TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Tutorial on MPI programming, Foundations

Victor Eijkhout eijkhout@tacc.utexas.edu

TACC training, 2017

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

Overview

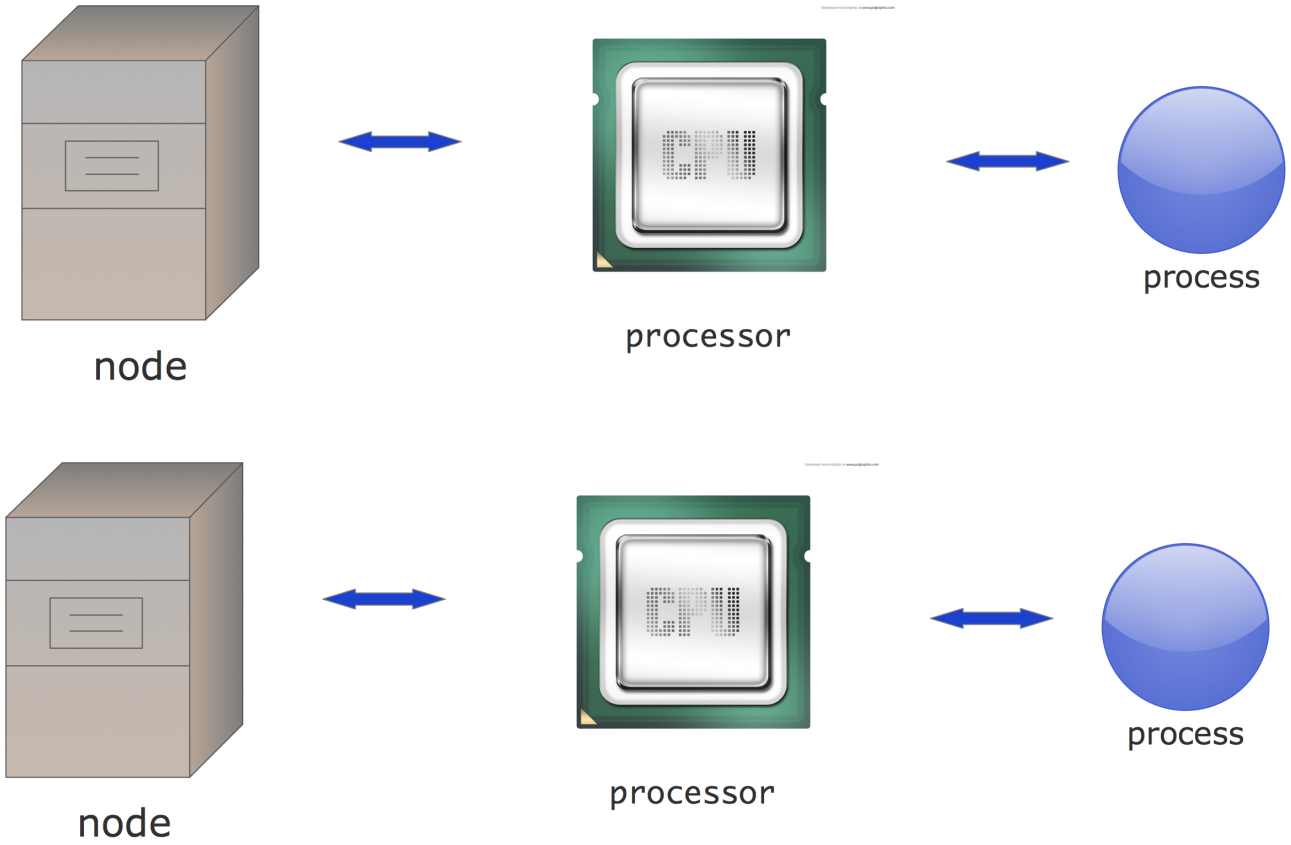
In this section you will learn how to think about parallelism in MPI.

Commands learned:

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

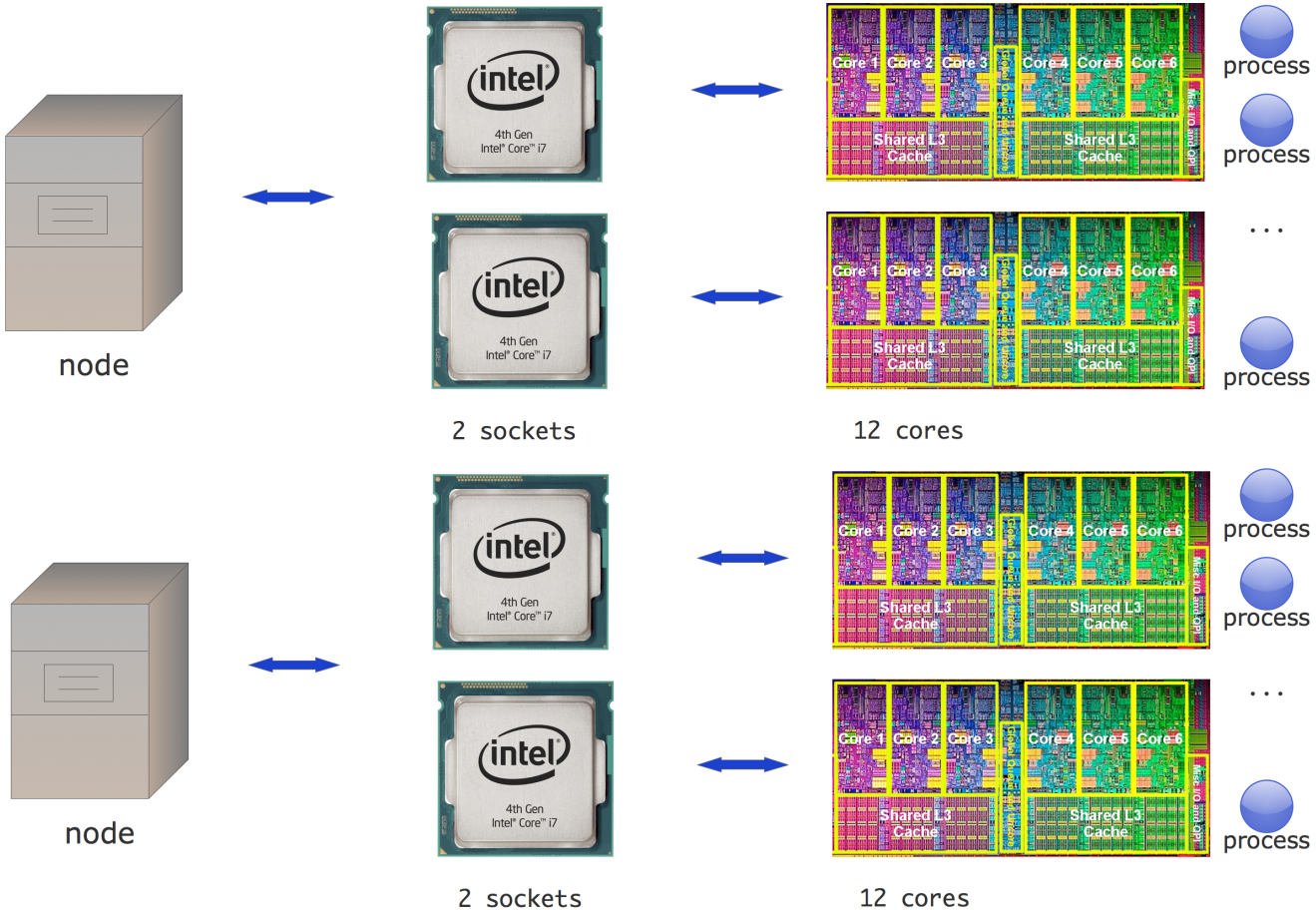
Table of Contents

Computers when MPI was designed



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

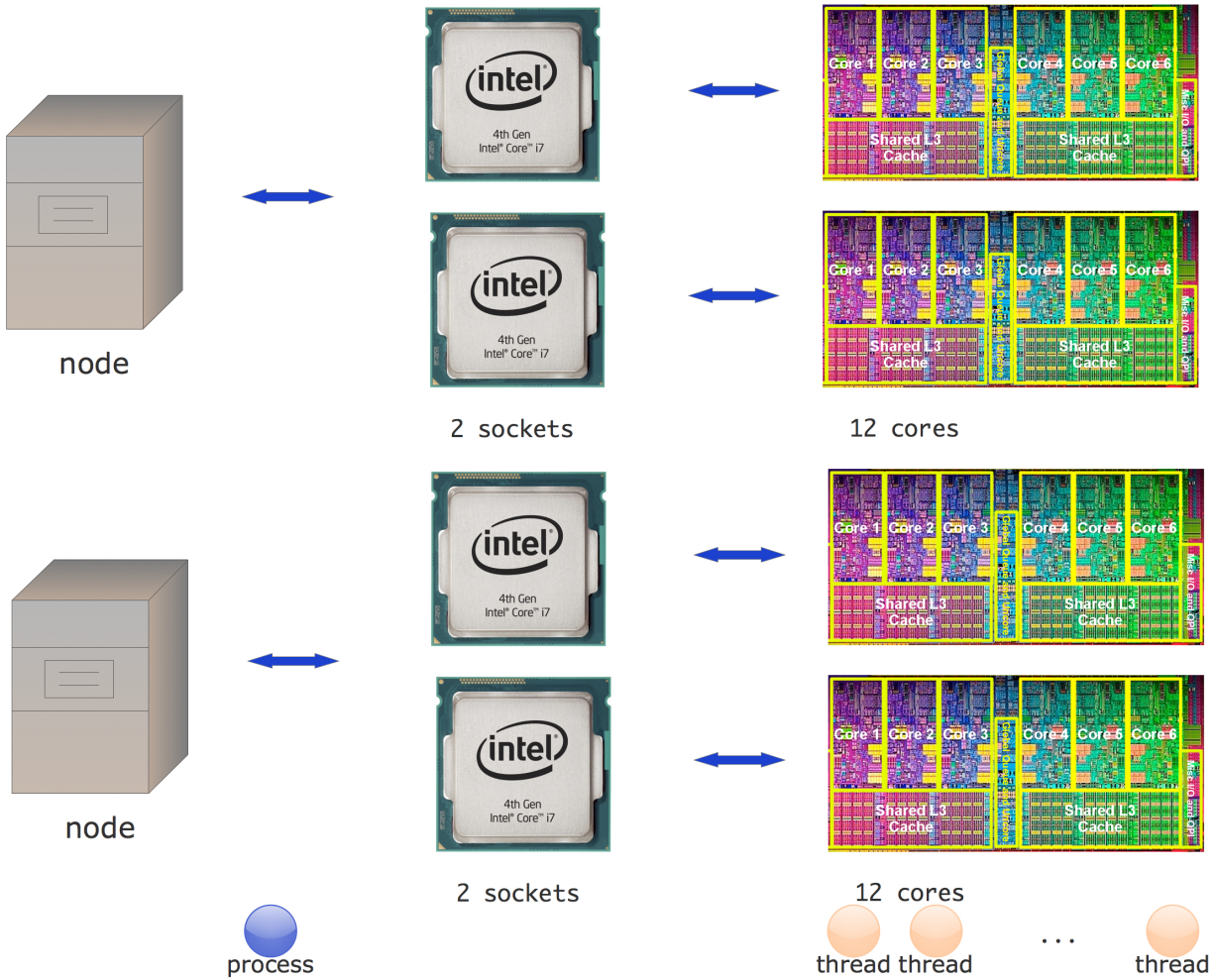
Pure MPI



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.
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.

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 neighbours.

Table of Contents

Compiling and running

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.

Run your program with something like

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

At TACC:

```
ibrun yourprog
```

the number of processes is determined by SLURM.

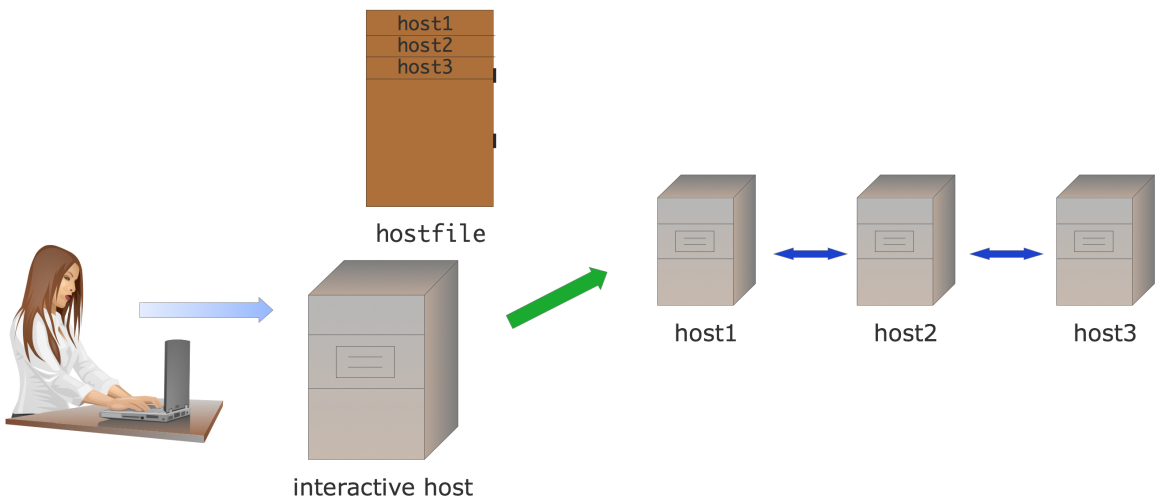
Do I need a supercomputer?

- 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. . .

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*.

How to make exercises

- Directory: `exercises-mpi-c` or `cxx` or `f` or `p`
- If a slide has a `(exercisename)` over it, there will be a template program `exercisename.c` (or `F90` or `py`).
- Type `make exercisename` to compile it
- Python: no compilation needed. Run:
  `ibrun python yourprogram`
- Add an exercise of your own to the makefile: add the name to the `EXERCISES`

Exercise 1 (`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.

(On TACC machines such as stampede, use `ibrun`, no processor count.)

In a picture

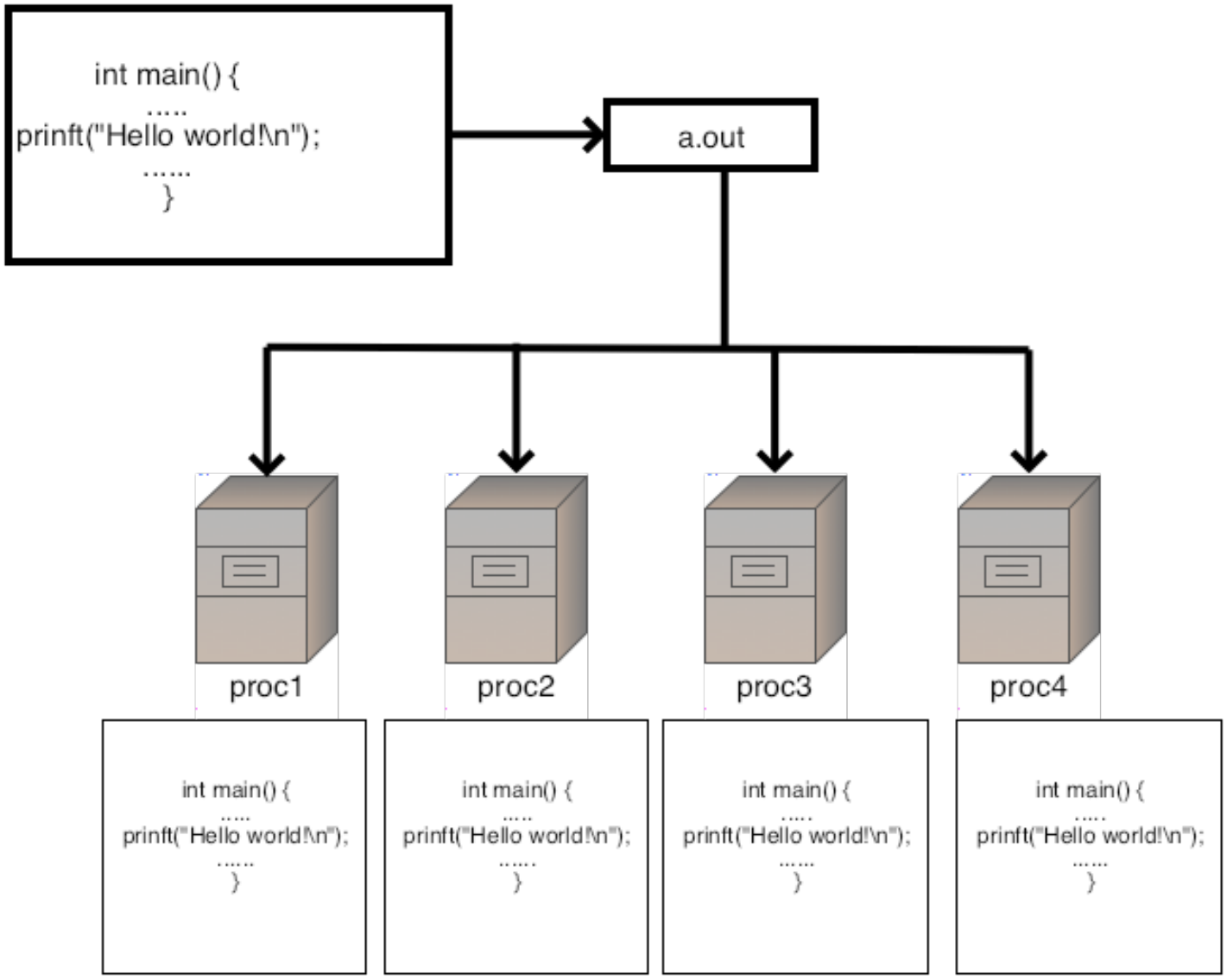

int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}

Table of Contents

MPI definitions

You need an include file:

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

- There are no real C++ bindings.
- There are true Fortran bindings, but only 2008 standard, and not widely supported yet.

MPI Init / Finalize

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 load python
```

```
from mpi4py import MPI
```

Run:

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

No initialization needed.

Exercise 2

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

Exercise (optional) 3

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

(The character buffer needs to be allocated by you, it is not created by MPI, with size at least `MPI_MAX_PROCESSOR_NAME`.) TACC nodes have a hostname `cRRR-CNN`, where RRR is the rack number, C is the chassis number in the rack, and NN is the node number within the chassis. Communication is faster inside a rack than between racks!

```
C:
int MPI_Get_processor_name(char *name, int *resultlen)
name : buffer char[MPI_MAX_PROCESSOR_NAME]

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

Python:
MPI.Get_processor_name()
```

How to read routine prototypes: ***??****.*

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_size( comm,&nprocs );
```

(but forget about that error code most of the time)

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)
```

Process identification

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

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

For now, the communicator will be `MPI_COMM_WORLD`.

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

```
Semantics:
MPI_COMM_SIZE(comm, size)
IN comm: communicator (handle)
OUT size: number of processes in the group of comm (integer)

C:
int MPI_Comm_size(MPI_Comm comm, int *size)

Fortran:
MPI_Comm_size(comm, size, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Comm.Get_size(self)
```

How to read routine prototypes: **??**.

```
Semantics:
MPI_COMM_RANK(comm, rank)
IN comm: communicator (handle)
OUT rank: rank of the calling process in group of comm (integer)

C:
int MPI_Comm_rank(MPI_Comm comm, int *rank)

Fortran:
MPI_Comm_rank(comm, rank, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python:
MPI.Comm.Get_rank(self)
```

How to read routine prototypes: **??**.

Exercise 4 (`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 5 (`commrank`)

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

In a picture

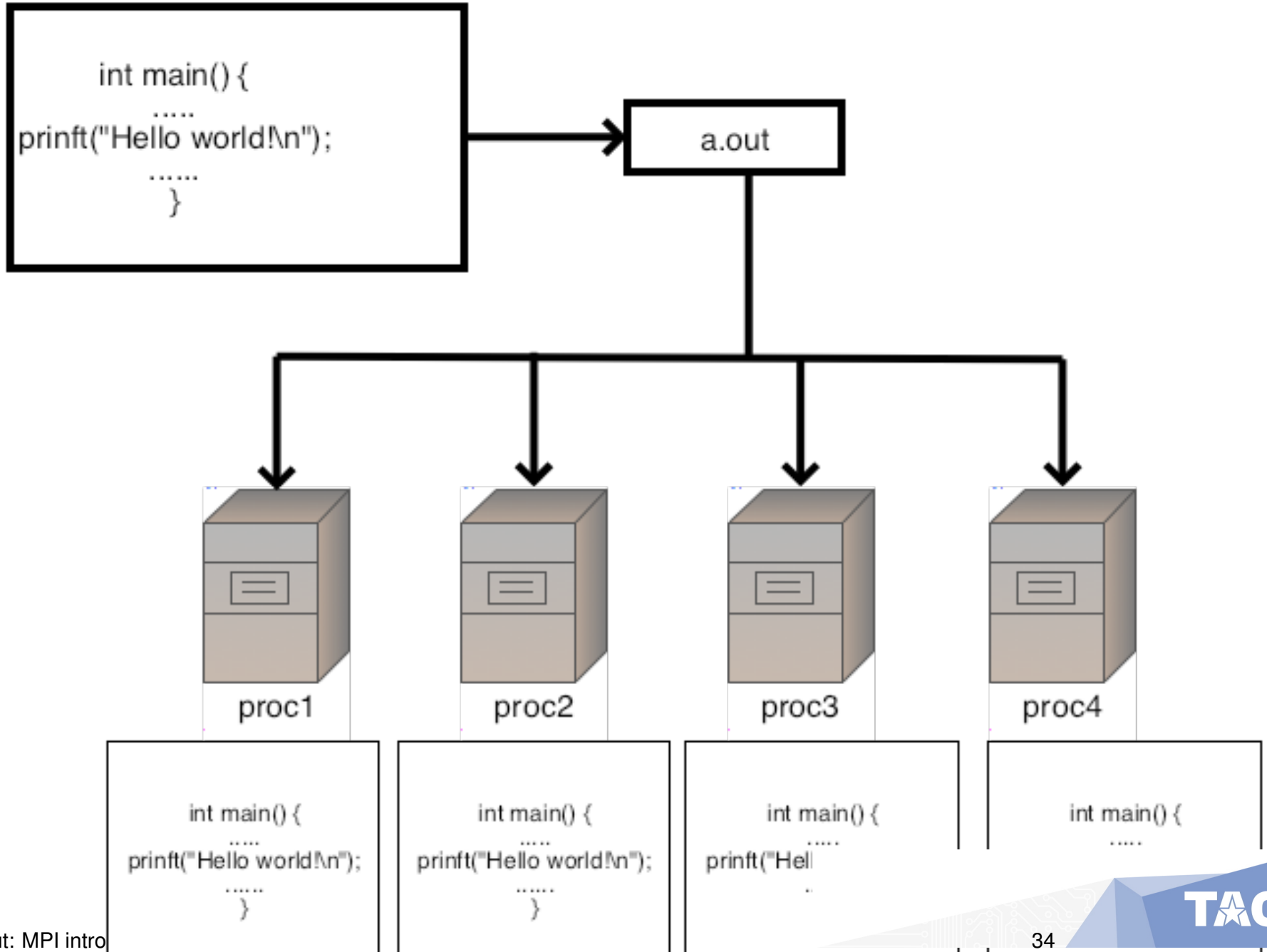

int main() {
.....
prinft("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hello world!\n");
.....
}
int main() {
.....
prinft("Hell
int main() {
.....

Table of Contents

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 6 (`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 $\sqrt{N} \approx 45,200$.

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

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`

Table of Contents

Collectives

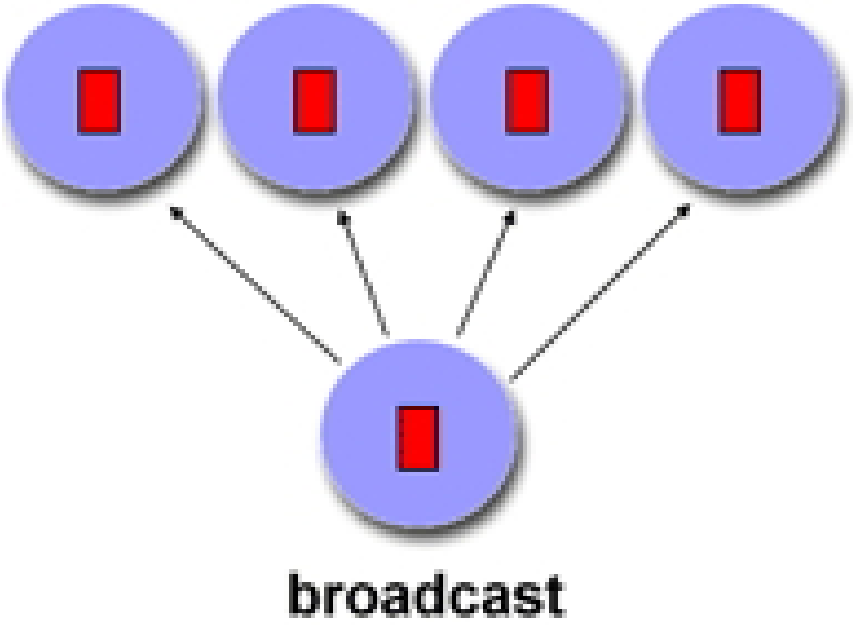
Gathering and spreading information:

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

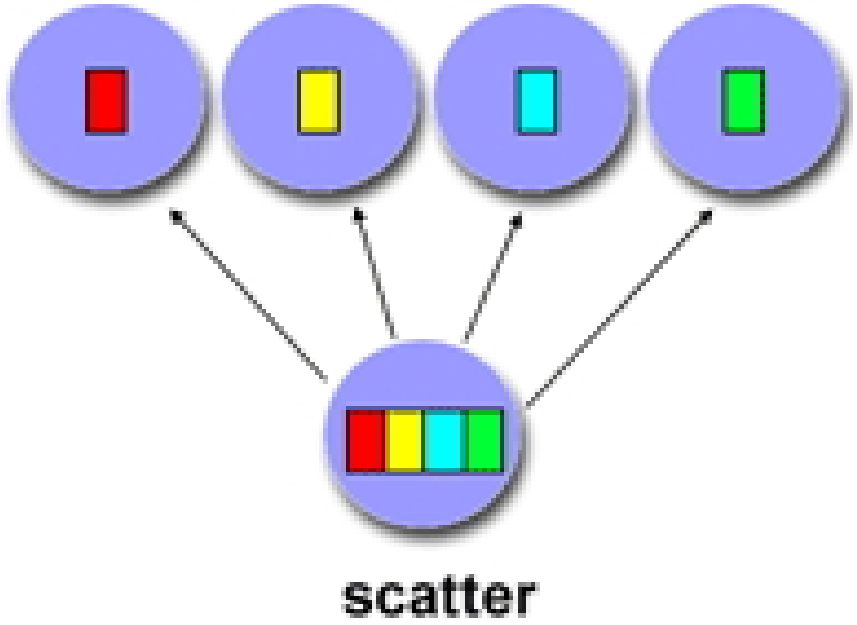
Root process: the one doing the collecting or disseminating.

Basic cases:

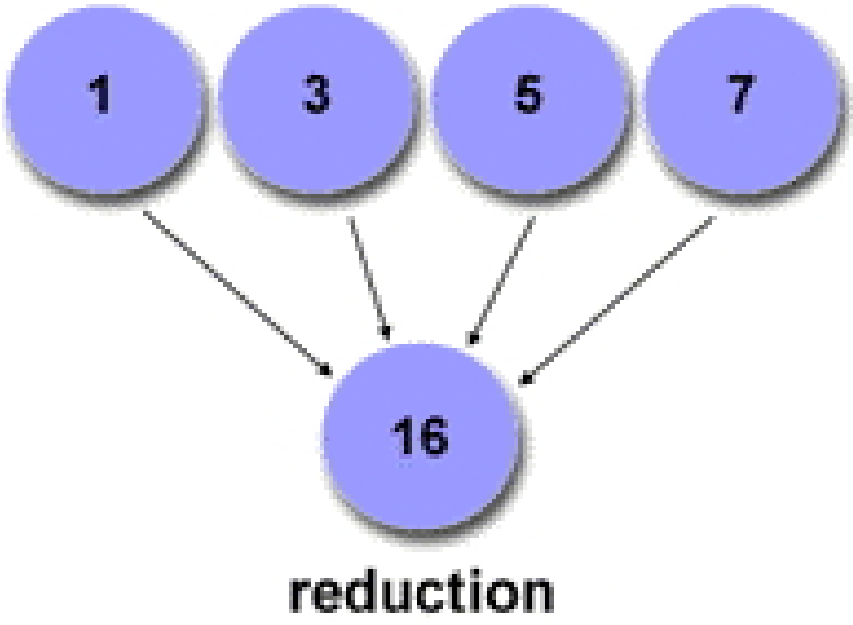
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



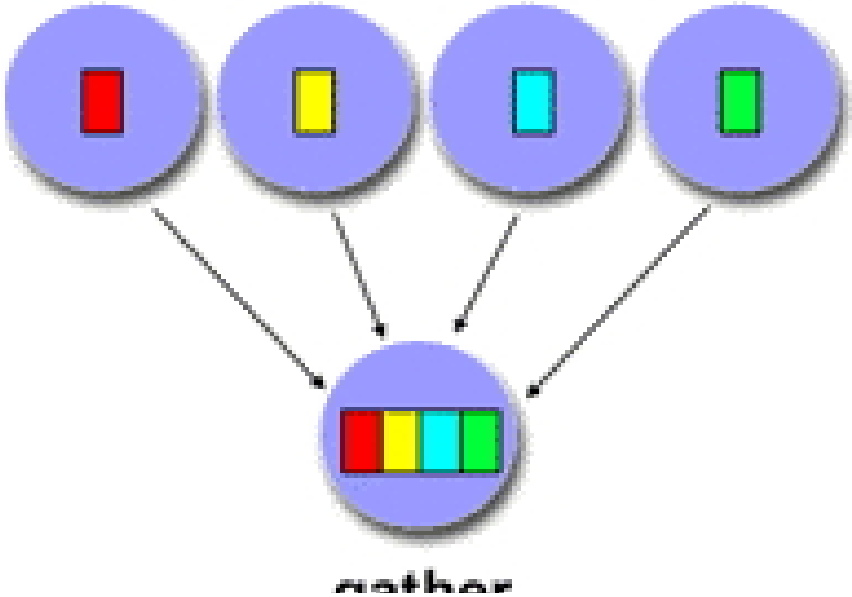
broadcast



scatter



reduction



gather

Exercise 7

How would you realize the following scenarios with MPI collectives?

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

More collectives

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

...and more

Table of Contents

Motivation for 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 processor stores just one x_i value.

1. The calculation of the average μ 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 processors.
3. $\sum_i (x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Might as well use alreduce.

Allreduce

Often: everyone needs the result of a reduction

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

- 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.

Reduction to single process

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

Allreduce syntax

```
int MPI_Allreduce(
  const void* sendbuf,
  void* recvbuf, int count, MPI_Datatype datatype,
  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.
- `MPI_Datatype` is `MPI_INT`, `MPI_REAL8` et cetera.
- `MPI_Op` is `MPI_SUM`, `MPI_MAX` et cetera.

Elementary datatypes

C	Fortran	meaning
MPI_CHAR MPI_SHORT MPI_INT MPI_FLOAT MPI_DOUBLE	MPI_CHARACTER MPI_BYTE MPI_INTEGER MPI_REAL MPI_DOUBLE_PRECISION MPI_COMPLEX MPI_LOGICAL	
		optional
	MPI_REAL4 MPI_REAL8	

A bunch more.

MPI operators

MPI operator	description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor

A couple more.

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.

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

  ```
  result = comm.recv(...)
  ```

  this uses `pickle`: slow.
- uppercase: communicates `numpy` objects;
  input and output are function argument.

  ```
  result = np.empty(.....)
  comm.Recv(result, ...)
  ```

  basicaly wrapper around C code: fast

Exercise 8 (`randommax`)

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

(The operator is `MPI_SUM` for C/Fortran, or `MPI.SUM` for Python.)

Each process then scales its value by this sum. Compute the sum of the scaled numbers and check that it is 1.

Random numbers in C

```
// Initialize the random number generator
srand(procno*(double)RAND_MAX/nprocs);
// compute a random number
randomfraction = (rand() / (double)RAND_MAX);
```

Random numbers in 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*procno
end do
call random_seed(put=randseed)
```

Exercise (optional) 9

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

Reduction

```
int MPI_Reduce
  (void *sendbuf, void *recvbuf,
   int count, MPI_Datatype datatype,
   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.

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`

Gauss-Jordan elimination

https://youtu.be/aQYuwatlWME

Exercise 10 (`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 $c = 1, \ldots, n$
            $A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{rc}$

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) 11

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

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`

Exercise 12

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

Table of Contents

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 $\sum_{q<p} n_q$.

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

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

Python:
res = Intracomm.scan( sendobj=None,recvobj=None,op=MPI.SUM)
res = Intracomm.exscan( sendobj=None,recvobj=Nor
```

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(*), displs(*), root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recv
TYPE(MPI_Comm), INTENT(IN) :: comm
```

All-to-all

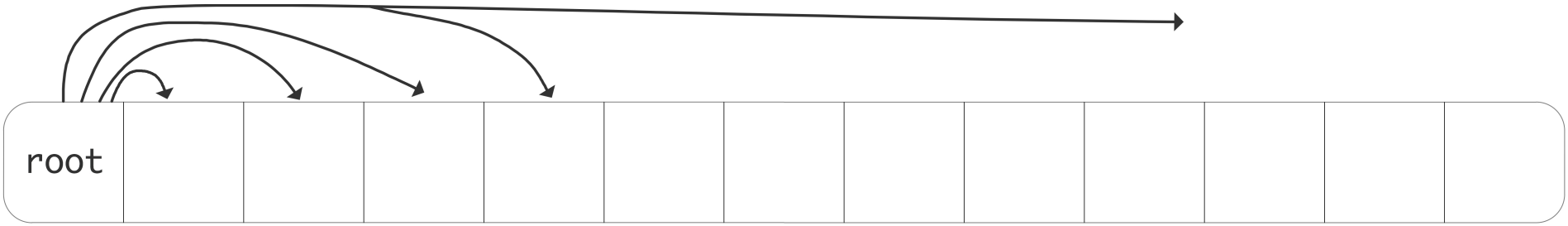
- Every process does a scatter;
- each individual data
- Example: data transposition in FFT

Barrier

- Synchronize processes:
- 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:

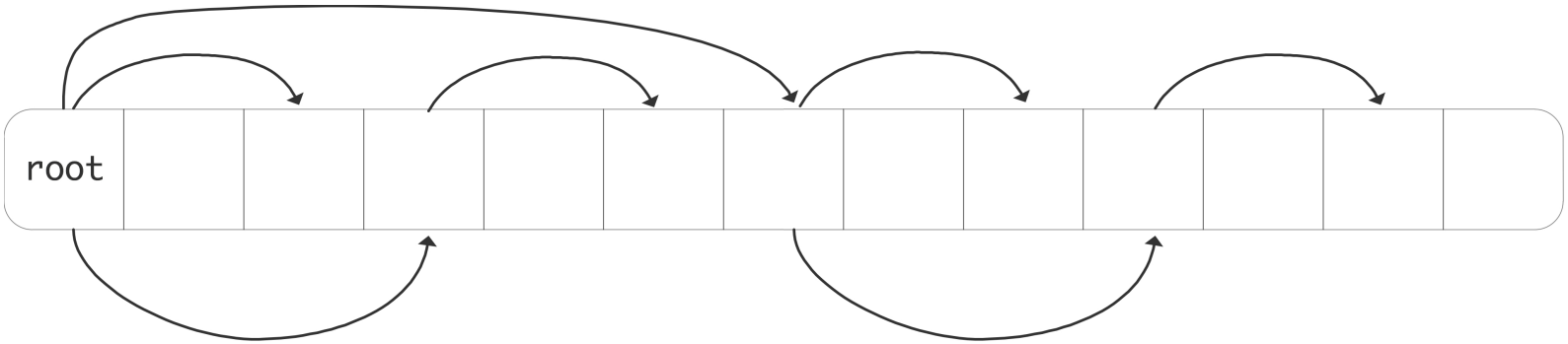


Message time is modeled as

$$\alpha + \beta n$$

Time for collective? Can you improve on that?

Better implementation of collective



What is the running time now?

Point-to-point communication

Overview

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

Commands learned:

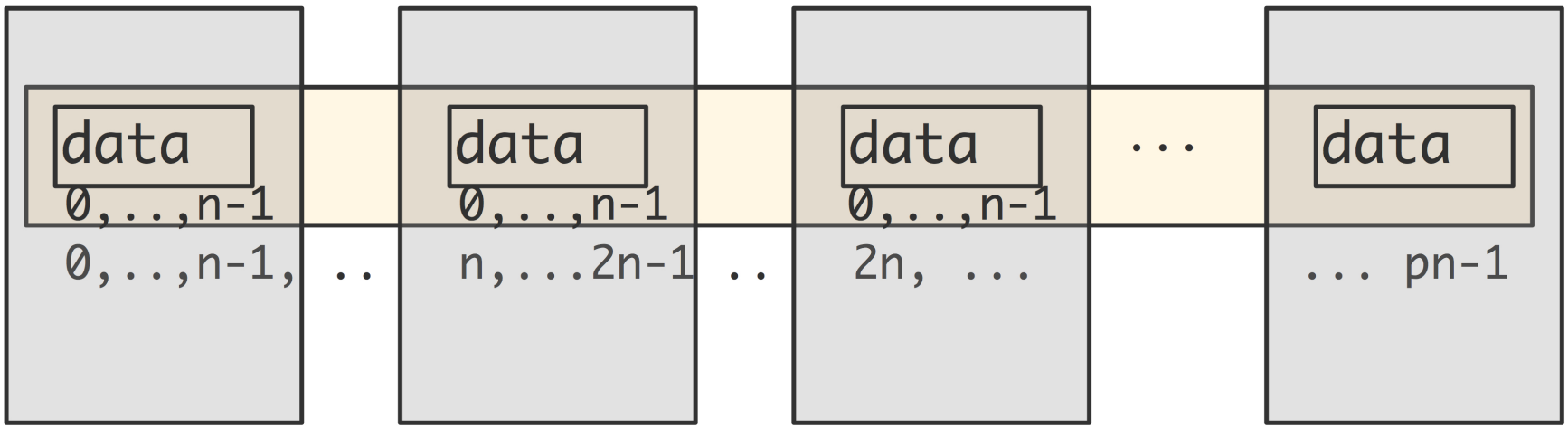
- `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Isend`, `MPI_Irecv`
- `MPI_Wait...`
- Mention of `MPI_Test`, `MPI_B/S/Rsend`.

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 13 (`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.)

Set a variable `N` for the total length of the array, and compute the local number of elements. Make sure you handle the case where N does not divide perfectly by the number of processes.

- Now allocate the local parts: each processor should allocate only local elements, not the whole vector.
  (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?

(Note that computer arithmetic is not exact: the computed sum will only be accurate up to some relative accuracy.)

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:

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

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 process 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 ...
```

Exercise (optional) 14 (`inprod`)

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

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

Table of Contents

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.

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
- So we need a point-to-point mechanism

Operating on distributed data

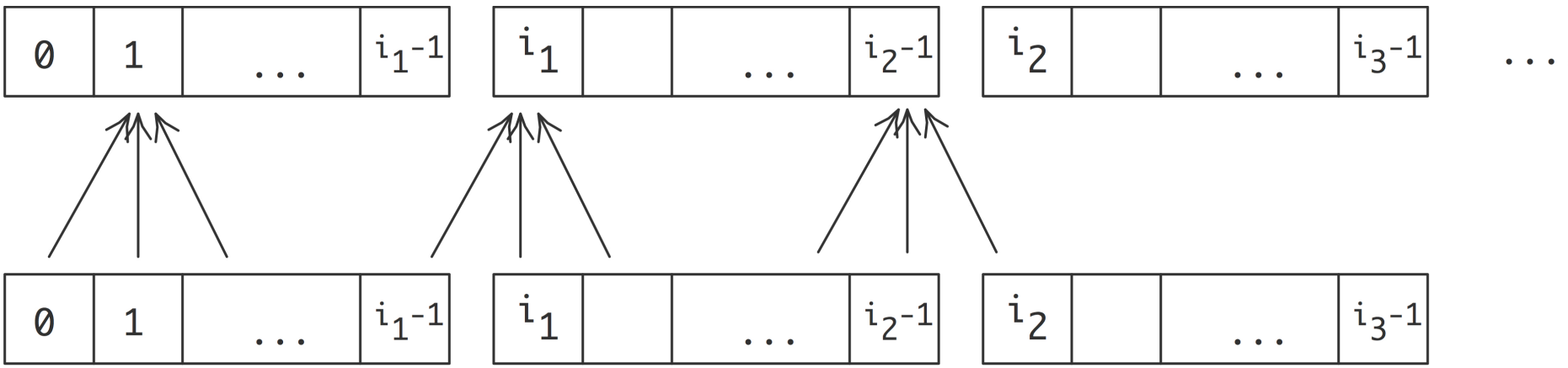
Array of numbers x_i: $i = 0, \ldots, N$

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`
- Receive buffer size is actually upper bound, not exact
- Use status object to retrieve actual description of the message
  use `MPI_STATUS_IGNORE` if the above does not apply

Exercise 15 (`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?

C:

```
double MPI_Wtime(void);
```

Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

Python:

```
MPI.Wtime()
```

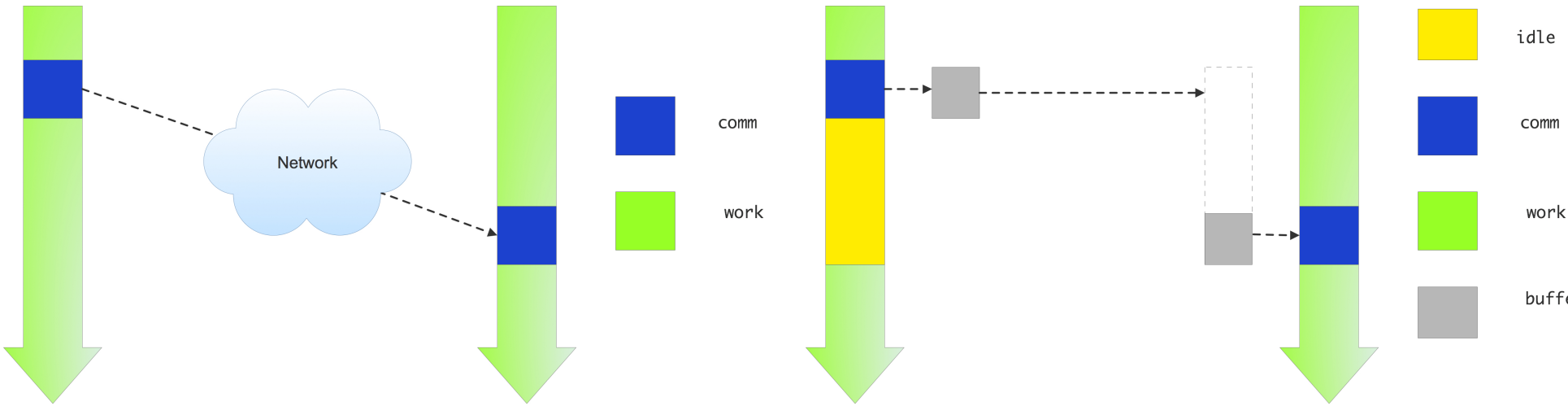
How to read routine prototypes: **??**.

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-procno; /* 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-procno; /* 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.
(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.

Exercise 16

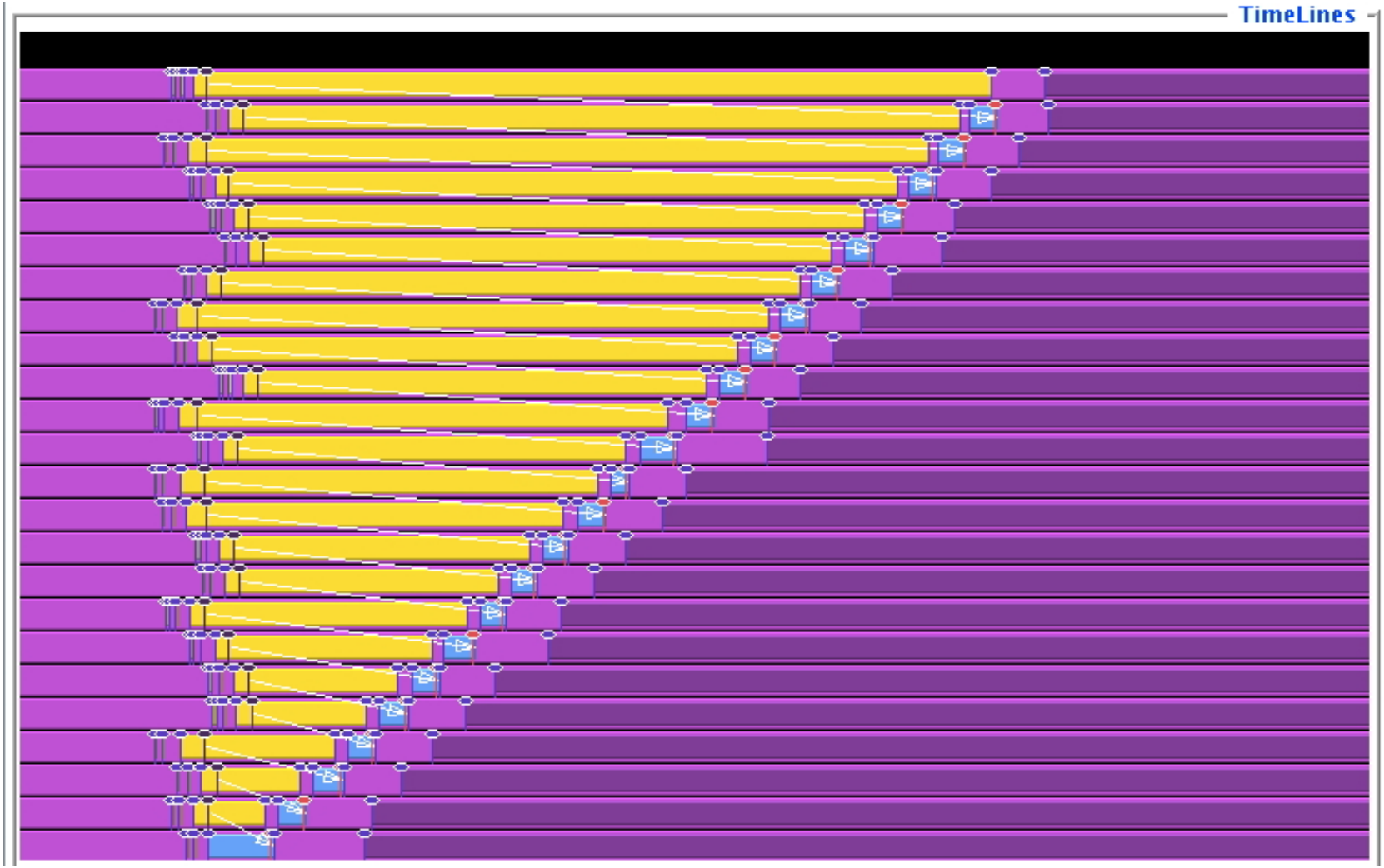
(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

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

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 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.

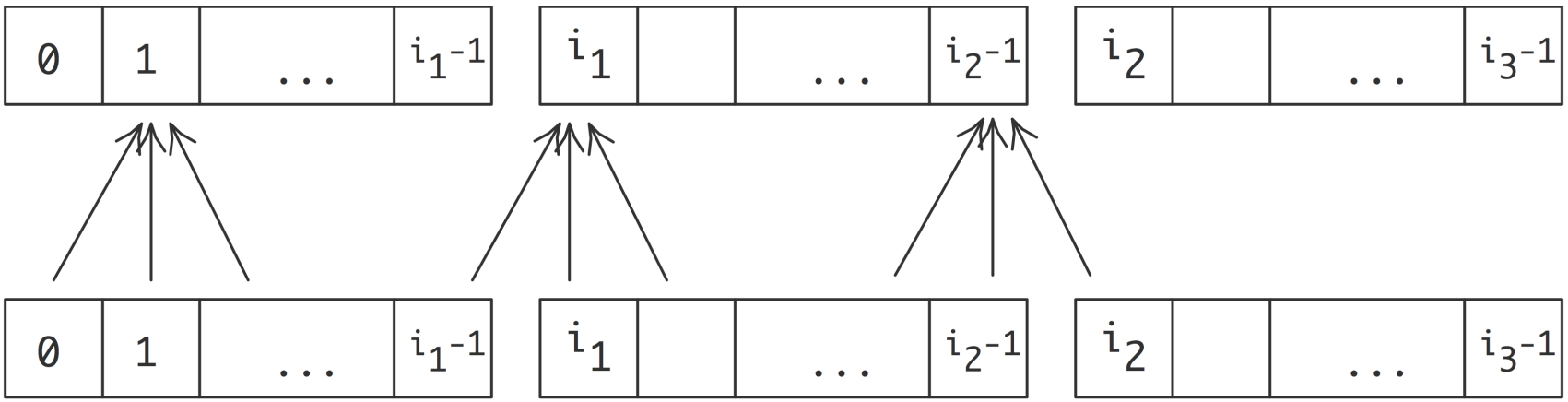
Beware! Blocking sends/receives can be trouble.
(How would you solve this particular case?)

Table of Contents

Operating on distributed data

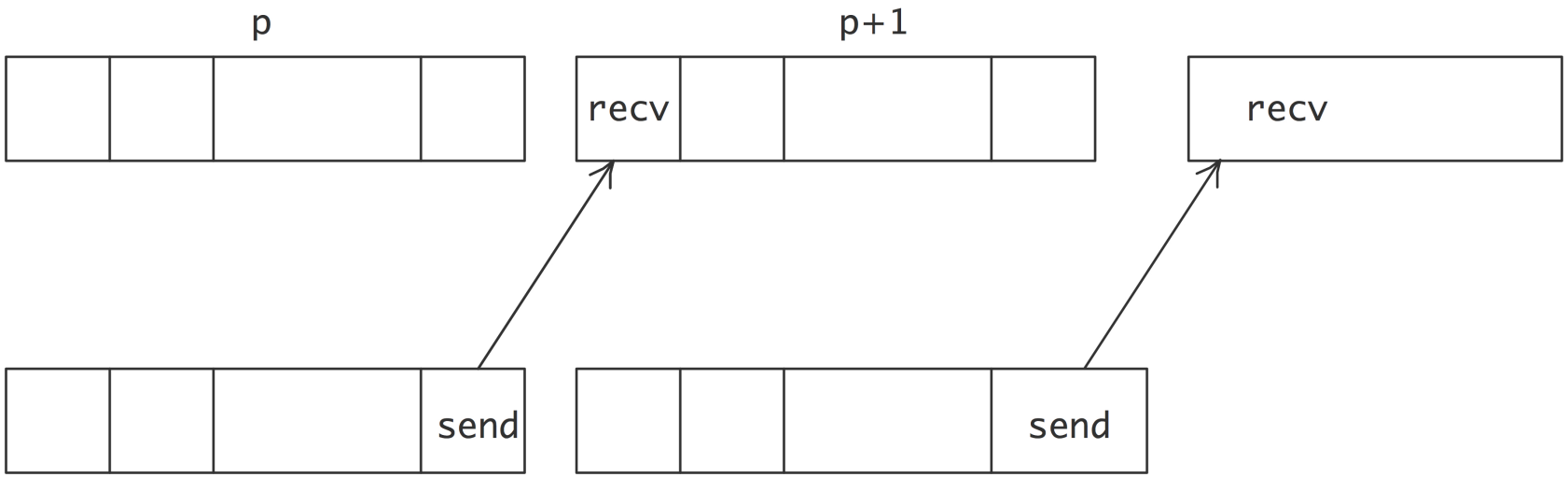
Take another look:

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



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

Not a good solution



First do all the data movement to the right.

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

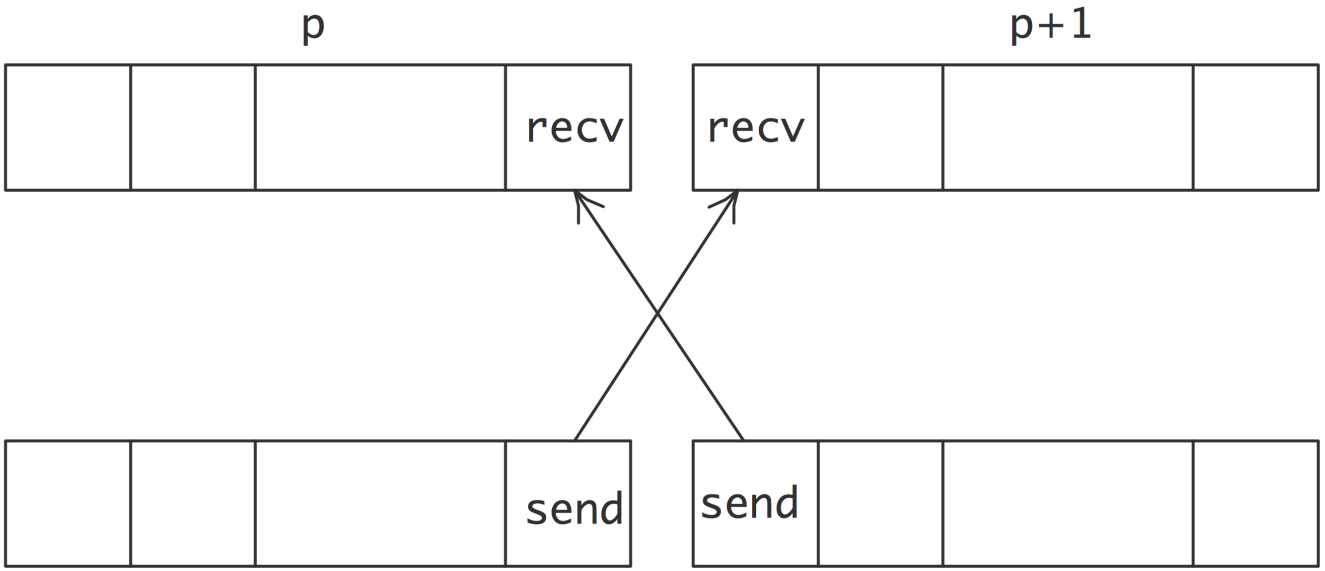
Sendrecv

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)
```

Pairwise exchange

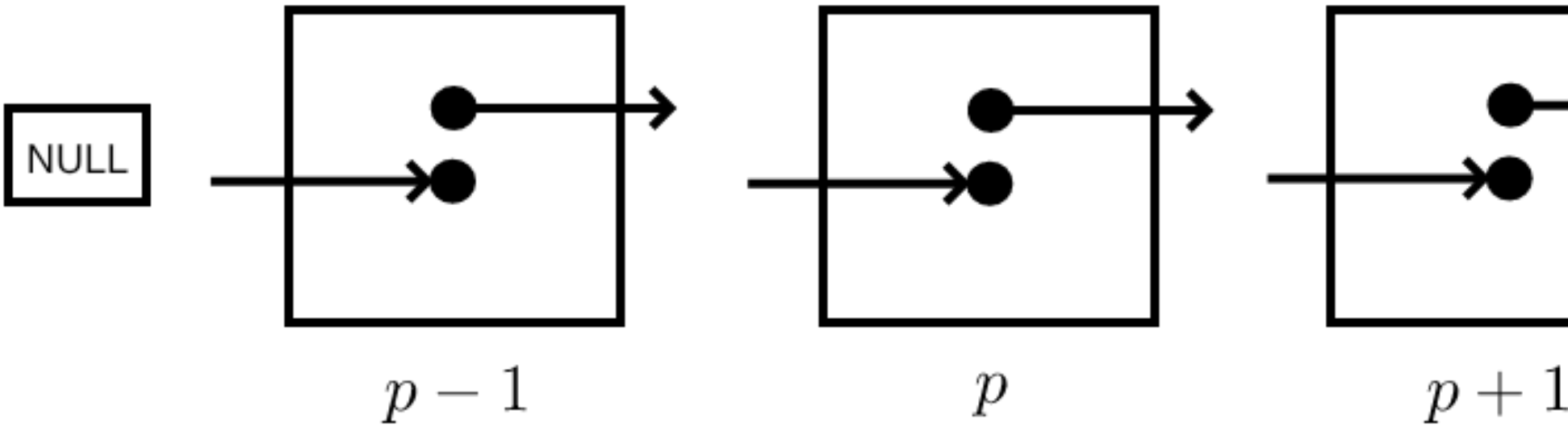


Each *p* sends to right, receives from right;
then same to the left. (Other possibilities possible.)

Sendrecv with incomplete pairs

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

In a picture



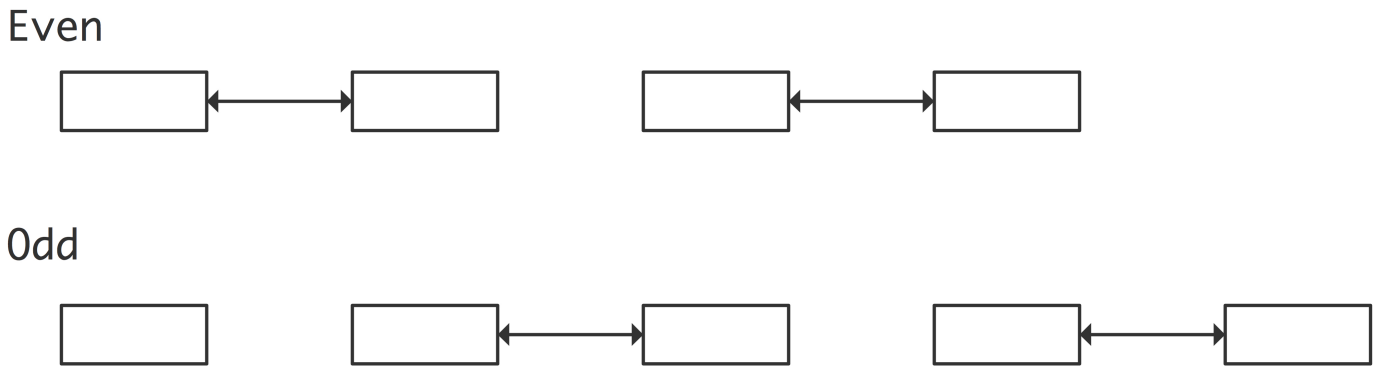
Exercise 17 (`sendrecv`)

Implement the above three-point combination 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 (optional) 18

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



The exchange sort algorithm is split in even and odd stages, 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.

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 print the global state of the distributed array at the beginning and end of the sorting proce

Table of Contents

Sending with irregular connections

Graph operations:

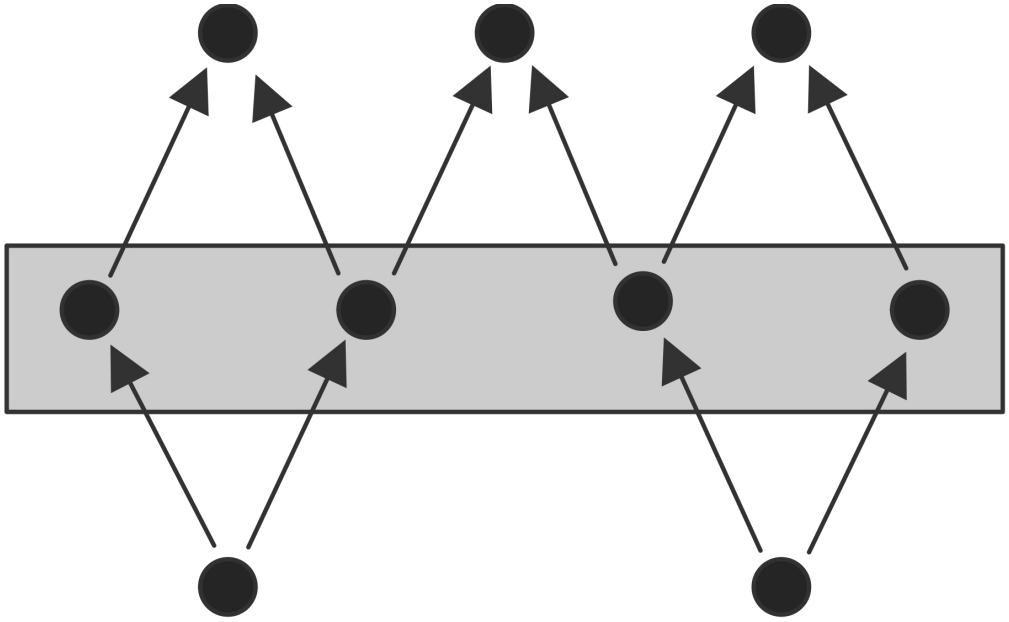
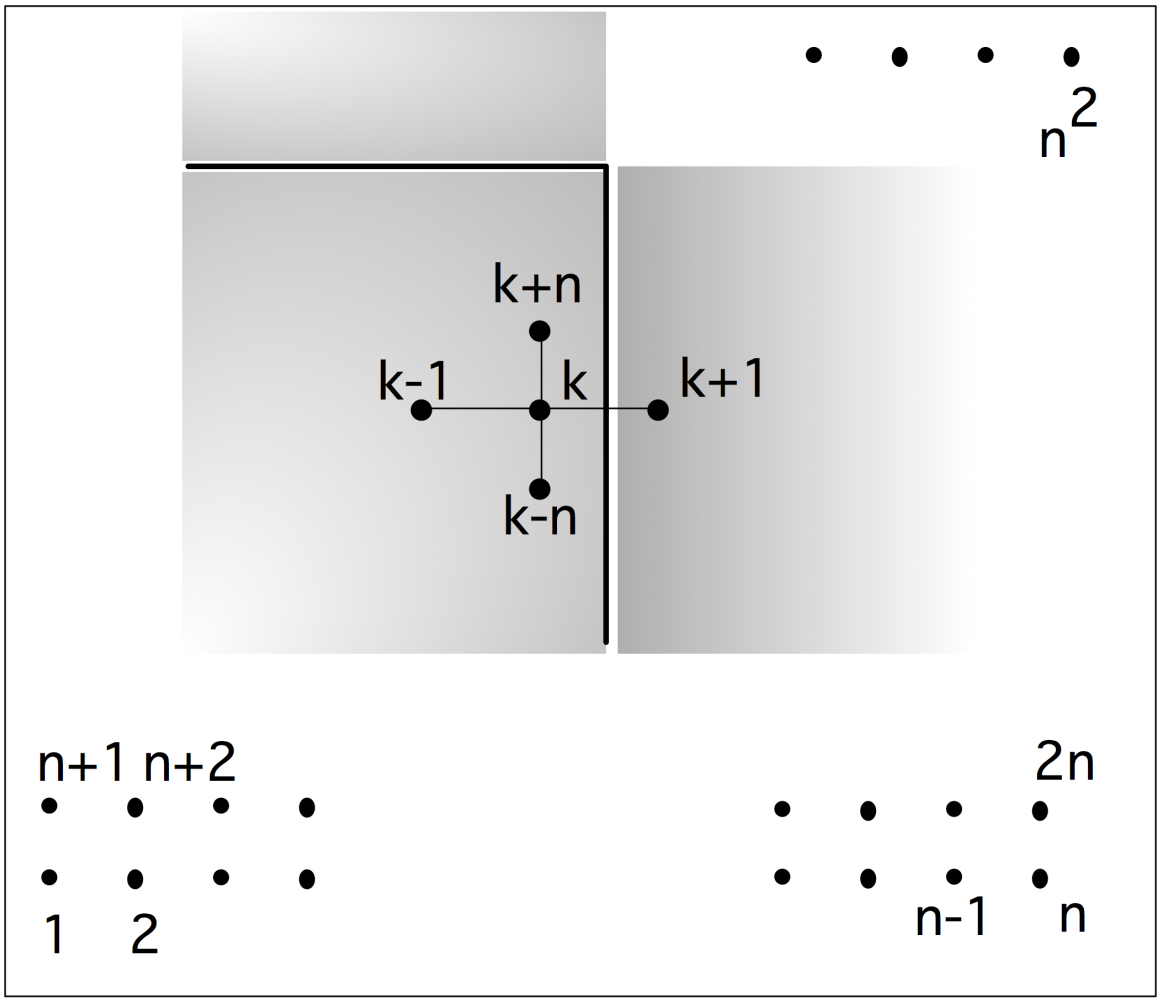


Table of Contents

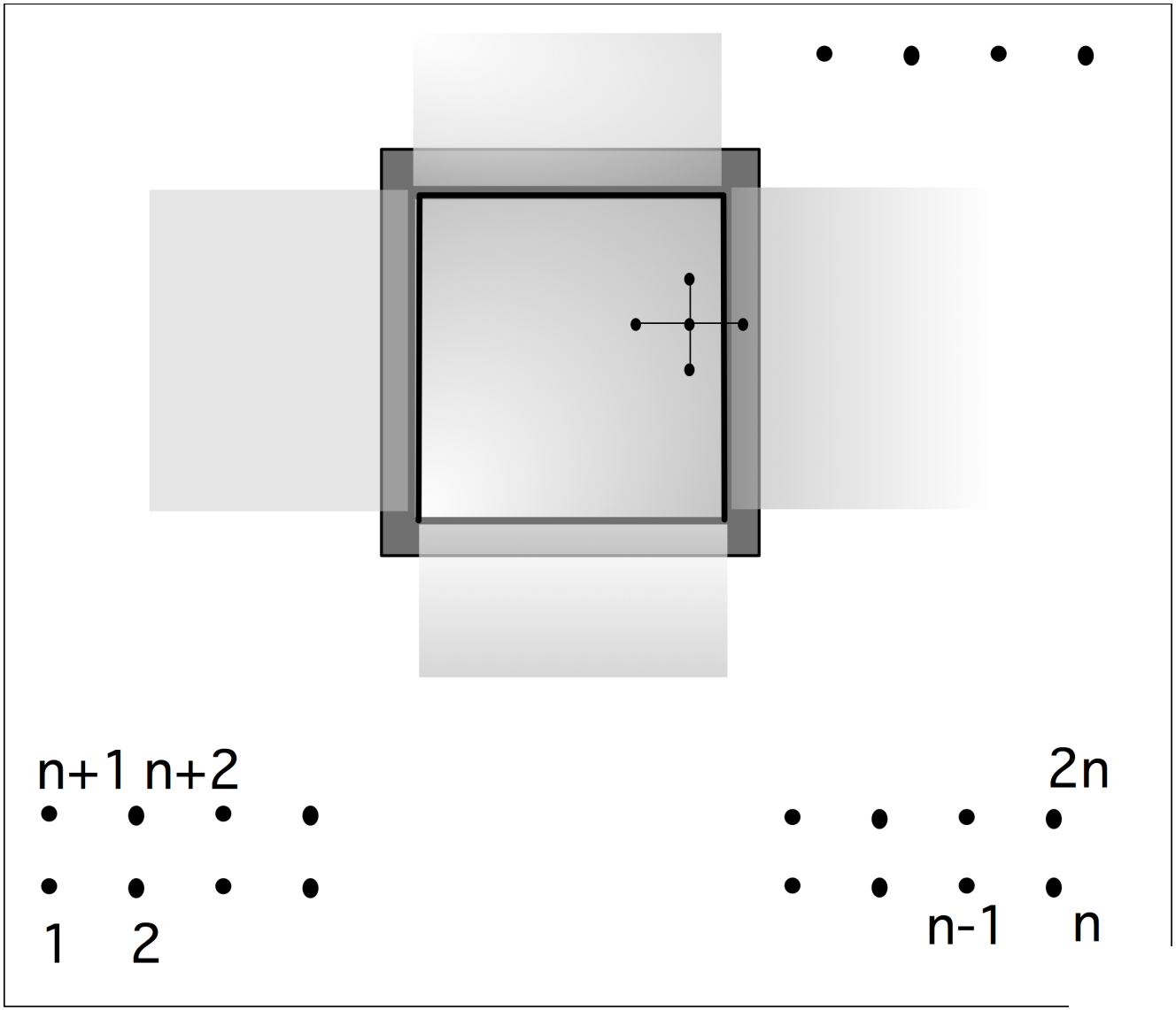
PDE, 2D case

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



Halo region

The halo region of a process, induced by a stencil



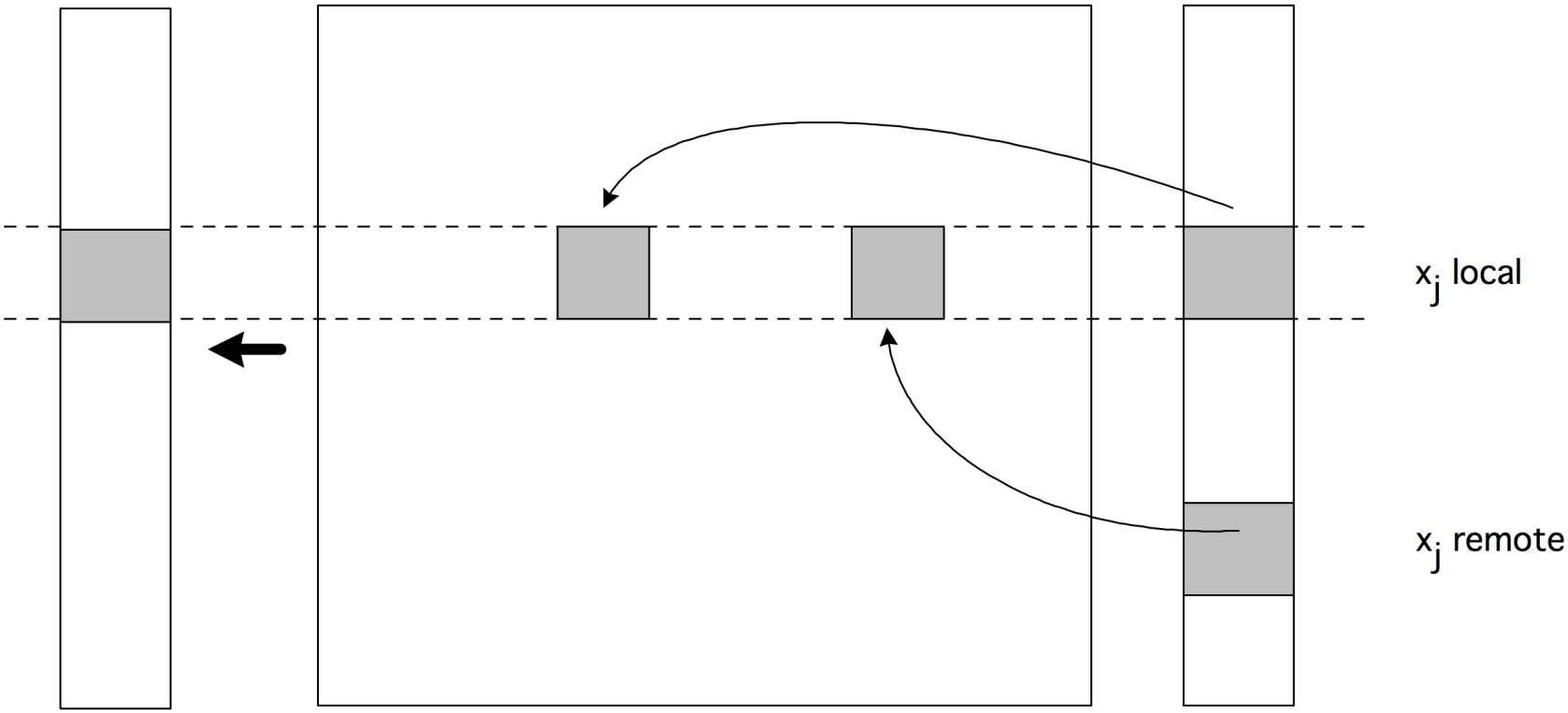
PDE matrix

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

Matrices in parallel

$$y \leftarrow Ax$$

and A, x, y all distributed:



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

```
// 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[])
```

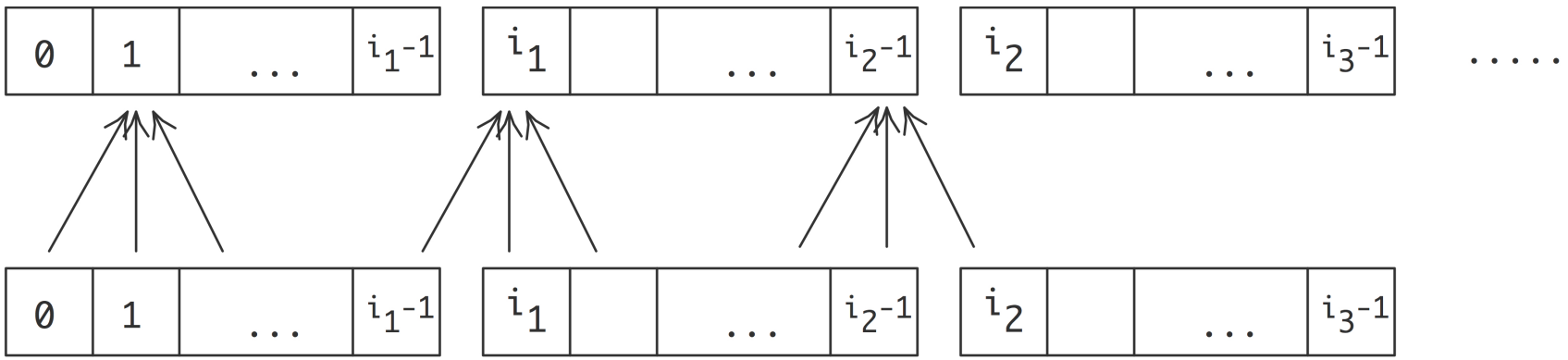
- ignore status: `MPI_STATUSES_IGNORE`
- also `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome`

Exercise 19 (`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. (Hint: use `MPI_PROC_NULL` at the ends.)



(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;
- 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 processes
4. Incorporate incoming data.

Exercise 20 (`isendirecvarray`)

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. (Hint: use `MPI_PROC_NULL` at the ends.)

Write your code so that it can achieve latency hiding.

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
- Persistent communication: repeated instance of same proc/data description.

too obscure to go into.

Advanced topics

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

One-sided communication

Overview

This section concernes one-sided operations, which allows ‘shared memory’ type programming.

Commands learned:

- `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
- Active target synchronization `MPI_Win_create`, `MPI_Win_fence`
- `MPI_Post/Wait/Start/Complete`
- Passive target synchronization `MPI_Win_lock/unlock`
- Atomic operations: `MPI_Fetch_and_op`

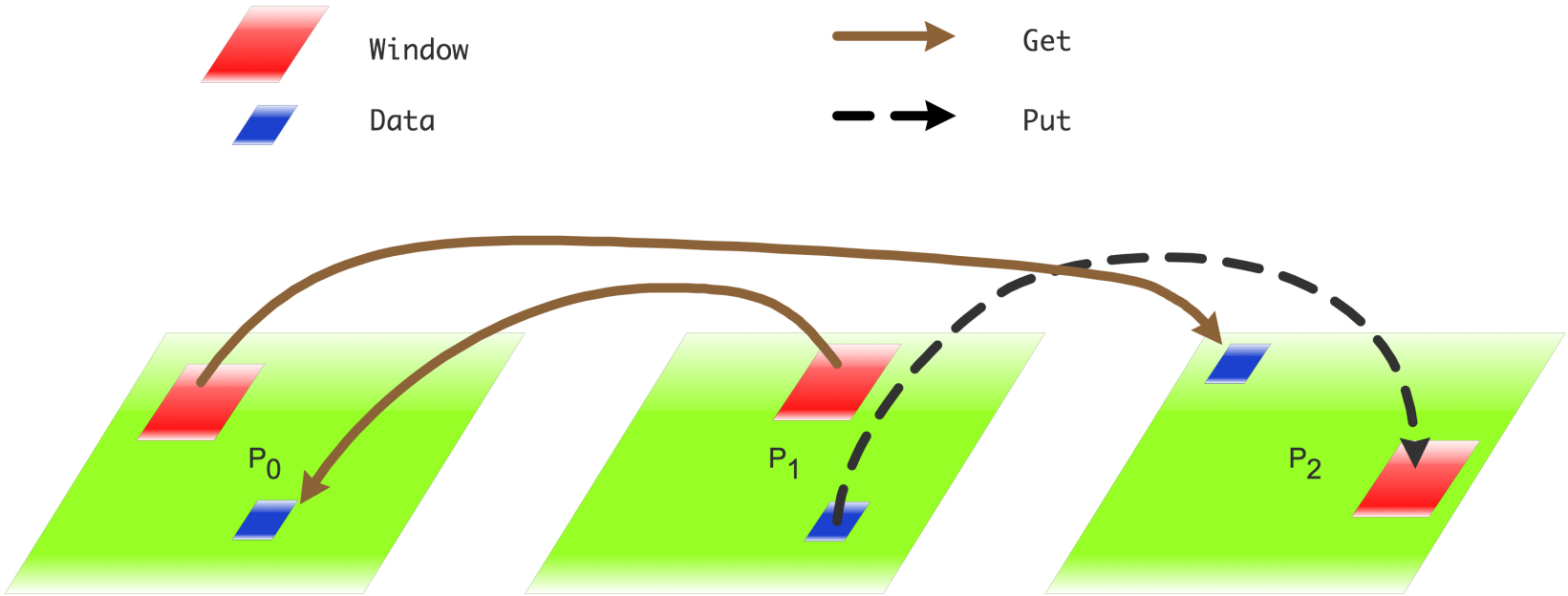
Table of Contents

Motivation

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

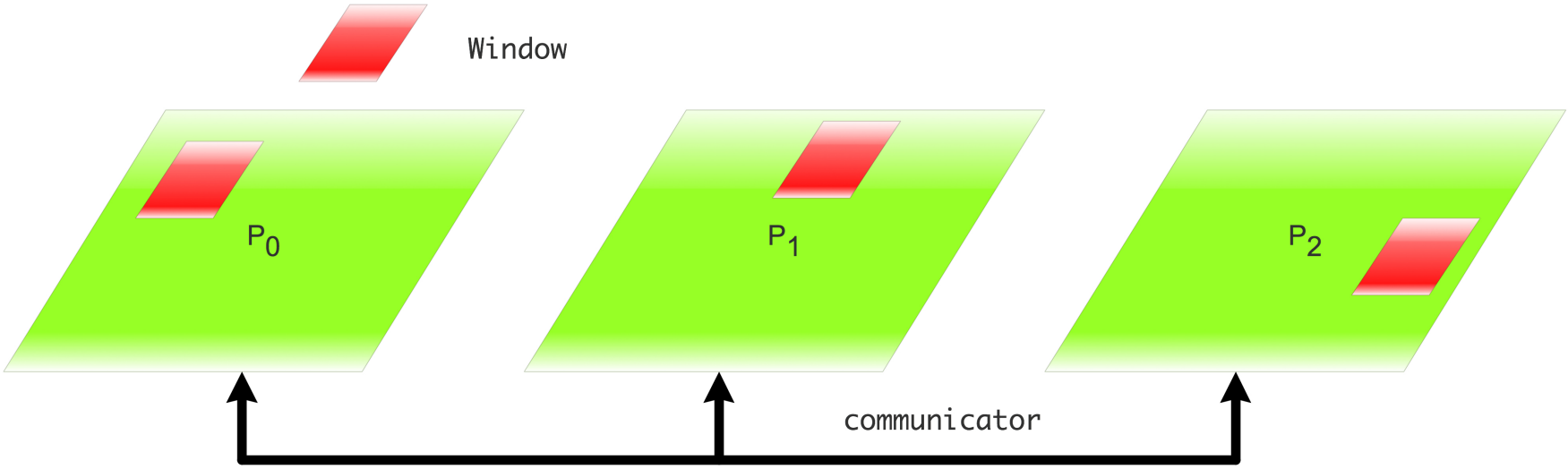
- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbours. Uncertain when this happens.
- Other irregular data structures: distributed hash tables.

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.

Window creation




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

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

Also call `MPI_Win_free` when done. This is important!

Window allocation

Instead of passing buffer, let MPI allocate:

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

Active target synchronization

All processes call `MPI_Win_fence`. Epoch is between fences:

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
if (procno==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.

```
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, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: ta
TYPE(MPI_Win), INTENT(IN) :: win
```

Exercise 21 (`randomput`)

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

Exercise (optional) 22 (`randomput`)

Replace `MPI_Win_create` by `MPI_Win_allocate`.

```
C:
int MPI_Accumulate
   (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_Op op, MPI_Win win)
int MPI_Raccumulate
   (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_Op op, MPI_Win win,MPI_Request *request)

Input Parameters

origin_addr : Initial address of buffer (choice).
origin_count : Number of entries in buffer (nonnegative integer).
origin_datatype : Data type of each buffer entry (handle).
target_rank : Rank of target (nonnegative integer).
target_disp : Displacement from start of window to beginning of target buffer
target_count : Number of entries in target buffer (nonnegative integer).
target_datatype : Data type of each entry in target buffer (handle).
op : Reduce operation (handle).
win : Window object (handle).

Output Parameter
```

Exercise (optional) 23 (`countdown`)

Implement a shared counter:

- One process maintains a counter;
- Iterate: all others at random moments update this counter.
- When the counter is zero, everyone stops iterating.

The problem here is data synchronization: does everyone see the counter the same way?

Atomic operations

Coherence problem in read/write:
result of `MPI_Get` is only known after the fence.

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

```
// passive.cxx
if (procno==repository) {
  // Repository processor creates a table of inputs
  // and associates that with the window
}
if (procno!=repository) {
  float contribution=(float)procno,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);
}
```

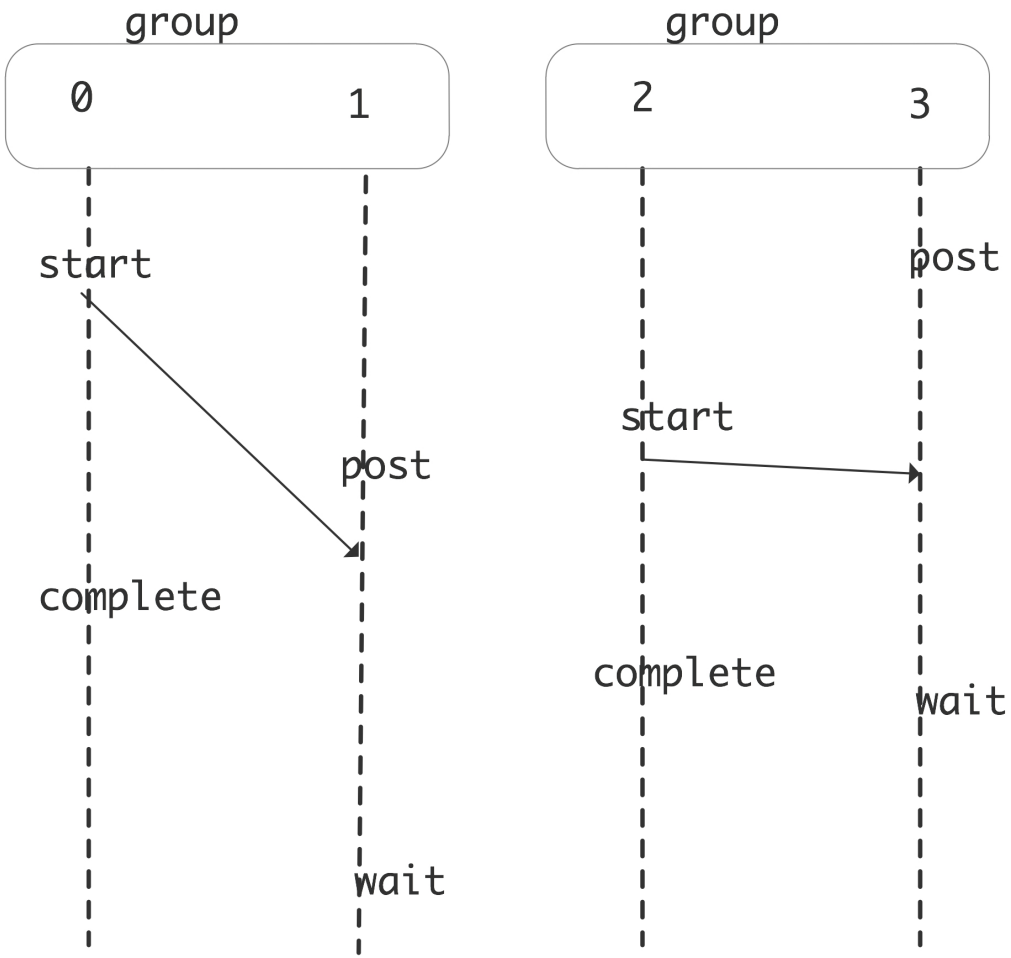
Exercise (optional) 24

Redo exercise **??** using `MPI_Fetch_and_op`. The problem is again to make sure all processes have the same view of the shared counter.

Does it work to make the fetch-and-op conditional? Is there a way to do it unconditionally? What should the 'break' test be, seeing that multiple processes can update the counter at the same time?

A second active synchronization

Use `Post,Wait,Start,Complete` calls



More fine-grained than fences.

Table of Contents

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)
```

Exercise 25 (`onesidedbuild`)

- Let each process have an empty array of sufficient length and a stack pointer that maintains the first free location.
- Now let each process randomly put data in a free location of another process' array.
- Use window locking. (Why is active target synchronization not possible?)

Table of Contents

Shared memory interface

- Use `MPI_Comm_split_type` to find processes on the same shared memory
- Use `MPI_Win_allocate_shared` to create a window between processes on the same shared memory
- Use `MPI_Win_shared_query` to get pointer to another process' window data.
- You can now use `memcpy` instead of `MPI_Put`.