# MPI Basics

Victor Eijkhout

2025 TACC APPI

# Materials

Textbooks and repositories:
`https://theartofhpc.com`

# 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.

# Supercomputer clusters

# Cluster setup

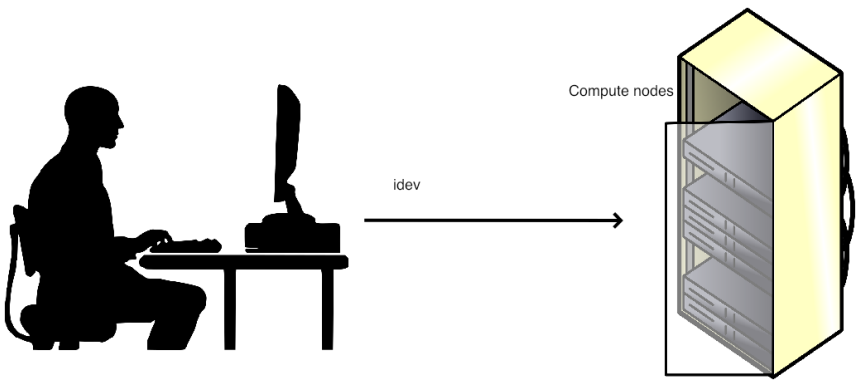
Typical cluster:

- Login nodes, where you ssh into; usually shared with 100 (or so) other people. You don't run your parallel program there!
- Compute nodes: where your job is run. They are often exclusive to you: no other users getting in the way of your program.

Hostfile: the description of where your job runs. Usually generated by a *job scheduler*.

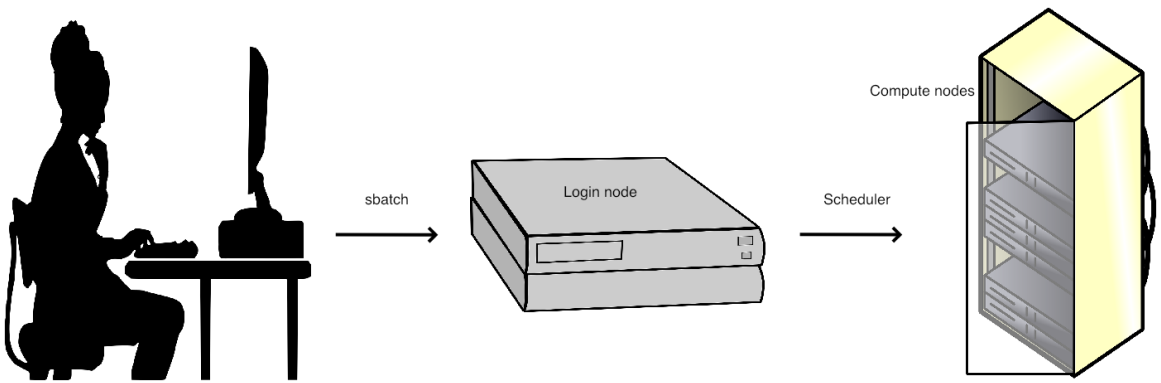
# Interactive run

- Do not run your programs on a login node.
- Acquire compute nodes with `idev`
- Caveat: only small short jobs; nodes may not be available.



# Batch run

- Submit batch job with *sbatch* or *qsub*
- Your job will be executed . . . Real Soon Now.
- See userguide for details about queues, sizes, runtimes, . . .



# Exercise 1

- Connect to your favorite cluster
  what is the hostname? how many users are logged in?
- Start an interactive session with `idev`;
  what is the hostname? how many users are logged in?
- Run: `ibrun hostname`
  also `ibrun -n 3 hostname`
- Same, but `idev` on two nodes.
- Create a job script that will run on 10 nodes;
  again let it run the `hostname` command.

# The SPMD model

# Overview

In this section you will learn how to think about parallelism in MPI.

Commands learned:

- `MPI_Init`, `MPI_Finalize`,
- `MPI_Comm_size`, `MPI_Comm_rank`
- `MPI_Get_processor_name`,

# Practicalities

# Lab setup

- Clone the repository `https://github.com/VictorEijkhout/TheArtOfHPC_vol2_parallelprogramming`
- Directory: `exercises-mpi-c` or `cxx` or `f` or `f08` or `p` or `mpl`
- Open a terminal window on a TACC cluster.
- Type `idev -N 2 -n 10 -t 2:0:0` which gives you an interactive session of 2 nodes, 10 cores, for the next 2 hours.
- Type `make exercisename` to compile it
- Run with `ibrun` or `mpiexec`

# Compiling

MPI compilers are usually called *mpicc*, *mpif90*, *mpicxx*.

These are not separate compilers, but scripts around the regular C/Fortran compiler. You can use all the usual flags.

```
$ mpicc -show
icx -I/intel/include/stuff -L/intel/lib/stuff -Wwarnings # et cetera
```

(Python of course no compilation needed)

(For CMake see slide 202.)

# Running (at TACC)

In your batch job indicate node and core count

```
1 #SBATCH -N 4
2 #SBATCH -n 200
3 ibrun yourprog
```

No specification on `ibrun` needed!

# Running (in general)

On non-TACC machines:

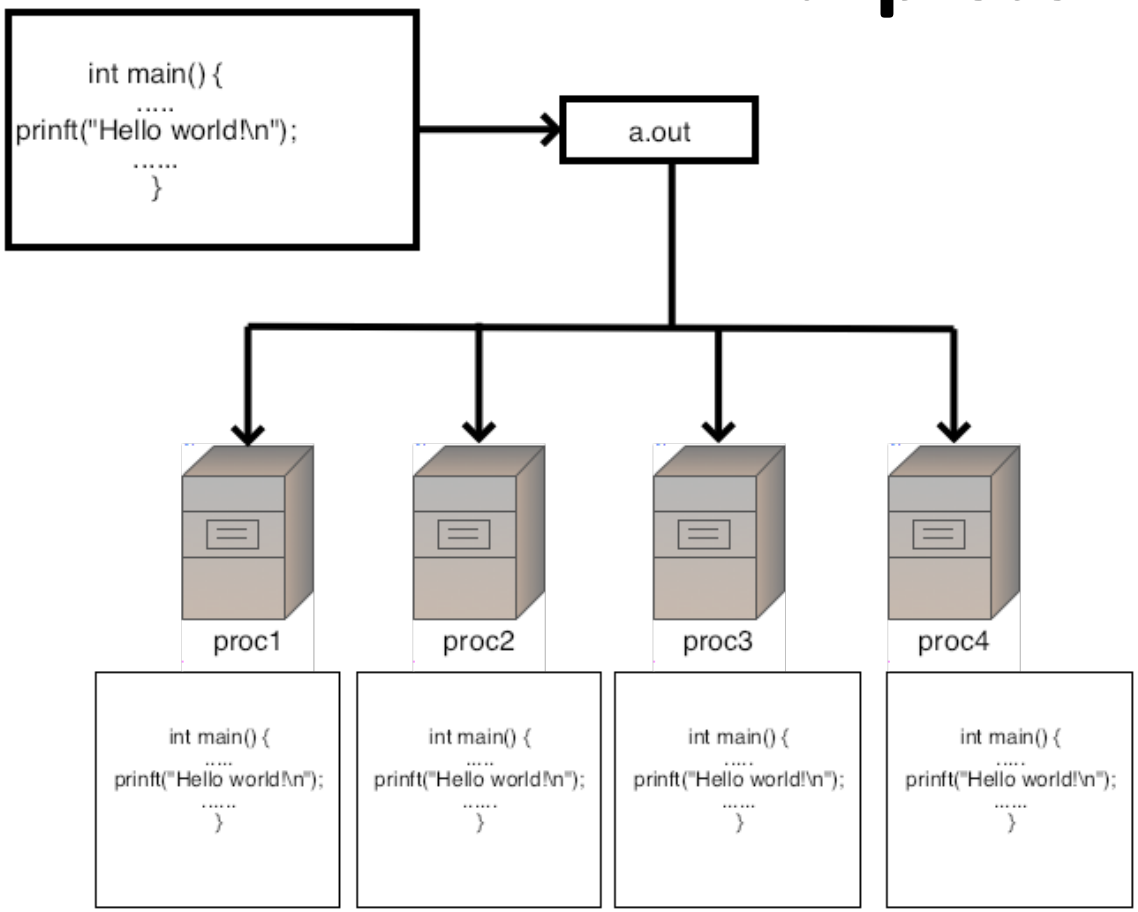
```
mpiexec -n 4 hostfile ... yourprogram arguments
mpirun -np 4 hostfile ... yourprogram arguments
```

# Exercise 2 (`hello`)

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

1. In the directories `exercises-mpi-xxx` do `make hello` to compile;
2. do `ibrun hello` to execute.

# In a picture


int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}

# The MPI worldview: SPMD

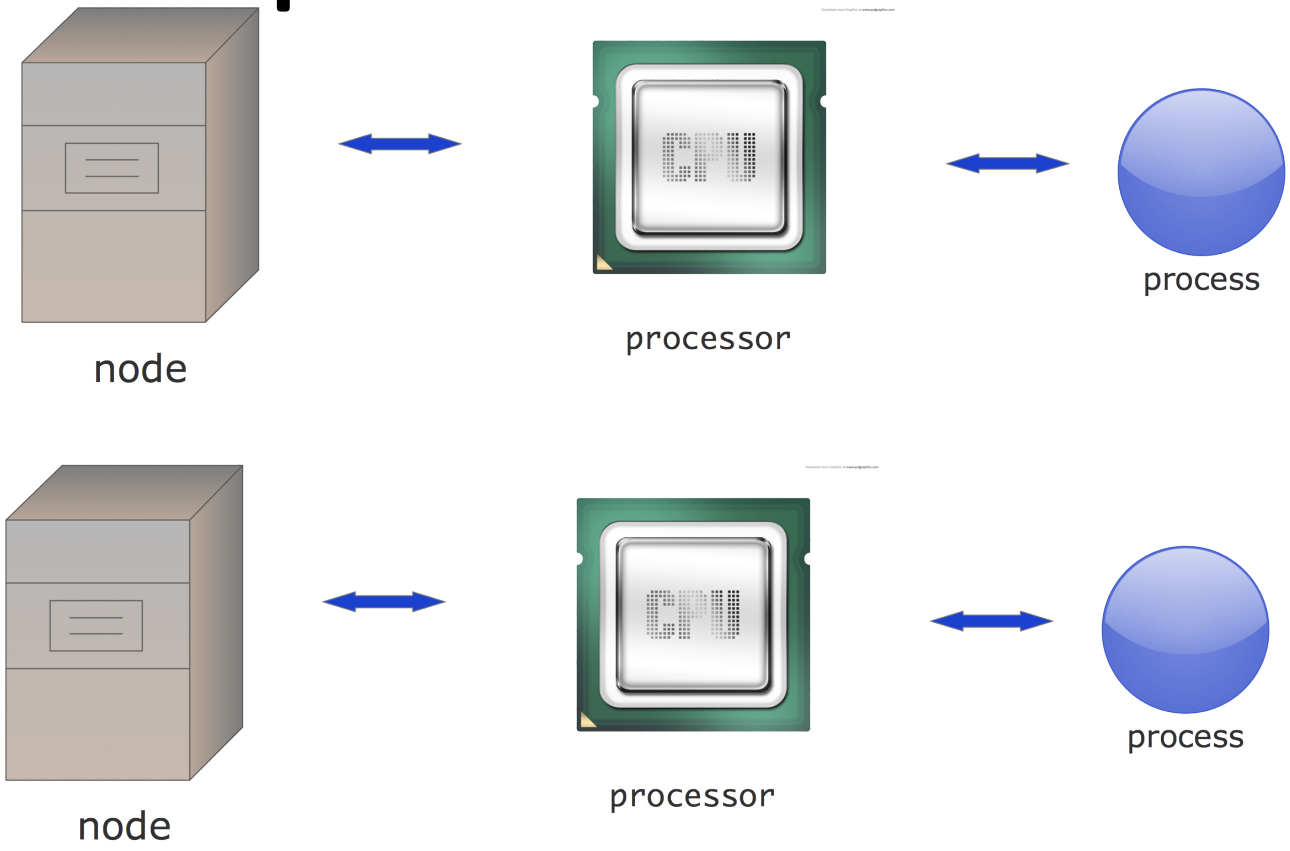
# SPMD

The basic model of MPI is
'Single Program Multiple Data':
each process is an instance of the same program.

Symmetry: There is no 'master process', all processes are equal, start and end at the same time.

Communication calls do not see the cluster structure:
data sending/receiving is the same for all neighbors.

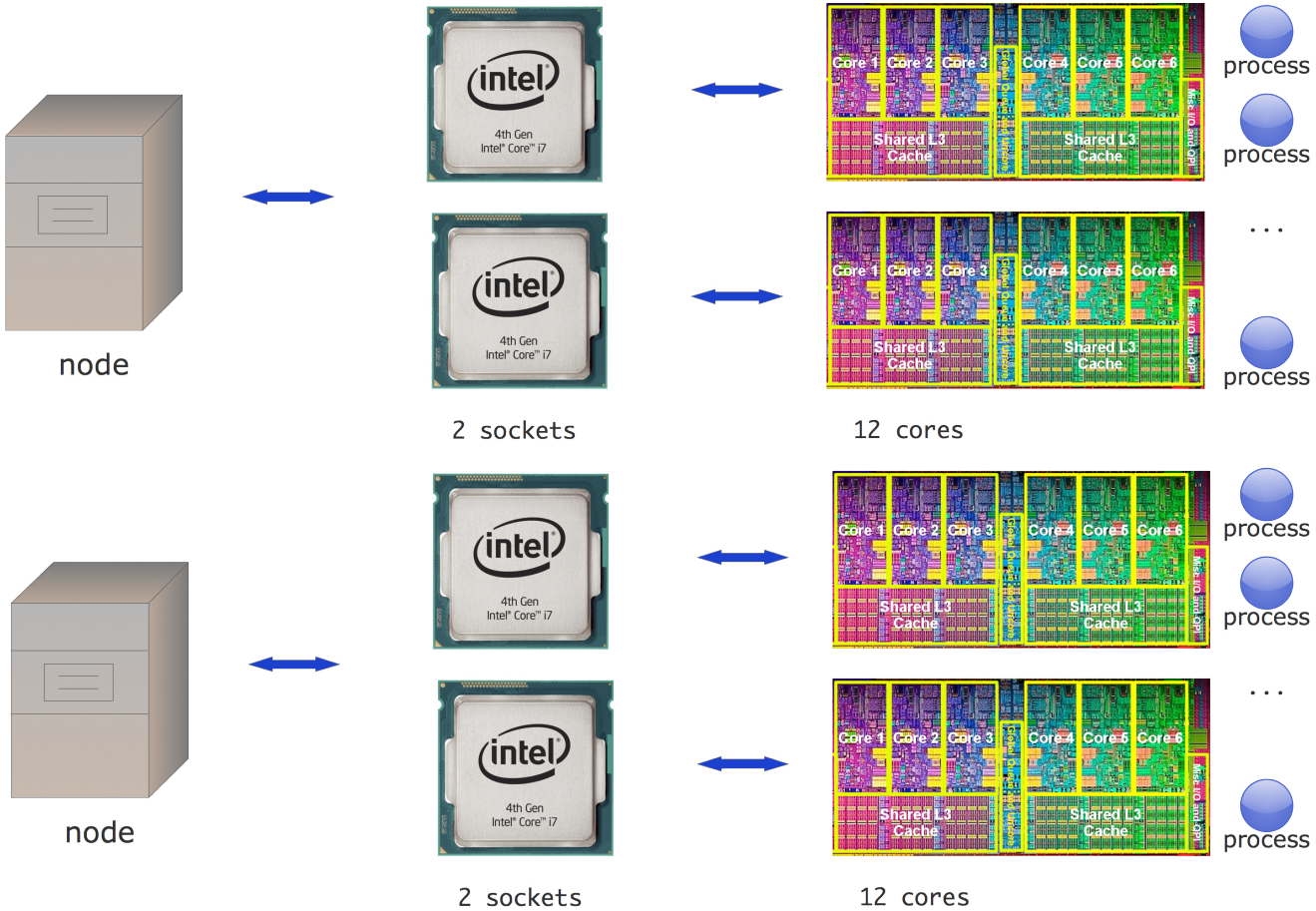
# Computers when MPI was designed




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

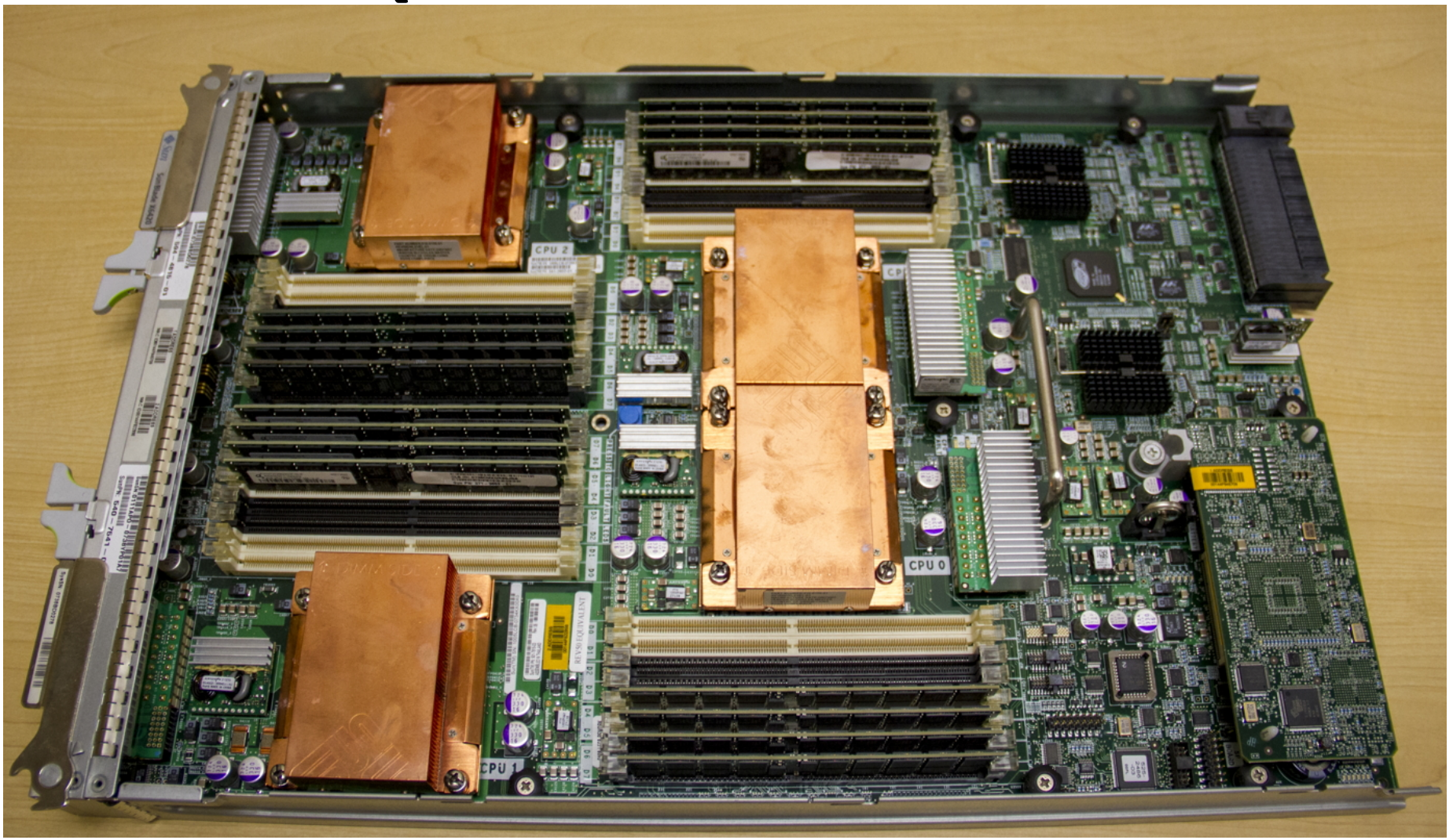
$\Rightarrow$ process model, no data sharing.

# Pure MPI

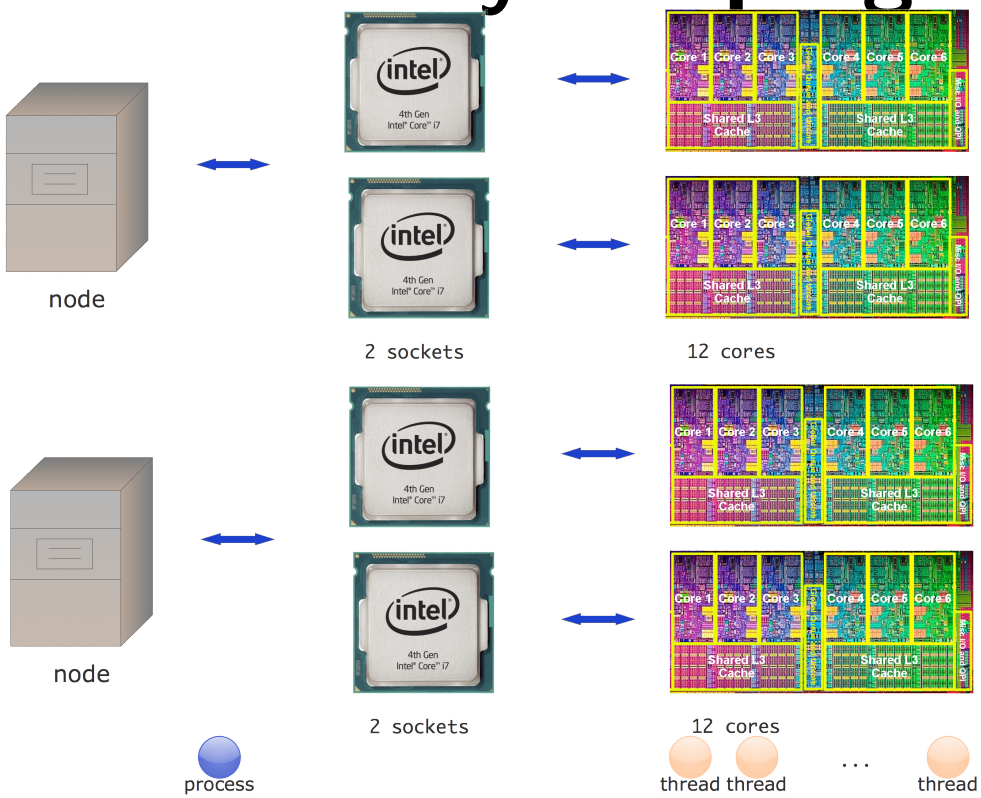



A node has multiple sockets, each with multiple cores.
Pure MPI puts a process on each core: pretend shared memory doesn't exist.

# Quad socket node



# Hybrid programming




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

Not in this course. . .

# Terminology

'Processor' is ambiguous: is that a chip or one independent instruction processing unit?

- Socket: the processor chip
- Processor: we don't use that word
- Core: one instruction-stream processing unit
- Process: preferred terminology in talking about MPI.

# Do I need a supercomputer?

- With `mpiexec` and such, you start a bunch of processes that execute your MPI program.
- Does that mean that you need a cluster or a big multicore?
- No! You can start a large number of MPI processes, even on your laptop. The OS will use 'time slicing'.
- Of course it will not be very efficient...

# Installing your own MPI

It is convenient to do MPI development on your laptop/desktop.

- Use a package manager
  Apple: brew or macports
  Linux: yum, aptget, ...
  Windows: I'll have to get back to you on that
- ... or download and compile from source `mpich.org`

# We start learning MPI!

# About library calls and bindings

# Bindings

The standard defines interfaces to MPI from C and Fortran.
These look very similar; sometimes we will only show the C variant.

MPI can also be used from C++ and Python

# MPI headers: C

You need an include file:

```
1 #include "mpi.h"
```

This defines all routines and constants.

# MPI Init / Finalize

These calls need to be around the MPI part of your code:

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

This is not a 'parallel region'.
Only internal library initialization:
allocate buffers, discover network, ...

# Exercise 3 (`hello`)

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.

# Ranks

# Process identification

- Processes are organized in 'communicators'.
- For now only the 'world' communicator
- Each process has a unique 'rank' wrt the communicator.

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

Lowest number is always zero.

This is a logical view of parallelism: mapping to physical processors/cores is invisible here.

# World communicator

For now, the communicator will be `MPI_COMM_WORLD`.

C:

```c
MPI_Comm comm = MPI_COMM_WORLD;
```

Fortran:

```fortran
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
! legacy
Integer :: comm = MPI_COMM_WORLD
```

Python:

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
```

MPL:

```
    const mpl::communicator &comm_world =
        mpl::environment::comm_world();
```

# MPI_Comm_size

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Comm_size ( | | | | |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | size | number of processes in the group of comm | int* | INTEGER |
| | ) | | | |

# MPI_Comm_rank

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Comm_rank ( | | | | |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | rank | rank of the calling process in group of comm | int* | INTEGER |
| | ) | | | |

# About routine signatures: C/C++

Signature:

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

Use:

```
1 MPI_Comm comm = MPI_COMM_WORLD;
2 int nprocs;
3 int errorcode;
4 errorcode = MPI_Comm_size( comm,&nprocs );
```

But forget about that error code most of the time:

```
MPI_Comm_size( comm,&nprocs );
```

# Exercise 4 (`commrank`)

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

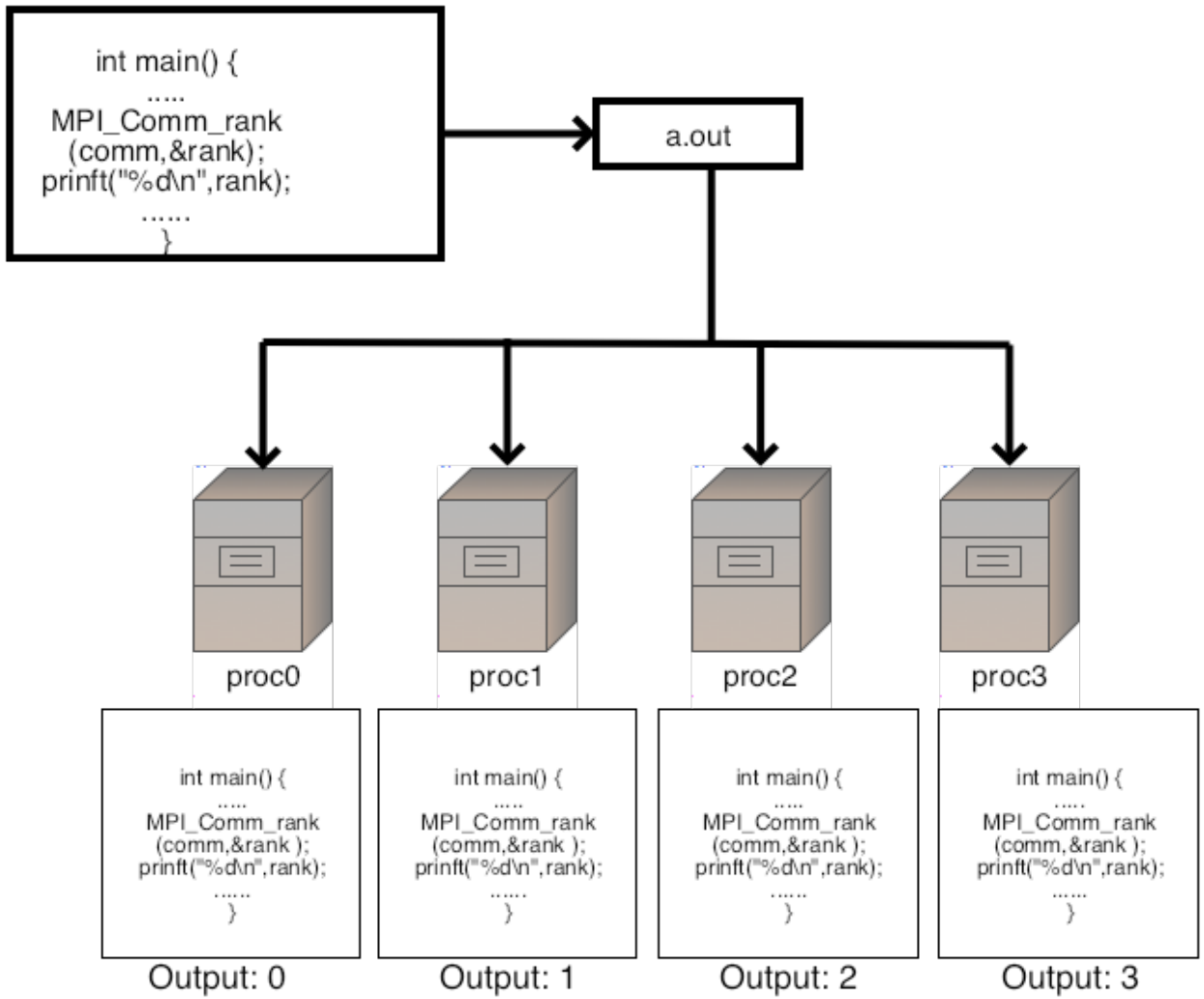
```
Hello from process 2 out of 5!
```

Write a second version of this program, where each process opens a unique file and writes to it.

# Exercise 5 (`commrank`)

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

# Illustration



# About errors

MPI routines invoke an error handler; (slide **??**) Default action: abort

Every routine is defined as returning integer error code

- In C: function result.

```
1 ierr = MPI_Init(0,0);
2 if (ierr!=MPI_SUCCESS) /* do something */
```

  But really: can often be ignored; is ignored in this course.

```
1 MPI_Init(0,0);
```

- In Fortran: as optional (F08 only) parameter.
- MPL: throws exceptions
- In Python: throwing exception.

There's not a lot you can do with an error code:
very hard to recover from errors in parallel.
~~By default code bombs with (hopefully informative) message.~~

**Processor name**

# MPI_Get_processor_name

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Get_processor_name ( | | | | |
| | name | A unique specifier for the actual (as opposed to virtual) node. | char* | CHARACTER |
| | resultlen | Length (in printable characters) of the result returned in name | int* | INTEGER |
| | ) | | | |

# Exercise 6

Use the command `MPI_Get_processor_name`. 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!

Go to `examples/mpi/xxx` and do `make procname`, then `ibrun procname`

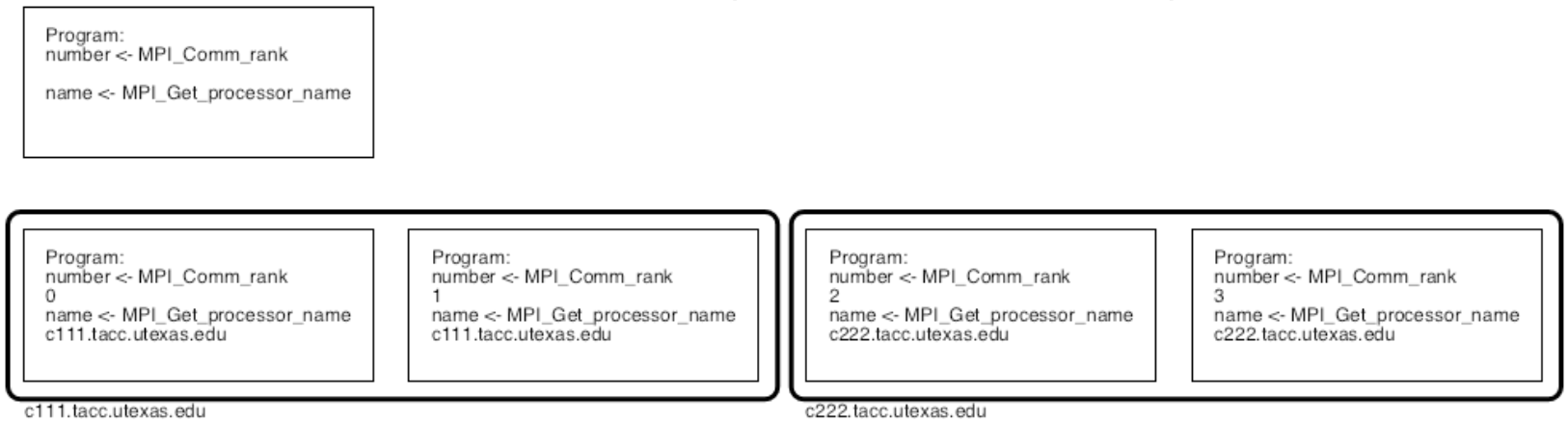
# Processor name

Processes (can) run on physically distinct locations.

```
1 // procname.c
2 int name_length = MPI_MAX_PROCESSOR_NAME;
3 char proc_name[name_length];
4 MPI_Get_processor_name(proc_name,&name_length);
5 printf("Process %d/%d is running on node <<%s>>\n",
6          procid,nprocs,proc_name);
```

# In a picture

Four processes on two nodes (`idev -N 2 -n 4`)



# Your first useful parallel program

# Functional Parallelism

Parallelism by letting each process do a different thing.

Example: divide up a search space.

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

# Exercise 7 (`prime`)

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

(Hint: `i%0` probably gives a runtime error.)

Can you find more than one solution?

# Exercise 8

Allocate on each process an array:

```
1 int my_ints[10];
```

and fill it so that process 0 has the integers $0 \cdots 9$, process 1 has $10 \cdots 19$, et cetera.

Let each process print out its array. It may be hard to print the output in a non-messy way.

# Collectives

# Overview

In this section you will learn 'collective' operations, that combine information from all processes.

Commands learned:

- `MPI_Bcast`, `MPI_Reduce`, `MPI_Gather`, `MPI_Scatter`
- *`MPI_All_...`* variants, *`MPI_....v`* variants
- `MPI_Barrier`, `MPI_Alltoall`, `MPI_Scan`

# Technically

Routines can be 'collective on a communicator':

- They involve a communicator;
- if one process calls that routine, every process in that communicator needs to call it
- Mostly about combining data, but also opening shared files, declaring 'windows' for one-sided communication.

# Collectives

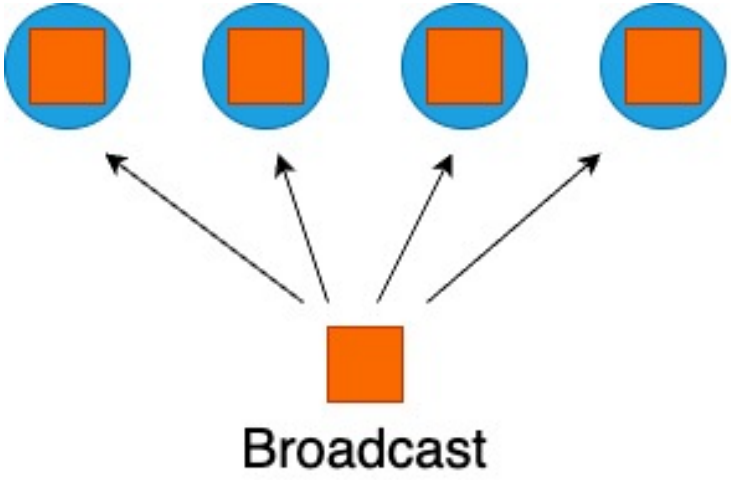
Gathering and spreading information:

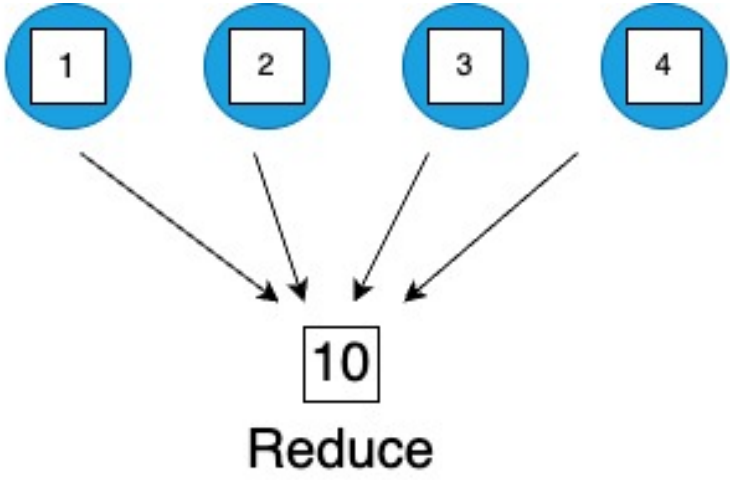
- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

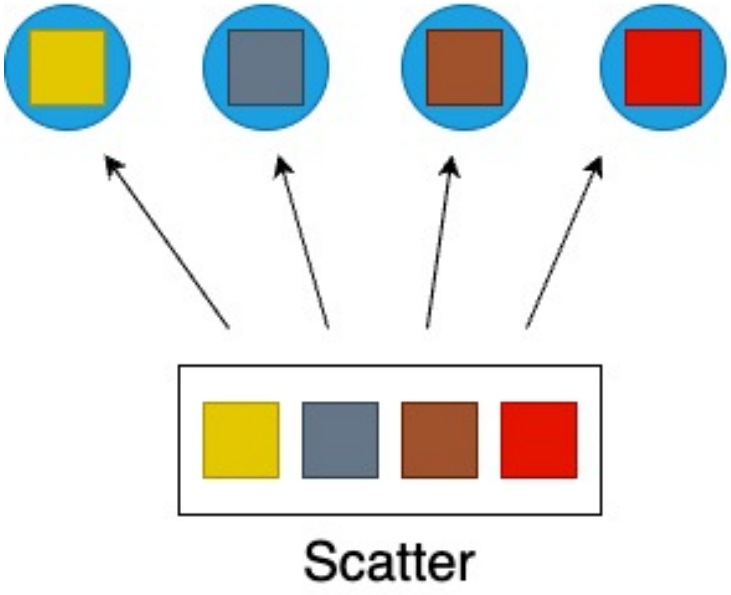
Root process: the one doing the collecting or disseminating.

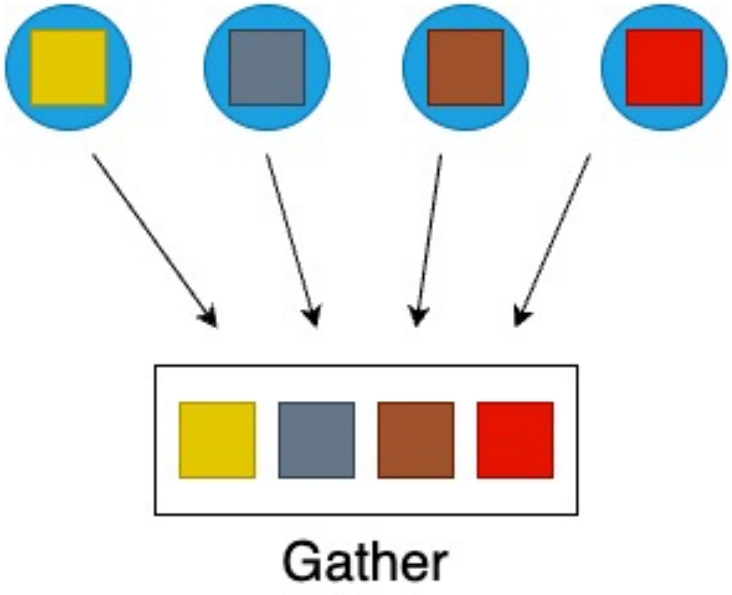
Basic cases:

- 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


1
2
3
4
10
Reduce


Scatter


Gather

# Exercise 9

How would you realize the following scenarios with MPI collectives?

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

Think about time and space complexity of your suggestions.

# Allreduce: reduce-to-all

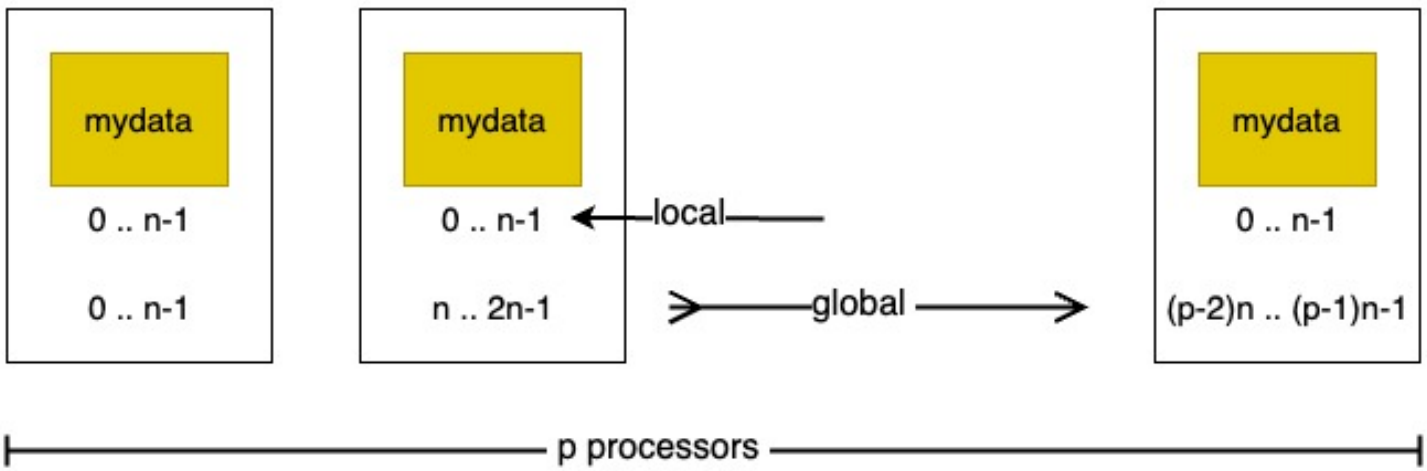
Exercise 2 above contains a common case:
do a reduction, but everyone needs the result.

- `MPI_Allreduce` does the same as:
  `MPI_Reduce` (reduction) followed by `MPI_Bcast` (broadcast)
- Same running time as either, half of
  reduce-followed-by-broadcast
  (no proof given here)
- Common use case, symmetrical expression.

# Motivation for allreduce

Example: normalizing a vector

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



# Structure of allreduce

- Vectors $x$,$y$ are distributed: every process has certain elements
- The norm calculation is an all-reduce: every process gets same value
- Every process scales its part of the vector.
- Question: what kind of reduction do you use for an inf-norm? One-norm? Two-norm?

# Another Allreduce

Standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i}^{N}(x_i - \mu)} \qquad \text{where} \qquad \mu = \frac{\sum_i^N x_i}{N}$$

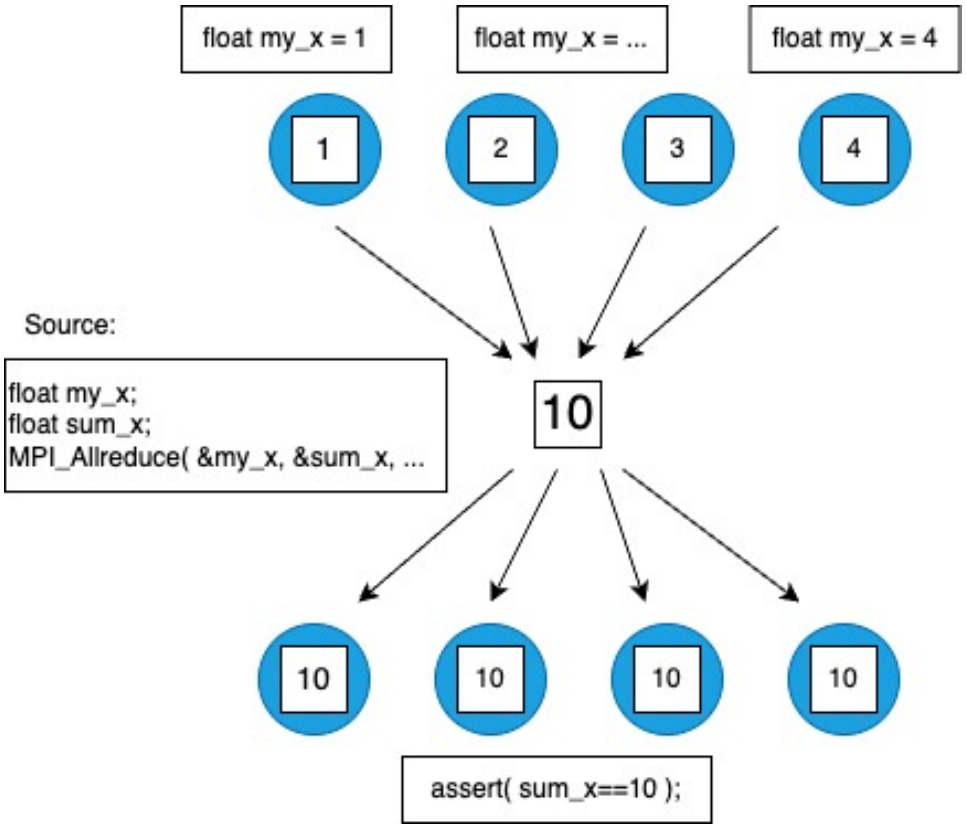
and assume that every process stores just one $x_i$ value.

How do we compute this?

1. The calculation of the average $\mu$ is a reduction.
2. Every process needs to compute $x_i - \mu$ for its value $x_i$, so use allreduce operation, which does the reduction and leaves the result on all processes.
3. $\sum_i (x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Might as well use allreduce.

# Conceptual picture

Recall SPMD: every process has the input and output variable




(What actually happens is a different story!)

# Allreduce syntax

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

- All processes have send and recv buffer
- (No root argument)
- *count* is number of items in the buffer: 1 for scalar.
  $> 1$: pointwise application of the reduction operator
- `MPI_Datatype` is `MPI_INT`, `MPI_FLOAT`, `MPI_REAL8` et cetera.
- `MPI_Op` is `MPI_SUM`, `MPI_MAX` et cetera.

# MPI_Allreduce

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Allreduce ( | | | | |
| MPI_Allreduce_c ( | | | | |
| | sendbuf | starting address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | recvbuf | starting address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in send buffer | [ int<br>MPI_Count | INTEGER |
| | datatype | datatype of elements of send buffer | MPI_Datatype | TYPE(MPI_Datatype) |
| | op | operation | MPI_Op | TYPE(MPI_Op) |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Exercise 10 (`randommax`)

Let each process compute a random number, and compute the sum of these numbers using the `MPI_Allreduce` routine.

$$\xi = \sum_i x_i$$

Each process then scales its value by this sum.

$$x_i' \leftarrow x_i / \xi$$

Compute the sum of the scaled numbers

$$\xi' = \sum_i x_i'$$

and check that it is 1.

# Buffers

# 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

```
1 double x;
2 MPI_Bcast( &x, .... );
3 double x[5];
4 MPI_Bcast( x, .... );
```

# Large buffers

As of MPI-4 a buffer can be longer than $2^{31}$ elements.

- Use `MPI_Count` for count
- In C: use `MPI_Reduce_c`
- in Fortran: polymorphism means no change to the call.
- MPL: *long int* and *size_t* supported for layouts.

```
1 MPI_Count buffersize = 1000;
2 double *indata,*outdata;
3 indata = (double*) malloc( buffersize*sizeof(double) );
4 outdata = (double*) malloc( buffersize*sizeof(double) );
5 MPI_Allreduce_c(indata,outdata,buffersize,
6                 MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
```

# Exercise 11

Extend exercise 10 to letting each process have an array.

# Collective basics

# Elementary datatypes

| C | Fortran | Python | meaning |
|---|---|---|---|
| MPI_CHAR | MPI_CHARACTER | | only for text |
| MPI_SHORT | MPI_BYTE | | 8 bits |
| MPI_INT | MPI_INTEGER | | like the C/F types |
| MPI_FLOAT | MPI_REAL | | |
| MPI_DOUBLE | MPI_DOUBLE_PRECISION | *MPI.DOUBLE* | |
| | MPI_COMPLEX | | |
| | MPI_LOGICAL | | |
| unsigned | extensions | | |
| | | | MPI_Aint |
| | | | MPI_Offset |

A bunch more.

# Reduction operators

| MPI type | | meaning | applies to |
|---|---|---|---|
| | MPI.Op | | |
| MPI_MAX | *MPI.MAX* | maximum | integer, floating point |
| MPI_MIN | *MPI.MIN* | minimum | |
| MPI_SUM | *MPI.SUM* | sum | integer, floating point, complex, r |
| MPI_PROD | *MPI.PROC* | product | |
| MPI_REPLACE | *MPI.REPLACE* | overwrite | |
| MPI_NO_OP | *MPI.OP_NULL* | no change | |
| MPI_LAND | *MPI.LAND* | logical and | C integer, logical |
| MPI_LOR | *MPI.LOR* | logical or | |
| MPI_LXOR | *MPI.LXOR* | logical xor | |
| MPI_BAND | *MPI.BAND* | bitwise and | integer, byte, multilanguage types |
| MPI_BOR | *MPI.BOR* | bitwise or | |
| MPI_BXOR | *MPI.BXOR* | bitwise xor | |
| MPI_MAXLOC | *MPI.MAXLOC* | max value and location | MPI_DOUBLE_INT and such |
| MPI_MINLOC | *MPI.MINLOC* | min value and location | |

# Reduction to single process

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

# Reduction to root

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

- Buffers: *sendbuf*, *recvbuf* are ordinary variables/arrays.
- Every process has data in its *sendbuf*,
  Root combines it in *recvbuf* (ignored on non-root processes).
- *count* is number of items in the buffer: 1 for scalar.
- MPI_Op is MPI_SUM, MPI_MAX et cetera.

# In-place operations

```
1 // allreduceinplace.c
2 for (int irand=0; irand<nrandoms; irand++)
3         myrandoms[irand] = (float) rand()/(float)RAND_MAX;
4 // add all the random variables together
5 MPI_Allreduce(MPI_IN_PLACE,myrandoms,
6                     nrandoms,MPI_FLOAT,MPI_SUM,comm);
```

# More in-place operations

```
1 if (procno==root)
2   MPI_Reduce(MPI_IN_PLACE,myrandoms,
3                     nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
4 else
5   MPI_Reduce(myrandoms,MPI_IN_PLACE,
6                     nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

or

```
1 float *sendbuf,*recvbuf;
2 if (procno==root) {
3   sendbuf = MPI_IN_PLACE; recvbuf = myrandoms;
4 } else {
5   sendbuf = myrandoms; recvbuf = MPI_IN_PLACE;
6 }
7 MPI_Reduce(sendbuf,recvbuf,
8           nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

# Broadcast

```
1 int MPI_Bcast(
2     void *buffer, int count, MPI_Datatype datatype,
3     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`

# Gauss-Jordan elimination

https://youtu.be/aQYuwatlWME

# MPI_Bcast

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Bcast ( | | | | |
| MPI_Bcast_c ( | | | | |
| | buffer | starting address of buffer | void* | TYPE(*), DIMENSION(..) |
| | count | number of entries in buffer | int<br>MPI_Count | INTEGER |
| | datatype | datatype of buffer | MPI_Datatype | TYPE(MPI_Datatype) |
| | root | rank of broadcast root | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Exercise 12 (`jordan`)

The *Gauss-Jordan algorithm* for solving a linear system with a matrix $A$ (or computing its inverse) runs as follows:

for pivot $k = 1, \ldots, n$
    let the vector of scalings $\ell_i^{(k)} = A_{ik}/A_{kk}$
    for row $r \neq k$
        for column $c = 1, \ldots, n$
            $A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{kc}$

where we ignore the update of the righthand side, or the formation of the inverse.

Let a matrix be distributed with each process storing one column.
Implement the Gauss-Jordan algorithm as a series of broadcasts: in iteration $k$ process $k$ computes and broadcasts the scaling vector $\{\ell_i^{(k)}\}_i$.
Replicate the right-hand side on all processors.

# Exercise (optional) 13

Bonus exercise: can you extend your program to have multiple columns per process?

# Scan

# Scan

Scan or 'parallel prefix': reduction with partial results

- Useful for indexing operations:
- Each process has an array of $n_p$ elements;
- My first element has global number $\sum_{q<p} n_q$.
- Two variants: `MPI_Scan` inclusive, and `MPI_Exscan` exclusive.

# In vs Exclusive

| | | | | | |
|---|---|---|---|---|---|
| process : | 0 | 1 | 2 | $\cdots$ | $p-1$ |
| data : | $x_0$ | $x_1$ | $x_2$ | $\cdots$ | $x_{p-1}$ |
| inclusive : | $x_0$ | $x_0 \oplus x_1$ | $x_0 \oplus x_1 \oplus x_2$ | $\cdots$ | $\oplus_{i=0}^{p-1} x_i$ |
| exclusive : | unchanged | $x_0$ | $x_0 \oplus x_1$ | $\cdots$ | $\oplus_{i=0}^{p-2} x_i$ |

# MPI_Scan

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Scan ( | | | | |
| MPI_Scan_c ( | | | | |
| | sendbuf | starting address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | recvbuf | starting address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in input buffer | [ int<br>MPI_Count | INTEGER |
| | datatype | datatype of elements of input buffer | MPI_Datatype | TYPE(MPI_Datatype) |
| | op | operation | MPI_Op | TYPE(MPI_Op) |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# MPI_Exscan

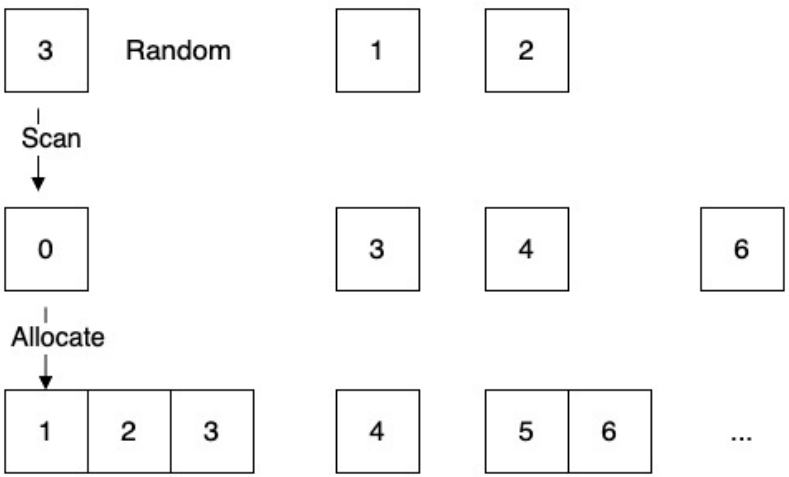
| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Exscan ( | | | | |
| MPI_Exscan_c ( | | | | |
| | sendbuf | starting address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | recvbuf | starting address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in input buffer | [ int<br>MPI_Count | INTEGER |
| | datatype | datatype of elements of input buffer | MPI_Datatype | TYPE(MPI_Datatype) |
| | op | operation | MPI_Op | TYPE(MPI_Op) |
| | comm | intra-communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Exercise 14 (`scangather`)

- Let each process compute a random value $n_{\text{local}}$, and allocate an array of that length. Define

$$N = \sum n_{\text{local}}$$

- Fill the array with consecutive integers, so that all local arrays, laid end-to-end, contain the numbers $0 \cdots N-1$. (See figure 14.)



## Gather/Scatter, Barrier, and others

# MPI_Gather

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Gather ( | | | | |
| MPI_Gather_c ( | | | | |
| | sendbuf | starting address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | sendcount | number of elements in send buffer | [ int<br>MPI_Count | INTEGER |
| | sendtype | datatype of send buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | recvbuf | address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | recvcount | number of elements for any single receive | [ int<br>MPI_Count | INTEGER |
| | recvtype | datatype of recv buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | root | rank of receiving process | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

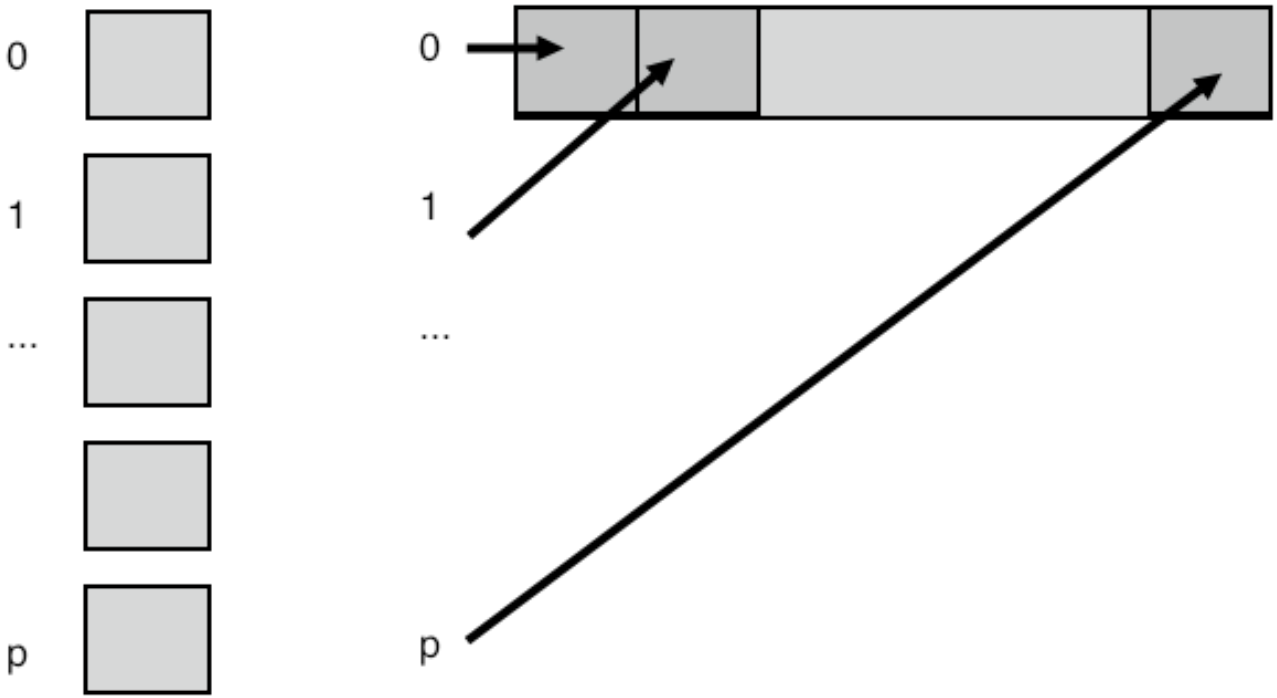
# MPI_Scatter

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Scatter ( | | | | |
| MPI_Scatter_c ( | | | | |
| | sendbuf | address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | sendcount | number of elements sent to each process | [ int<br>MPI_Count | INTEGER |
| | sendtype | datatype of send buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | recvbuf | address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | recvcount | number of elements in receive buffer | [ int<br>MPI_Count | INTEGER |
| | recvtype | datatype of receive buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | root | rank of sending process | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Gather/Scatter

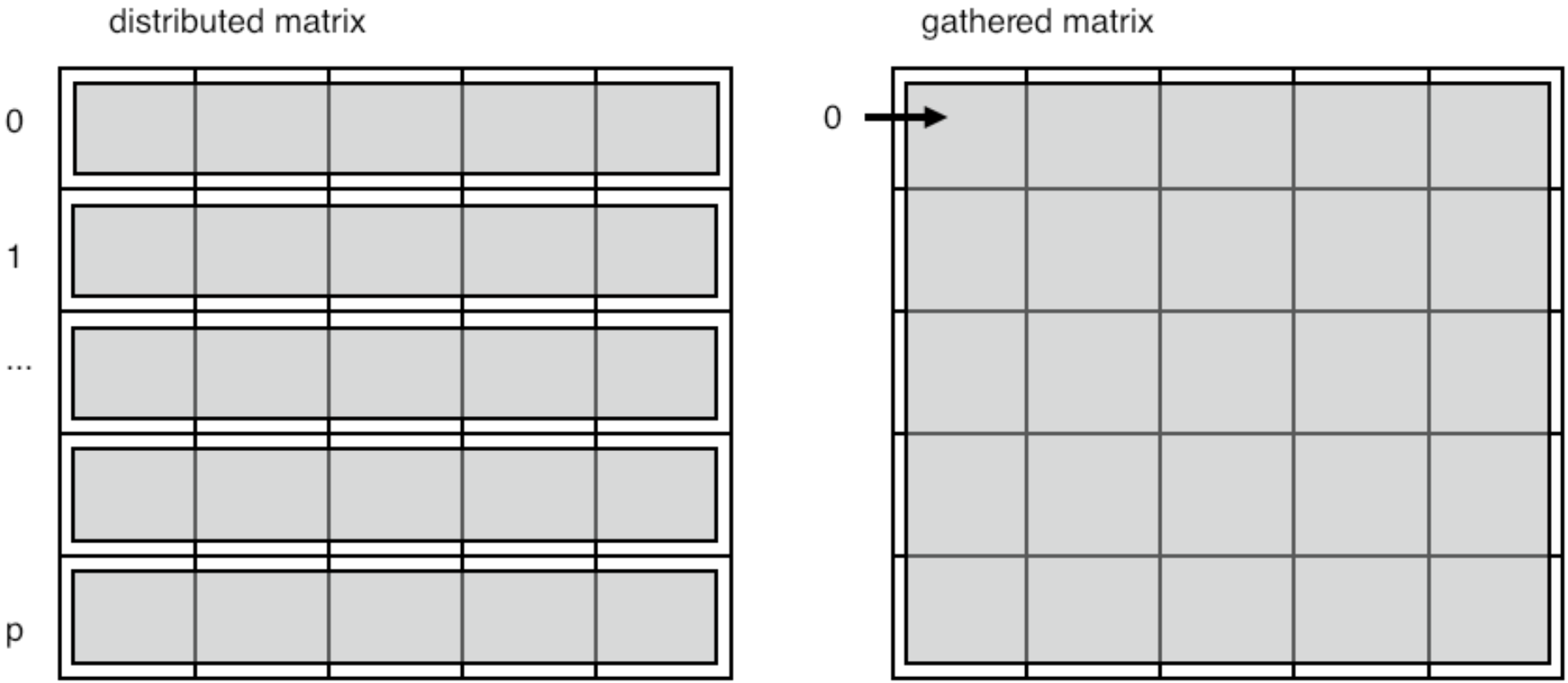
- Compare buffers to reduce
- Scatter: the `sendcount` / Gather: the `recvcount`:
  this is not, as you might expect, the total length of the buffer;
  instead, it is the amount of data to/from each process.

# Gather pictured



# Popular application of gather

Matrix is constructed distributed, but needs to be brought to one process:

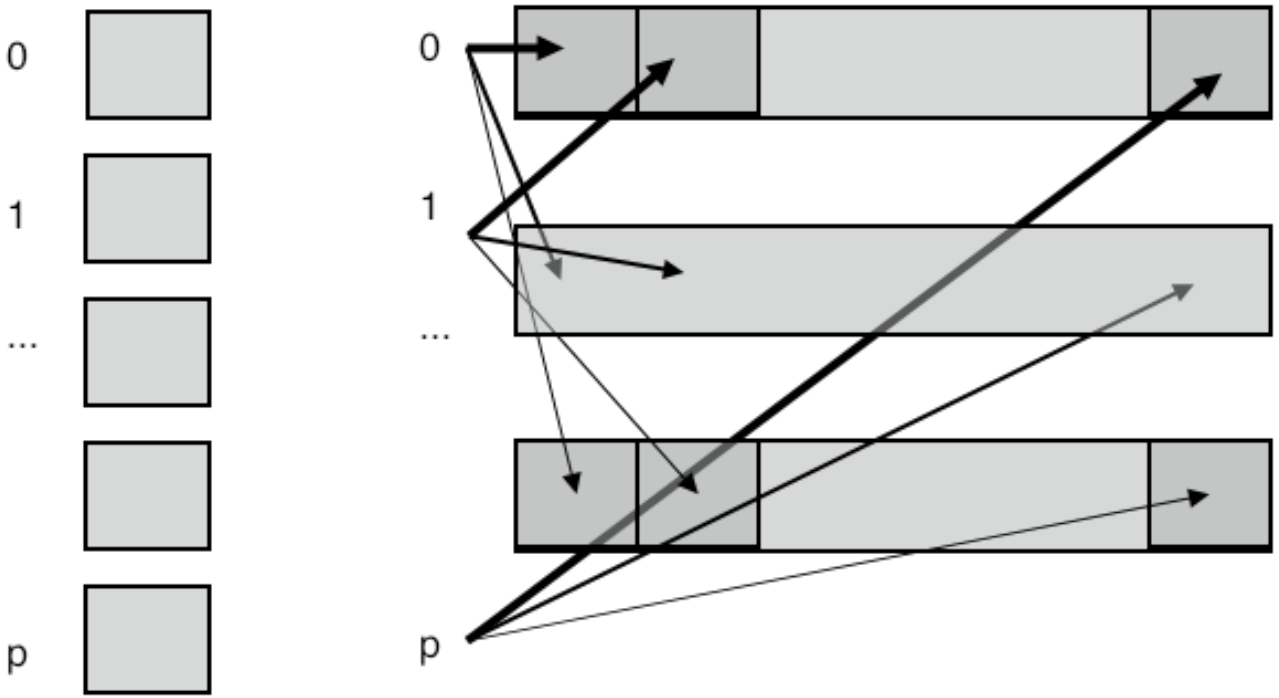



This is not efficient in time or space. Do this only when strictly necessary. Remember SPMD: try to keep everything symmetrically

# MPI_Allgather

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Allgather ( | | | | |
| MPI_Allgather_c ( | | | | |
| | sendbuf | starting address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | sendcount | number of elements in send buffer | int<br>MPI_Count | INTEGER |
| | sendtype | datatype of send buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | recvbuf | address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | recvcount | number of elements received from any process | int<br>MPI_Count | INTEGER |
| | recvtype | datatype of receive buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Allgather pictured



# V-type collectives

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

# MPI_Gatherv

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Gatherv ( | | | | |
| MPI_Gatherv_c ( | | | | |
| | sendbuf | starting address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | sendcount | number of elements in send buffer | [ int<br>MPI_Count | INTEGER |
| | sendtype | datatype of send buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | recvbuf | address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | recvcounts | non-negative integer array (of length group size) containing the number of elements that are received from each process | [ const int[]<br>MPI_Count[] | INTEGER(*) |
| | displs | integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i | [ const int[]<br>MPI_Aint[] | INTEGER(*) |
| | recvtype | datatype of recv buffer elements | MPI_Datatype | TYPE(MPI_Datatype) |
| | root | rank of receiving process | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Exercise 15 (`scangather`)

Take the code from exercise 14 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires `MPI_Gatherv`.

How do you construct the lengths and displacements arrays?

# Review 1

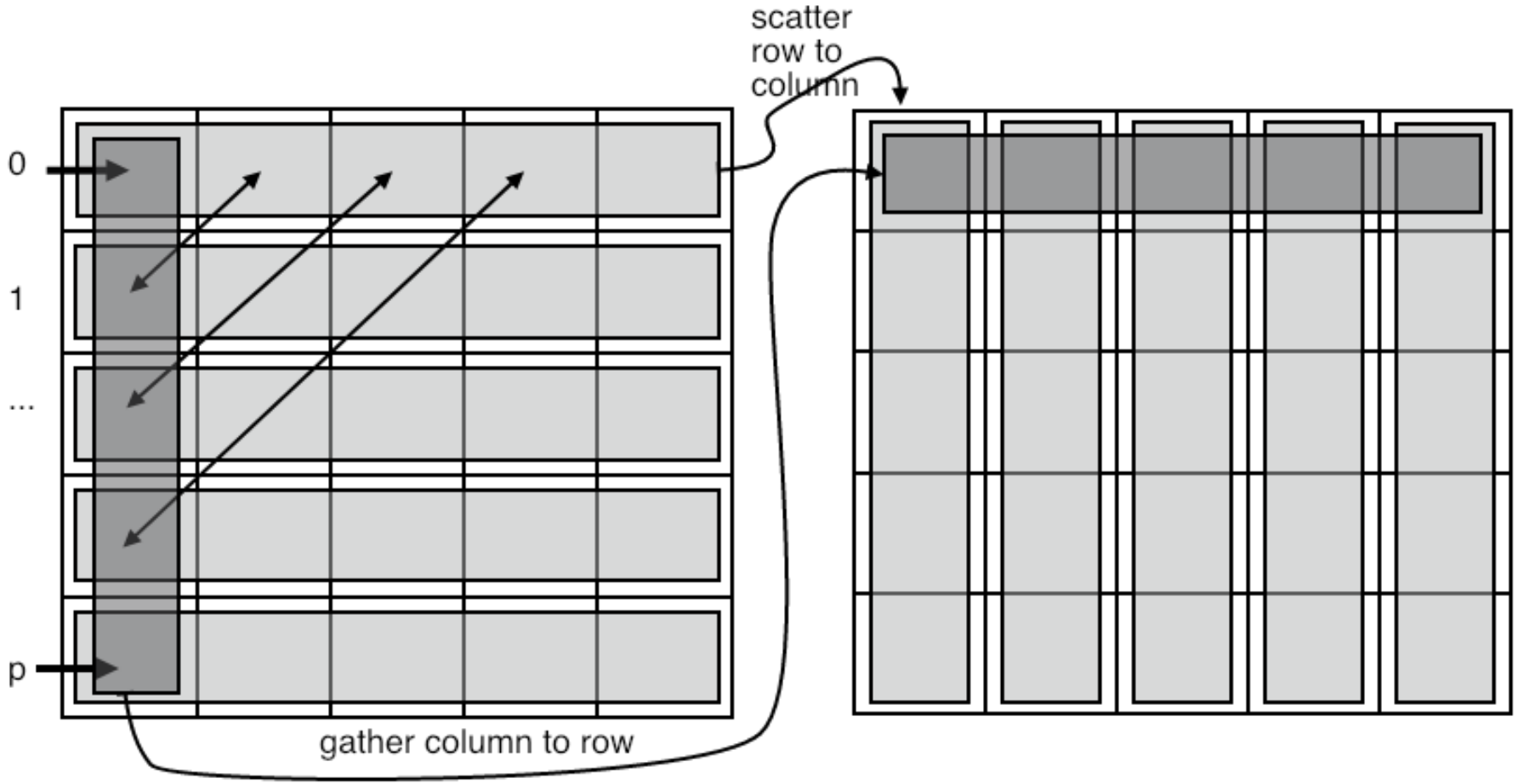
An `MPI_Scatter` call puts the same data on each process

```
/poll "A scatter call puts the same data on each process" "T" "F"
```

# All-to-all

- Every process does a scatter;
- (equivalently: every process gather)
- each individual data, but amounts are identical
- Example: data transposition in FFT

# Data transposition




Example: each process knows who to send to,
all-to-all gives information who to receive from

# All-to-allv

- Every process does a scatter or gather;
- each individual data and individual amounts.
- Example: radix sort by least-significant digit.

# Radix sort

Sort 4 numbers on two processes:

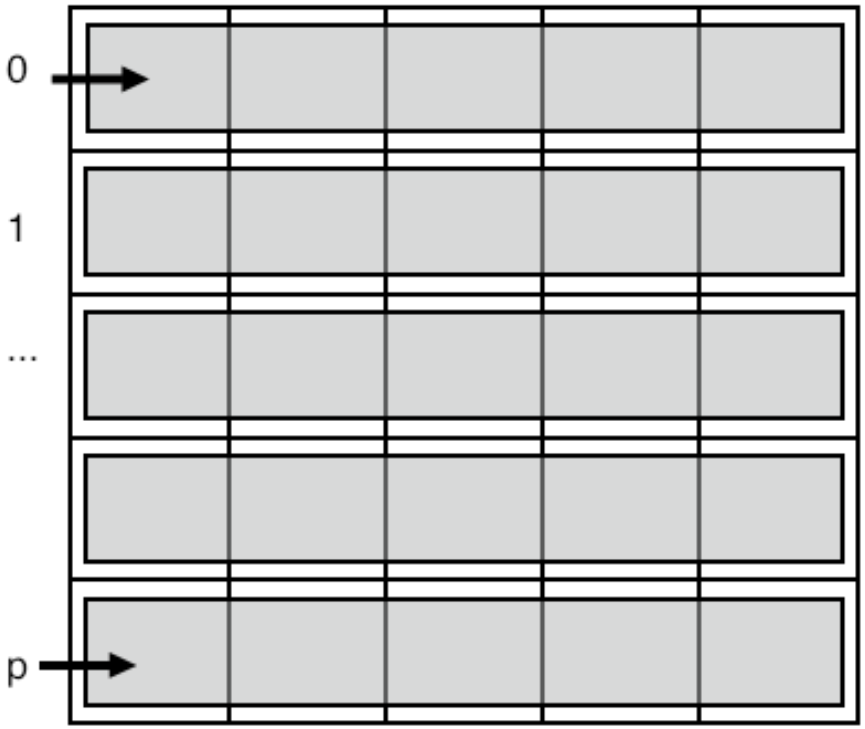
| | proc0 | | proc1 | | |
|---|---|---|---|---|---|
| array | 2 | 5 | 7 | 1 | |
| binary | 010 | 101 | 111 | 001 | |
| stage 1 | | | | | |
| last digit | 0 | 1 | 1 | 1 | |
| | (this serves as bin number) | | | | |
| sorted | 010 | | 101 | 111 | 001 |
| stage 2 | | | | | |
| next digit | 1 | | 0 | 1 | 0 |
| | (this serves as bin number) | | | | |
| sorted | 101 | 001 | 010 | 111 | |
| stage 3 | | | | | |
| next digit | 1 | 0 | 0 | 1 | |
| | (this serves as bin number) | | | | |
| sorted | 001 | 010 | 101 | 111 | |
| decimal | 1 | 2 | 5 | 7 | |

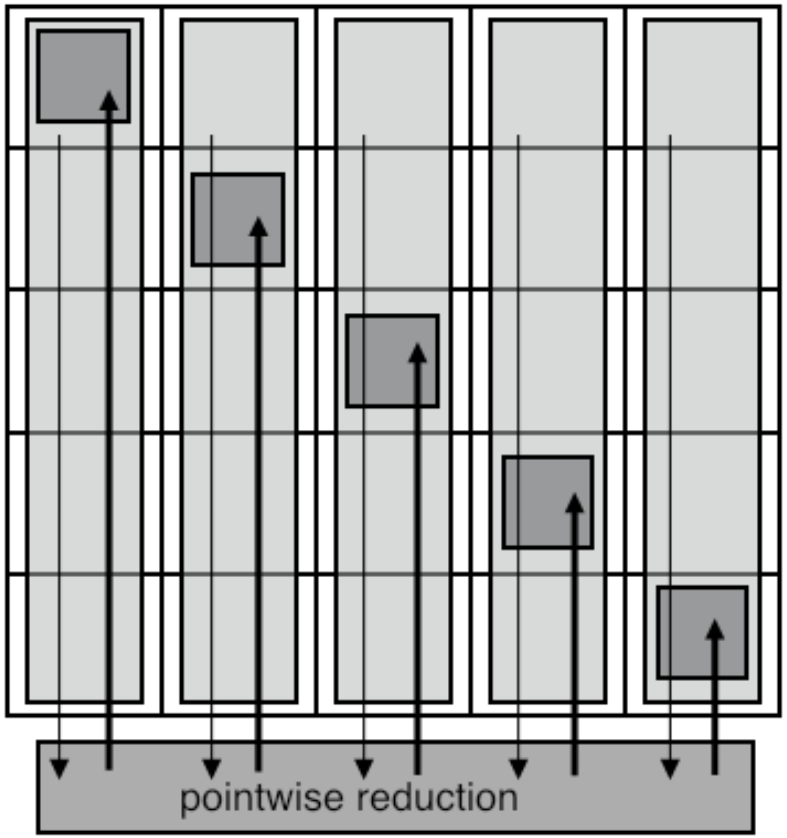
# Reduce-scatter

- Pointwise reduction (one element per process) followed by scatter
- Somewhat related to all-to-all: data transpose but reduced information, rather than gathered.
- Applications in both sparse and dense matrix-vector product.

# Example: sparse matrix setup

Example: each process knows who to send to,
all-to-all gives information how many messages to expect
reduce-scatter leaves only relevant information



# Barrier

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

- Synchronize processes:
- each process waits at the barrier until all processes have reached the barrier
- **This routine is almost never needed**:
  collectives are already a barrier of sorts, two-sided communication is a local synchronization
- One conceivable use: timing

# User-defined operators

# MPI Operators

Define your own reduction operator

- Define operator between partial result and new operand

```
1 typedef void MPI_User_function
2     ( void *invec, void *inoutvec, int *len,
3       MPI_Datatype *datatype);
```

- Don't forget to free:

```
1 int MPI_Op_free(MPI_Op *op)
```

- Make your own reduction scheme `MPI_Reduce_local`

# MPI_Op_create

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Op_create ( | | | | |
| MPI_Op_create_c ( | | | | |
| | user_fn | user defined function | MPI_User_function* / MPI_User_function_c* | PROCEDURE (MPI_User_function |
| | commute | true if commutative; false otherwise. | int | LOGICAL |
| | op | operation | MPI_Op* | TYPE(MPI_Op) |
| | ) | | | |

# Example

Smallest nonzero:

```
1 *(int*)inout = m;
2 }
```

# Review 2

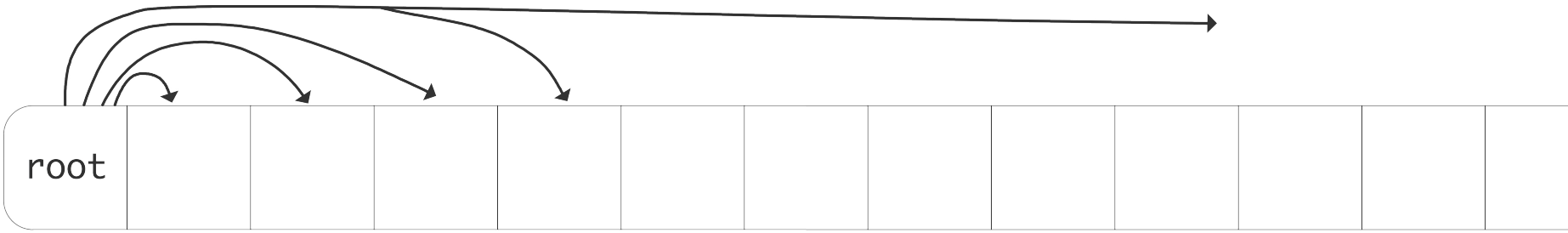
The $\|\cdot\|_2$ norm (sum of squares) needs a custom operator.

```
/poll "The sum of squares norm needs a custom operators" "T" "F"
```

# Performance of collectives

# Naive realization of collectives

Broadcast:




Single message:

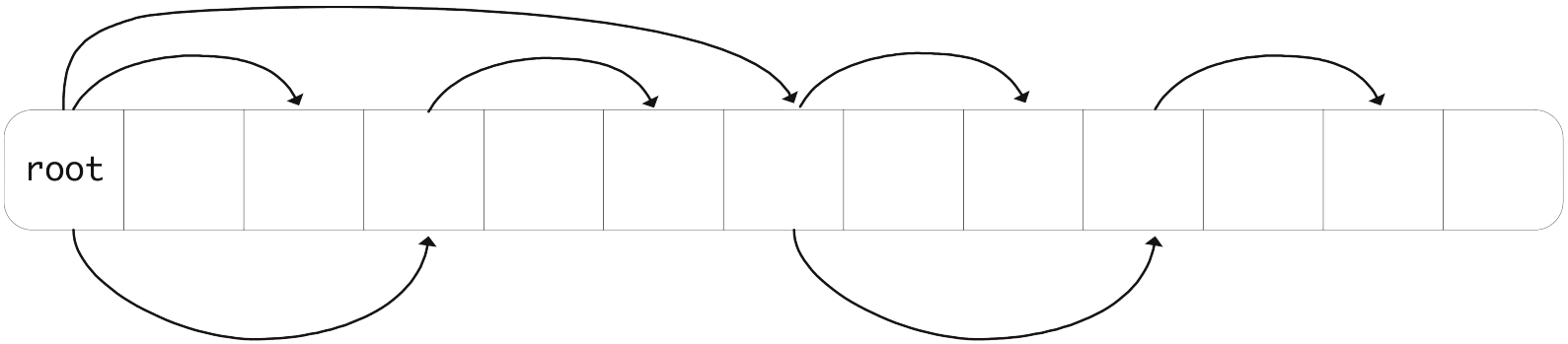
$\alpha =$ message startup $\approx 10^{-6}s$, $\beta =$ time per word $\approx 10^{-9}s$

- Time for message of $n$ words:

$$\alpha + \beta n$$

- Time for collective? Can you improve on that?

# Better implementation of collective




- What is the running time now?
- Can you come up with lower bounds on the $\alpha, \beta$ terms? Are these achieved here?
- How about the case of really long buffers?

# Implementation of Reduce

| | $t = 1$ | $t = 2$ | $t = 3$ |
|---|---|---|---|
| $p_0$ | $x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$ | $x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$ | $x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0}$ |
| $p_1$ | $x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$ | | |
| $p_2$ | $x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$ | $x_0^{(2:3)} \uparrow, x_1^{(2:3)} \uparrow, x_2^{(2:3)} \uparrow, x_3^{(2:3)} \uparrow$ | |
| $p_3$ | $x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$ | | |

# Implementation of Allreduce

| | $t = 1$ | $t = 2$ | $t = 3$ |
|---|---|---|---|
| $p_0$ | $x_0^{(0)} \downarrow, x_1^{(0)} \downarrow, x_2^{(0)} \downarrow, x_3^{(0)} \downarrow$ | $x_0^{(0:1)} \downarrow\downarrow, x_1^{(0:1)} \downarrow\downarrow, x_2^{(0:1)} \downarrow\downarrow, x_3^{(0:1)} \downarrow\downarrow$ | $x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3}$ |
| $p_1$ | $x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$ | $x_0^{(0:1)} \downarrow\downarrow, x_1^{(0:1)} \downarrow\downarrow, x_2^{(0:1)} \downarrow\downarrow, x_3^{(0:1)} \downarrow\downarrow$ | $x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3}$ |
| $p_2$ | $x_0^{(2)} \downarrow, x_1^{(2)} \downarrow, x_2^{(2)} \downarrow, x_3^{(2)} \downarrow$ | $x_0^{(2:3)} \uparrow\uparrow, x_1^{(2:3)} \uparrow\uparrow, x_2^{(2:3)} \uparrow\uparrow, x_3^{(2:3)} \uparrow\uparrow$ | $x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3}$ |
| $p_3$ | $x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$ | $x_0^{(2:3)} \uparrow\uparrow, x_1^{(2:3)} \uparrow\uparrow, x_2^{(2:3)} \uparrow\uparrow, x_3^{(2:3)} \uparrow\uparrow$ | $x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3}$ |

# Review 3

True of false: there are collectives that do not communicate data

```
/poll "there are collectives that do not communicate data" "T" "F"
```

## Reduction operators

# User-defined operators

Given a reduction function:

```
typedef void user_function
    ( void *invec, void *inoutvec, int *len,
      MPI_Datatype *datatype);
```

create a new operator:

```
MPI_Op rwz;
MPI_Op_create(user_function,1,&rwz);
MPI_Allreduce(data+procno,&positive_minimum,1,MPI_INT,rwz,comm);
```

# Exercise 16 (`onenorm`)

Write the reduction function to implement the *one-norm* of a vector:

$$\|x\|_1 \equiv \sum_i |x_i|.$$

# Point-to-point communication

# Overview

This section concerns direct communication between two processes. Discussion of distributed work, deadlock and other parallel phenomena.

Commands learned:

- `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Isend`, `MPI_Irecv`
- `MPI_Wait...`
- Mention of `MPI_Test`, `MPI_Bsend`/*Ssend*/*Rsend*.

# Point-to-point communication

# 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 in MPI

Remember SPMD:

```
1 if ( /* I am process A */ ) {
2   MPI_Send( /* to: */ B ..... );
3   MPI_Recv( /* from: */ B ... );
4 } else if ( /* I am process B */ ) {
5   MPI_Recv( /* from: */ A ... );
6   MPI_Send( /* to: */ A ..... );
7 }
```

# Sample send and recv calls

```
1 double x[10],y[10];
2 MPI_Send( x,10,MPI_DOUBLE, tgt,0,comm );
3 MPI_Status status;
4 MPI_Recv( y,10,MPI_DOUBLE, src,0,comm,&status );
```

# MPI_Send

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Send ( | | | | |
| MPI_Send_c ( | | | | |
| | buf | initial address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in send buffer | [ int<br>MPI_Count | INTEGER |
| | datatype | datatype of each send buffer element | MPI_Datatype | TYPE(MPI_Datatype) |
| | dest | rank of destination | int | INTEGER |
| | tag | message tag | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# MPI_Recv

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Recv ( | | | | |
| MPI_Recv_c ( | | | | |
| | buf | initial address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in receive buffer | int<br>MPI_Count | INTEGER |
| | datatype | datatype of each receive buffer element | MPI_Datatype | TYPE(MPI_Datatype) |
| | source | rank of source or MPI_ANY_SOURCE | int | INTEGER |
| | tag | message tag or MPI_ANY_TAG | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | status | status object | MPI_Status* | TYPE(MPI_Status) |
| | ) | | | |

# Status object

Use `MPI_STATUS_IGNORE` unless . . .

- Receive call can have various wildcards:
  `MPI_ANY_SOURCE`, `MPI_ANY_TAG`
- Receive buffer size is actually upper bound, not exact
- Use status object to retrieve actual description of the message

```
1 int s = status.MPI_SOURCE;
2 int t = status.MPI_TAG;
3 MPI_Get_count(status,MPI_FLOAT,&c);
```

# Exercise 17 (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 $\alpha, \beta$?

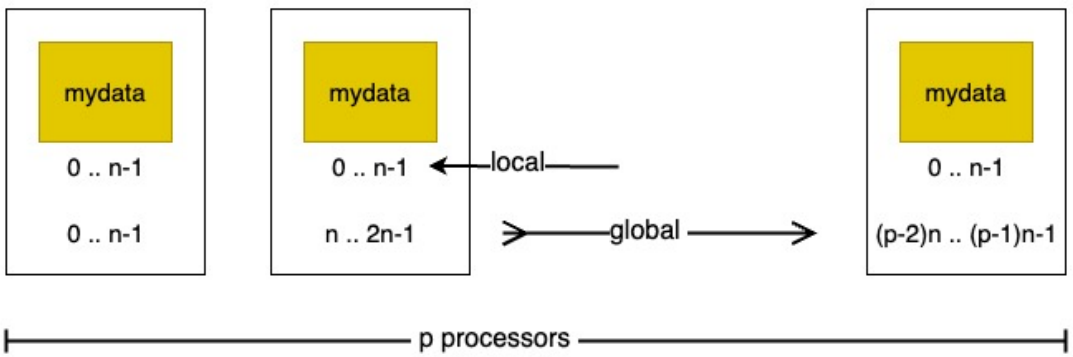
# MPI_Wtime

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| `MPI_Wtime (`<br>`)` | | | | |

# Distributed data

# Distributed data

Distributed array: each process stores disjoint local part




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:

```
1 int myfirst = .....;
2 for (int ilocal=0; ilocal<nlocal; ilocal++) {
3    int iglobal = myfirst+ilocal;
4    array[ilocal] = f(iglobal);
5 }
```

# Exercise (optional) 18

Implement a (very simple-minded) Fourier transform: if $f$ is a function on the interval $[0, 1]$, then the $n$-th Fourier coefficient is

$$f_n \hat{=} \int_0^1 f(t) e^{-2\pi x}\, dx$$

which we approximate by

$$f_n \hat{=} \sum_{i=0}^{N-1} f(ih) e^{-in\pi/N}$$

- Make one distributed array for the $e^{-inh}$ coefficients,
- make one distributed array for the $f(ih)$ values
- calculate a couple of coefficients

# Load balancing

If the distributed array is not perfectly divisible:

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

This gives a load balancing problem. Better solution?

# (for future reference)

Let

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

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

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

# Local information exchange

# Motivation

Partial differential equations:

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

Simple case:

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

Finite difference approximation:

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

Finite dimensional: $u_i \equiv u(ih)$.

# Motivation (continued)

Equations

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

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

So we are interested in sparse/banded matrices.

# Matrix vector product

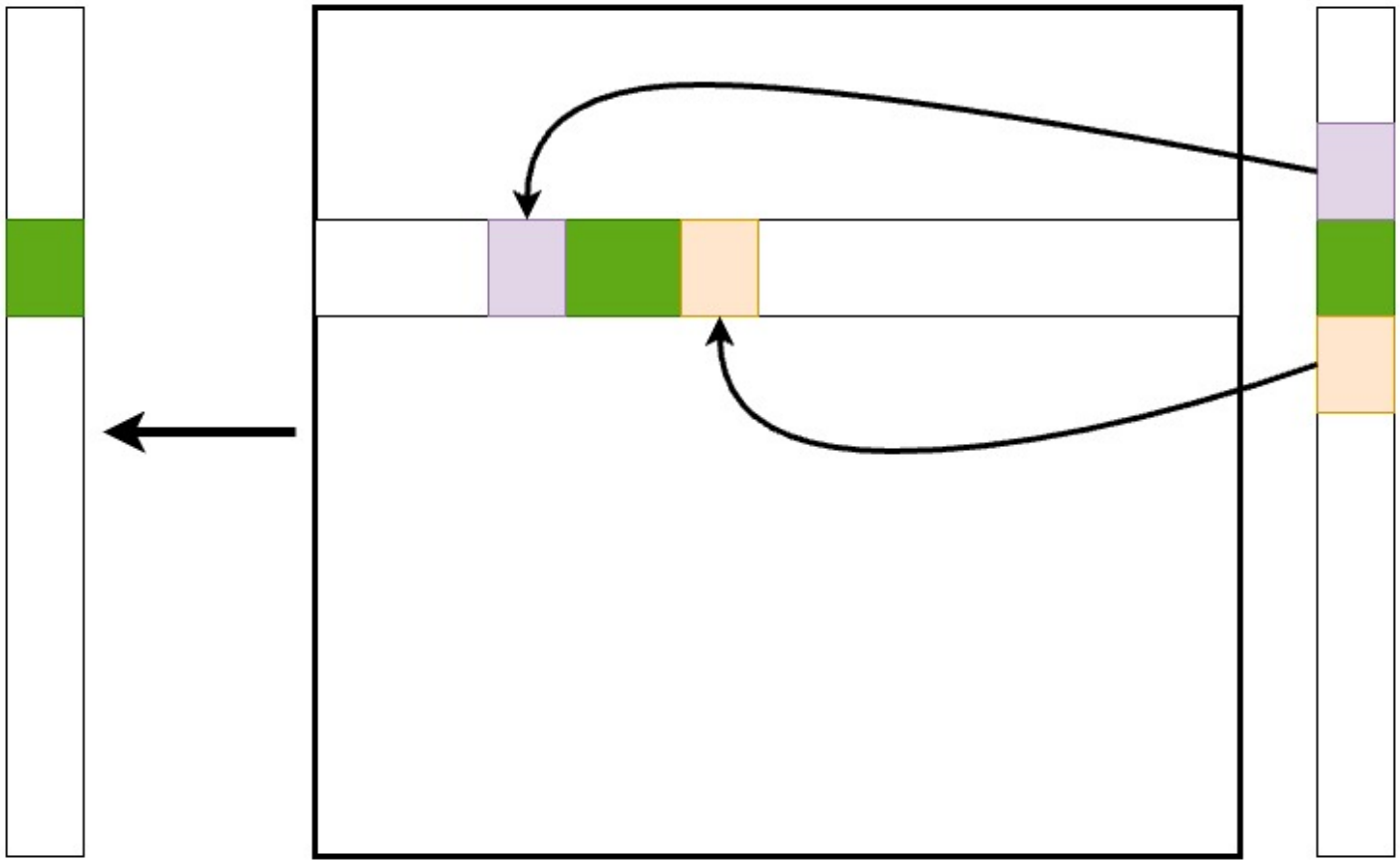
Most common operation: matrix vector product

$$y \leftarrow Ax, \qquad A = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix}$$

- Component operation: $y_i = 2x_i - x_{i-1} - x_{i+1}$
- Parallel execution: each process has range of $i$-coordinates
- $\Rightarrow$ segment of vector, block row of matrix

# Partitioned matrix-vector product

We need a point-to-point mechanism:



each process with ones before/after it.

# Operating on distributed data

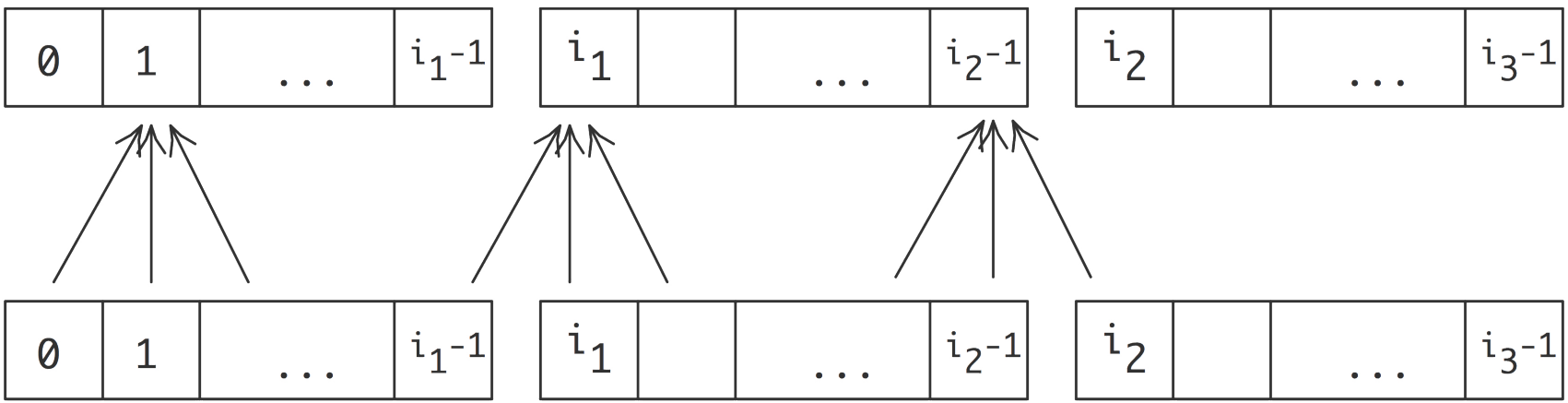
Array of numbers $x_i : i = 0, \ldots, N$

compute

$$y_i = -x_{i-1} + 2x_i - x_{i+1} : i = 1, \ldots, N-1$$

'owner computes'

This leads to communication:



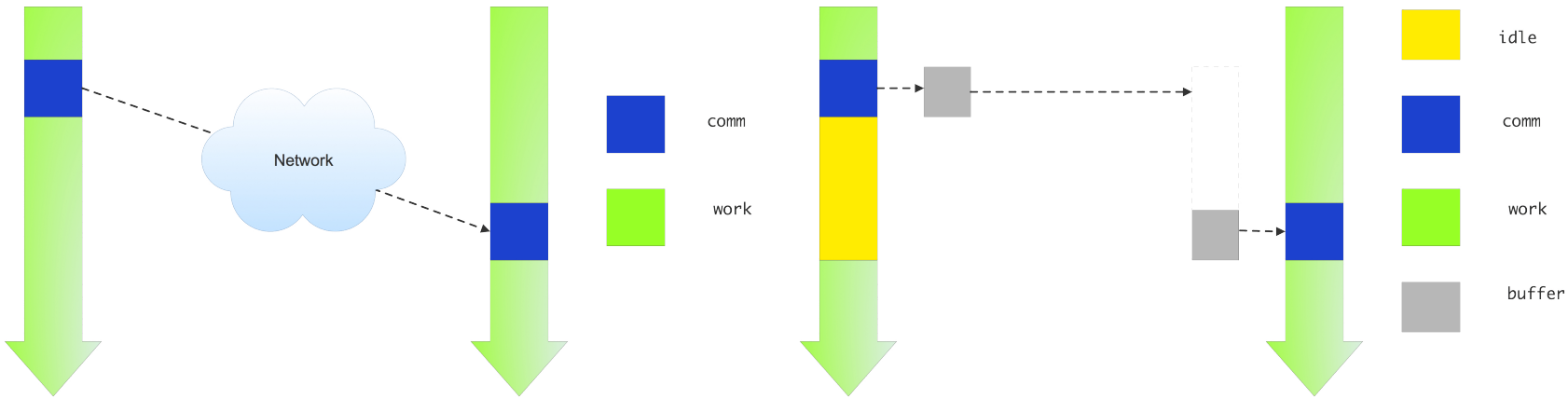

so we need a point-to-point mechanism.

# 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

Exchange between two processes:

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

A subtlety.

This code may actually work:

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

Small messages get sent even if there is no corresponding receive.
(Often a system parameter)

# 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.’

Small messages bypass this: ‘eager’ send.
Definition of ‘small message’ controlled by environment variables:

*I_MPI_EAGER_THRESHOLD MV2_IBA_EAGER_THRESHOLD*

# Exercise 19

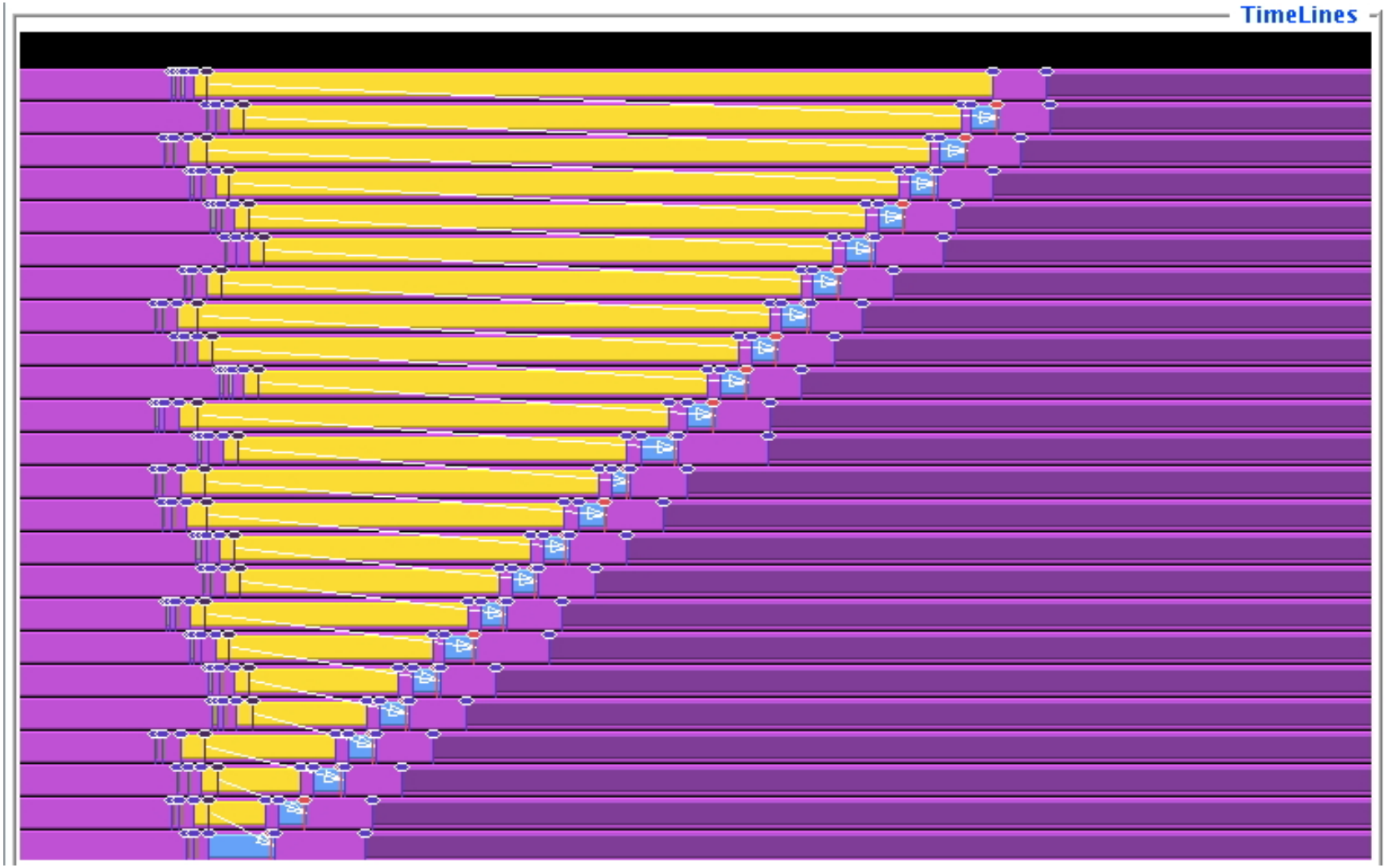
(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

1. Give the paper to your right neighbor;
2. Accept the paper from your left neighbor.

Including boundary conditions for first and last process, that becomes the following program:

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

# 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?)

Food for thought: what happens if you flip the send and receive call?

# Exercise 20 (`rightsend`)

Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 150. If you don't have TAU, can you show this serialization behavior using timings, for instance running it on an increasing number of processes?

# Synchronous send

Synchronous send:

- sender and receiver synchronize
- No 'eager' sends
- $\Rightarrow$ enforced always blocking behavior

# MPI_Ssend

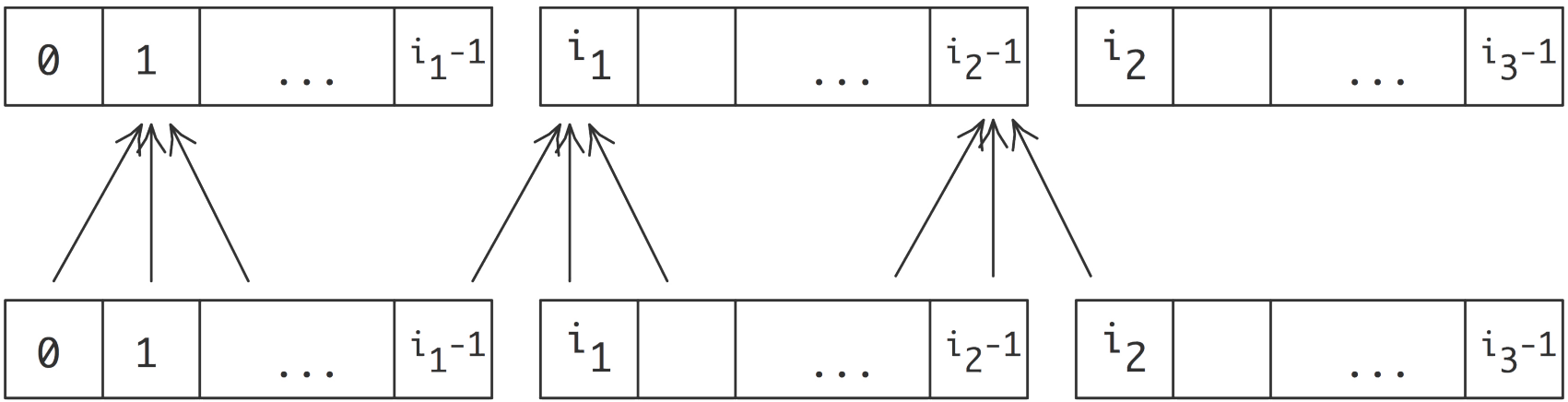
| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Ssend ( | | | | |
| MPI_Ssend_c ( | | | | |
| | buf | initial address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in send buffer | [ int<br>MPI_Count | INTEGER |
| | datatype | datatype of each send buffer element | MPI_Datatype | TYPE(MPI_Datatype) |
| | dest | rank of destination | int | INTEGER |
| | tag | message tag | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | ) | | | |

# Pairwise exchange

# Operating on distributed data

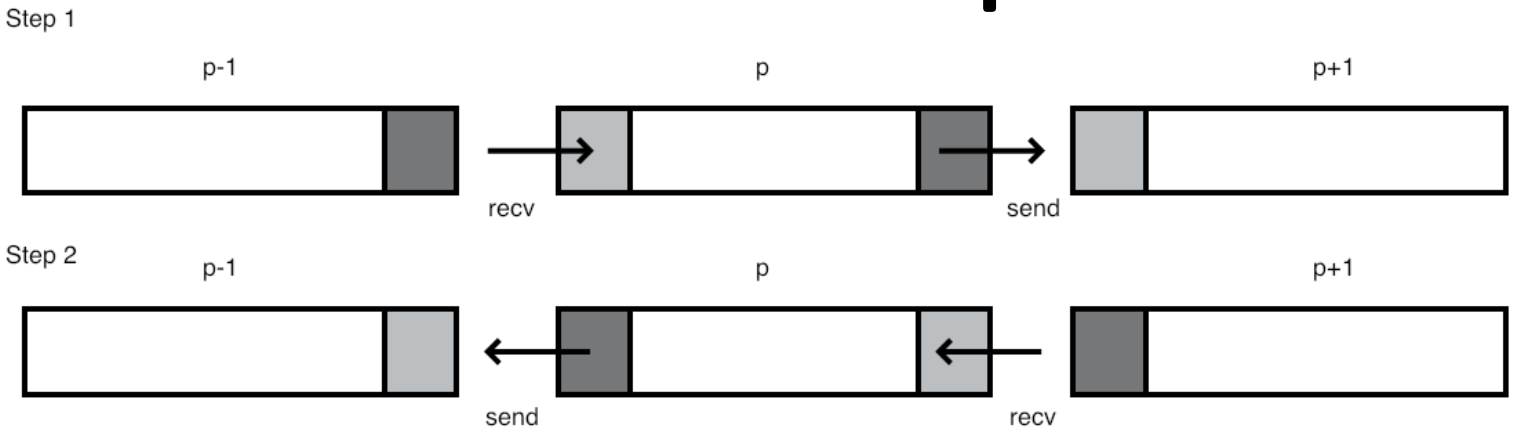
Take another look:

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




- One-dimensional data and linear process numbering;
- Operation between neighboring indices: communication between neighboring processes.

# Two steps




First do all the data movement to the right, later to the left.

- Each process does a send and receive
- So everyone does the send, then the receive? We just saw the problem with that.
- Better solution coming up!

# Sendrecv

Instead of separate send and receive: use

`MPI_Sendrecv`

Combined calling sequence of send and receive;
execute such that no deadlock or sequentialization.

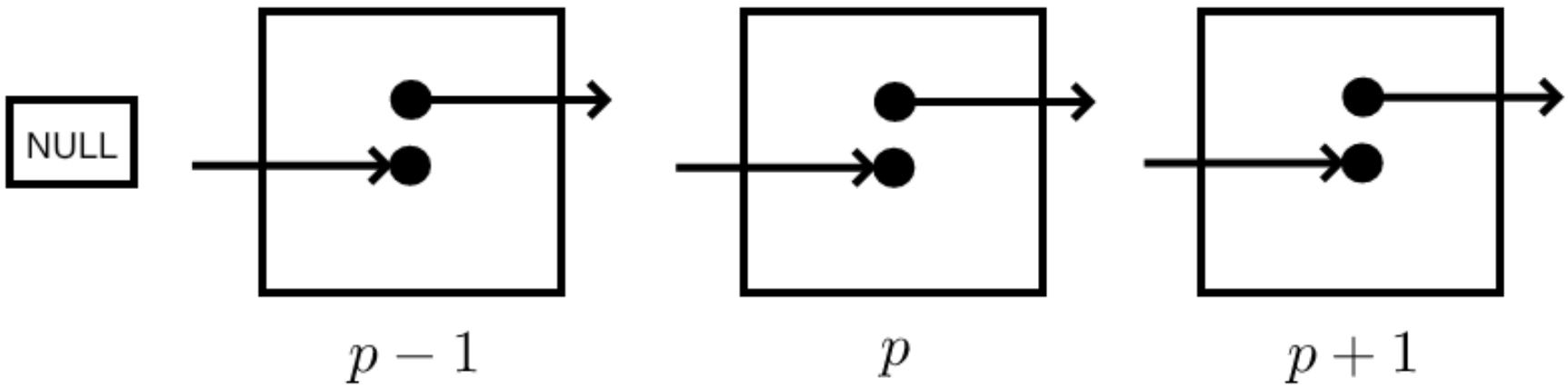
(Also: `MPI_Sendrecv_replace` with single buffer.)

# MPI_Sendrecv

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Sendrecv ( | | | | |
| MPI_Sendrecv_c ( | | | | |
| | sendbuf | initial address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | sendcount | number of elements in send buffer | [ int<br>MPI_Count | INTEGER |
| | sendtype | type of elements in send buffer | MPI_Datatype | TYPE(MPI_Datatype) |
| | dest | rank of destination | int | INTEGER |
| | sendtag | send tag | int | INTEGER |
| | recvbuf | initial address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | recvcount | number of elements in receive buffer | [ int<br>MPI_Count | INTEGER |
| | recvtype | type of elements receive buffer element | MPI_Datatype | TYPE(MPI_Datatype) |
| | source | rank of source or MPI_ANY_SOURCE | int | INTEGER |
| | recvtag | receive tag or MPI_ANY_TAG | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | status | status object | MPI_Status* | TYPE(MPI_Status) |
| | ) | | | |

# SPMD picture

What does process $p$ do?



# Sendrecv with incomplete pairs

```
1  MPI_Comm_rank( .... &procno );
2  if ( /* I am not the first process */ )
3    predecessor = procno-1;
4  else
5    predecessor = MPI_PROC_NULL;
6
7  if ( /* I am not the last process */ )
8    successor = procno+1;
9  else
10   successor = MPI_PROC_NULL;
11
12 sendrecv(from=predecessor,to=successor);
```

(Receive from `MPI_PROC_NULL` succeeds without altering the receive buffer.)

# A point of programming style

The previous slide had:

- a conditional for computing the sender and receiver rank;
- a single Sendrecv call.

Also possible:

```
1 if ( /* i am first */ )
2   Sendrecv( to=right, from=NULL );
3 else if ( /* i am last */
4   Sendrecv( to=NULL,  from=left );
5 else
6   Sendrecv( to=right, from=left );
```

```
1 if ( /* i am first */ )
2   Send( to=right );
3 else if ( /* i am last */
4   Recv( from=left );
5 else
6   Sendrecv( to=right, from=left );
```

But:
Code duplication is error-prone, also
chance of deadlock by missing a case

# Exercise (optional) 21 (`rightsend`)

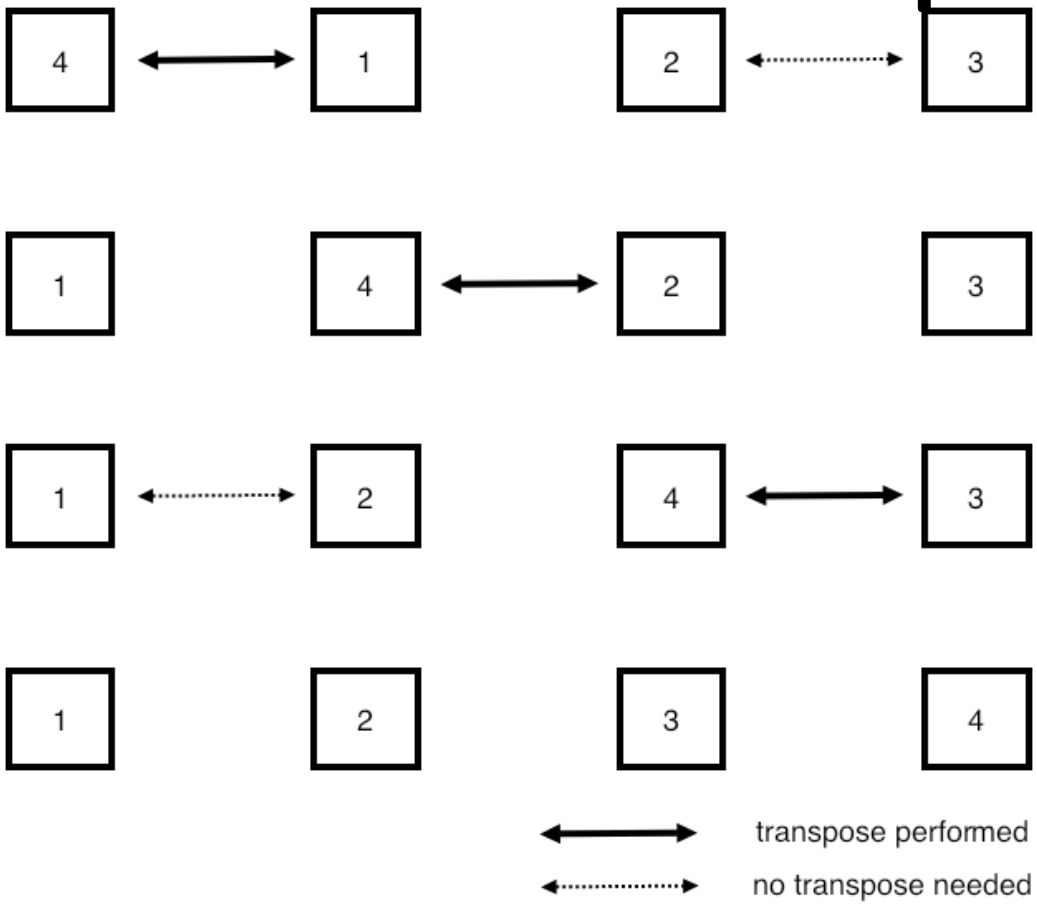
Revisit exercise 19 and solve it using `MPI_Sendrecv`.

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 22 (`sendrecv`)

Implement the above three-point combination scheme using `MPI_Sendrecv`; every processor only has a single number to send to its neighbor.

# Odd-even transposition sort




Odd-even transposition sort on 4 elements.

# Exercise (optional) 23

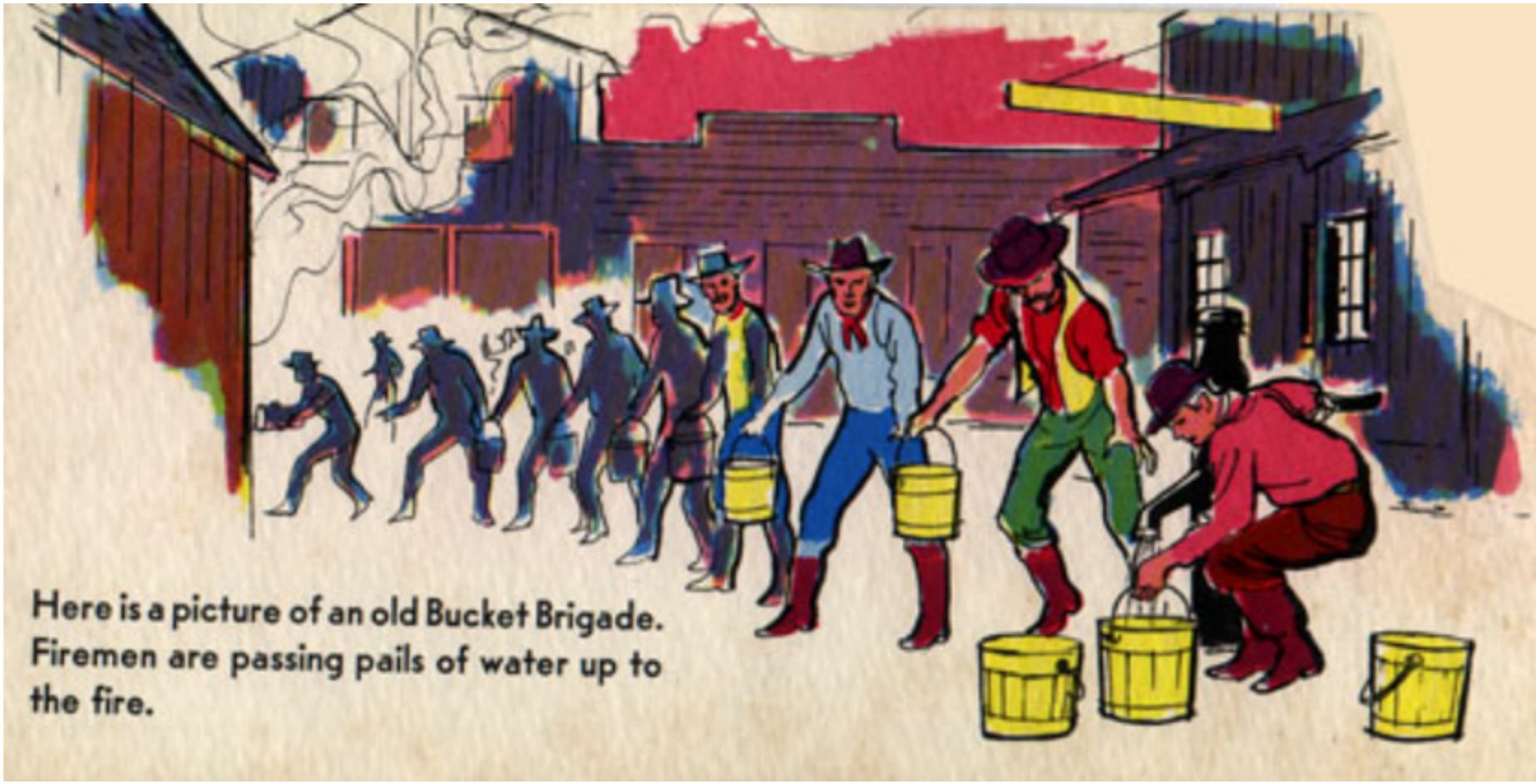
A very simple sorting algorithm is *swap sort* or *odd-even transposition sort*: pairs of processors compare data, and if necessary exchange. The elementary step is called a *compare-and-swap*: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the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

The transposition sort algorithm is split in even and odd stages, where in the even stage processors $2i$ and $2i+1$ compare and swap data, and 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; see figure 165.

Implement this algorithm using `MPI_Sendrecv`. (Use `MPI_PROC_NULL` for the edge cases if needed.) Use a gather call to print the global state of the distributed array at the beginning and end of the sorting process.

# Bucket brigade

Sometimes you really want to pass information from one process to the next: 'bucket brigade'



# Exercise 24 (`bucketblock`)

Take the code of exercise 20 and modify it so that the data from process zero gets propagated to every process. Specifically, compute all partial sums $\sum_{i=0}^{p} i^2$:

$$\begin{cases} x_0 = 1 & \text{on process zero} \\ x_p = x_{p-1} + (p+1)^2 & \text{on process } p \end{cases}$$

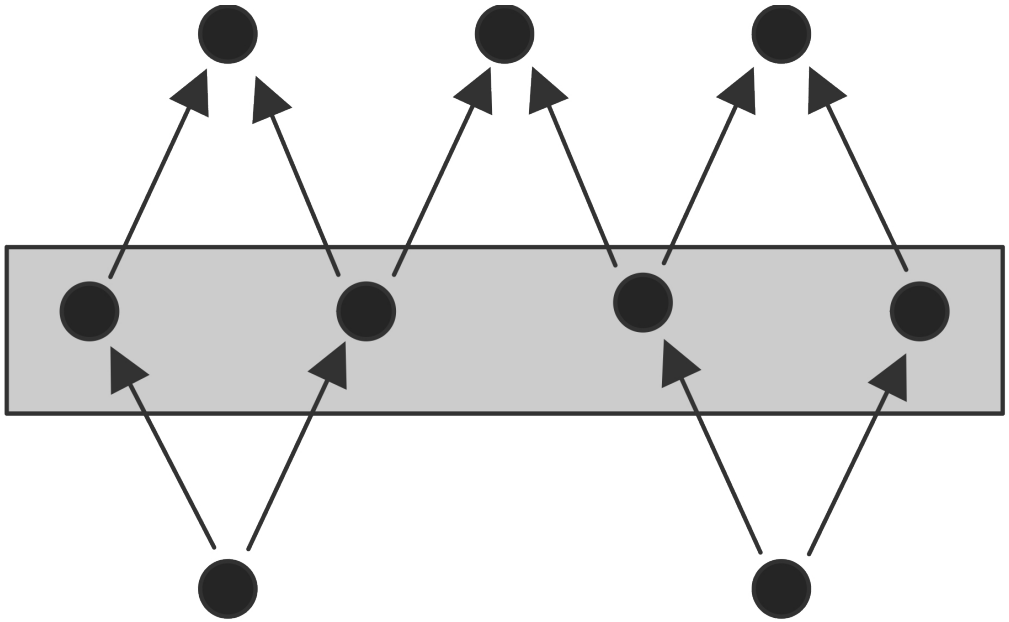
Use `MPI_Send` and `MPI_Recv`; make sure to get the order right.

Food for thought: all quantities involved here are integers. Is it a good idea to use the integer datatype here?

# Irregular exchanges: non-blocking communication

# Sending with irregular connections

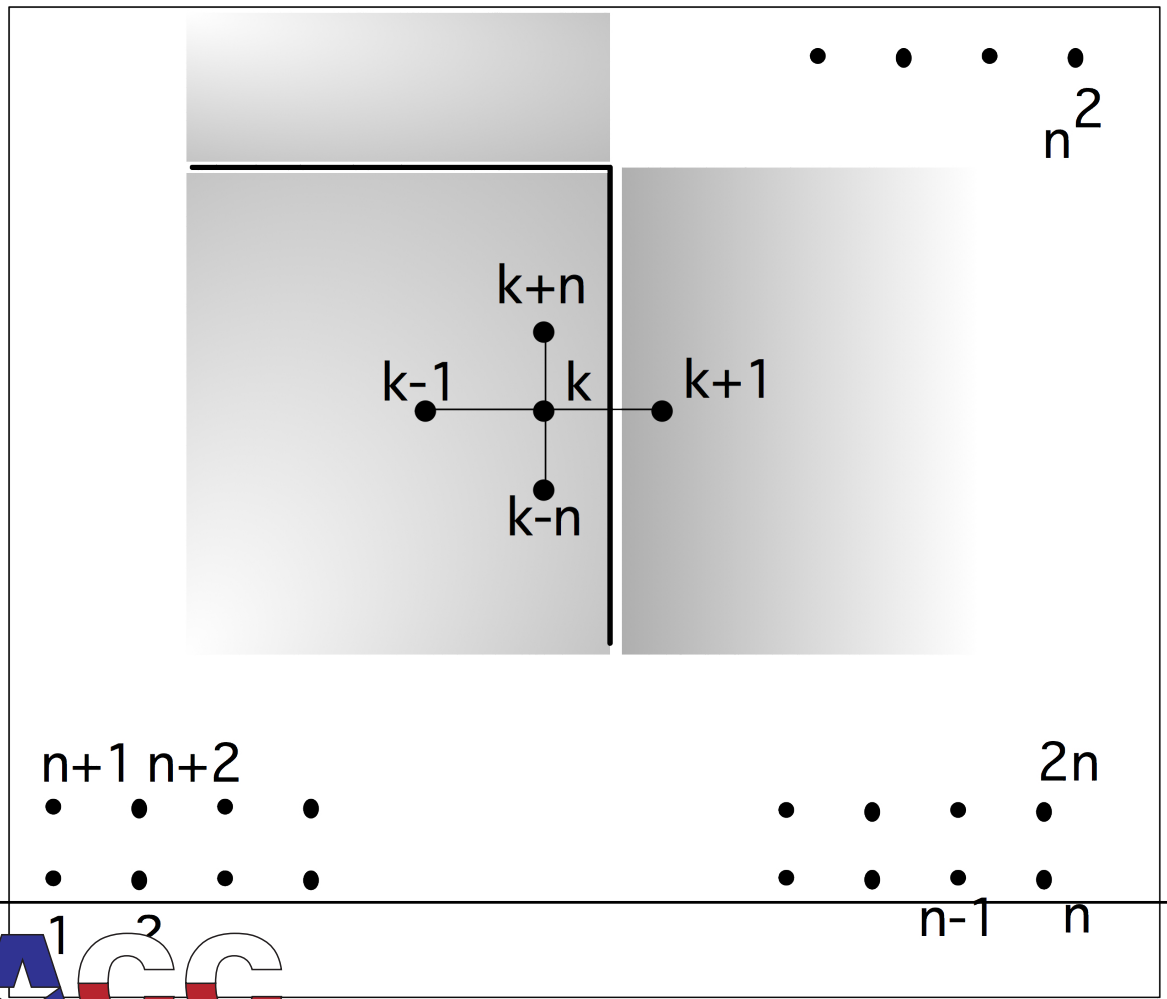
Graph operations:



# Communicating other than in pairs

# PDE, 2D case

A difference stencil applied to a two-dimensional square domain, distributed over processes. A cross-process connection is indicated
$\Rightarrow$ complicated to express pairwise

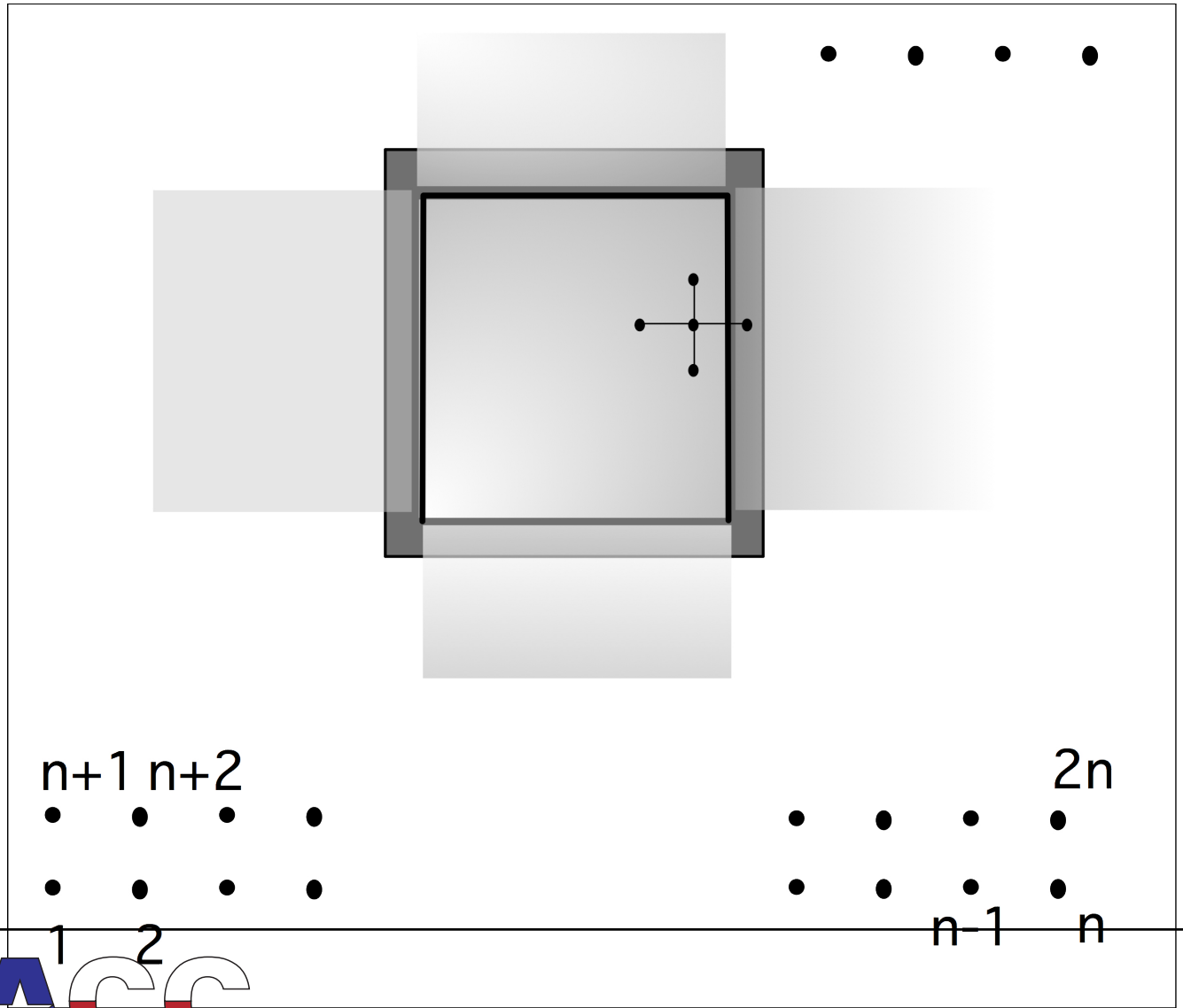


# PDE matrix

$$
A = \left(\begin{array}{cccccc|cccccc|cc}
4 & -1 & & & \emptyset & & -1 & & & & \emptyset & & & \\
-1 & 4 & -1 & & & & & -1 & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & \ddots & & & & & \\
 & & \ddots & \ddots & -1 & & & & & \ddots & & & & \\
\emptyset & & & -1 & 4 & & \emptyset & & & & -1 & & & \\
\hline
-1 & & & & \emptyset & & 4 & -1 & & & & & -1 & \\
 & -1 & & & & & -1 & 4 & -1 & & & & & -1 \\
 & \uparrow & \ddots & & & & \uparrow & \uparrow & \uparrow & & & & \uparrow & \\
 & k-n & & & & & k-1 & k & k+1 & & -1 & & k+n & \\
 & & & & -1 & & & & & -1 & 4 & & & \\
\hline
 & & & & & & \ddots & & & & & & \ddots &
\end{array}\right)
$$

# Halo region

The halo region of a process, induced by a stencil



# 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 process 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

- `MPI_Isend` / `MPI_Irecv` does not send/receive:
- They declare a buffer.
- The buffer contents are there after a wait call.
- In between the `MPI_Isend` and `MPI_Wait` the data may not have been sent.
- In between the `MPI_Irecv` and `MPI_Wait` the data may not have arrived.

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

# MPI_Isend

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Isend ( | | | | |
| MPI_Isend_c ( | | | | |
| | buf | initial address of send buffer | const void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in send buffer | int<br>MPI_Count | INTEGER |
| | datatype | datatype of each send buffer element | MPI_Datatype | TYPE(MPI_Datatype) |
| | dest | rank of destination | int | INTEGER |
| | tag | message tag | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | request | communication request | MPI_Request* | TYPE(MPI_Request) |
| | ) | | | |

# MPI_Irecv

| Name | Param name | Explanation | C type | F type |
|---|---|---|---|---|
| MPI_Irecv ( | | | | |
| MPI_Irecv_c ( | | | | |
| | buf | initial address of receive buffer | void* | TYPE(*), DIMENSION(..) |
| | count | number of elements in receive buffer | int<br>MPI_Count | INTEGER |
| | datatype | datatype of each receive buffer element | MPI_Datatype | TYPE(MPI_Datatype) |
| | source | rank of source or MPI_ANY_SOURCE | int | INTEGER |
| | tag | message tag or MPI_ANY_TAG | int | INTEGER |
| | comm | communicator | MPI_Comm | TYPE(MPI_Comm) |
| | request | communication request | MPI_Request* | TYPE(MPI_Request) |
| | ) | | | |

# Request waiting

Basic wait:

```
1 MPI_Wait( MPI_Request*, MPI_Status* );
```

Most common way of waiting for completion:

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

- ignore status: `MPI_STATUSES_IGNORE`
- also `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome`

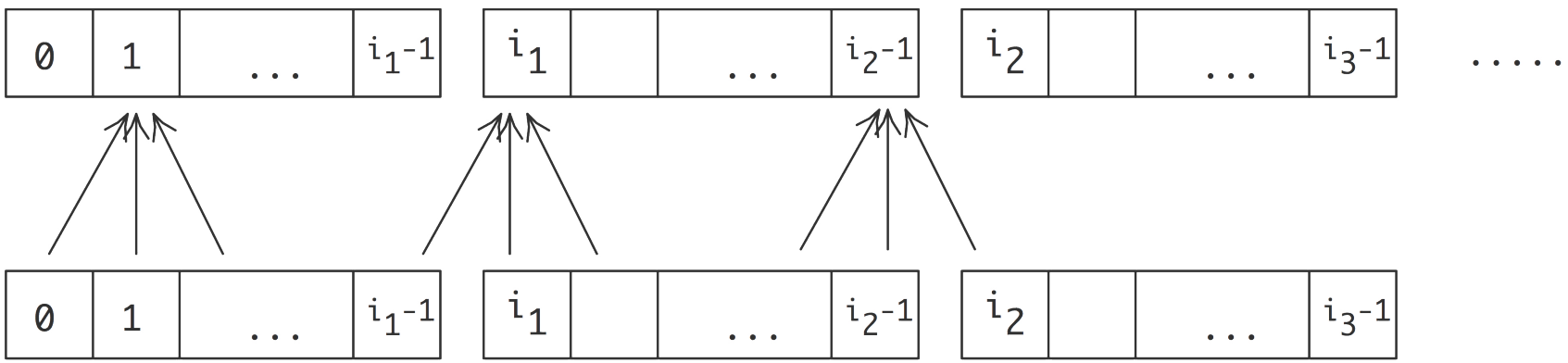
# Exercise 25 (`isendirecv`)

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. There are two approaches to the first and last process:

1. you can use `MPI_PROC_NULL` for the 'missing' communications;
2. you can skip these communications altogether, but now you have to count the requests carefully.




(Can you think of a different way of handling the end points?)

# Comparison

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

# Buffer use in blocking/non-blocking case

Blocking:

```
1 double *buffer;
2 // allocate the buffer
3 for ( ... p ... ) {
4    buffer = // fill in the data
5    MPI_Send( buffer, ... /* to: */ p );
```

Non-blocking:

```
1 double **buffers;
2 // allocate the buffers
3 for ( ... p ... ) {
4    buffers[p] = // fill in the data
5    MPI_Isend( buffers[p], ... /* to: */ p );
6 MPI_Waitsomething(.....)
```

# Pitfalls

- Strictly one request/wait per `isend`/`irecv`:
  can not use one request for multiple simultaneous `isends`
- Some people argue:

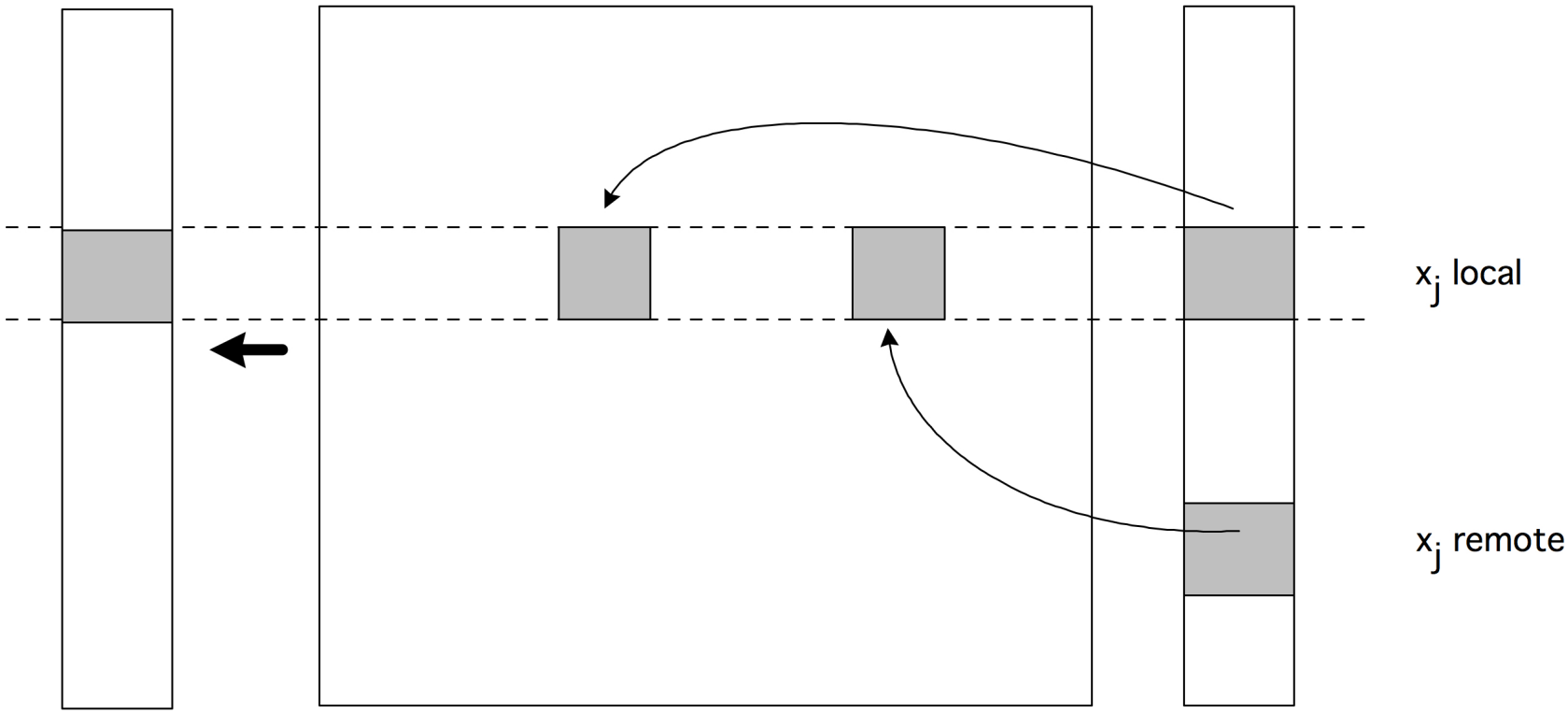
  > *Wait for the send is not necessary: if you wait for the receive, the message has arrived safely*

  This leads to memory leaks! The `wait` call deallocates the request object.

# Matrices in parallel

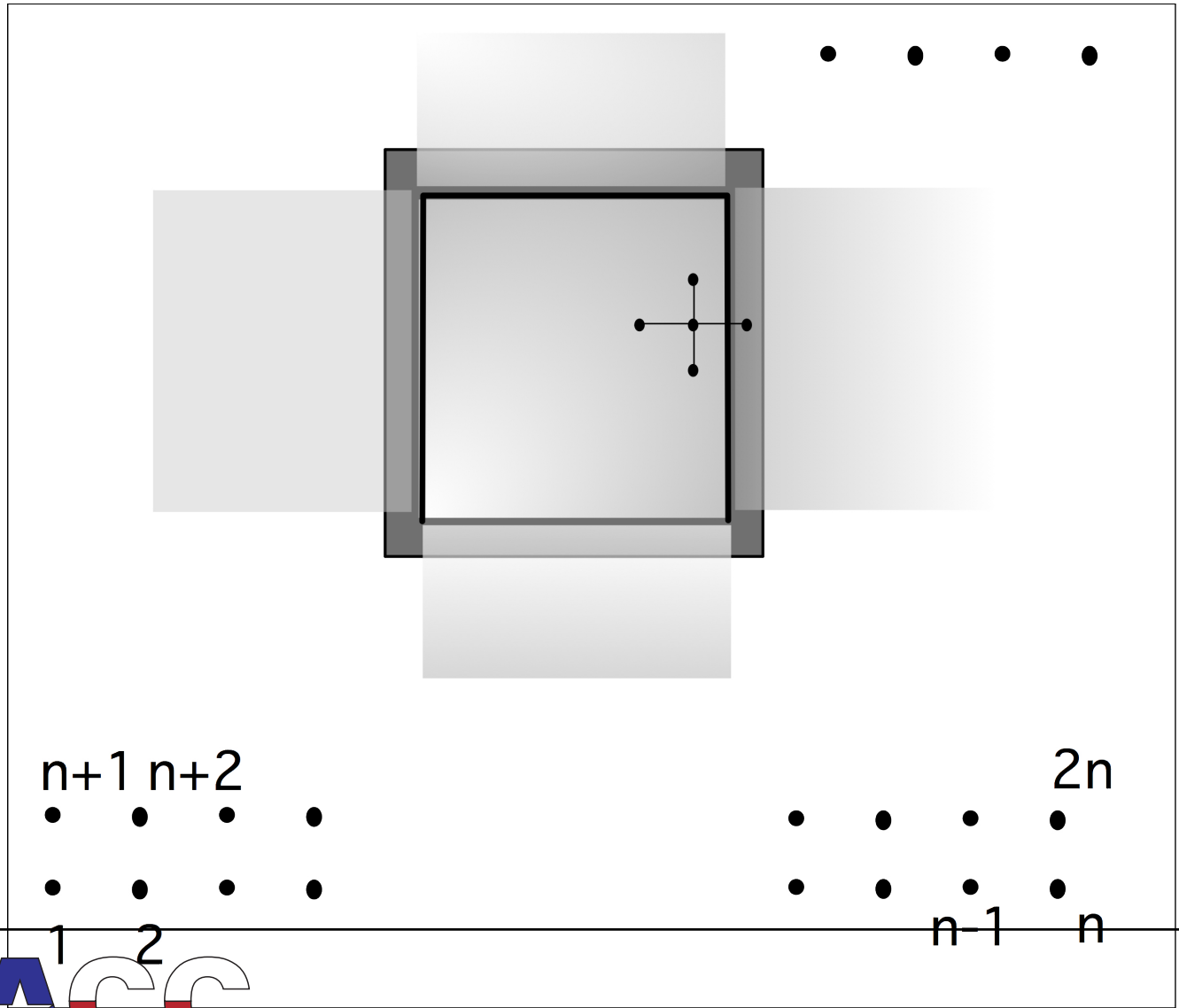
$$y \leftarrow Ax$$

and $A, x, y$ all distributed:



# Hiding the halo

Interior of a process domain can overlap with halo transfer:



# 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 neighbor processes
4. Incorporate incoming data.

# Exercise 26 (`isendirecvarray`)

Take your code of exercise 25 and modify it to use latency hiding. Operations that can be performed without needing data from neighbors should be performed in between the `MPI_Isend` / `MPI_Irecv` calls and the corresponding `MPI_Wait` calls.

Write your code so that it can achieve latency hiding.

# Mix and match

You can match blocking and non-blocking:

```
1 if ( /* I am Process A */ ) {
2   MPI_Irecv( /* from B */, &req1 );
3   MPI_Isend( /* to B */, &req2 );
4   MPI_Waitall( /* requests 1 and 2 */ );
5 } else if ( /* I am Process B */ ) {
6   MPI_Recv( /* from A */ );
7   MPI_Send( /* to A */ );
8 }
```

# Test: non-blocking wait

- Post non-blocking receives
- test on the request(s) for incoming messages
- if nothing comes in, do local work

```
1 while (1) {
2   MPI_Test( some_request, &flag );
3   if (flag)
4     // do something with incoming message
5   else
6     // do local work
7 }
```

Local operation.

Also `MPI_Testall` et cetera.

# Probe for message

Is there a message?

```
1 // probe.c
2 if (procno==receiver) {
3   MPI_Status status;
4   MPI_Probe(sender,0,comm,&status);
5   int count;
6   MPI_Get_count(&status,MPI_FLOAT,&count);
7   float recv_buffer[count];
8   MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE)
       ;
9 } else if (procno==sender) {
10  float buffer[buffer_size];
11  ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(
       ierr);
12 }
```

(Also non-blocking `MPI_Iprobe`.)

These commands force global progress.

# The Pipeline Pattern

- Remember the bucket brigade: data propagating through processes
- If you have many buckets being passed: pipeline
- This is very parallel: only filling and draining the pipeline is not completely parallel
- Application to long-vector broadcast: pipelining gives overlap

# Exercise (optional) 27 (`bucketpipenonblock`)

Implement a pipelined broadcast for long vectors:
use non-blocking communication to send the vector in parts.

# Exercise 28 (`setdiff`)

Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.

How could you solve this with an `MPI_Allgather` call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.

Consider: `MPI_Send` and `MPI_Recv` vs `MPI_Sendrecv` vs `MPI_Sendrecv_replace` vs `MPI_Isend` and `MPI_Irecv`

# The wheel of reinvention

The circular bucket brigade is the idea behind the 'Horovod' library, which is the key to efficient parallel Deep Learning.

# More sends and receive

- `MPI_Bsend`, `MPI_Ibsend`: buffered send
- `MPI_Ssend`, `MPI_Issend`: synchronous send
- `MPI_Rsend`, `MPI_Irsend`: ready send
- Persistent communication: repeated instance of same proc/data description.

  **MPI-4:**

  - *Partitioned sends.*

too obscure to go into.

# Review 4

Does this code deadlock?

```
1 for (int p=0; p<nprocs; p++)
2   if (p!=procid)
3     MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
4 for (int p=0; p<nprocs; p++)
5   if (p!=procid)
6     MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
```

/poll "This code deadlocks" "Yes" "No" "Maybe"

# Review 5

Does this code deadlock?

```
1 int ireq = 0;
2 for (int p=0; p<nprocs; p++)
3   if (p!=procid)
4     MPI_Isend(sbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
5 for (int p=0; p<nprocs; p++)
6   if (p!=procid)
7     MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
8 MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
```

```
/poll "This code deadlocks" "Yes" "No" "Maybe"
```

# Review 6

Does this code deadlock?

```
1 int ireq = 0;
2 for (int p=0; p<nprocs; p++)
3   if (p!=procid)
4     MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
5 MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
6 for (int p=0; p<nprocs; p++)
7   if (p!=procid)
8     MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
```

```
/poll "This code deadlocks" "Yes" "No" "Maybe"
```

# Review 7

Does this code deadlock?

```
1 int ireq = 0;
2 for (int p=0; p<nprocs; p++)
3   if (p!=procid)
4     MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
5 for (int p=0; p<nprocs; p++)
6   if (p!=procid)
7     MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
8 MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
```

```
/poll "This code deadlocks" "Yes" "No" "Maybe"
```

# Other topics

# Where to go from here...

- Derived data types: send strided/irregular/inhomogeneous data
- Sub-communicators: work with subsets of `MPI_COMM_WORLD`
- I/O: efficient file operations
- One-sided communication: 'just' put/get the data somewhere
- Process management
- Non-blocking collectives
- Graph topology and neighborhood collectives
- Shared memory

# cmake

. MPI is discoverable by cmake.

# CMake for C

```
1  cmake_minimum_required( VERSION 3.12 )
2  project( ${PROJECT_NAME} VERSION 1.0 )
3
4  # https://cmake.org/cmake/help/latest/module/FindMPI.html
5  find_package( MPI )
6
7  add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.c )
8  target_include_directories(
9          ${PROJECT_NAME} PUBLIC
10         ${MPI_C_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
11 target_link_libraries(
12         ${PROJECT_NAME} PUBLIC
13         ${MPI_C_LIBRARIES} )
14
15 install( TARGETS ${PROJECT_NAME} DESTINATION . )
```