Collective operations

Victor Eijkhout

2026 PCSE

Overview

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

Commands learned:

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

Technically

Routines can be 'collective on a communicator':

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

Collectives

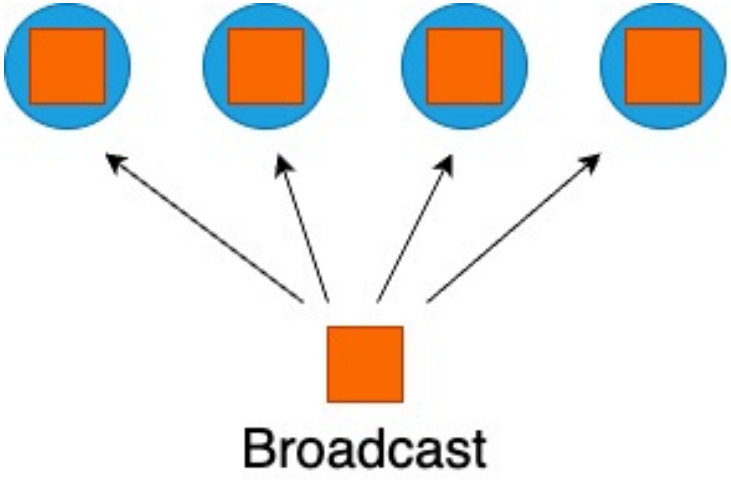
Gathering and spreading information:

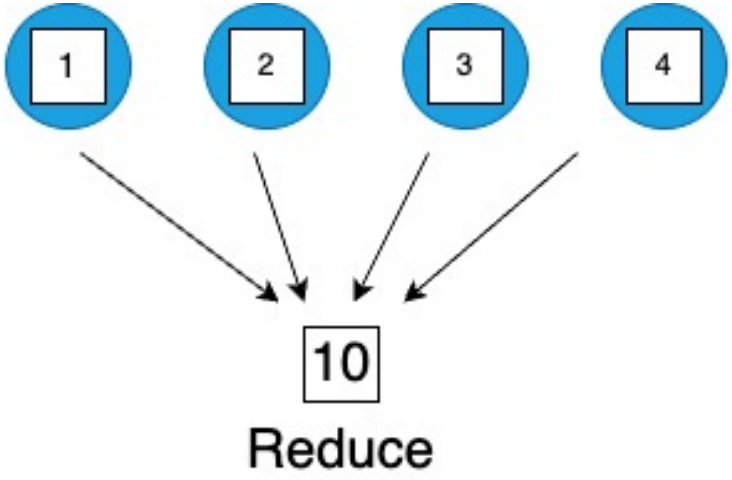
- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

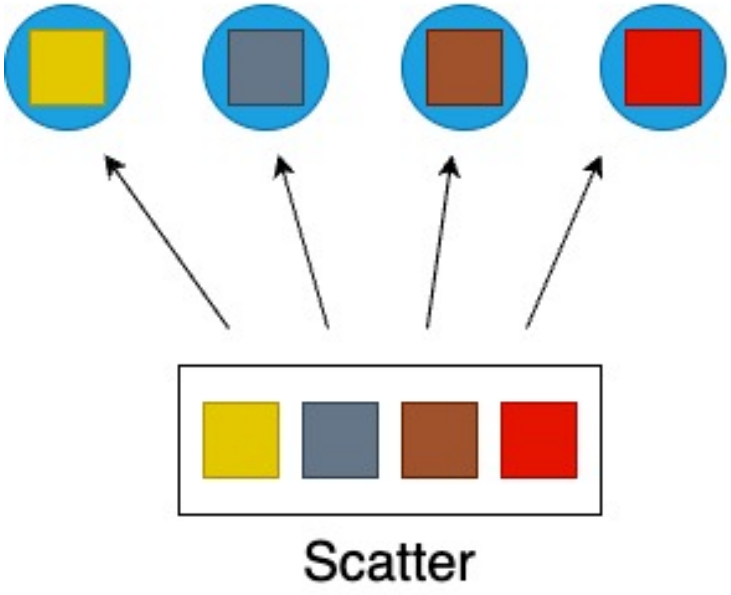
Root process: the one doing the collecting or disseminating.

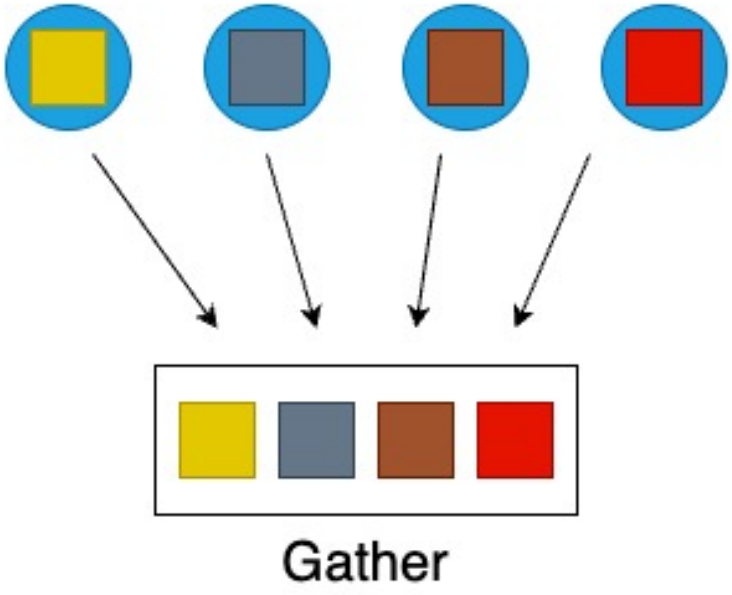
Basic cases:

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


Broadcast


1
2
3
4
10
Reduce


Scatter


Gather

Exercise 1

How would you realize the following scenarios with MPI collectives?

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

Think about time and space complexity of your suggestions.

Allreduce: reduce-to-all

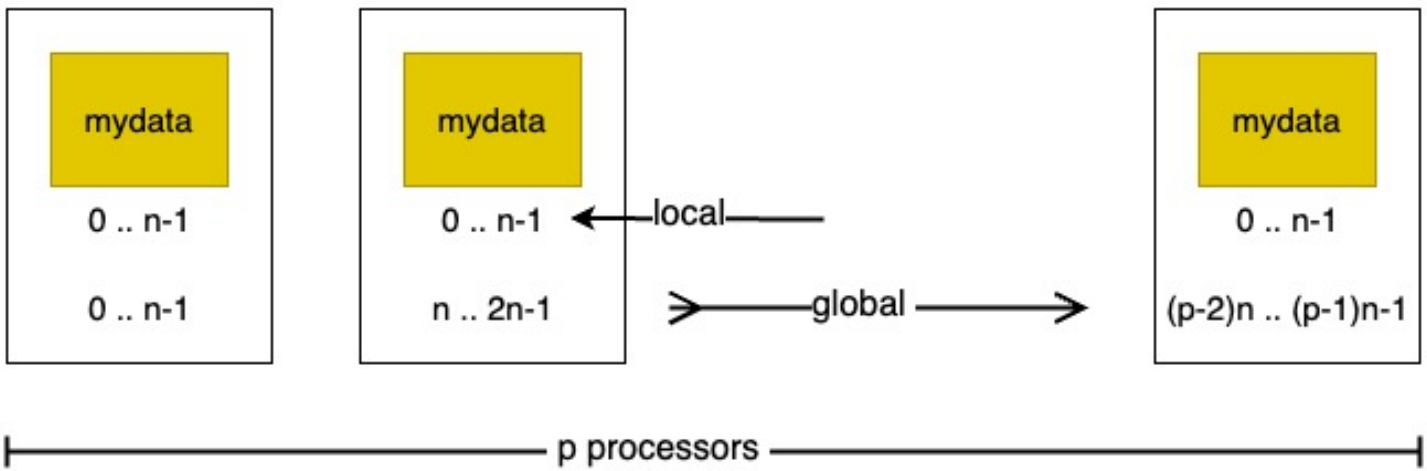
Case 2 in the exercise above contains a common case:
do a reduction, but everyone needs the result.

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

Motivation for allreduce

Example: normalizing a vector

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



Structure of allreduce

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

Exercise 2

How many objections can you come up to this strategy:

1. Gather vector x on some root process;
2. Compute the reduction on that root;
3. Construct the scaled vector on the root;
4. Scatter the scaled vector.

Another Allreduce

Standard deviation:

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

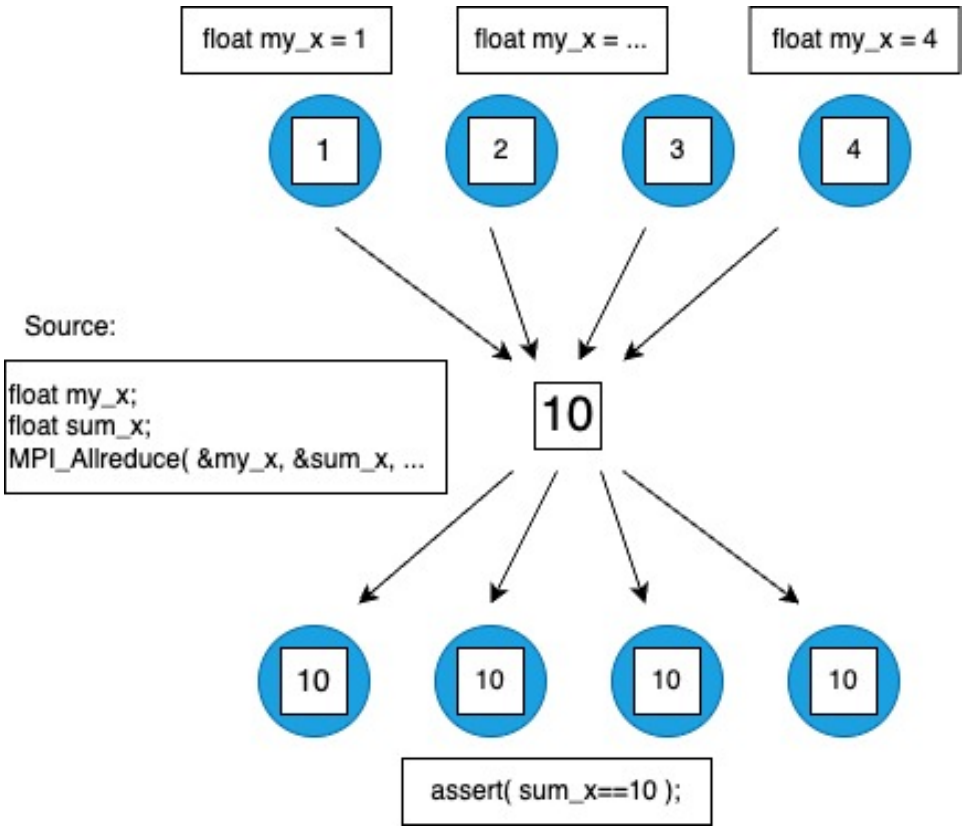
and assume that every process stores just one x_i value.

How do we compute this?

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

Conceptual picture

Recall SPMD: every process has the input and output variable



(What actually happens is a different story!)

Allreduce syntax

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

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

MPI_Allreduce

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

MPL: MPI_Allreduce

```
1 template<typename T , typename F >
2 void mpl::communicator::allreduce
3    ( F, const T &, T & ) const;
4    ( F, const T *, T *,
5      const contiguous_layout< T > & ) const;
6    ( F, T & ) const;
7    ( F, T *, const contiguous_layout< T > & ) const;
8 F : reduction function
9 T : type
```

Python: MPI_Allreduce

```
1 # native:
2 recvobj = MPI.Comm.allreduce(self, sendobj, op=SUM)
3 # numpy:
4 MPI.Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)
```

Exercise 3 `randommax`

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

$$\xi = \sum_i x_i$$

Each process then scales its value by this sum.

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

Compute the sum of the scaled numbers

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

and check that it is 1.

Buffers

Buffers in C

General principle: buffer argument is address in memory of the data.

- Buffer is void pointer:
- write `&x` or `(void*)&x` for scalar
- write `x` or `(void*)x` for array

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

Buffers in Fortran

General principle: buffer is address in memory of the data.

- Fortran always passes by reference:
- write `x` for scalar
- write `x` for array

Buffer arguments in Fortran

```
1 integer :: i
2 integer,dimension(:) :: iar
3 call MPI_Bcast( i, .... );
4 call MPI_Bcast( iar, .... );
```

Buffers in C++

- Scalars same as in C.
- Use of *std::vector* or *std::array*:

```
1 vector<float> xx(25);
2 MPI_Send( xx.data(),25,MPI_FLOAT, .... );
3 MPI_Send( &xx[0],25,MPI_FLOAT, .... );
4 MPI_Send( &xx.front(),25,MPI_FLOAT, .... );
```

- Can not send from iterator / let recv determine size/capacity.

Buffers in MPL

Two mechanisms:

1. Scalars; type derived through overloading
2. Automatic (static) arrays; type derived through overloading.
3. Layouts: contiguous or otherwise; see later.

MPL buffers through layout

You can pass a C-style array as buffer, requiring a layout:

```
1 // vector of 50 floats
2 vector<float> ar(50);
3 auto root = 0;
4 auto data = ar.data(); // or &(ar[0]) or &ar.front()
5 auto layout = mpl::contiguous_layout<float>(50)
6 comm_world::bcast( root,data,layout );
```

Large buffers

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

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

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

Large buffers in MPL

Large buffer communication is supported:

```
1 // bigint.cxx
2 size_t s;
3   vector<char> buffer(s);
4   mpl::contiguous_layout<char> buffersize(s);
5     comm_world.send( buffer.data(),buffersize,processB );
```

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 4

Extend exercise 3 to letting each process have an array.

Collective basics

Elementary datatypes

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

A bunch more.

MPL datatypes

Elementary types derived through overloading / templating.

Python datatypes

- Elementary types not needed: type can be deduced from the Numpy buffer
- Buffer / count / datatype triples can be used in exceptional circumstances.

Reduction operators

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

MPL operators

Operators need to have type:

```
T(T&)
```

Elementary operators:

```
comm_world.allreduce(mpl::plus<float>(), rank2p2p1, p2layout);
```

User-defined operator:

```
comm_world.reduce(lcm<int>(), 0, v, result);
```

MPL operators

Available: *max*, *min*, *plus*, *multiplies*, *logical_and*, *logical_or*, *logical_xor*, *bit_and*, *bit_or*, *bit_xor*.

```
1 // separate recv buffer
2 comm_world.allreduce(mpl::plus<float>(), proc_data,reduce_data);
3 // in place
4 comm_world.allreduce(mpl::plus<float>(), proc_data);
```

Reduction to single process

Reduce with a single root process: great for printing out summary information at the end of your job.

Can you think of a case were a rooted reduce is appropriate or unavoidable? (Hint: tree)

Reduction to root

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

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

In-place operations

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

More in-place operations

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

or

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

In-place (Fortran)

```
!! reduceinplace.F90
  call random_number(mynumber)
  target_proc = ntids-1;
  ! add all the random variables together
  if (mytid.eq.target_proc) then
     result = mytid
     call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
          target_proc,comm)
  else
     mynumber = mytid
     call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
          target_proc,comm)
  end if
```

or

```
!! reduceinplaceptr.F90
  in_place_val = MPI_IN_PLACE
  if (mytid.eq.target_proc) then
     ! set pointers
     result_ptr => result
     mynumber_ptr => in_place_val
     ! target sets value in receive buffer
     result_ptr = mytid
  else
     ! set pointers
     mynumber_ptr => mynumber
     result_ptr => in_place_val
     ! non-targets set value in send buffer
     mynumber_ptr = mytid
  end if
  call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
```

In-place (MPL)

Scalar:

```
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), proc_data,reduce_data);
// in place
comm_world.allreduce(mpl::plus<float>(), proc_data);
```

Buffer:

```
// collectbuffer.cxx
vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
comm_world.allreduce
  (mpl::plus<float>(), rank2p2p1.data(),reduce2p2p1.data(),two_floats);
if ( iprint )
  cout << "Got: " << reduce2p2p1.at(0) << ","
       << reduce2p2p1.at(1) << endl;
```

Broadcast

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

- All processes call with the same argument list
- *root* is the rank of the process doing the broadcast
- Each process allocates buffer space;
  root explicitly fills in values,
  all others receive values through broadcast call.
- Datatype is *MPI_FLOAT*, *MPI_INT* et cetera, different between C/Fortran.
- *comm* is usually *MPI_COMM_WORLD*

Gauss-Jordan elimination

https://youtu.be/aQYuwatlWME

MPI_Bcast

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

MPL: MPI_Bcast

```
1 template<typename T >
2 void mpl::communicator::bcast
3    ( int  root, T &  data ) const
4    ( int  root, T *  data, const layout< T > &  l ) const
```

Python: MPI_Bcast

```
1 # native:
2 rbuf = MPI.Comm.bcast(self, obj=None, int root=0)
3 # numpy:
4 MPI.Comm.Bcast(self, buf, int root=0)
```

Exercise 5 `jordan`

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

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

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

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

Exercise (optional) 6

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

Scan

Scan

Scan or 'parallel prefix': reduction with partial results

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

In vs Exclusive

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

MPI_Scan

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

MPL: MPI_Scan

```
1 template<typename T , typename F >
2 void mpl::communicator::scan
3    ( F, const T &, T & ) const;
4    ( F, const T *, T *,
5      const contiguous_layout< T > & ) const;
6    ( F, T & ) const;
7    ( F, T *, const contiguous_layout< T > & ) const;
8 F : reduction function
9 T : type
```

Python: MPI_Scan

```
1 res = Intracomm.scan( sendobj=None,recvobj=None,op=MPI.SUM)
2 res = Intracomm.exscan( sendobj=None,recvobj=None,op=MPI.SUM)
```

MPI_Exscan

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

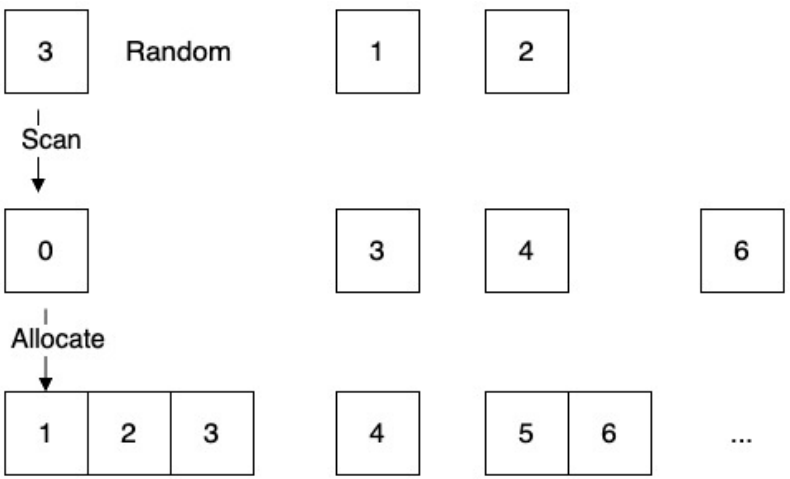
MPL: MPI_Exscan

```
1 template<typename T , typename F >
2 void mpl::communicator::exscan
3    ( F, const T &, T & ) const;
4    ( F, const T *, T *,
5      const contiguous_layout< T > & ) const;
6    ( F, T & ) const;
7    ( F, T *, const contiguous_layout< T > & ) const;
8 F : reduction function
9 T : type
```

Python: MPI_Exscan

Exercise 7 `scangather`

- Let each process compute a random value n_{local}, and allocate an array of that length. Define
$$N = \sum n_{\text{local}}$$
- Fill the array with consecutive integers, so that all local arrays, laid end-to-end, contain the numbers $0 \cdots N-1$. (See figure 7.)



Gather/Scatter, Barrier, and others

MPI_Gather

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

MPL: MPI_Gather

```
1 void mpl::communicator::gather
2     ( int  root_rank, const T * senddata, const layout< T > &  sendl ) const
3     ( int  root_rank, const T * senddata, const layout< T > &  sendl,
4                             T * recvdata, const layout< T > &  recvl ) const
5     // non-root versions:
6     ( int  root_rank, const T & senddata ) const
7     ( int  root_rank, const T & senddata, T *  recvdata ) const
```

Python: MPI_Gather

```
1 MPI.Comm.Gather
2     (self, sendbuf, recvbuf, int root=0)
```

MPI_Scatter

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

MPL: MPI_Scatter

```
1 void mpl::communicator::scatter
2     ( int  root_rank, const T *  send_data, const layout< T > &  sendl,
3       T *  recv_data, const layout< T > &  recvl ) const
4     ( int  root_rank, const T *  send_data,
5       T &  recv_data ) const
6     // non-root versions:
7     ( int  root_rank, T &  recv_data ) const
8     ( int  root_rank, T *  recv_data, const layout< T > &  recvl ) const
```

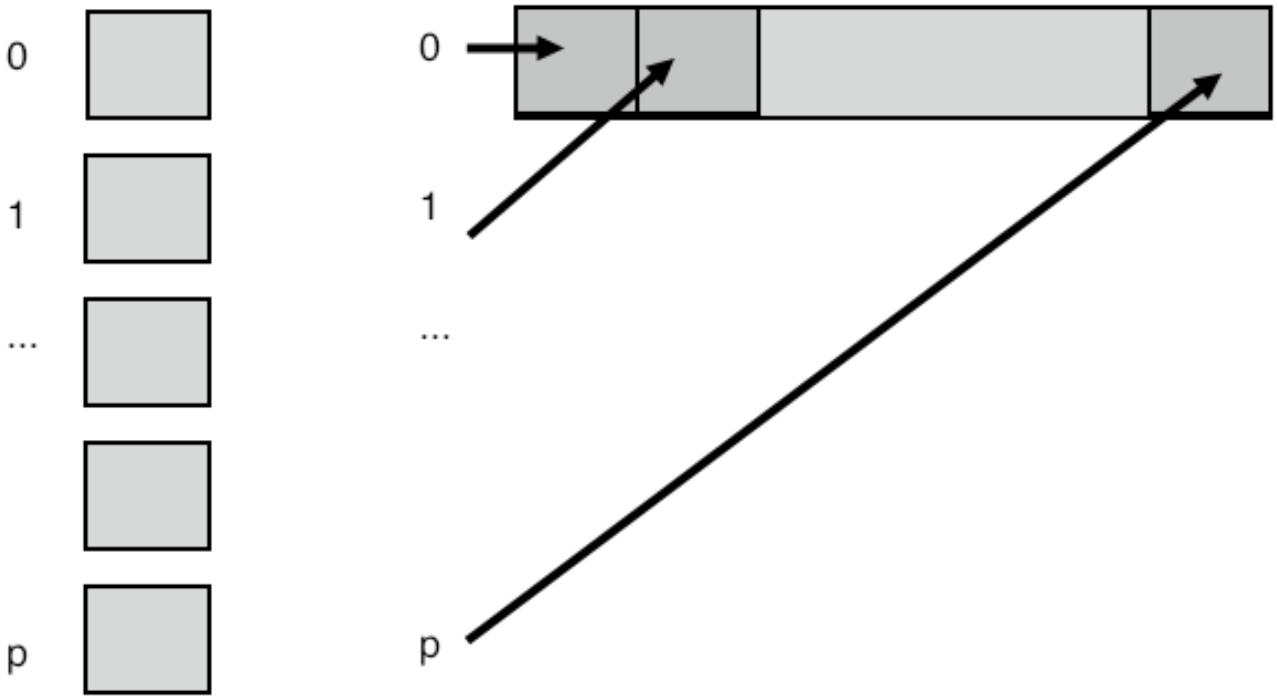
Python: MPI_Scatter

Missing Python proto: standardp/mpi˙scatter

Gather/Scatter

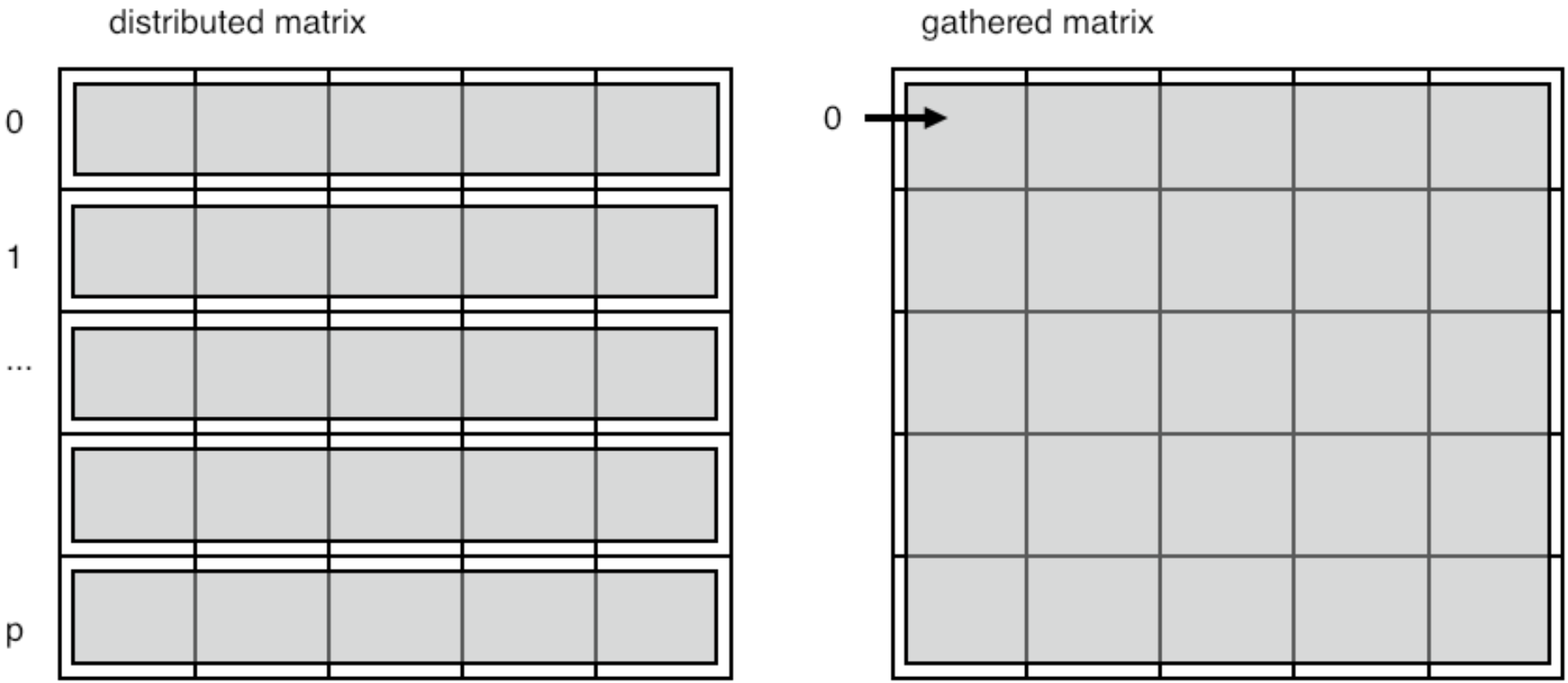
- Compare buffers to reduce
- Scatter: the `sendcount` / Gather: the `recvcount`:
  this is not, as you might expect, the total length of the buffer; instead, it is the amount of data to/from each process.

Gather pictured



Popular application of gather

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



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

MPI_Allgather

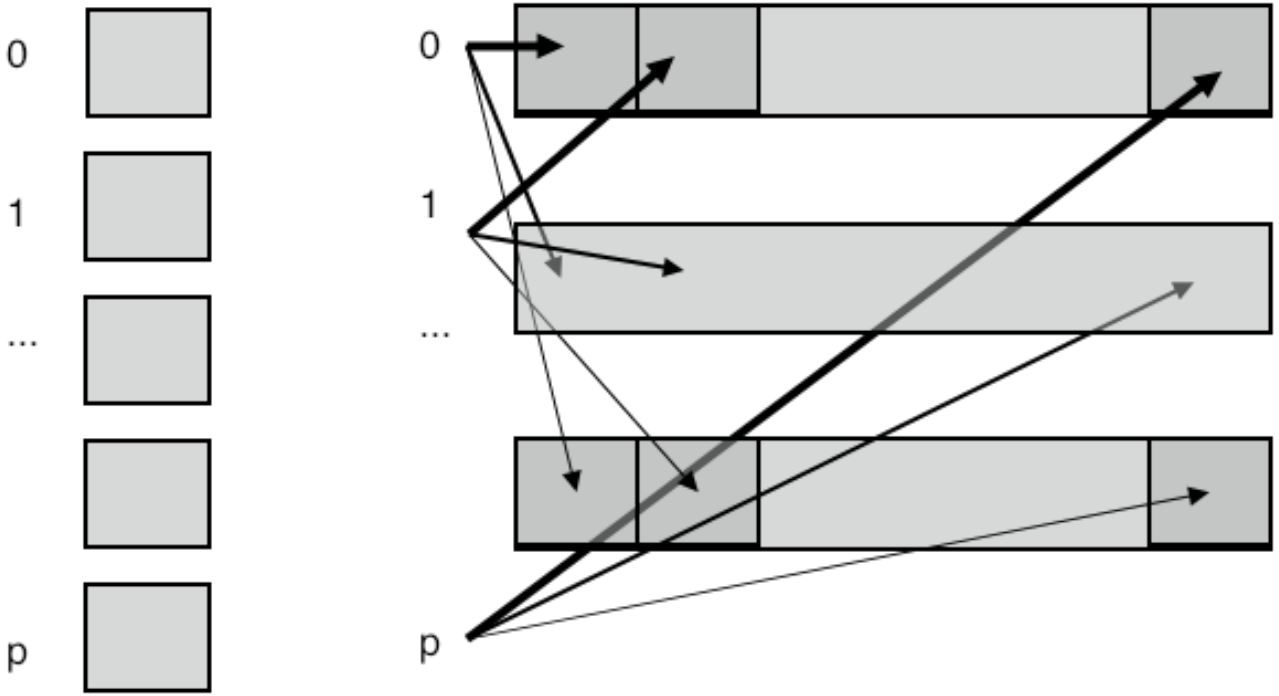
Name	Param name	Explanation	C type	F type
MPI_Allgather (				
MPI_Allgather_c (				
	sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)
	sendcount	number of elements in send buffer	[int MPI_Count	INTEGER
	sendtype	datatype of send buffer elements	MPI_Datatype	TYPE(MPI_Datatype)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

MPL: MPI_Allgather

```
1 void allgather
2     ( const T & send_data, T * recv_data ) const
3     ( const T * send_data, const layout< T > & sendl,
4             T * recv_data, const layout< T > & recvl ) const
```

Python: MPI_Allgather

Allgather pictured



V-type collectives

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

MPI_Gatherv

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

MPL: MPI_Gatherv

```
1 template<typename T>
2 void gatherv
3    (int root_rank, const T *senddata, const layout<T> &sendl,
4     T *recvdata, const layouts<T> &recvls, const displacements &recvdispls) const
5    (int root_rank, const T *senddata, const layout<T> &sendl,
6     T *recvdata, const layouts<T> &recvls) const
7    (int root_rank, const T *senddata, const layout<T> &sendl ) const
```

Python: MPI_Gatherv

```
1 Gatherv(self, sendbuf, [recvbuf,counts], int root=0)
```

Exercise 8 `scangather`

Take the code from exercise 7 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires *`MPI_Gatherv`*.

How do you construct the lengths and displacements arrays?

Review 1

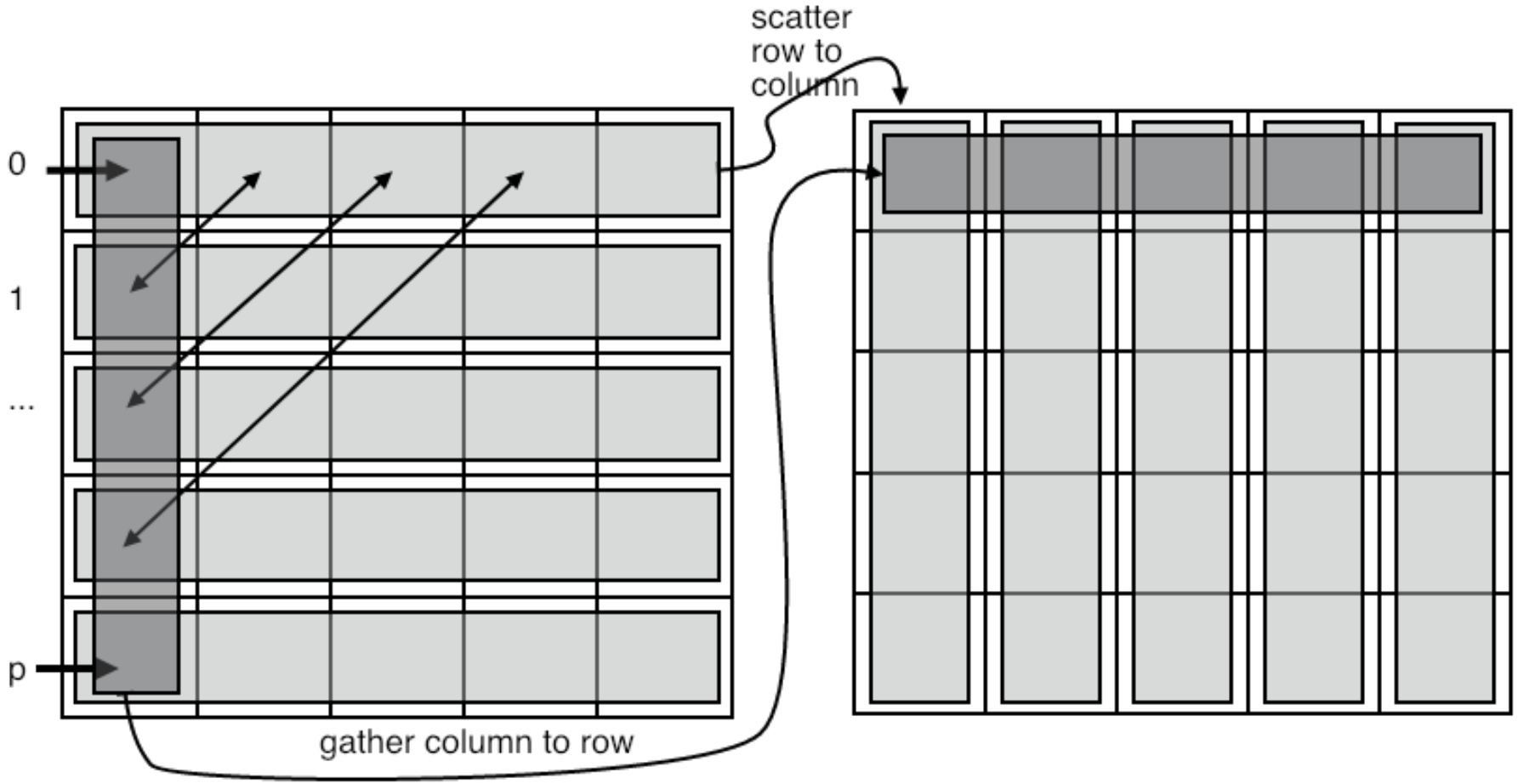
An *MPI_Scatter* call puts the same data on each process

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

All-to-all

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

Data transposition



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

All-to-allv

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

Radix sort

Sort 4 numbers on two processes:

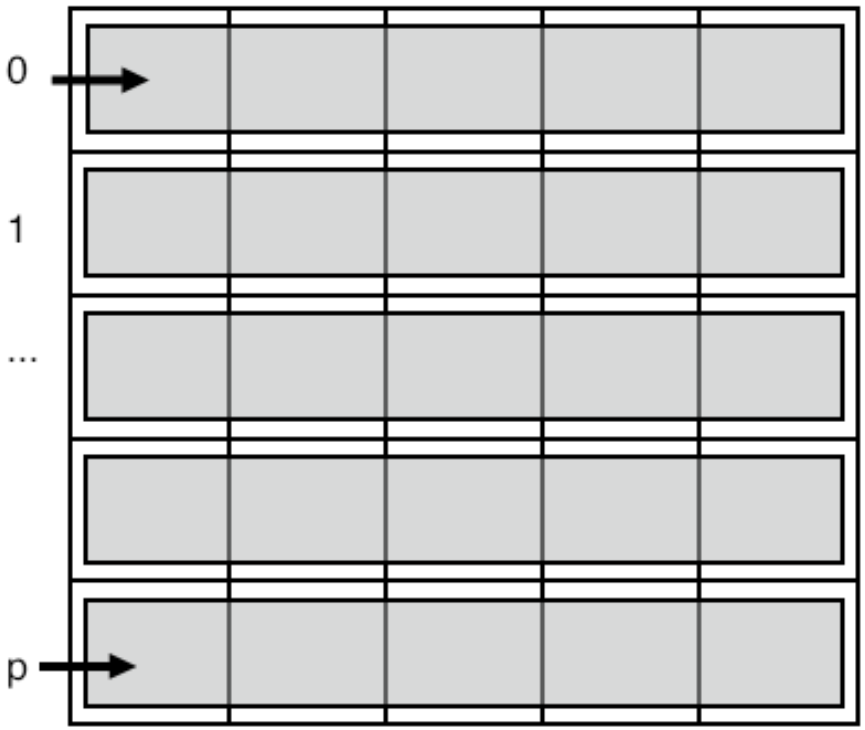
	proc0		proc1		
array	2	5	7	1	
binary	010	101	111	001	
stage 1					
last digit	0	1	1	1	
	(this serves as bin number)				
sorted	010		101	111	001
stage 2					
next digit	1		0	1	0
	(this serves as bin number)				
sorted	101	001	010	111	
stage 3					
next digit	1	0	0	1	
	(this serves as bin number)				
sorted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

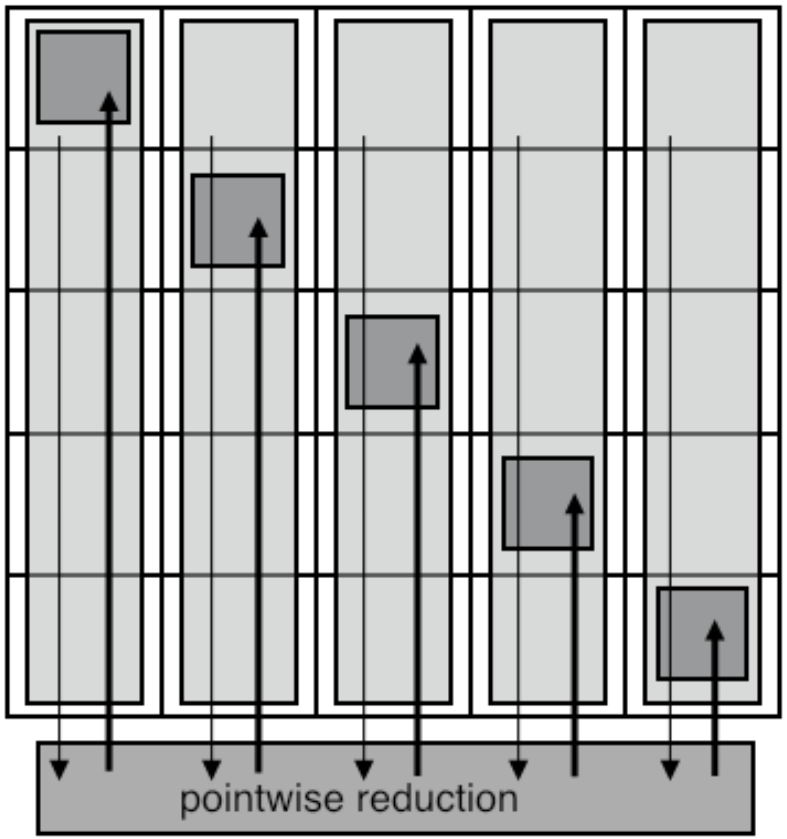
Reduce-scatter

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

Example: sparse matrix setup

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





Barrier

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

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

User-defined operators

MPI Operators

Define your own reduction operator

- Define operator between partial result and new operand

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

- Don't forget to free:

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

- Make your own reduction scheme *MPI_Reduce_local*

User defined operators, Fortran

```
1 FUNCTION user_function( invec(*), inoutvec(*), length, mpitype)
2 <fortrantype> :: invec(length), inoutvec(length)
3 INTEGER :: length, mpitype
```

MPI_Op_create

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

MPL: MPI_Op_create

Missing MPL proto: mpi˙op˙create

Python: MPI_Op_create

```
1 MPI.Op.create(cls,function,bool commute=False)
```

Example

Smallest nonzero:

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

Review 2

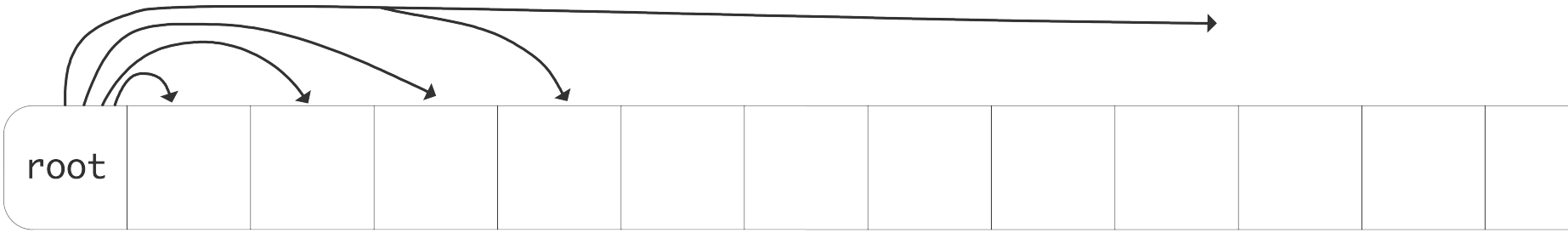
The $\|\cdot\|_2$ norm (sum of squares) needs a custom operator.

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

Performance of collectives

Naive realization of collectives

Broadcast:



Single message:

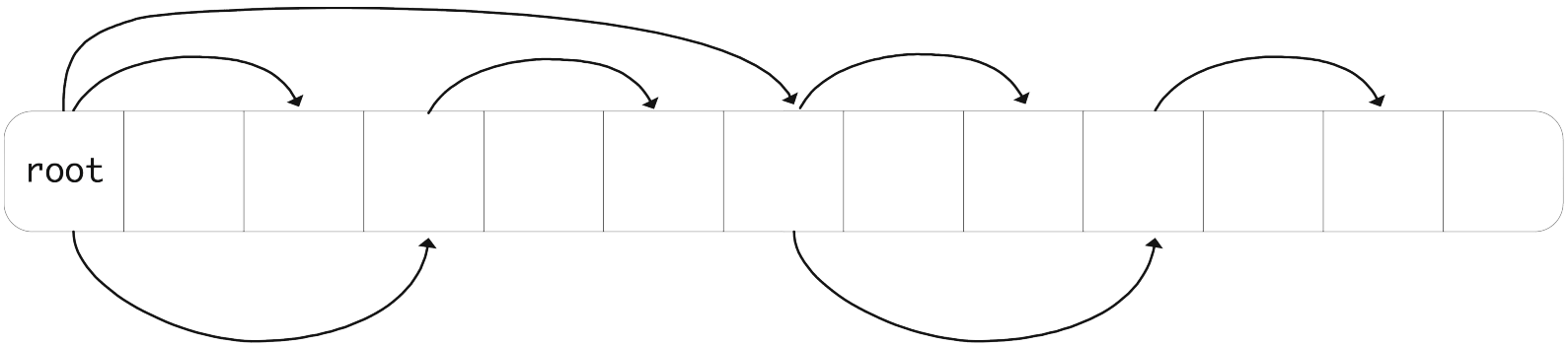
$$\alpha = \text{message startup} \approx 10^{-6}s, \qquad \beta = \text{time per word} \approx 10^{-9}s$$

- Time for message of n words:

$$\alpha + \beta n$$

- Time for collective? Can you improve on that?

Better implementation of collective



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

Implementation of Reduce

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

Implementation of Allreduce

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

Review 3

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

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

Reduction operators

User-defined operators

Given a reduction function:

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

create a new operator:

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

Exercise 9 `onenorm`

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

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