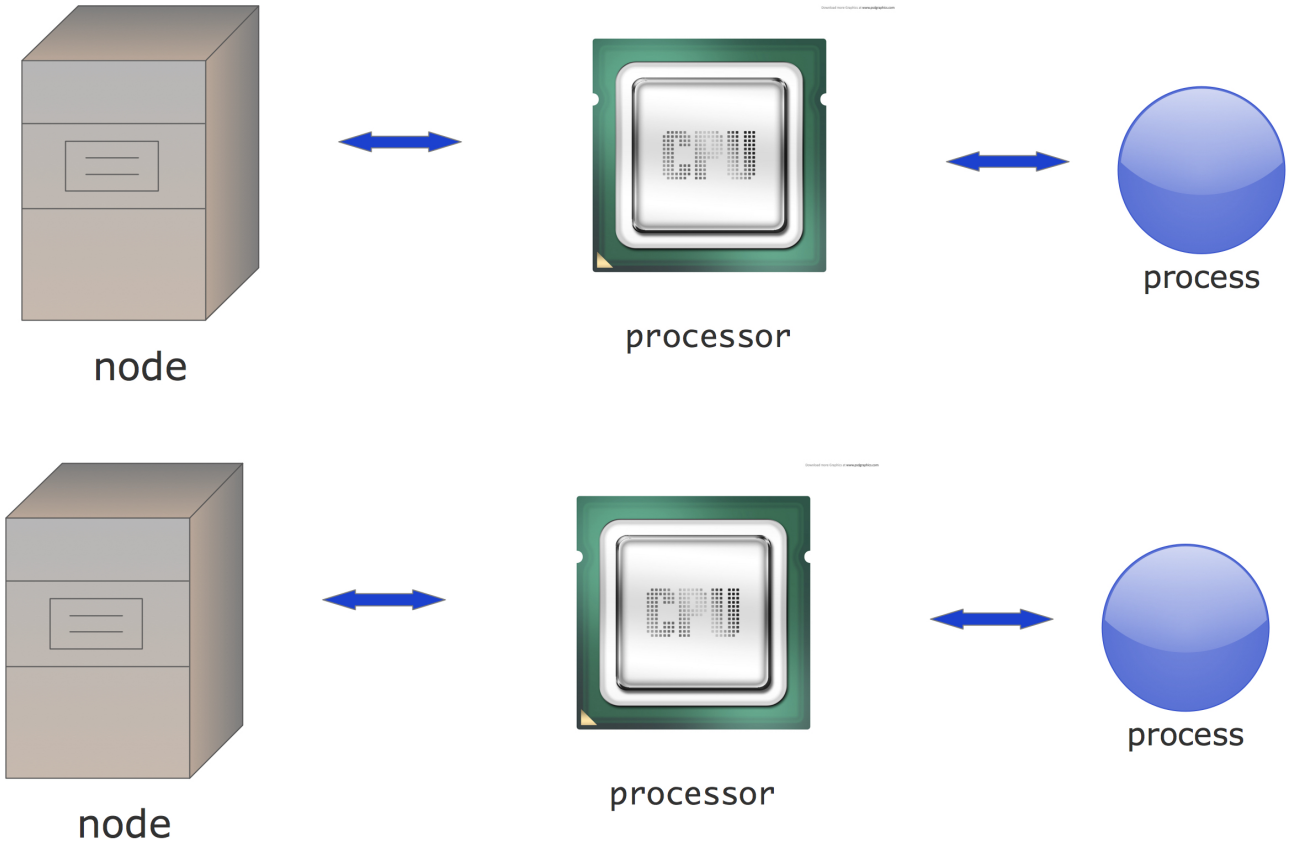
TACC

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.

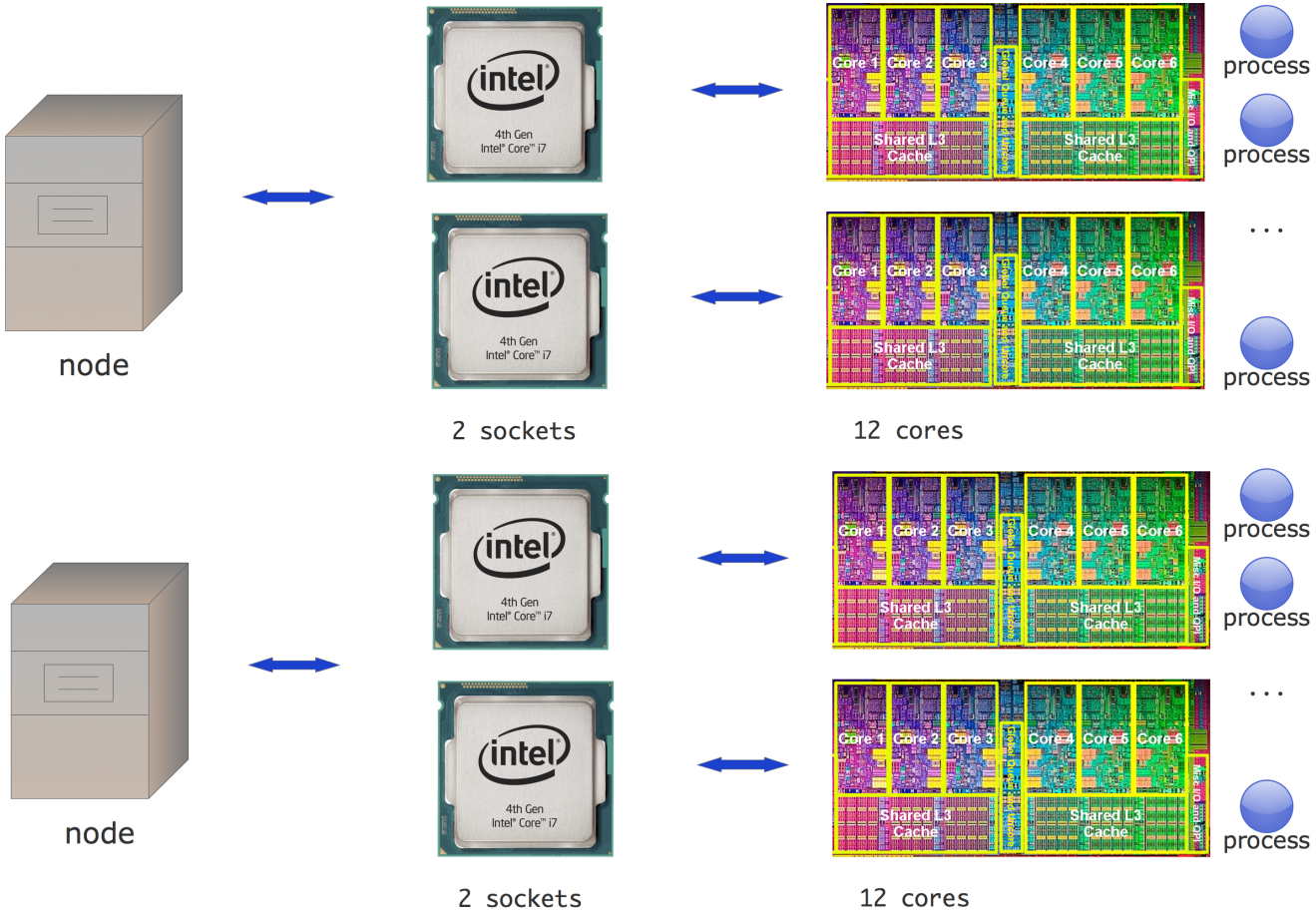
The SPMD model

Computers when MPI was designed



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

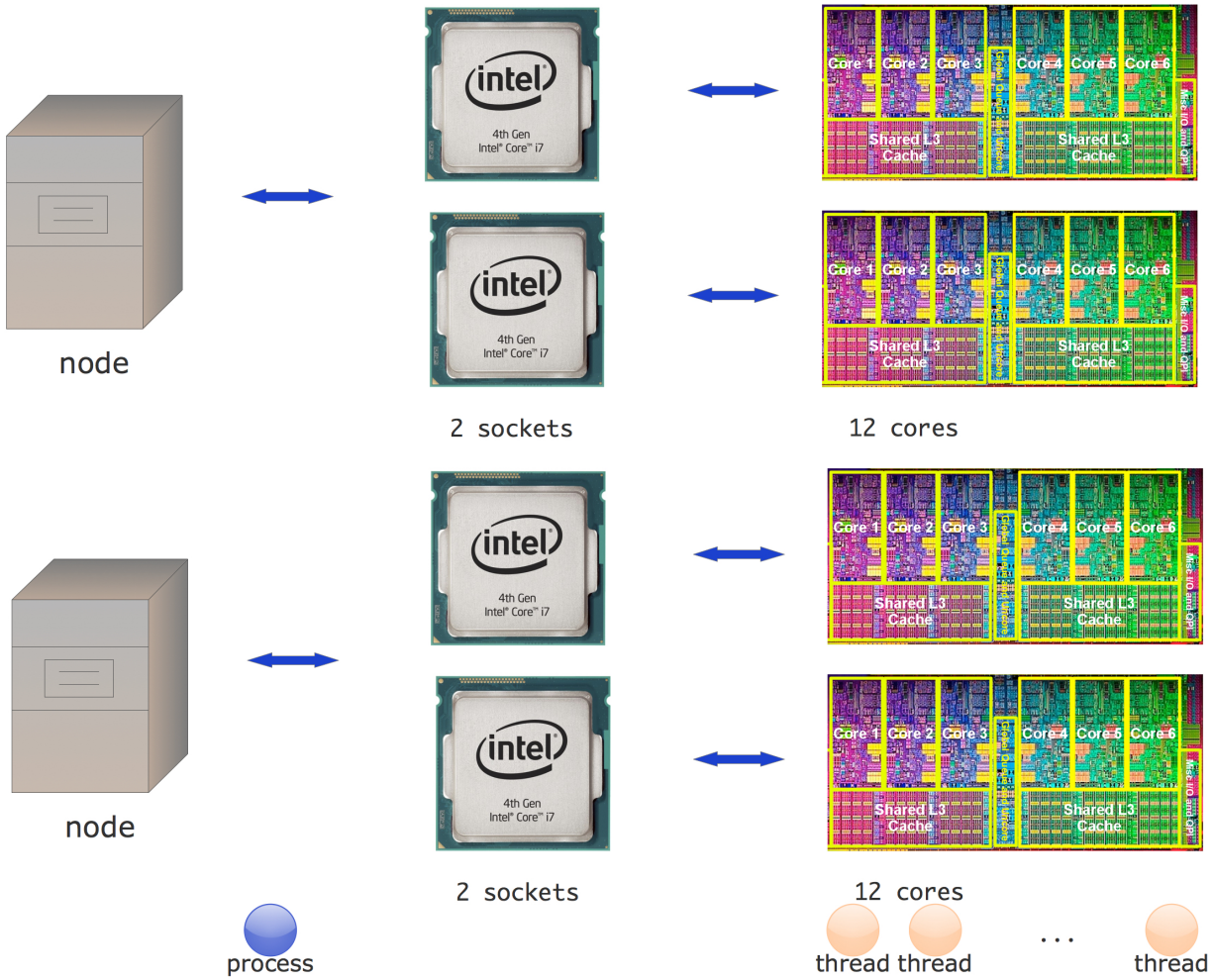
Pure MPI



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.

Compiling running

MPI compilers are usually called `mpicc`, `mpif90`, `mpicxx`. Use `mpicc` and such. These are not separate compilers, but scripts around the regular C/Fortran compiler.

Run your program with

```
mpiexec -n 4 hostfile ... yourprogram arguments
```

At TACC: `ibrun yourprog` without the number of procs.

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`, C++ is `mpicxx`, Fortran is `mpif90`. To run (on a compute node!) type `ibrun yourprog`.

No hostfiles or processor count needed!

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:

```
ierr = MPI_Init(&argc,&argv); // zeros allowed
// your code
ierr = MPI_Finalize();
```

and for Fortran:

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

About error codes

MPI routines return an integer error code

- In C: function result. Can be ignored.
- In Fortran: as parameter.
- In Python: throwing exception.

There's actually not a lot you can do with an error code:
very hard to recover from errors in parallel.

Python bindings

```
module python
```

```
from mpi4py import MPI
```

Run:

```
ibrun python-mpi yourprogram.py
```

No initialization needed.

About routine prototypes: C

Prototype:

```
int MPI_Comm_size(MPI_Comm comm,int *nprocs)
```

Use:

```
MPI_Comm comm = MPI_COMM_WORLD;
int nprocs;
int errorcode;
errorcode = MPI_Comm_world( MPI_COMM_WORLD,&nprocs
```

About routine prototypes: Fortran

Prototype

```
MPI_Comm_size(comm, size, ierror)
INTEGER, INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Use:

```
integer :: comm = MPI_COMM_WORLD
integer :: size
CALL MPI_Comm_size( comm, size, ierr )
```

- Final parameter always error parameter. Do not forget!
- Most `MPI_...` types are `INTEGER`.

About routine prototypes: Python

Prototype:

```
# object method
MPI.Comm.Send(self, buf, int dest, int tag=0)
# class method
MPI.Request.Waitall(type cls, requests, statuses=None)
```

Use:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
comm.Send(sendbuf,dest=other)
MPI.Request.Waitall(requests)
```

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 1 (`commrank`)

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 2 (`commrank`)

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 3 (`prime`)

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

Collectives

Table of Contents

Collectives

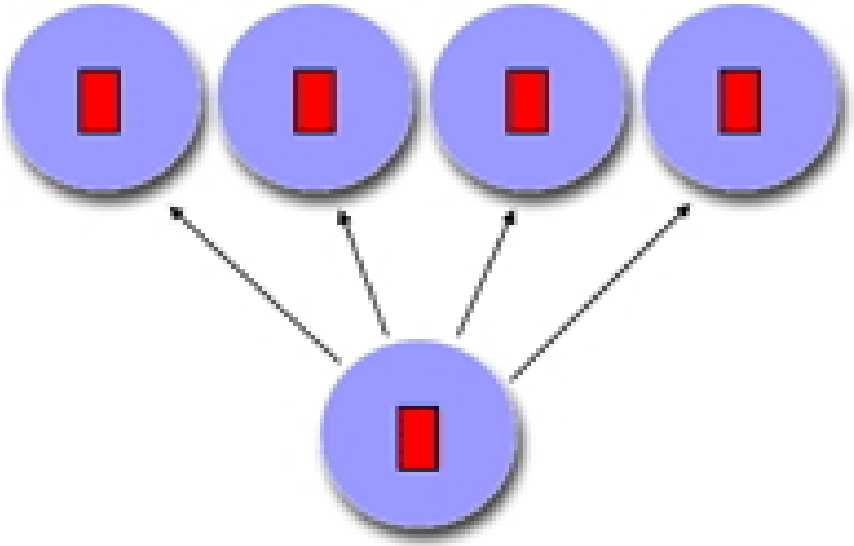
Gathering and spreading information:

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

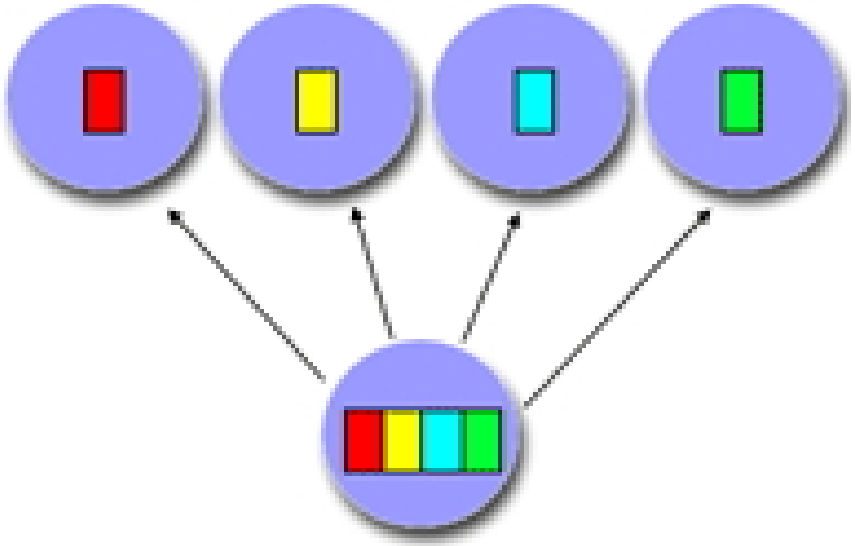
Root process: the one doing the collecting or disseminating.

Basic cases:

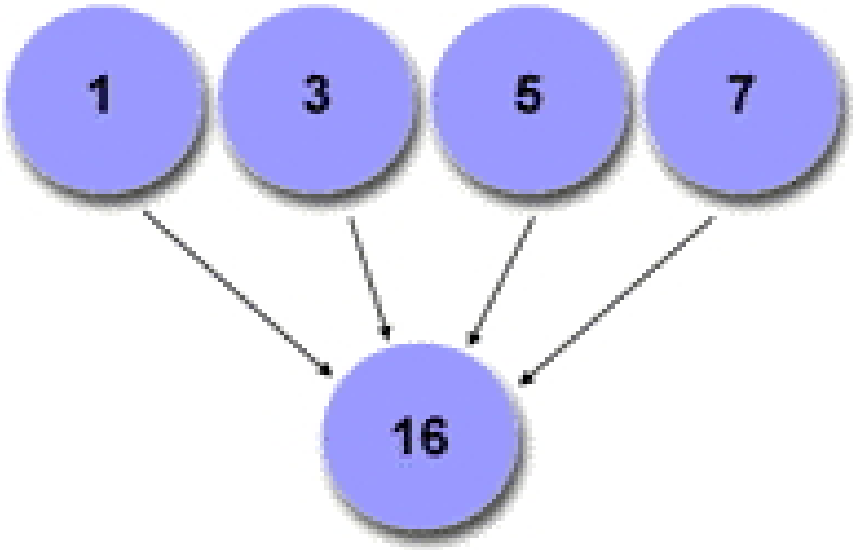
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



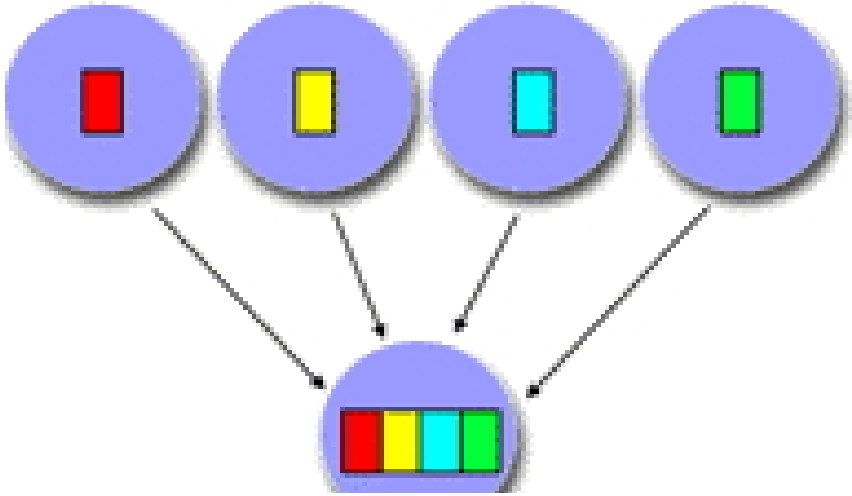
broadcast



scatter



reduction



Exercise 4

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

Broadcast

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

- All processes call with the same argument list
- `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.

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.

Allreduce

Regular reduce: great for printing out summary information at the end of your job.

Often: everyone needs the result of a reduction

$$y \leftarrow x/\|x\|$$

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

Why use allreduce?

Instead of reduce and broadcast.

- One line less code.
- Gives the implementation more possibilities for optimization.
- Is actually twice as fast: allreduce same time as reduce.

Exercise 5 (`randommax`)

Write a program where each process computes a random number, after which the maximum value over all processors is found. Each process then scales its value by this maximum. Use the `MPI_Allreduce` routine.

Random numbers

C:

```
// Initialize the random number generator
srand(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
```

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.

Also: `MPI_Allgather`

Table of Contents

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 $\sum_{q<p} n_q$.

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 root)
IN recvcounts: non-negative integer array (of length group size) containing t
IN displs: integer array (of length group size). Entry i specifies the displa
IN recvtype: data type of recv buffer elements (significant only at root) (ha
IN root: rank of receiving process (integer)
IN comm: communicator (handle)

Fortran:
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvty
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcounts(*),
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recv
TYPE(MPI_Comm), INTENT(IN) :: comm
```

All-to-all

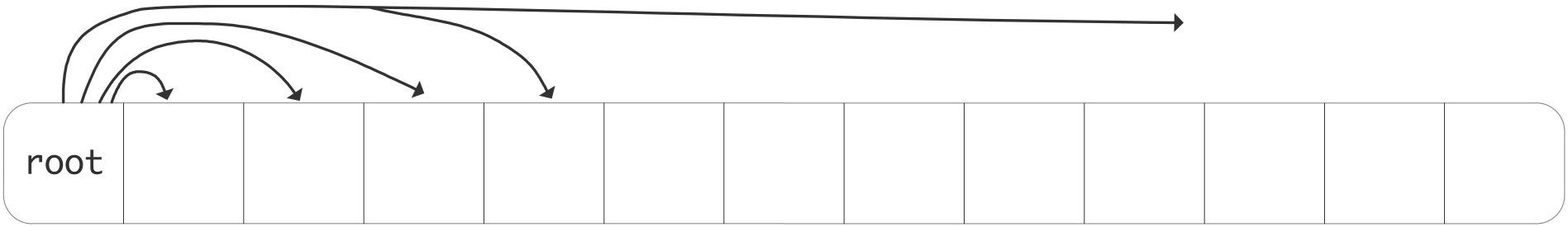
- Every process does a scatter;
- each individual data
- Very rarely needed.

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

Broadcast:

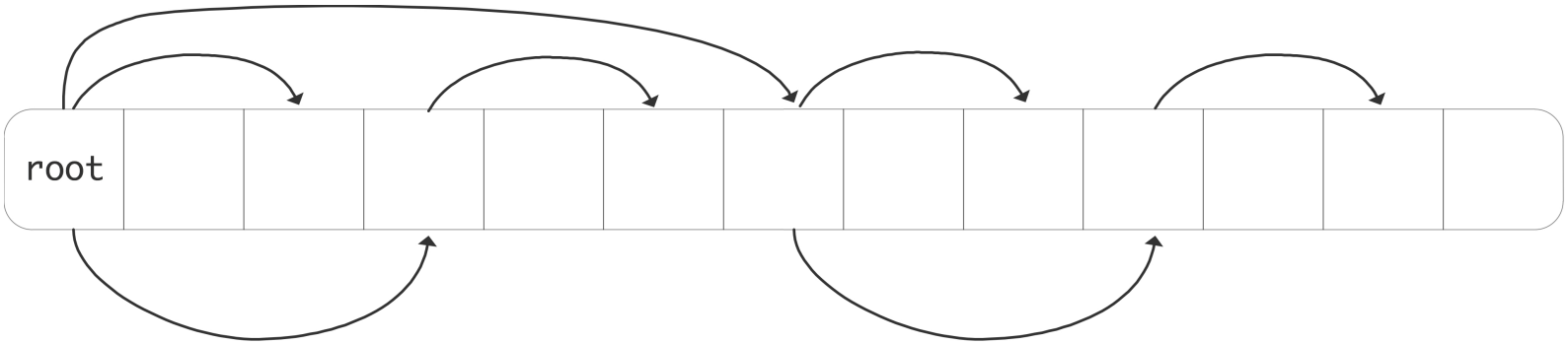


Message time is modeled as

$$\alpha + \beta n$$

Time for collective? Can you improve on that?

Better implementation of collective



What is the running time now?

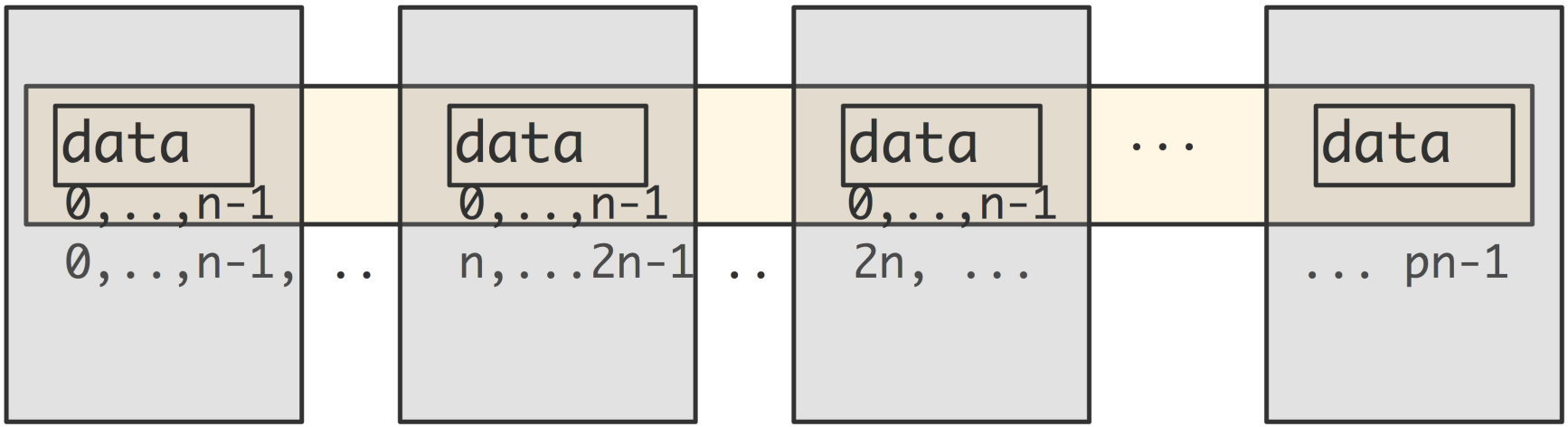
Point-to-point communication

Table of Contents

Distributed data

Distributed array: each process stores disjoint local part

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



Local numbering $0, \ldots, n_{\text{local}}$;
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 6 (`sumsquares`)

We want to compute $\sum_{n=1}^{N} n^2$, and we do that as follows by filling in an array and summing the elements. (Yes, you can do it without an array, but for purposes of the exercise do it with.)

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.
  (Allocate your array as real numbers. Why are integers not a good idea?)
- On each processor, initialize the local array so that the i-th location of the distributed array (for $i = 0, \ldots, N-1$) 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 $N = P$.

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?

(for future reference)

Let

$$f(i) = \lfloor iN/p \rfloor$$

and give processor i the points $f(i)$ up to $f(i+1)$.
Result:

$$\lfloor N/p \rfloor \leq f(i+1) - f(i) \leq \lceil N/p \rceil$$

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_Allreduce( &local_inprod, &global_inprod, 1,MPI_DOUBLE ...
```

Table of Contents

Motivation

Partial differential equations:

$$-\Delta u = -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{ for } \bar{x} \in \Omega = [0,1]^2 \text{ with } u(\bar{x}) = u_0 \text{ on } \delta\Omega.$$

Simple case:

$$-u_{xx} = f(x).$$

Finite difference approximation:

$$\frac{2u(x) - u(x+h) - u(x-h)}{h^2} = f(x, u(x), u'(x)) + O(h^2),$$

Motivation (continued)

Equations

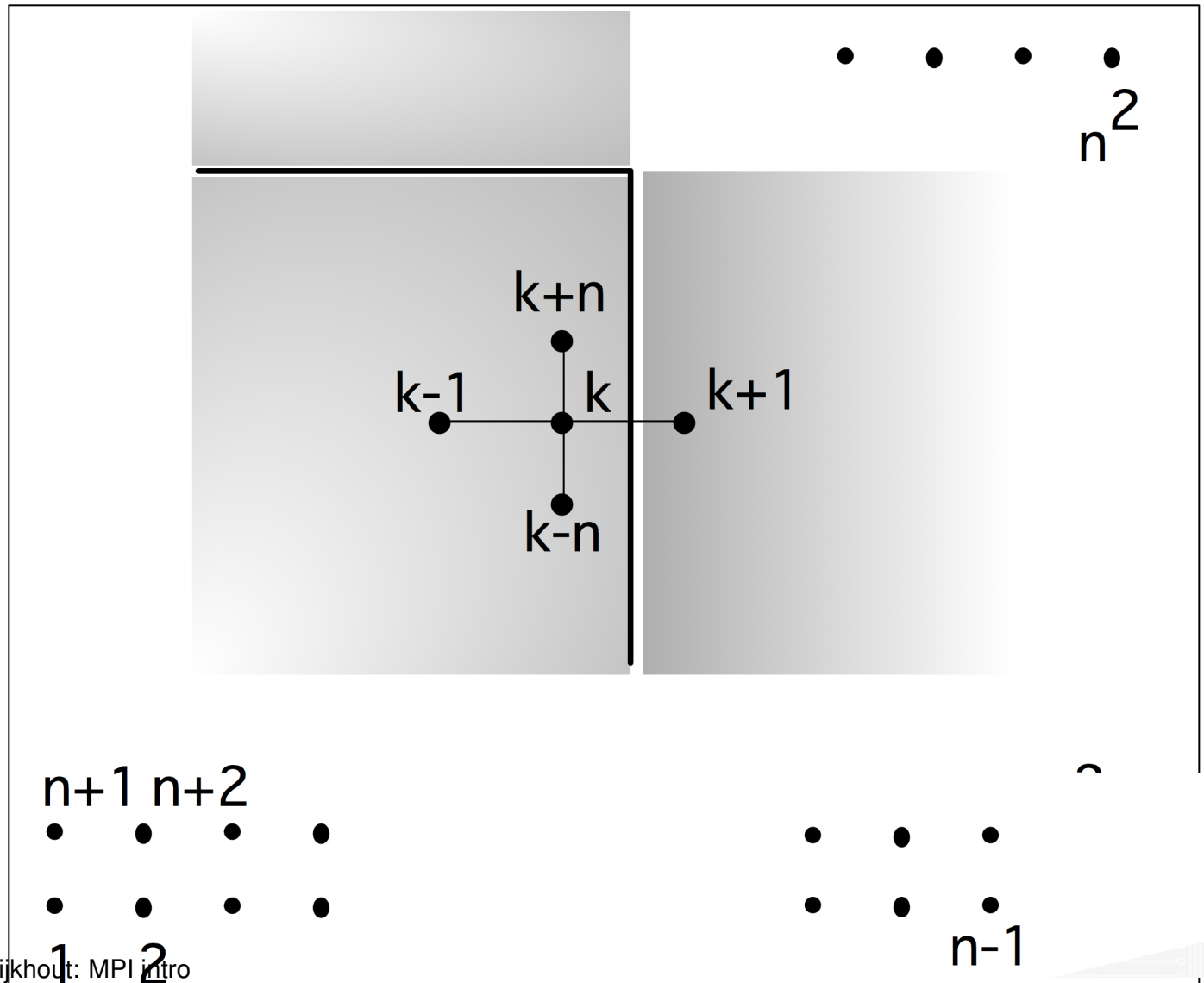
$$\begin{cases} -u_{i-1} + 2u_i - u_{i+1} = h^2 f(x_i) & 1 < i < n \\ 2u_1 - u_2 = h^2 f(x_1) + u_0 \\ 2u_n - u_{n-1} = h^2 f(x_n) + u_{n+1}. \end{cases}$$

$$\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} h^2 f_1 + u_0 \\ h^2 f_2 \\ \vdots \end{pmatrix} \tag{1}$$

So we are interested in sparse/banded matrices.

PDE, 2D case

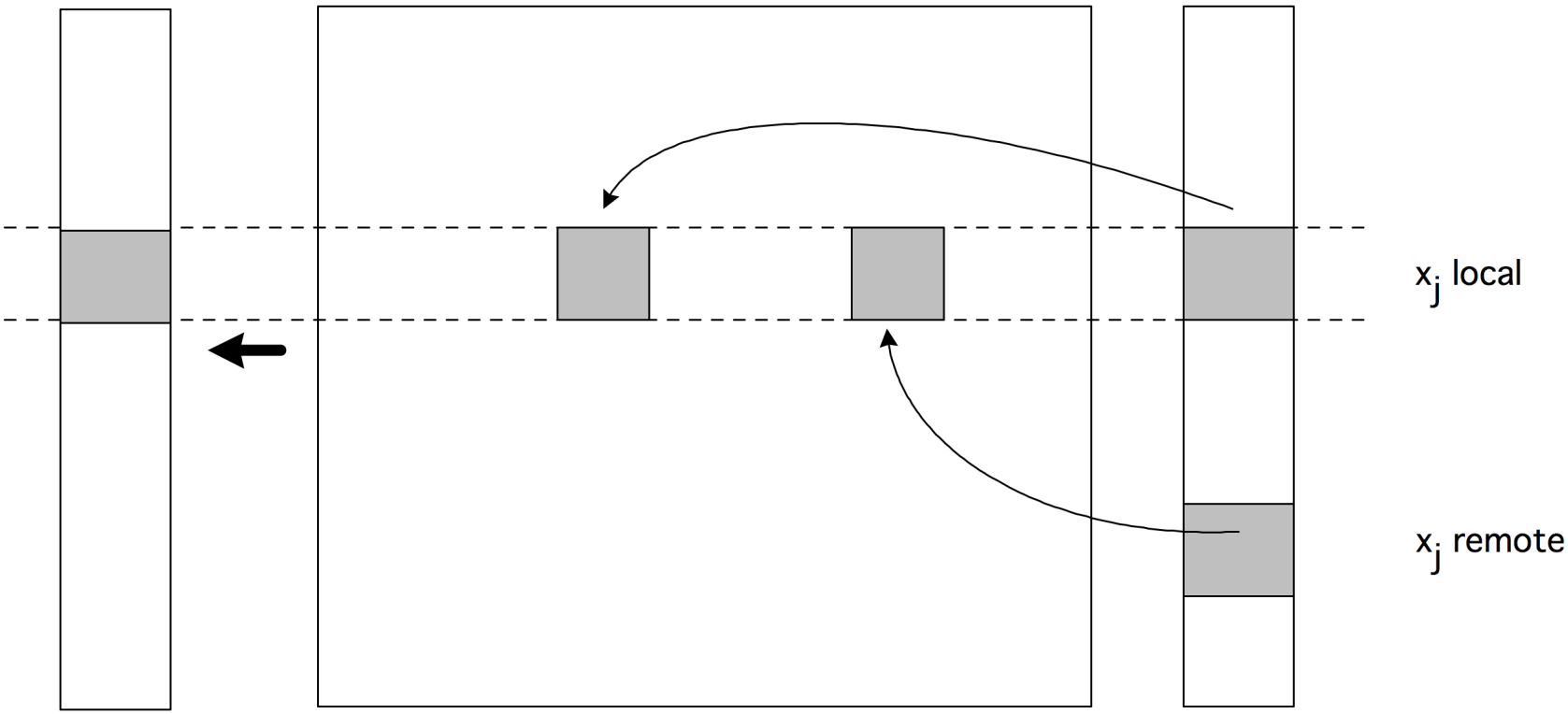
A difference stencil applied to a two-dimensional square domain, distributed over processors. A cross-processor connection is indicated.



Matrices in parallel

$$y \leftarrow Ax$$

and A, x, y all distributed:



Operating on distributed data

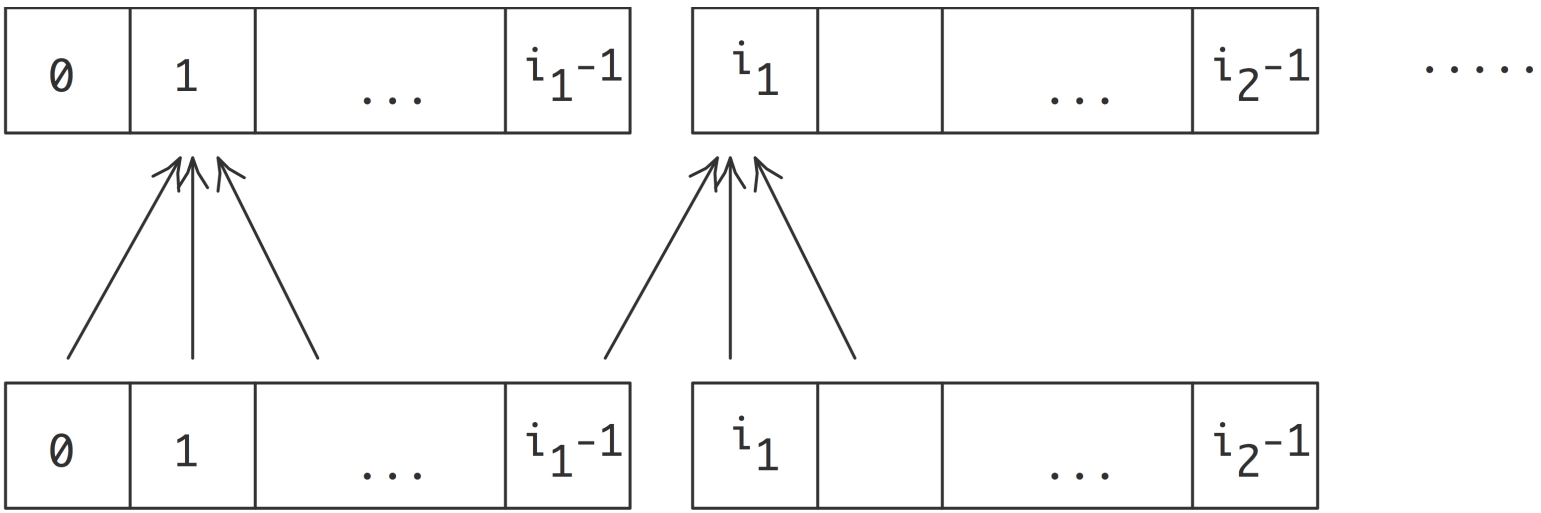
Array of numbers $x_i : i = 0, \ldots, N$

compute

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

'owner computes'

This leads to communication:



so we need a point-to-point mechanism.

MPI point-to-point mechanism

- Two-sided communication
- 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:
MPI.Comm.send(self, obj, int dest, int tag=0)
Python numpy:
```

```
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

Python native:
```

Status object

- Receive call can have various wildcards
- `MPI_ANY_SOURCE`, `MPI_ANY_TAG`
- use status object to retrieve actual description of the message
- use `MPI_STATUS_IGNORE` if the above does not apply

Exercise 7 (`pingpong`)

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

- Run multiple ping-pongs (say a thousand) and put the timer around the loop. The first run may take longer; try to discard it.
- Run your code with the two communicating processes first on the same node, then on different nodes. Do you see a difference?
- Then modify the program to use longer messages. How does the timing increase with message size?

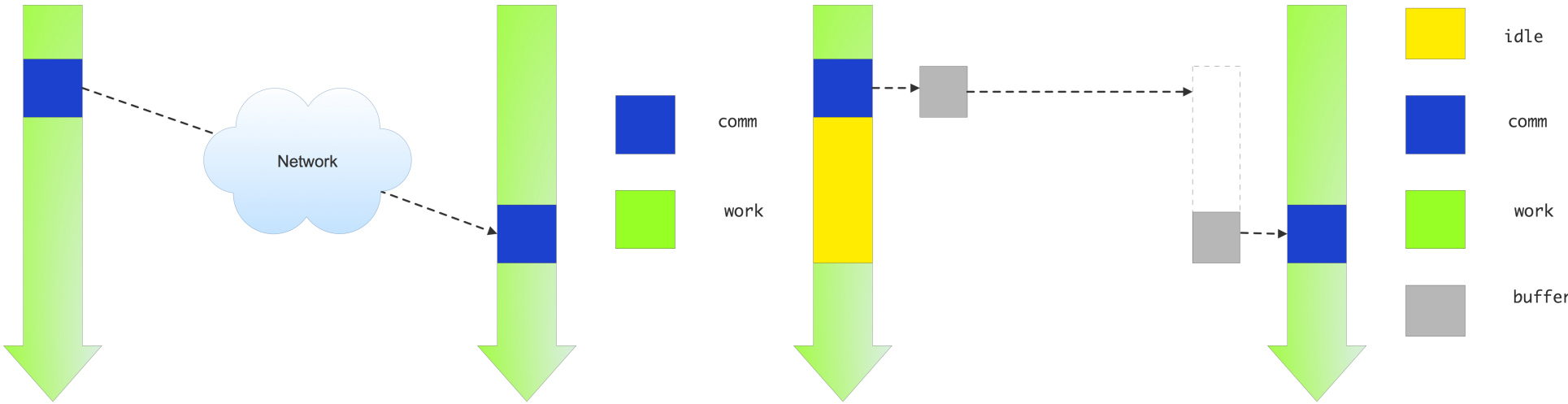
For bonus points, can you do a regression to determine α, β?

Table of Contents

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:

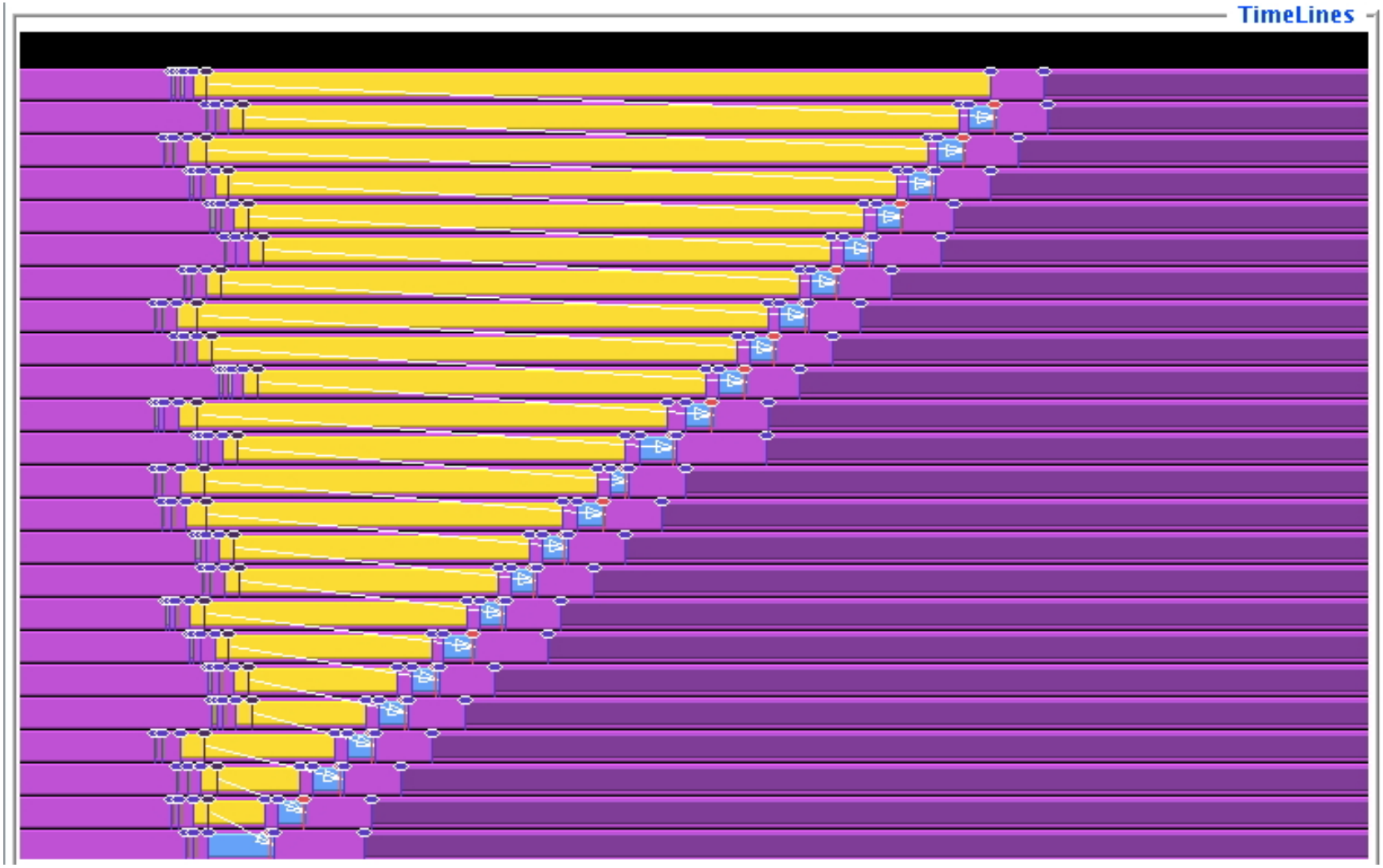
- Sender: ‘I have data for you’
- Receiver: ‘I have a buffer ready, send it over’
- Sender: ‘Ok, here it comes’
- Receiver: ‘Got it.’

Exercise 8

(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.

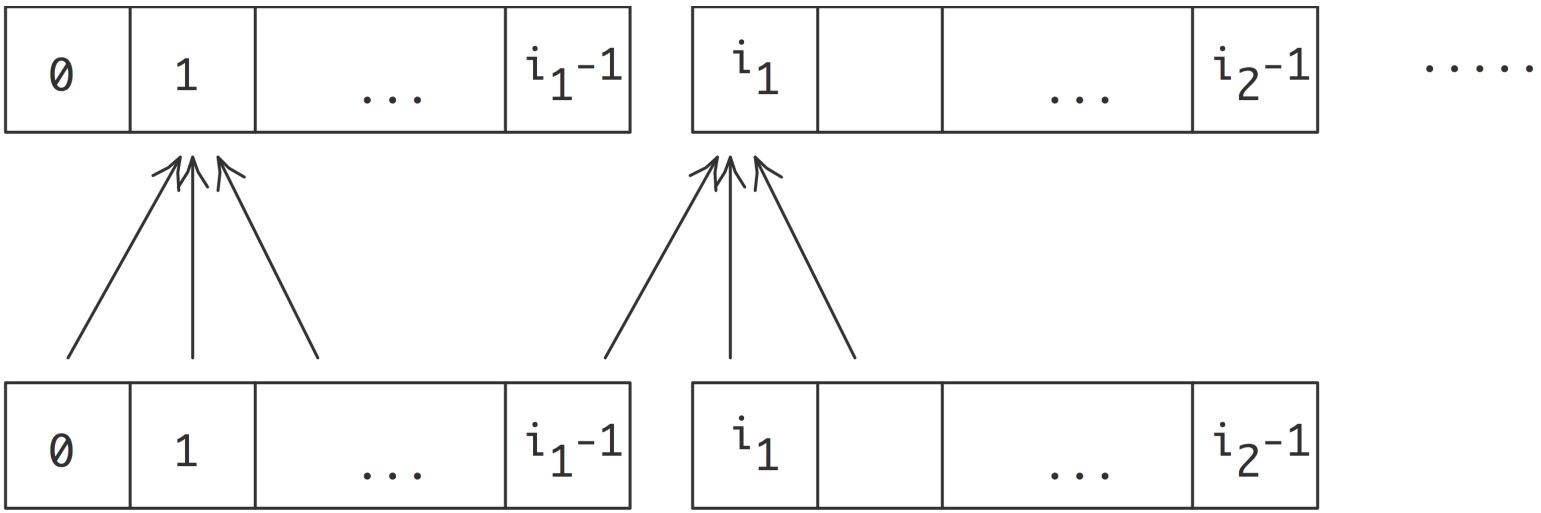
Beware! Blocking sends/receives can be trouble.
(How would you solve this particular case?)

Table of Contents

Operating on distributed data

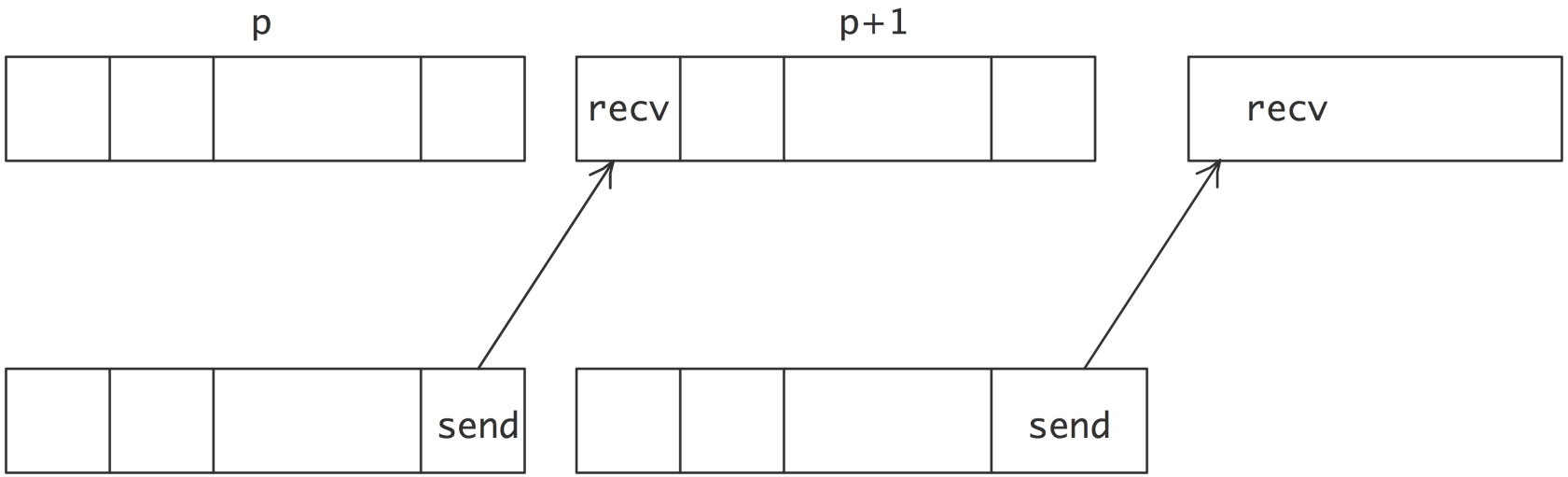
Take another look:

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



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

Not a good solution



First do all the data movement to the right.

- Each processor does a send and receive
- So every do send, then receive?
- We just saw the problem with that.

Sendrecv

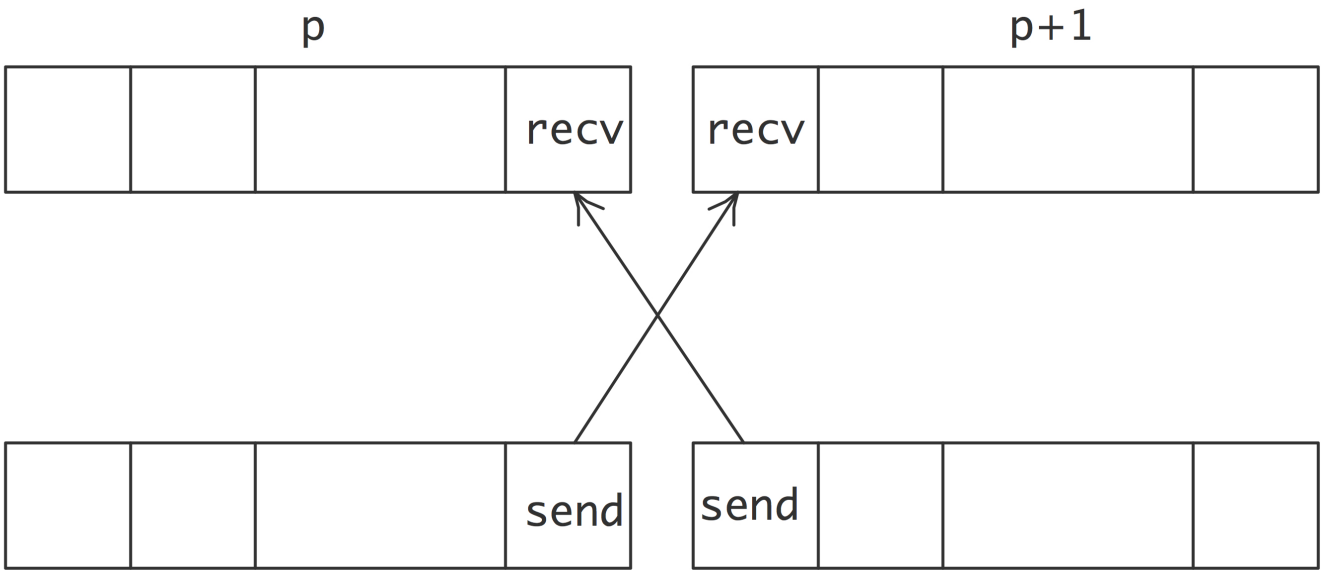
Instead of separate send and receive: use

```
Semantics:

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
recvcount, recvtype, source, recvtag, comm, status)
IN sendbuf: initial address of send buffer (choice)
IN sendcount: number of elements in send buffer (non-negative integer)
IN sendtype: type of elements in send buffer (handle)
IN dest: rank of destination (integer)
IN sendtag: send tag (integer)
OUT recvbuf: initial address of receive buffer (choice)
IN recvcount: number of elements in receive buffer (non-negative integer)
IN recvtype: type of elements in receive buffer (handle)
IN source: rank of source or MPI_ANY_SOURCE (integer)
IN recvtag: receive tag or MPI_ANY_TAG (integer)
IN comm: communicator (handle)
OUT status: status object (Status)

C:
int MPI_Sendrecv(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype
```

Pairwise exchange



Each *p* sends to right, receives from left;
then same to the left. (Other possbilities possible.)

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);
```

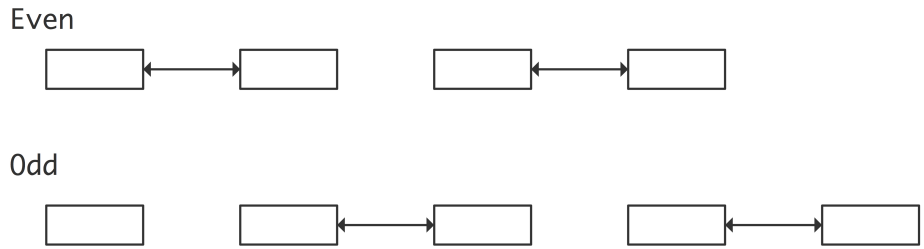
Exercise 9 (`sendrecv`)

Implement the above right-shift scheme using `MPI_Sendrecv`; every processor only has a single number to send to its neighbour.

If you have TAU installed, make a trace. Does it look different from the serialized send/recv code? If you don't have TAU, run your code with different numbers of processes and show that the runtime is essentially constant.

Exercise 10

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*[1]: 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.

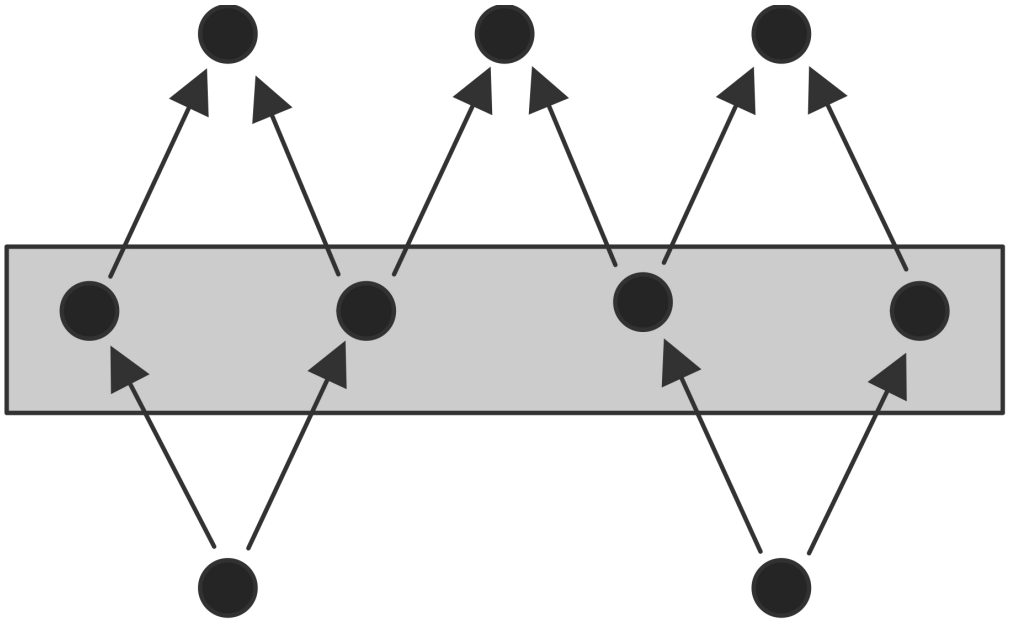
Implement this algorithm using `MPI_Sendrecv`. (You can use `MPI_PROC_NULL` for the edge cases, but that is not strictly necessary.) Use a gather call to p[illegible] distributed array at the beginning and end of the sorting proce[illegible]

[1] There is an `MPI_Compare_and_swap` call. Do not use that

Table of Contents

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 ▸ send and ▸ recv:

```
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[])
```

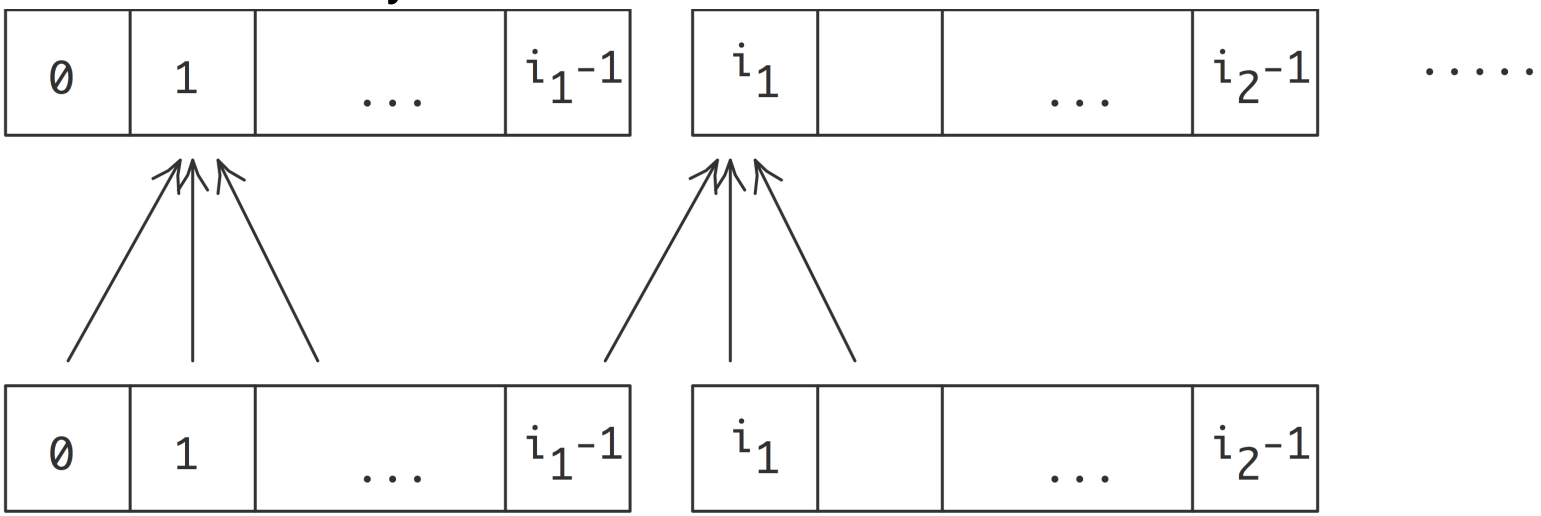
(also `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome`)

Exercise 11 (`isendirecv`)

Now use nonblocking send/receive routines to implement the three-point averaging operation

$$y_i = \left(x_{i-1} + x_i + x_{i+1}\right)/3 \colon\ i = 1, \ldots, N-1$$

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) {
  MPI_Test( /* from: */ ANY_SOURCE, &flag );
  if (flag)
    // 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

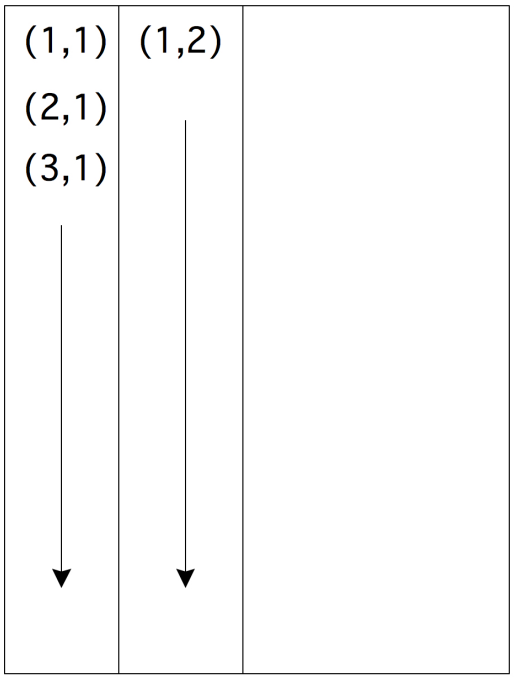
Table of Contents

Non-contiguous data

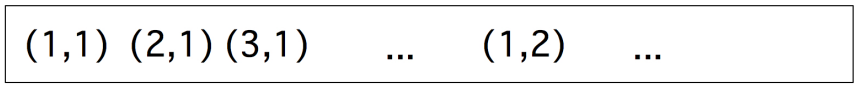
Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

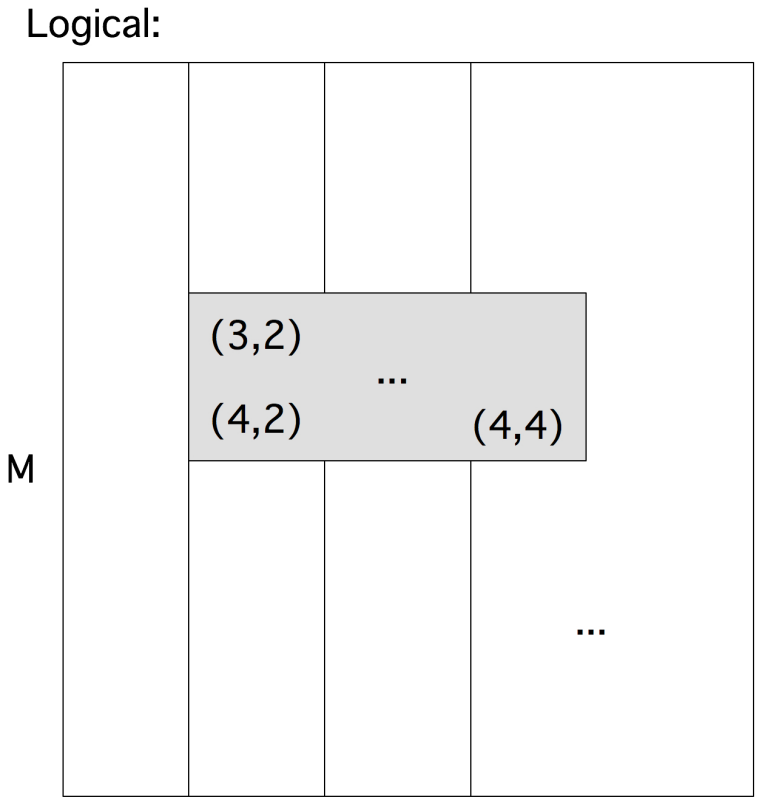
Logical:



Physical:



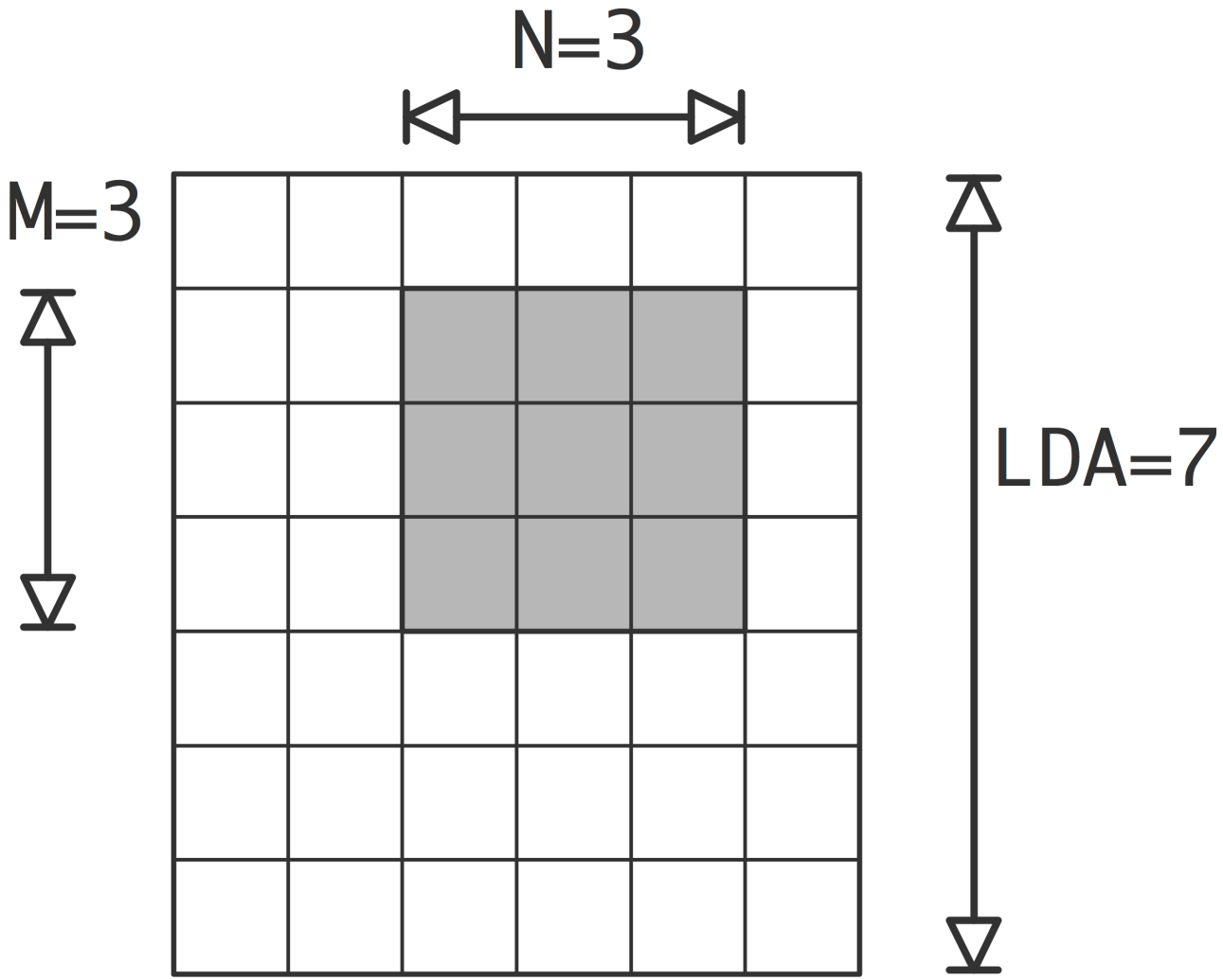
Submatrix storage



- Location of first element
- Stride, blocksize

BLAS/Lapack storage

Three parameter description:



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

Elementary datatypes

C/C++	Fortran
MPI_CHAR	MPI_CHARACTER
MPI_UNSIGNED_CHAR	
MPI_SIGNED_CHAR	
	MPI_LOGICAL
MPI_SHORT	
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 newtype;
MPI_Type_xxx( ... oldtype ... &newtype);
MPI_Type_commit ( &newtype );

  // code using the new type

MPI_Type_free ( &newtype );
```

The `oldtype` can be elementary or derived.
Recursively constructed types.

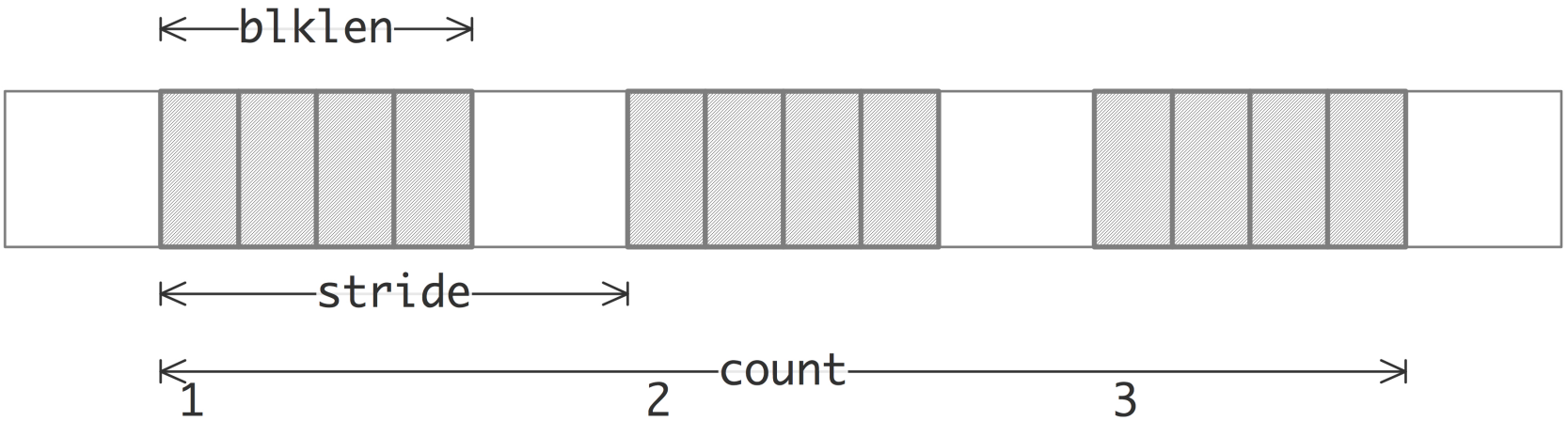
Contiguous type

```
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
);
```



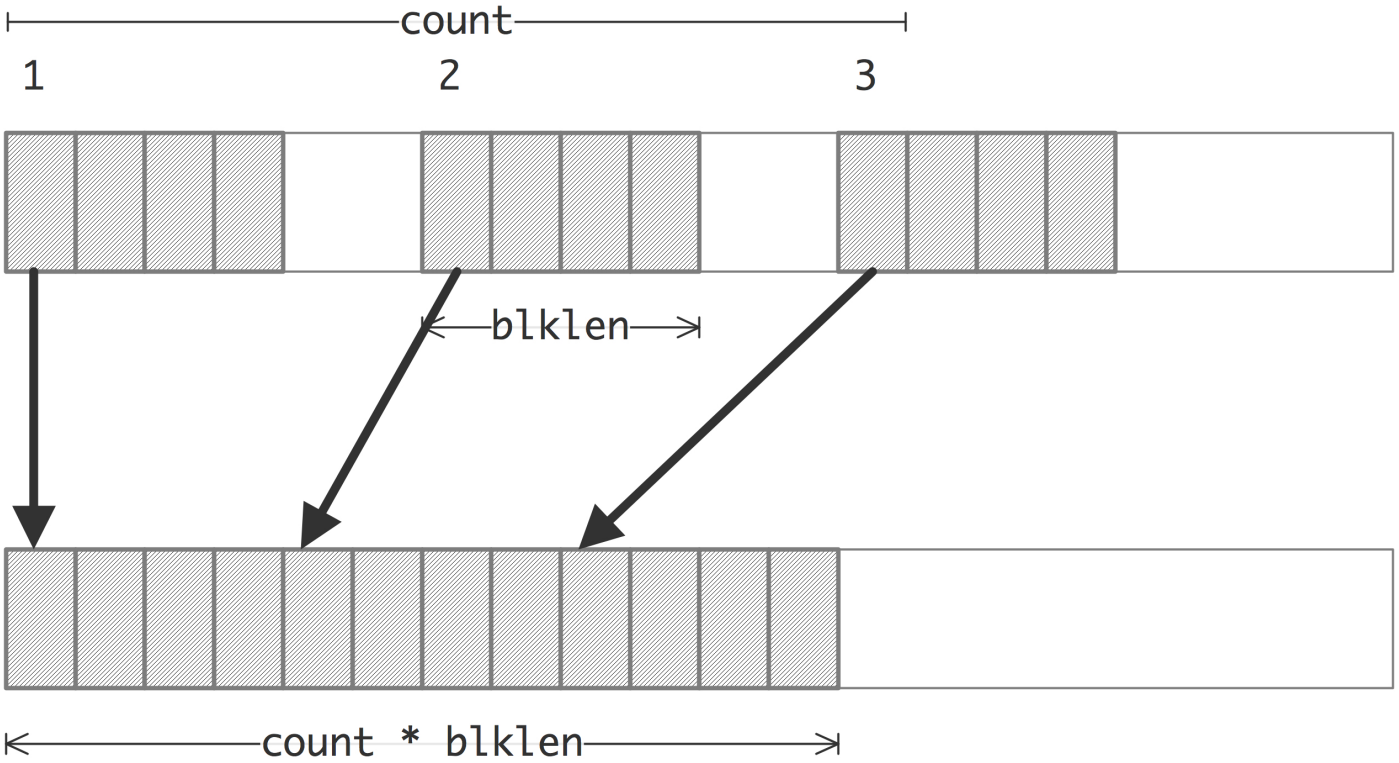
Used to pick a regular subset of elements from an array.

```
// 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);
}
```

Different send and receive types

Sender type: vector

receiver type: contiguous or elementary



Receiver has no knowledge of the stride of the sender.

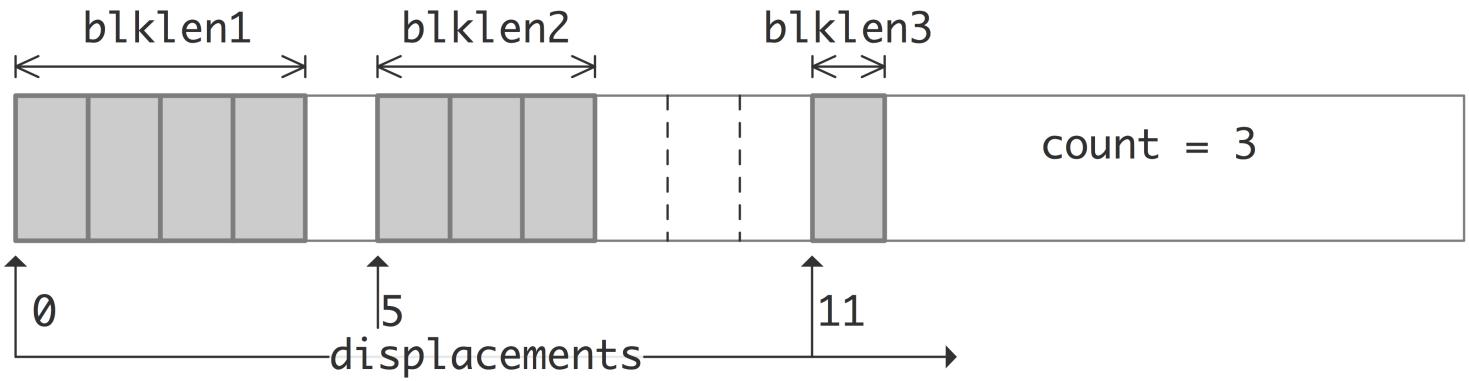
Exercise 12

Let processor 0 have an array x of length $10P$, where P is the number of processors. Elements $0, P, 2P, \ldots, 9P$ should go to processor zero, $1, P+1, 2P+1, \ldots$ 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 simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?

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

Indexed type

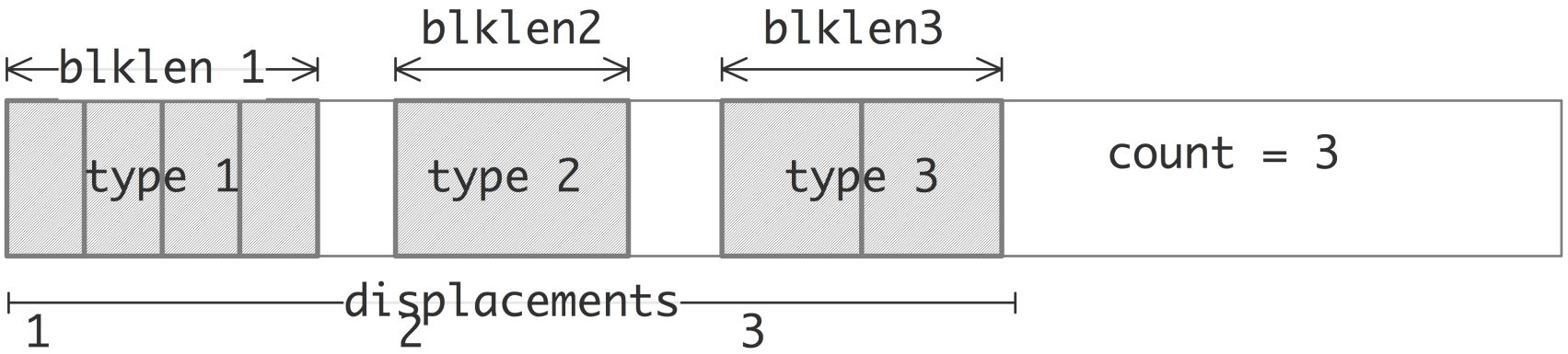



```
int MPI_Type_indexed(
  int count, int blocklens[], int displacements[],
  MPI_Datatype old_type, MPI_Datatype *newtype);
```

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

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 (int 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 (int i=0; i<nsends; i++) {
    MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,MPI_DOUBLE,comm
  }
  MPI_Unpack(buffer,buflen,&position,&irecv
  ASSERT(irecv_value==nsends);
```

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,ntids,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,ntids,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 13

Organize your processes in a grid, and make subcommunicators for the rows and columns. First let each processor print out its global rank, column number and rank, and row number and rank. Then, design a gather operation that lets the root print out the state of all processors as a nicely formatted matrix. For instance, a 2×3 processor grid should print:

```
Global ranks:
  0  1  2
  3  4  4
Row ranks:
  0  1  2
  0  1  2
```

Initialize all processes in the first row with their column number and the ones in the first column with their row number; all others should be set to zero. Use a gather operation to print out this state of affairs.

Now do a broadcast from the first row and column through the columns and rows respectively; processor (i, j) winds up with the numbers i and j. Again use a gather to print this out.

Run your code on different number of processes, for instance
that is a power of 2, or that is a prime number. This is one occ
np. 13: normally you would *never* put a processor count on

More

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

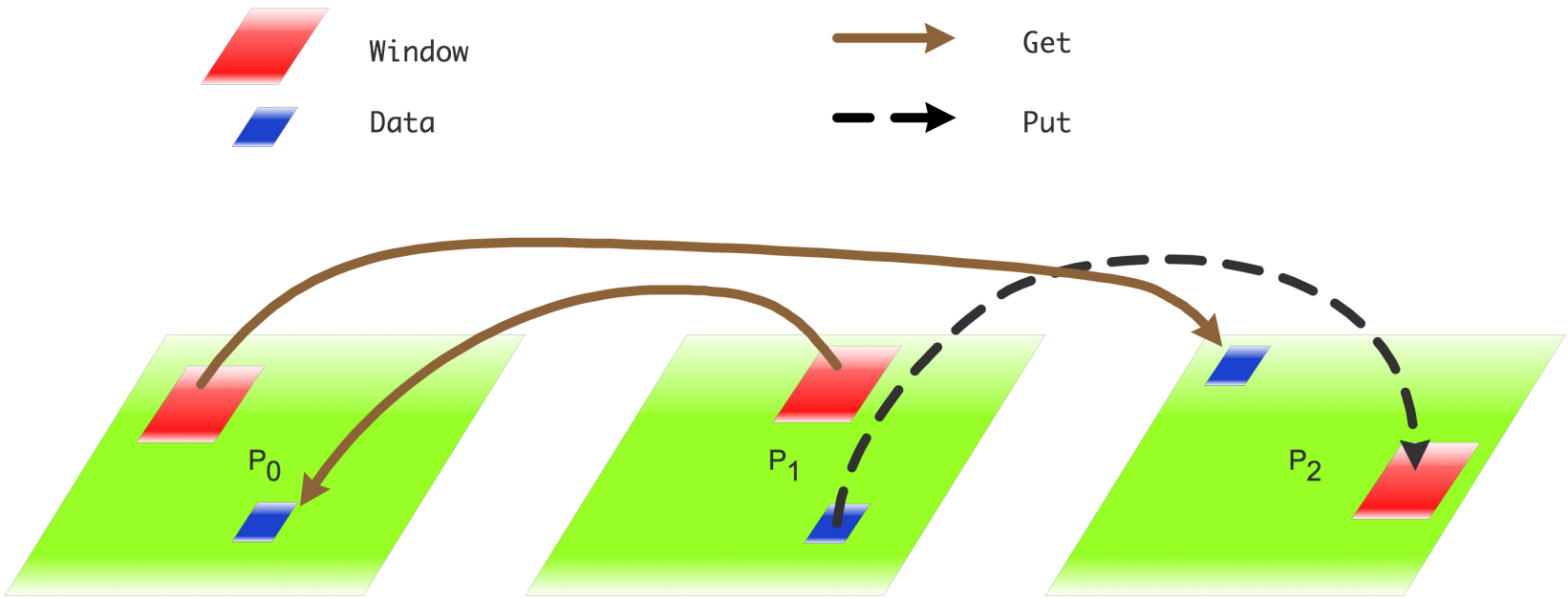
One-sided communication

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.

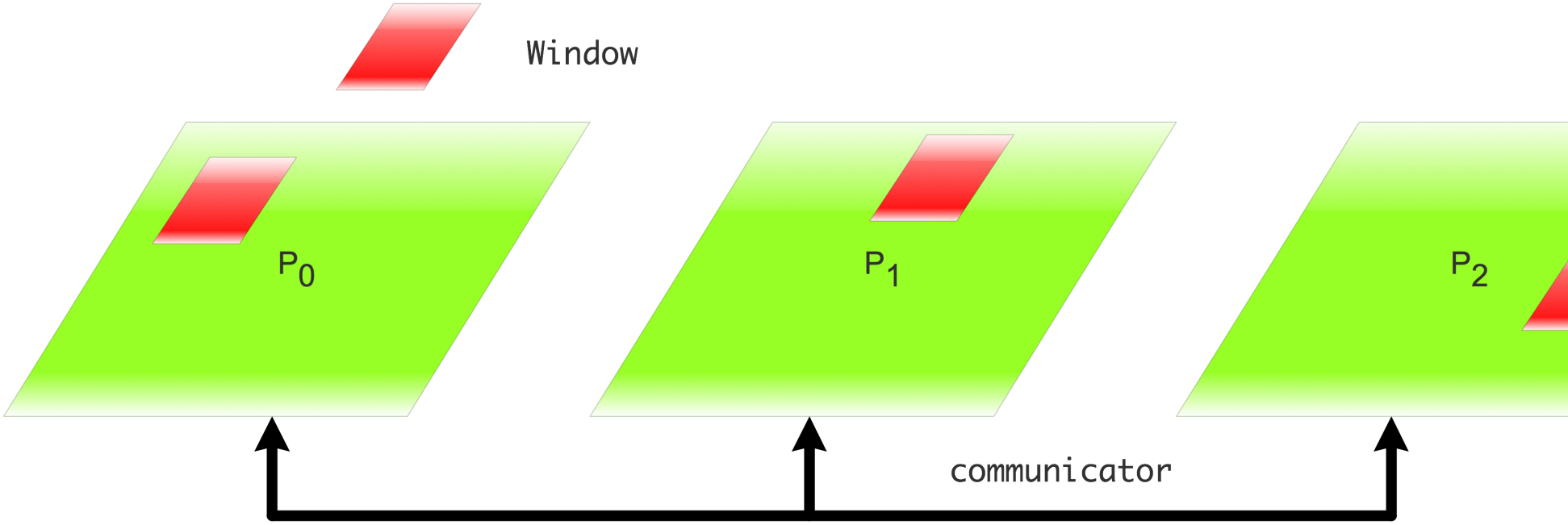
Active target synchronization

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 dedicate

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 targe
  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-negat
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
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatyp
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: ta
TYPE(MPI_Win), INTENT(IN) :: win
```

Exercise 14

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

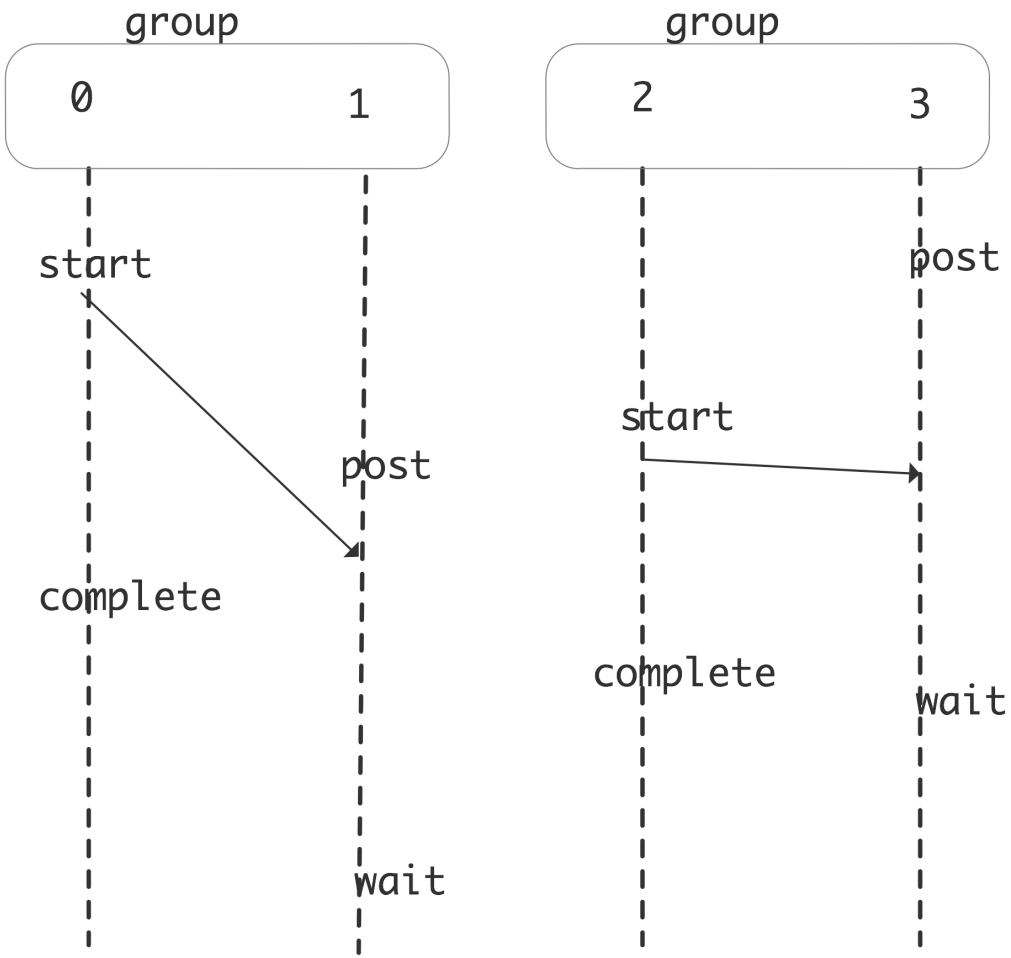
```
// randomput_skl.c
MPI_Win_create(&window_data,sizeof(int),sizeof(int),
               MPI_INFO_NULL,comm,&the_window);

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)
  printf("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) {
  // Repository processor 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 ze
  err = MPI_Fetch_and_op
    (&contribution,&table_element,MPI_FLOAT,
     repository,loc,MPI_SUM,the_window); CHK(err);
  MPI_Win_unlock(repository,the_window);
}
```

Index