MPI lecture and labs 3

Victor Eijkhout

2016

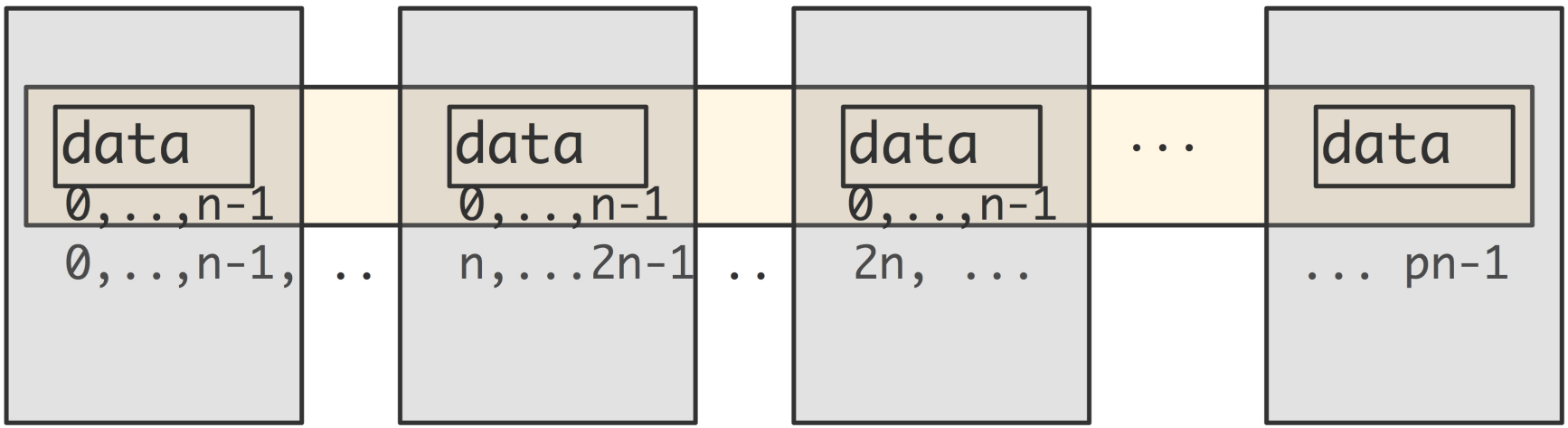
Point-to-point communication

Table of Contents

Distributed data

Distributed array: each process stores disjoint local part

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



Local numbering $0, \ldots, n_{\text{local}}$;
global numbering is 'in your mind'.

Local and global indexing

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

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

Exercise 1

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

Read in the global `N` parameter, and make sure that it is a multiple of the number P of processors. Your code should produce an error message and exit immediately if it doesn't.

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

To debug your program, first start with $N = P$.
(Did you allocate your array as real numbers? Why are integers not a good idea?)

Load balancing

If the distributed array is not perfectly divisible:

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

This gives a load balancing problem. Better solution?

Inner product calculation

Given vectors x, y:

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

Start out with a distributed vector.

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

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

Exercise 2

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.

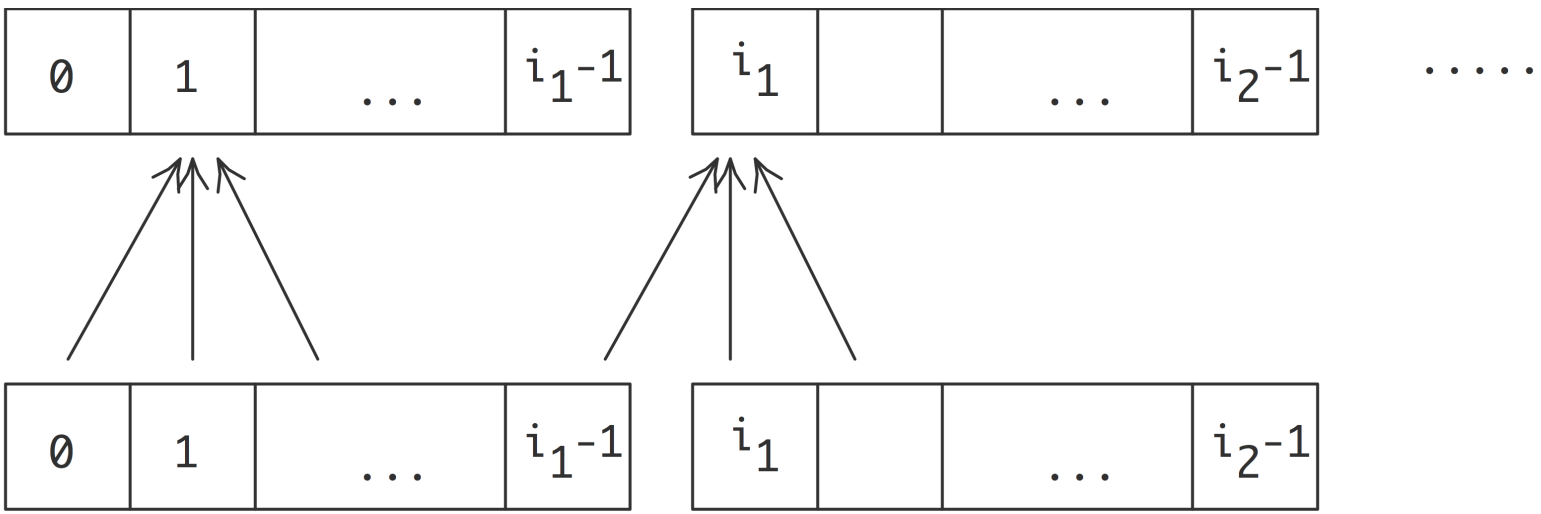
Table of Contents

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

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

Ping-pong

A sends to B, B sends back to A

What is the code for A? For B?

Ping-pong

A sends to B, B sends back to A

Process A executes the code

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

Process B executes

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

Ping-pong in MPI

Remember SPMD:

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

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

Semantics:
IN buf: initial address of send buffer (choice)
IN count: number of elements in send buffer (non-negative integer)
IN datatype: datatype of each send buffer element (handle)
IN dest: rank of destination (integer)
IN tag: message tag (integer)
IN comm: communicator (handle)

Fortran:
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Python native:
```

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

Semantics:
OUT buf: initial address of receive buffer (choice)
IN count: number of elements in receive buffer (non-negative integer)
IN datatype: datatype of each receive buffer element (handle)
IN source: rank of source or MPI_ANY_SOURCE (integer)
IN tag: message tag or MPI_ANY_TAG (integer)
IN comm: communicator (handle)
OUT status: status object (Status)

Fortran:
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
TYPE(*), DIMENSION(..) :: buf
INTEGER, INTENT(IN) :: count, source, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Status object

- Receive call can have various wildcards
- `MPI_ANY_SOURCE, MPI_ANY_TAG`
- use status object to retrieve actual description of the message

Exercise 3

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

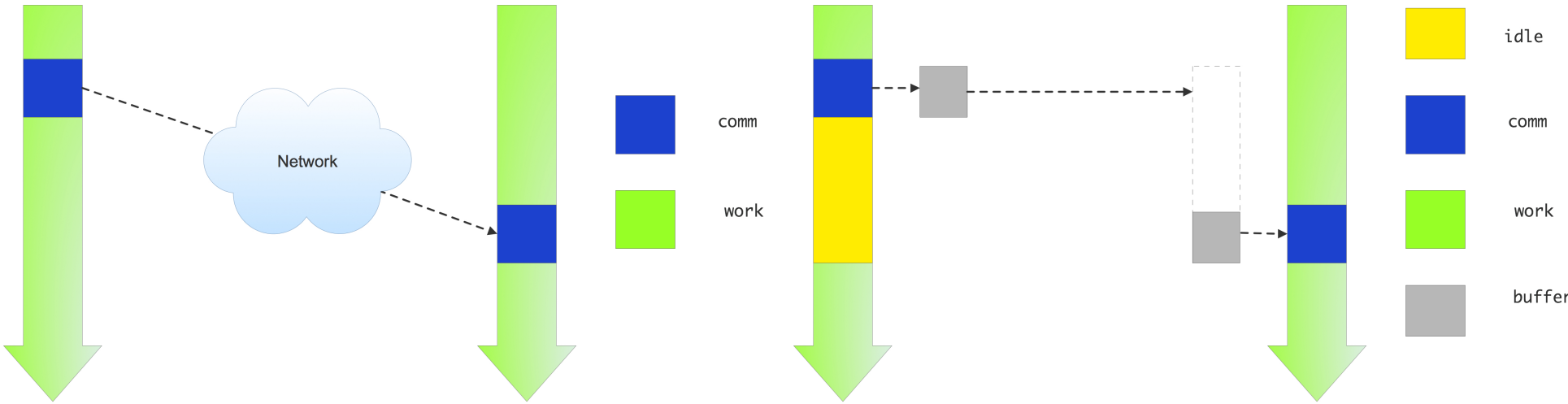
Then modify the program to use longer messages. How does the timing increase with message size?

Table of Contents

Blocking send/recv

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

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



Ideal vs actual send/recv behaviour.

Deadlock

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

A subtlety.
This code may actually work:

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

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

Protocol

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

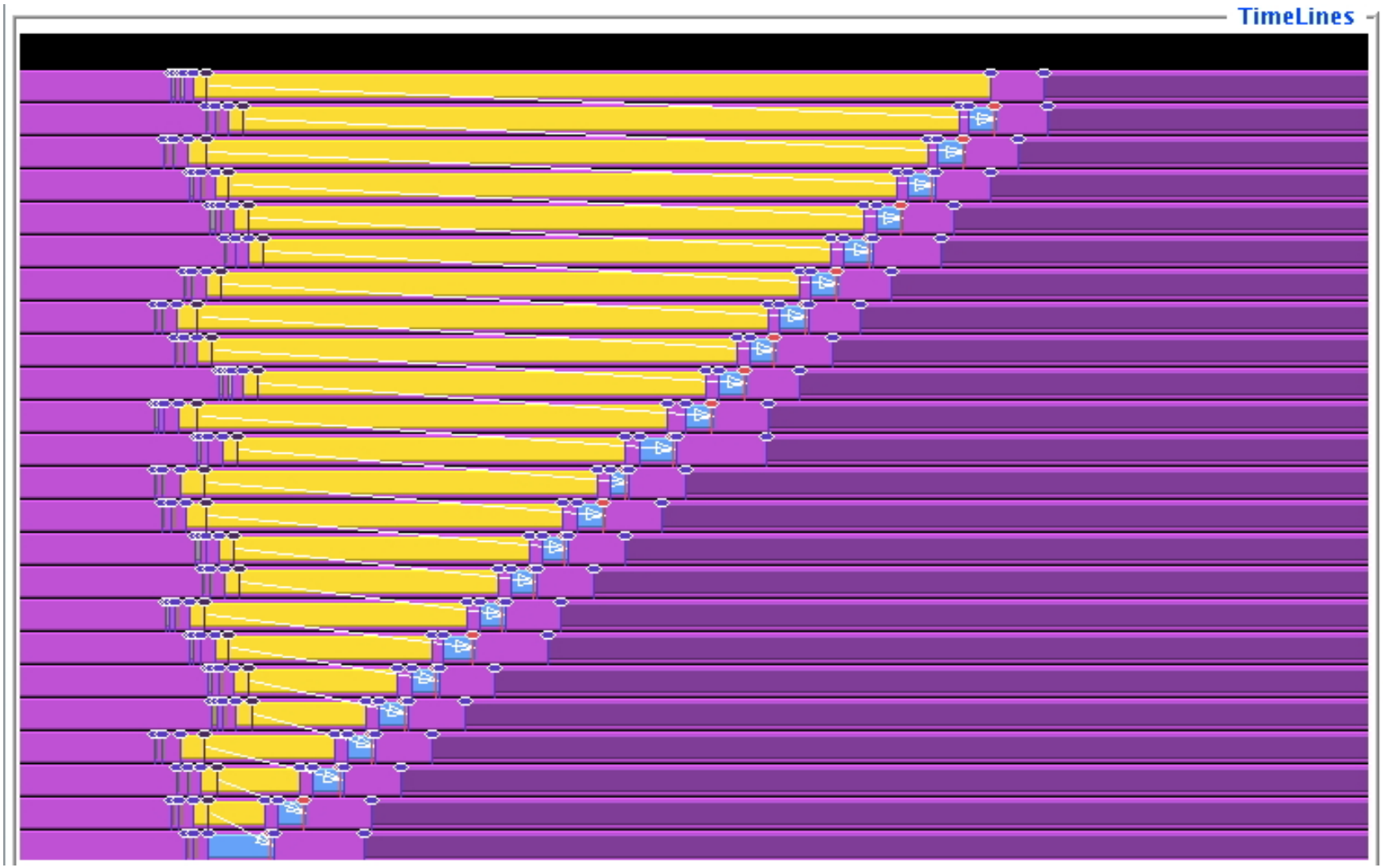
- Sender: ‘I have data for you’
- Receiver: ‘I have a buffer ready, send it over’
- Sender: ‘Ok, here it comes’
- Receiver: ‘Got it.’

Exercise 4

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

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

TAU trace: serialization



The problem here. . .

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

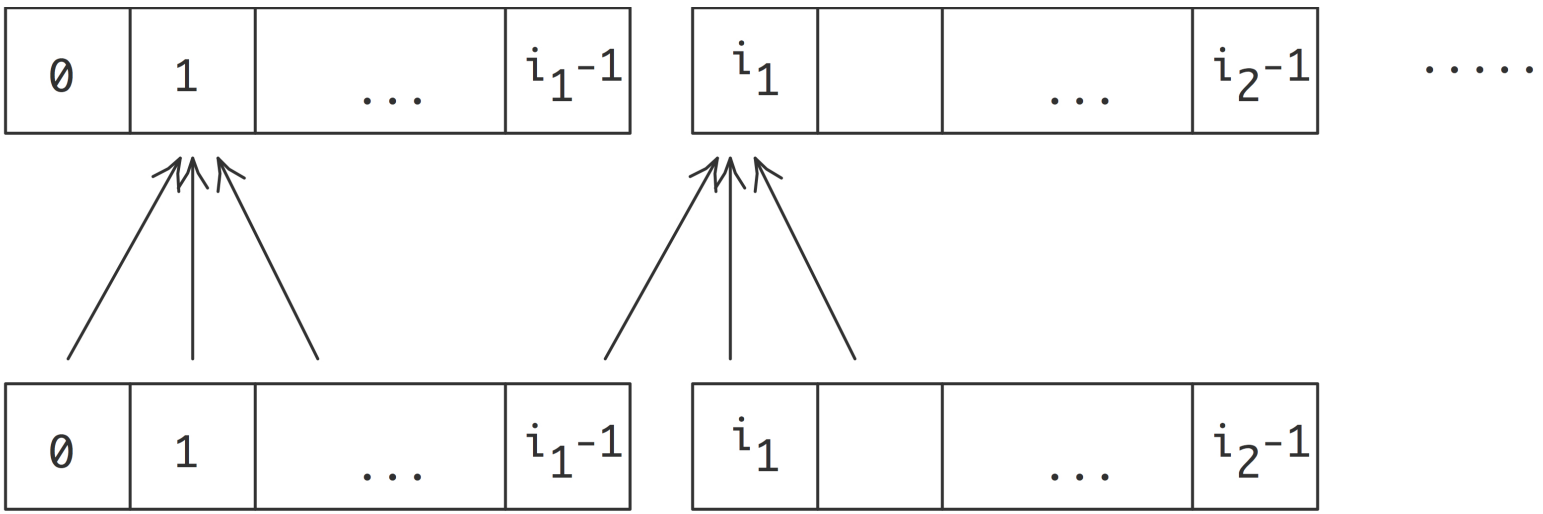
Beware! Blocking sends/receives can be trouble.
(How would you solve this particular case?)

Table of Contents

Operating on distributed data

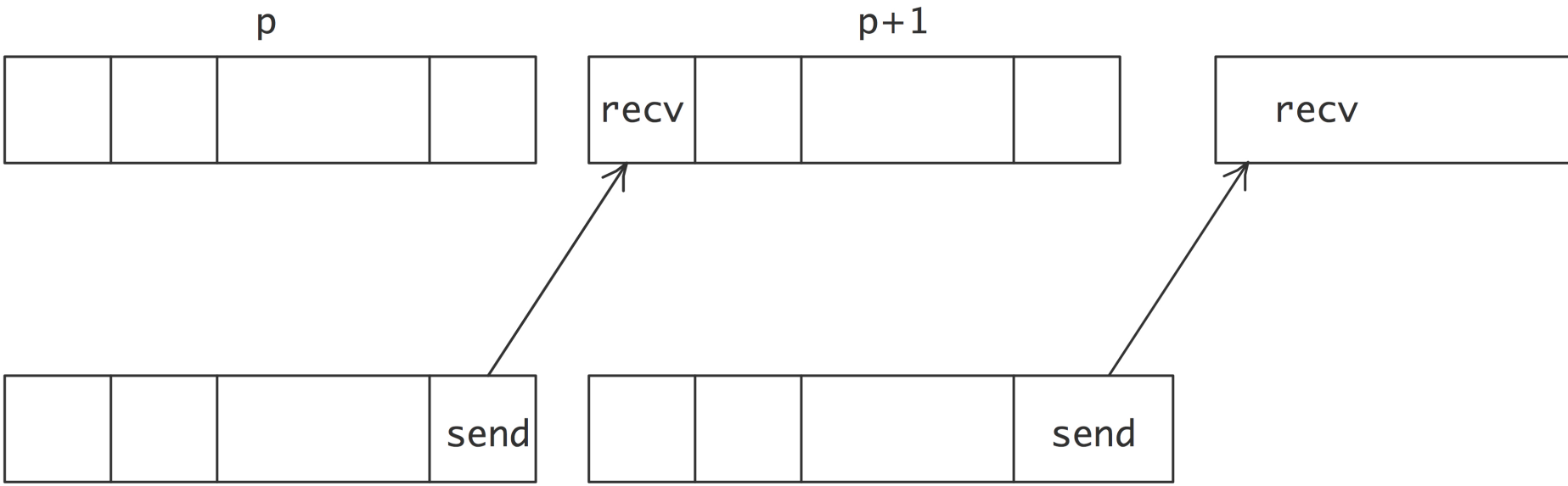
Take another look:

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



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

Not a good solution



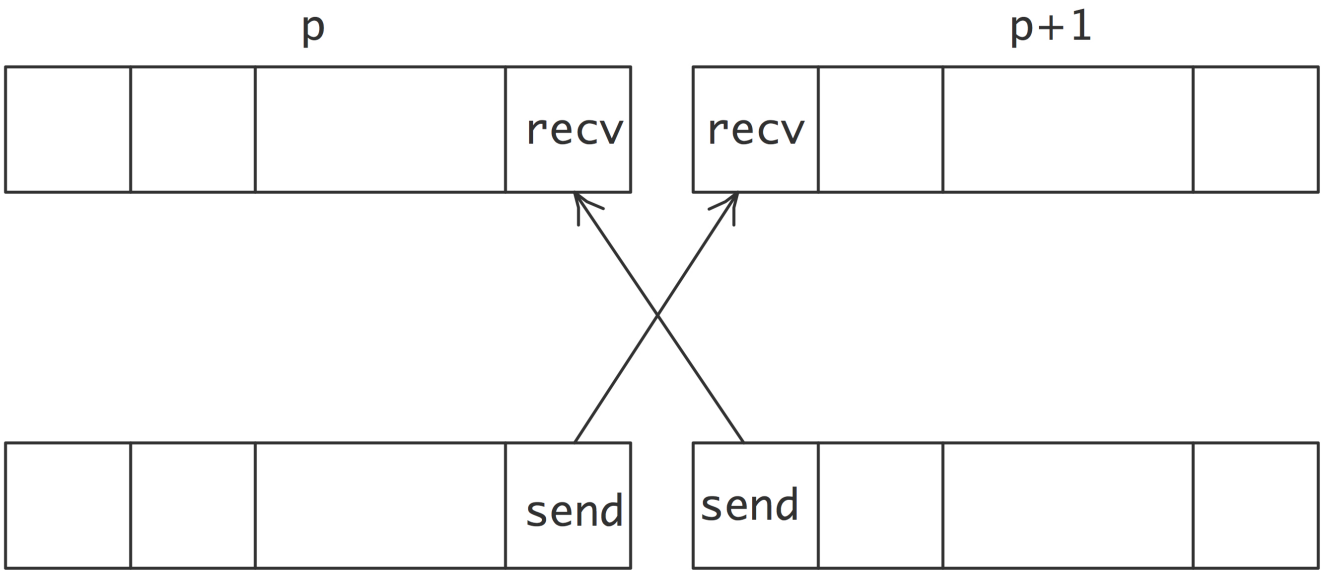
- Each arrow is a send and receive
- So maybe do first all the right arrows?
- Why is that a bad idea?

Sendrecv

Instead of separate send and receive: use

```
int MPI_Sendrecv(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
        int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int source, int recvtag,
  MPI_Comm comm, MPI_Status *status)
```

Pairwise exchange



Sendrecv with incomplete pairs

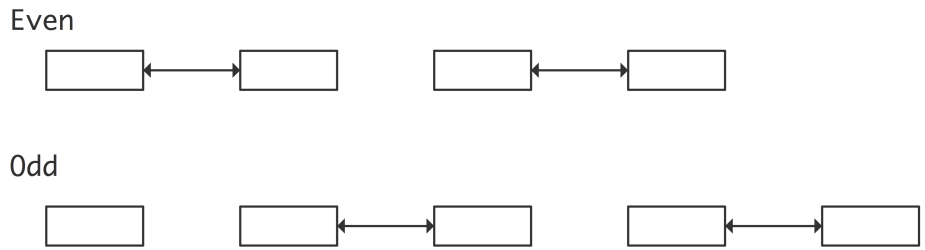
```
MPI_Comm_rank( .... &mytid );
if ( /* I am not the first processor */ )
  predecessor = mytid-1;
else
  predecessor = MPI_PROC_NULL;
if ( /* I am not the last processor */ )
  successor = mytid+1;
else
  successor = MPI_PROC_NULL;
sendrecv(from=predecessor,to=successor);
```

Exercise 5

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

Exercise 6

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



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

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

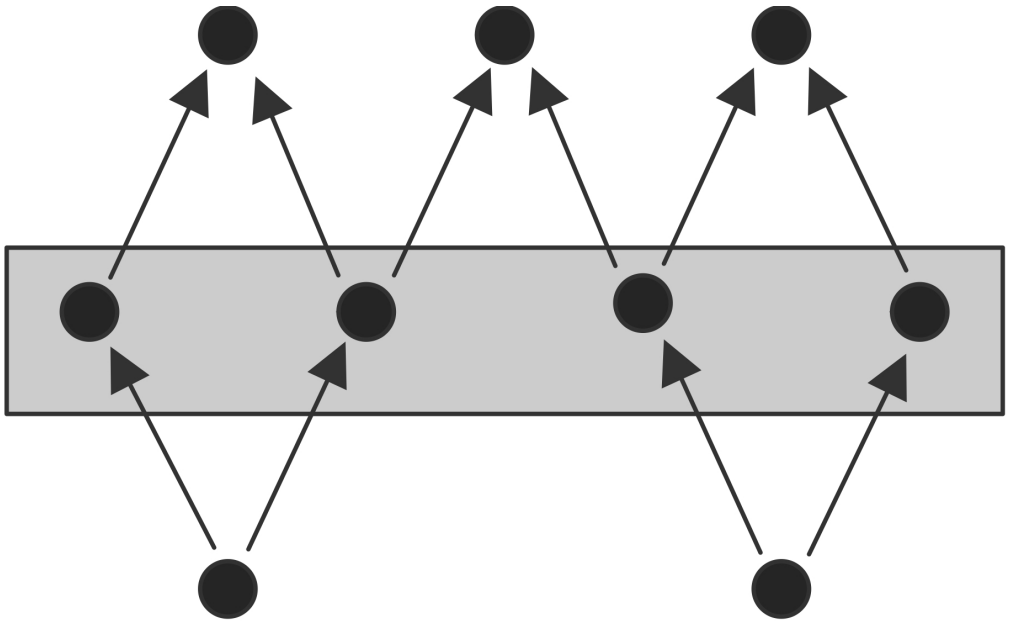
You need to repeat this $P/2$ times, where P is the number of processors.

Use `MPI_PROC_NULL` for the edge cases.

Table of Contents

Sending with irregular connections

Graph operations:



How do you approach this?

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

Instead:

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

Non-blocking send/recv

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

Syntax

Very much like blocking ▸ send and ▸ recv:

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

The `MPI_Request` can be tested:

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

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

Exercise 7

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

Comparison

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

Buffer use in blocking/non-blocking case

Blocking:

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

Non-blocking:

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

Latency hiding

Other motivation for non-blocking calls:
overlap of computation and communication, provided hardware support.

Also known as ‘latency hiding’.

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

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

Test: non-blocking wait

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

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

More sends and receive

- `MPI_Bsend`, `MPI_Ibsend`: buffered send
- `MPI_Ssend`, `MPI_Issend`: synchronous send
- `MPI_Rsend`, `MPI_Irsend`: ready send

too obscure to go into.