Point-to-point communication

Victor Eijkhout
Jack Gaither

2026 PCSE

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:

```
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 ..... );
}
```

Sample send and recv calls

```
double x[10],y[10];
MPI_Send( x,10,MPI_DOUBLE, tgt,0,comm );
MPI_Status status;
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 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)
	)			

```
template<typename T >
void mpl::communicator::send
   ( const T scalar&,int dest,tag = tag(0) ) const
   ( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
   ( iterT begin,iterT end,int dest,tag = tag(0) ) const
T : scalar type
begin : begin iterator
end : end iterator
```

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 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)
	)			

```
template<typename T >
status mpl::communicator::recv
   ( T &,int,tag = tag(0) ) const inline
   ( T *,const layout< T > &,int,tag = tag(0) ) const
   ( iterT  begin,iterT  end,int  source, tag  t = tag(0) ) const
```

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

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

Exercise 1 `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?

MPI_Wtime

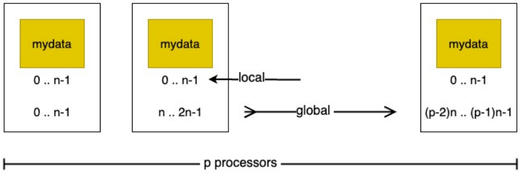
Name	Param name	Explanation	C type	F type
MPI_Wtime (				
)				

```
double mpl::environment::wtime() const;
double mpl::environment::wtick() const;
```

Distributed data

Distributed data

Distributed array: each process stores disjoint local part



Local numbering $0, \ldots, n_{\mathrm{local}}$;
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 (optional) 2

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:

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

Simple case:

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

Finite difference approximation:

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

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

Motivation (continued)

Equations

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

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

So we are interested in sparse/banded matrices.

Matrix vector product

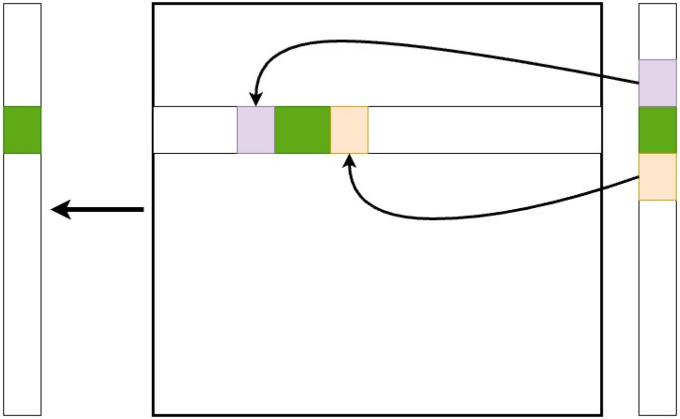
Most common operation: matrix vector product

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

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

Partitioned matrix-vector product

We need a point-to-point mechanism:



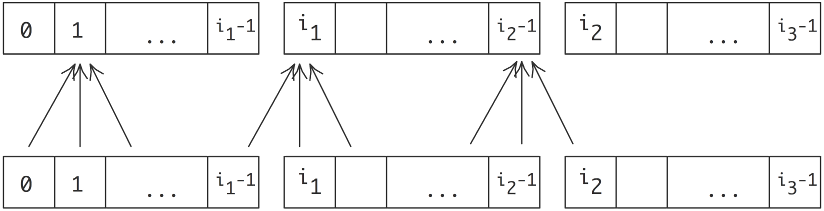
each process with ones before/after it.

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:



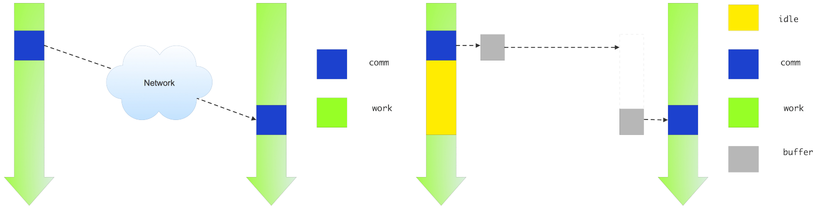
so we need a point-to-point mechanism.

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:

```
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: ‘eager’ send.
Definition of ‘small message’ controlled by environment variables:
`I_MPI_EAGER_THRESHOLD` `MV2_IBA_EAGER_THRESHOLD`

Exercise 3

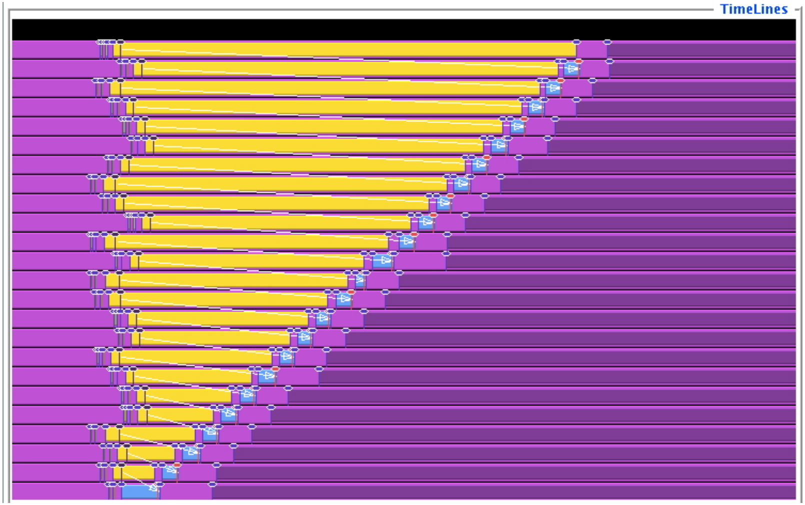
(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

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 4 `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 30. 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 / 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)
	)			

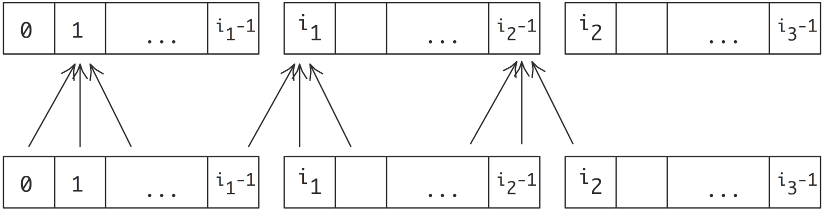
Missing MPL proto: mpi˙ssend

Pairwise exchange

Operating on distributed data

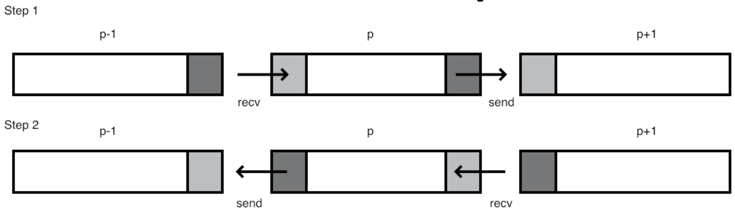
Take another look:

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



- 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

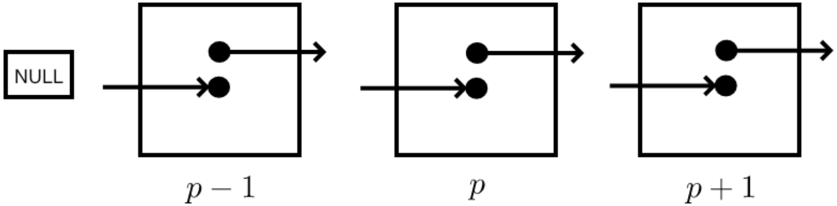
Name	Param name	Explanation	C type	F type
MPI_Sendrecv (				
MPI_Sendrecv_c (				
	sendbuf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)
	sendcount	number of elements in send buffer	[int / MPI_Count	INTEGER
	sendtype	type of elements in send buffer	MPI_Datatype	TYPE(MPI_Datatype)
	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 / 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)
	)			

MPL: MPI_Sendrecv

```
template<typename T >
status mpl::communicator::sendrecv
   ( const T & senddata, int dest,   tag sendtag,
           T & recvdata, int source, tag recvtag
   ) const
   ( const T * senddata, const layout< T > & sendl, int dest,   tag sendtag,
           T * recvdata, const layout< T > & recvl, int source, tag recvtag
   ) const
   ( iterT1 begin1, iterT1 end1, int dest,   tag sendtag,
     iterT2 begin2, iterT2 end2, int source, tag recvtag
   ) const
```

SPMD picture

What does process *p* do?



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);
```

(Receive from `MPI_PROC_NULL` succeeds without altering the receive buffer.)
Use `mpl::proc_null`.

A point of programming style

The previous slide had:

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

Also possible:

```
if ( /* i am first */ )
  Sendrecv( to=right, from=NULL );
else if ( /* i am last */
  Sendrecv( to=NULL,  from=left );
else
  Sendrecv( to=right, from=left );
```

```
if ( /* i am first */ )
  Send( to=right );
else if ( /* i am last */
  Recv( from=left );
else
  Sendrecv( to=right, from=left );
```

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

Exercise (optional) 5 (`rightsend`)

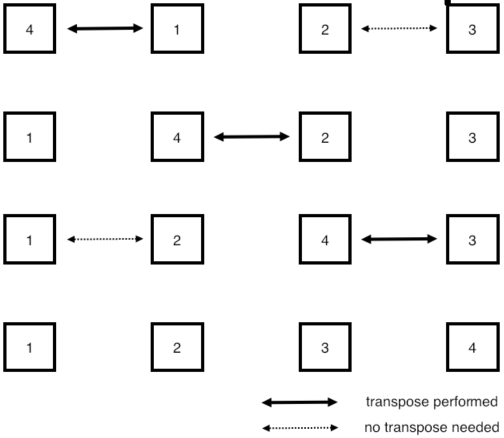
Revisit exercise 3 and solve it using *`MPI_Sendrecv`*.

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 6 `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) 7

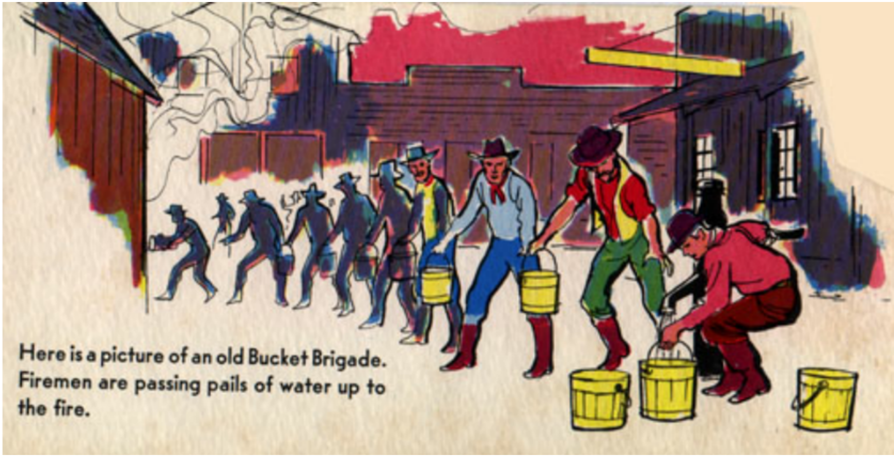
A very simple sorting algorithm is *swap sort* or *odd-even transposition sort*: pairs of processors compare data, and if necessary exchange. The elementary step is called a *compare-and-swap*: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the 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 46.

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'


Here is a picture of an old Bucket Brigade.
Firemen are passing pails of water up to the fire.

Exercise 8 `bucketblock`

Take the code of exercise 4 and modify it so that the data from process zero gets propagated to every process. Specifically, compute all partial sums $\sum_{i=0}^{p} i^2$:

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

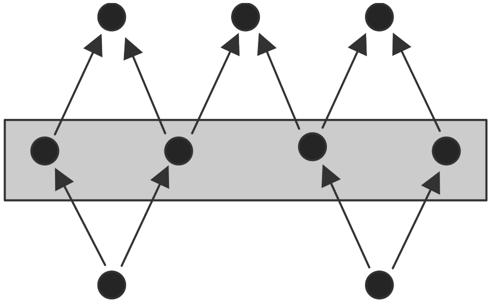
Use `MPI_Send` and `MPI_Recv`; make sure to get the order right.

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

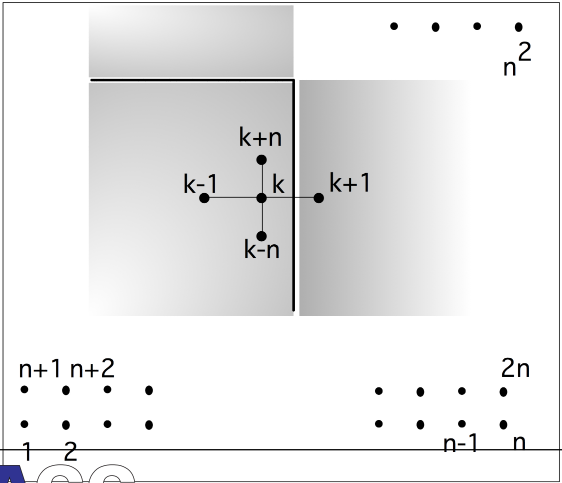
Graph operations:



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 ⇒ complicated to express pairwise

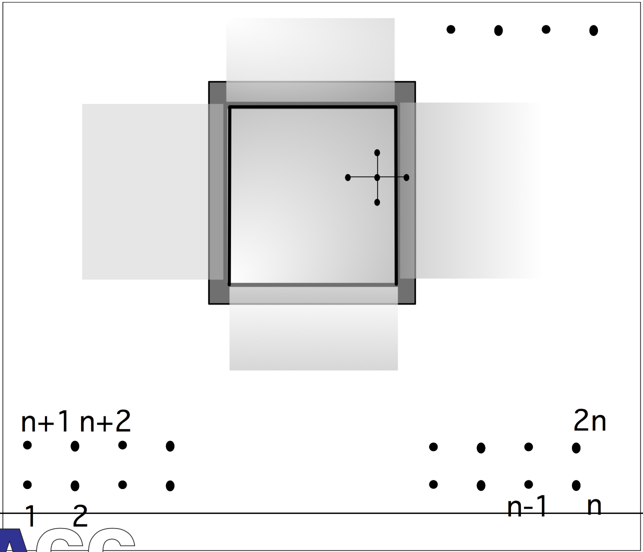


PDE matrix

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

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.

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

MPL: MPI_Isend

```
template<typename T >
irequest mpl::communicator::isend
   ( const T & data, int dest, tag t = tag(0) ) const;
   ( const T * data, const layout< T > &  l, int  dest, tag  t = tag(0) ) const;
   ( iterT  begin, iterT  end, int  dest, tag  t = tag(0) ) const;
```

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 / 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)
	)			

MPL: MPI_Irecv

```
template<typename T >
irequest mpl::communicator::irecv
   ( const T & data, int src, tag t = tag(0) ) const;
   ( const T * data, const layout< T > &  l, int  src, tag  t = tag(0) ) const;
   ( iterT  begin, iterT  end, int  src, tag  t = tag(0) ) const;
```

Request waiting

Basic wait:

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

Most common way of waiting for completion:

```
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`

MPL request object

Nonblocking routines have an `irequest` as function result. Note: not a parameter passed by reference, as in the C interface. The various wait calls are methods of the `irequest` class.

```
double recv_data;
mpl::irequest recv_request =
  comm_world.irecv( recv_data,sender );
recv_request.wait();
```

You can not default-construct the request variable:

```
// DOES NOT COMPILE:
mpl::irequest recv_request;
recv_request = comm.irecv( ... );
```

This means that the normal sequence of first declaring, and then filling in, the request variable is not possible.

Implementation note: The wait call always returns a `status_t` object; not assigning it means that the destructor is called on it.

Request pools

Instead of an array of requests, use an *irequest_pool* object, which acts like a vector of requests, meaning that you can *push* onto it.

```
// irecvsource.cxx
mpl::irequest_pool recv_requests;
for (int p=0; p<nprocs-1; p++) {
  recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
}
```

Commands such as *waitall* are methods on the pool; they don't return statuses; statuses are found by indexing:

```
// irecvall.cxx
recv_requests.waitall();
auto status_last = recv_requests.get_status( recv_requests.size()-1 );
```

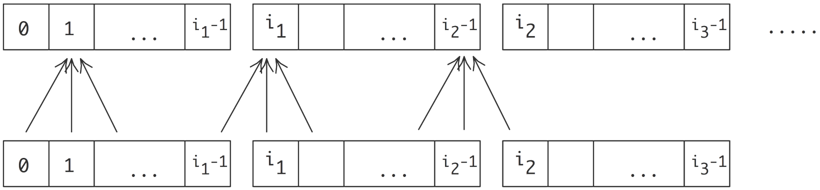
Exercise 9 `isendirecv`

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

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

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:

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

Non-blocking:

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

Pitfalls

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

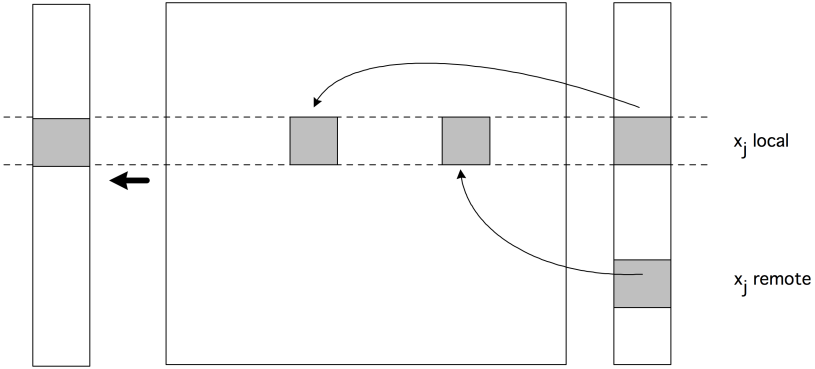
  > *Wait for the send is not necessary: if you wait for the receive, the message has arrived safely*

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

Matrices in parallel

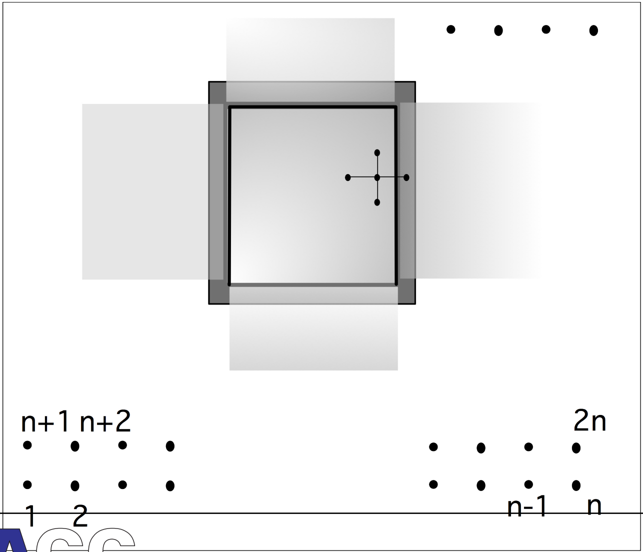
$$y \leftarrow Ax$$

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 10 `isendirecvarray`

Take your code of exercise 9 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:

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

Test: non-blocking wait

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

```
while (1) {
  MPI_Test( some_request, &flag );
  if (flag)
    // do something with incoming message
  else
    // do local work
}
```

Local operation.
Also `MPI_Testall` et cetera.

Probe for message

Is there a message?

```
// probe.c
if (procno==receiver) {
  MPI_Status status;
  MPI_Probe(sender,0,comm,&status);
  int count;
  MPI_Get_count(&status,MPI_FLOAT,&count);
  float recv_buffer[count];
  MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
} else if (procno==sender) {
  float buffer[buffer_size];
  ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
}
```

(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) 11 (`bucketpipenonblock`)

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

Exercise 12 `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 1

Does this code deadlock?

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

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

Review 2

Does this code deadlock?

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

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

Review 3

Does this code deadlock?

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

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

Review 4

Does this code deadlock?

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

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