

# MPI Course

Victor Eijkhout

2024 COE 379L / CSE 392



Textbooks and repositories:  
<https://theartofhpc.com>



The MPI library is the main tool for parallel programming on a large scale. This course introduces the main concepts through lecturing and exercises.



1. The SPMD Model *link*
2. Collectives *link*
3. Point-to-point *link*
4. Derived datatypes *link*
5. Communicators *link*
6. MPI I/O *link*
7. One-sided communication *link*
8. Big data *link*
9. Advanced collectives *link*
10. Shared memory *link*
11. Process management *link*
12. Process topologies *link*
13. Trace and performance *link*



# Basics



# Part I

## The SPMD model



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

Commands learned:

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



# The MPI worldview: SPMD



# Computers when MPI was designed

TACC



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

⇒ process model, no data sharing.



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



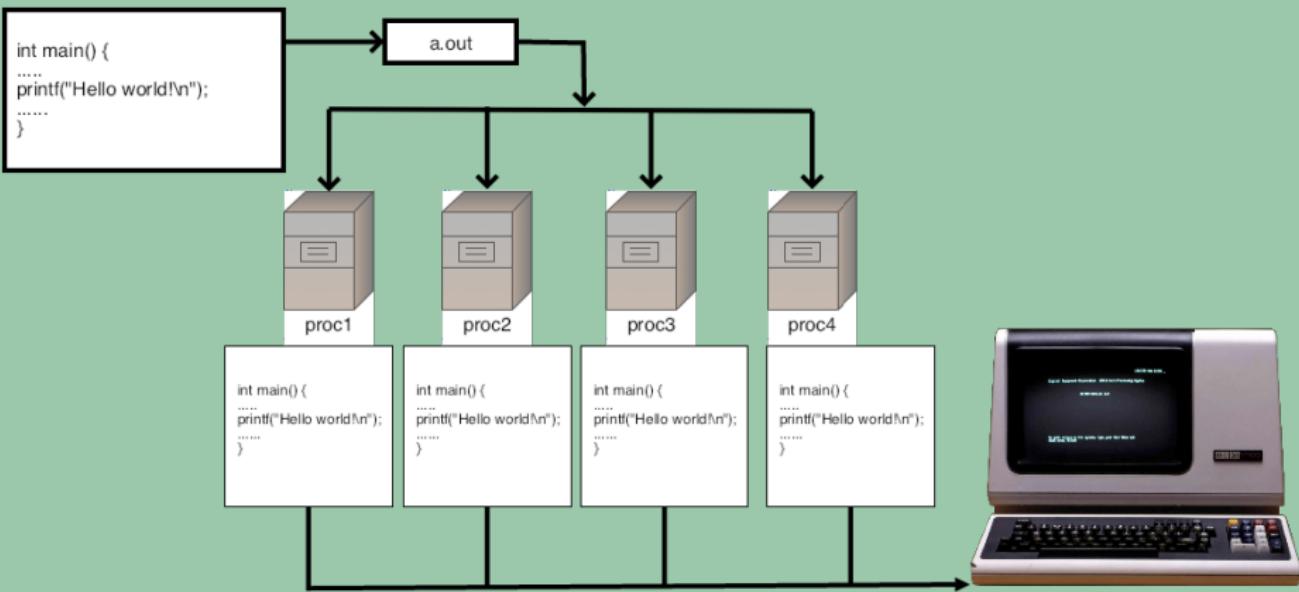
Write a ‘hello world’ program, without any MPI in it, and run it in parallel with `mpiexec` or your local equivalent. Explain the output.

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



# In a picture

TACC



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

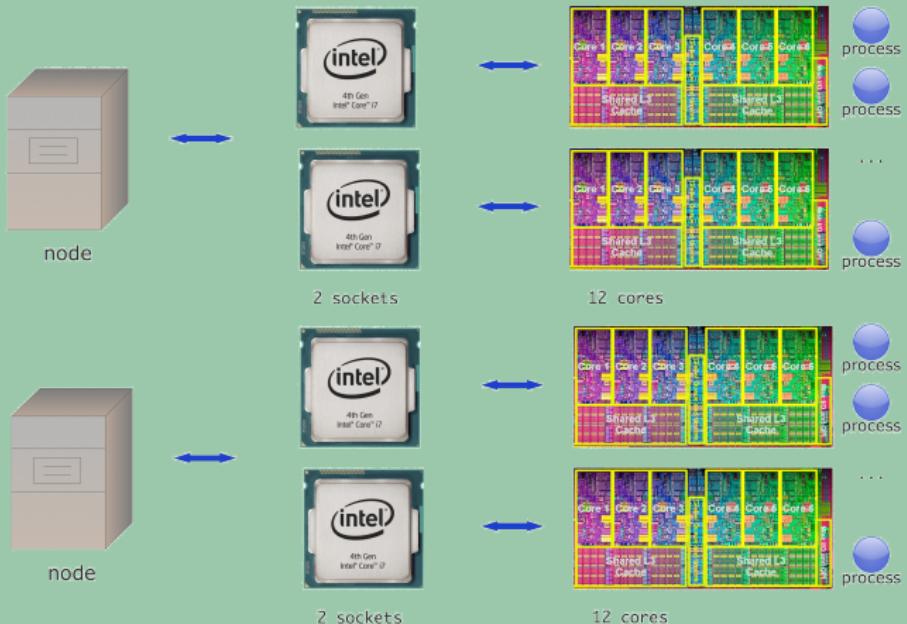
Symmetry: There is no 'master process', all processes are equal, start and end at the same time.

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



# Practicalities





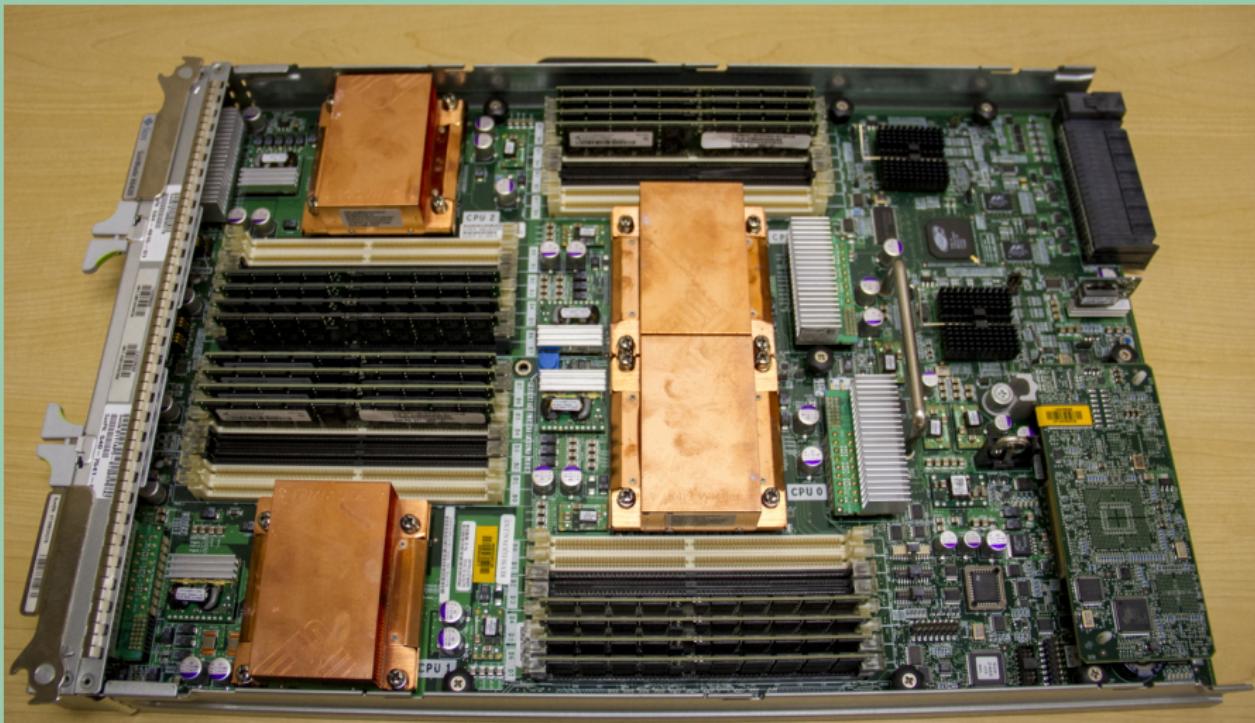
A node has multiple sockets, each with multiple cores.

Pure MPI puts a process on each core: pretend shared memory doesn't exist.



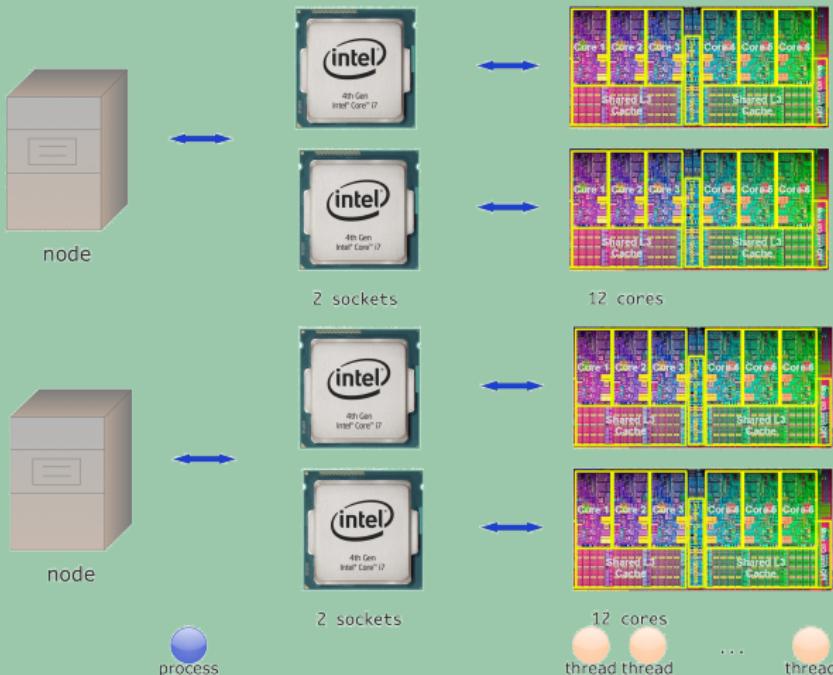
# Quad socket node

TACC



# Hybrid programming

TACC



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



'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.



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

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

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

(For CMake see slide 405.)



Running your program at TACC:

```
#SBATCH -N 4  
#SBATCH -n 200  
ibrun yourprog
```

the number of processes is determined by SLURM. General case of running code

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



- 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. . .



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

- Use a package manager

Apple: brew or macports

Linux: yum, aptget, ...

Windows: I'll have to get back to you on that

- ... or download and compile from source [mpich.org](http://mpich.org)



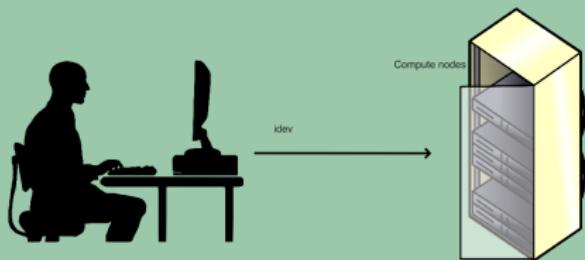
Typical cluster:

- 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.

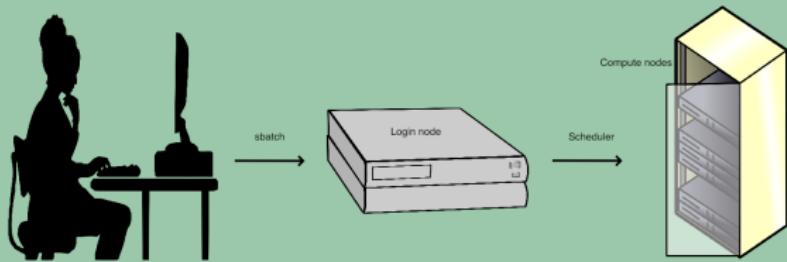
Hostfile: the description of where your job runs. Usually generated by a *job scheduler*.



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



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



# We start learning MPI!



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

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

This is not a ‘parallel region’.

Only internal library initialization:

allocate buffers, discover network, ...



No explicit init/finalize:

- init is done by the first command that needs it
- finalize in some destructor.



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.



# About library calls and bindings



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

MPI can also be used from C++ and Python



You need an include file:

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

This defines all routines and constants.



MPI-1 had C++ bindings, by MPI-2 they were deprecated, in MPI-3 they have been removed.

- Easy solution: use the C bindings unaltered.  
This is done in the cxx exercise directory; Ugly: very un-OO.
- There are private projects for C++ bindings.  
In particular MPL: <https://github.com/rabauke/mpl>  
(Exercises in mpl directory.)  
Very modern OO, Header-only  
Not a full MPI implementation: (I/O and one-sided mostly missing)

In your program:

```
1 #include <mpl/mpl.hpp>
2
```

Compiling:

```
mpicxx -o prog sources.cxx -I${TACC_MPL_INC}
```



# Ranks



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

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

Lowest number is always zero.

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



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

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

Write a second version of this program, where each process opens a unique file and writes to it. *On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.*



Signature:

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

Use:

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

But forget about that error code most of the time:

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



Signature:

```
int mpl::communicator::rank ( ) const
```

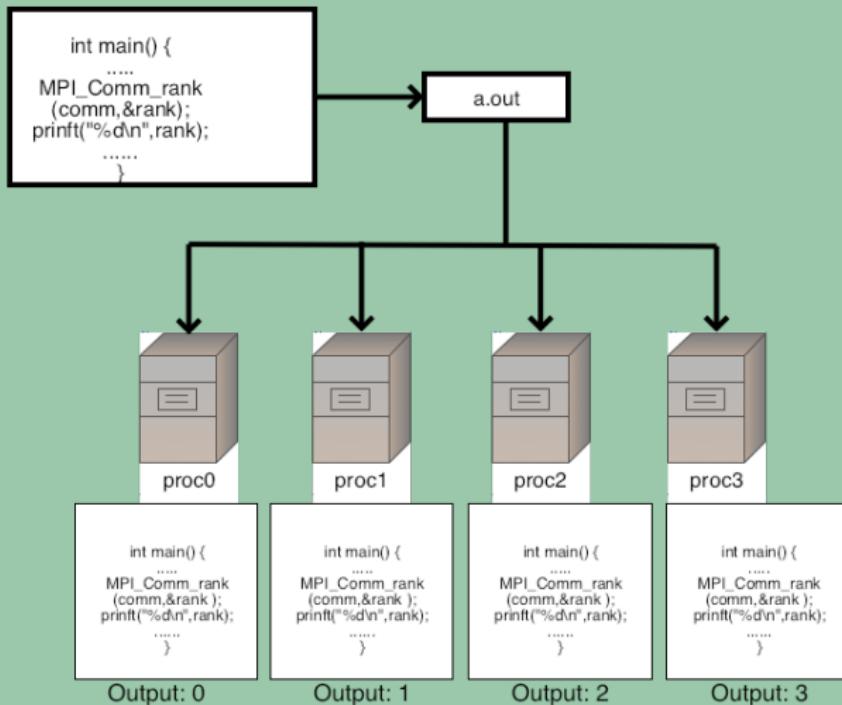
Use

```
1 const mpl::communicator &comm_world = mpl::environment::comm_world();  
2 int nprocs, procno;  
3 nprocs = comm_world.size();
```



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





MPI routines invoke an error handler (slide 408)

default action: abort

Every routine is defined as returning integer error code

- In C: function result.

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

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

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

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

There's not a lot you can do with an error code:

very hard to recover from errors in parallel.

By default code bombs with (hopefully informative) message.



For now, the communicator will be `MPI_COMM_WORLD`.

C:

```
1 MPI_Comm comm = MPI_COMM_WORLD;
```

F:

```
1 Type(MPI_Comm) :: comm = MPI_COMM_WORLD
```

P:

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
```

MPL:

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



T/F?

1. In C, the result of `MPI_Comm_rank` is a number from zero to number-of-processes-minus-one, inclusive.
2. In Fortran, the result of `MPI_Comm_rank` is a number from one to number-of-processes, inclusive.



Use the command `MPI_Get_processor_name`. Confirm that you are able to run a program that uses two different nodes.

TACC nodes have a hostname cRRR-CNN, where RRR is the rack number, C is the chassis number in the rack, and NN is the node number within the chassis. Communication is faster inside a rack than between racks!

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



Processes (can) run on physically distinct locations.

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



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

```
Program:  
number <- MPI_Comm_rank  
  
name <- MPI_Get_processor_name
```

```
Program:  
number <- MPI_Comm_rank  
0  
name <- MPI_Get_processor_name  
c111.tacc.utexas.edu
```

c111.tacc.utexas.edu

```
Program:  
number <- MPI_Comm_rank  
1  
name <- MPI_Get_processor_name  
c111.tacc.utexas.edu
```

c222.tacc.utexas.edu

```
Program:  
number <- MPI_Comm_rank  
2  
name <- MPI_Get_processor_name  
c222.tacc.utexas.edu
```

```
Program:  
number <- MPI_Comm_rank  
3  
name <- MPI_Get_processor_name  
c222.tacc.utexas.edu
```



Processes (can) run on physically distinct locations.

Code:

```
1 // procname.cxx
2 procno = comm_world.rank();
3 string procname =
4     mpl::environment::processor_name();
5 stringstream ss;
6 ss << "[" << procno << "] "
7     << " Running on: " << procname;
8 cout << ss.str() << '\n';
```

Output:

```
1 TACC: Starting up job
    ↪6051291
2 TACC: Starting
    ↪parallel
    ↪tasks...
3 [6] Running on:
    ↪c204-031.frontera.tacc.u
4 [7] Running on:
    ↪c204-031.frontera.tacc.u
5 [2] Running on:
    ↪c204-029.frontera.tacc.u
6 [4] Running on:
    ↪c204-030.frontera.tacc.u
7 [0] Running on:
    ↪c204-028.frontera.tacc.u
8 [3] Running on:
    ↪c204-029.frontera.tacc.u
9 [1] Running on:
    ↪c204-028.frontera.tacc.u
```



# Your first useful parallel program



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.



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

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

Can you find more than one solution?



Allocate on each process an array:

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

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

It may be hard to print the output in a non-messy way.



# Part II

## Collectives



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**



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.



# Concepts



Gathering and spreading information:

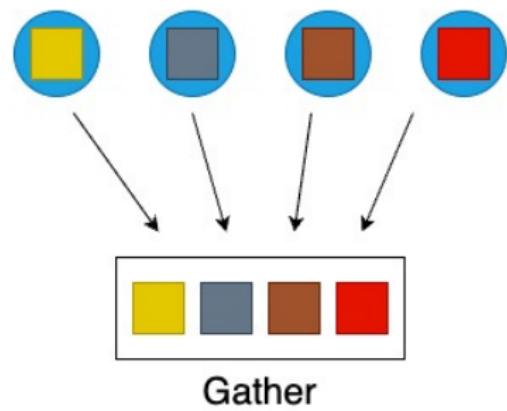
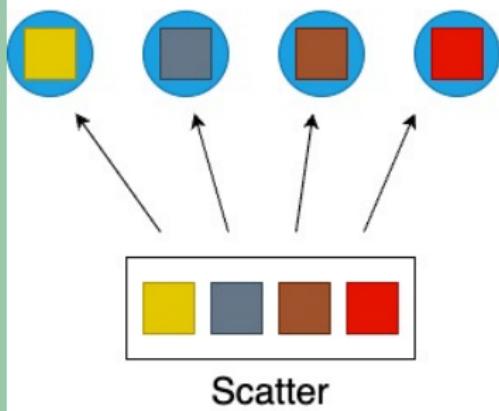
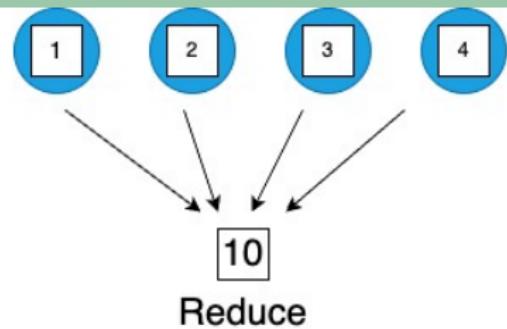
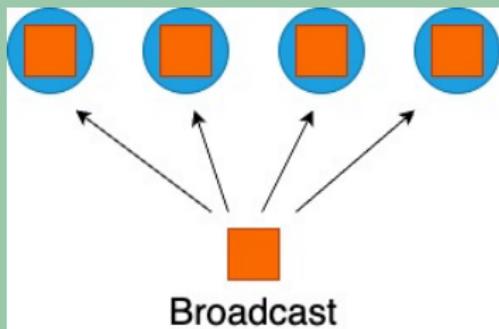
- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

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.





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.



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

- **MPI\_Allreduce** does the same as:  
**MPI\_Reduce** (reduction) followed by **MPI\_Bcast** (broadcast)
- Same running time as either, half of reduce-followed-by-broadcast  
(no proof given here)
- Common use case, symmetrical expression.



Example: normalizing a vector

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

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



- 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?



Standard deviation:

$$\sigma = \sqrt{\frac{1}{N} \sum_i^N (x_i - \mu)^2} \quad \text{where} \quad \mu = \frac{\sum_i^N x_i}{N}$$

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

How do we compute this?

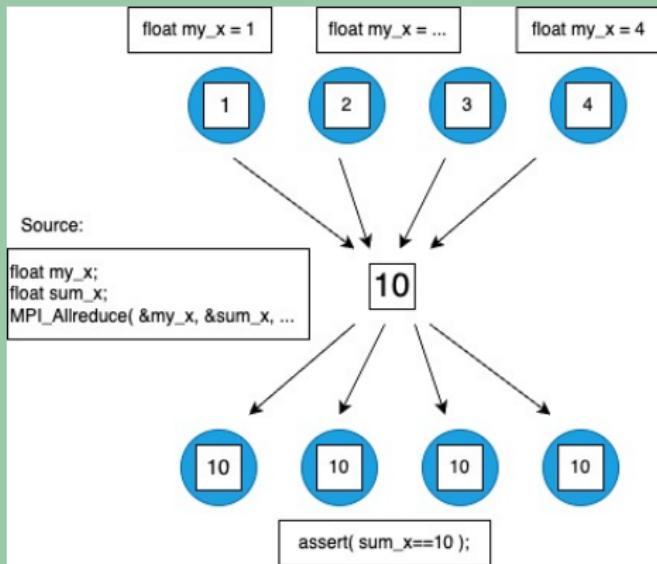
1. The calculation of the average  $\mu$  is a reduction.
2. Every process needs to compute  $x_i - \mu$  for its value  $x_i$ , so use allreduce operation, which does the reduction and leaves the result on all processes.
3.  $\sum_i (x_i - \mu)$  is another sum of distributed data, so we need another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



# Conceptual picture

TACC

Recall SPMD: every process has the input and output variable



(What actually happens is a different story!)



```
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.



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.



Extend the previous exercise to letting each process have an array.



Name	Param name	Explanation	C type	F type
MPI_Allreduce (				
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	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)
)				



```
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
```



# Buffers



- 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.



Two mechanisms:

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



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);
```



# Collective basics



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_COMPLEX MPI_LOGICAL	<i>MPI.DOUBLE</i>	
unsigned	extensions		
			MPI_Aint MPI_Offset

A bunch more.



Elementary types derived through polymorphism.



MPI_Op	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
MPI_MAXLOC	location of max
MPI_MINLOC	location of min

A couple more.



Operators need to have type:

```
1   T(T&)
```

Elementary operators:

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

User-defined operator:

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



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



```
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.



```
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);
```



```
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);
```



Scalar:

```
1 float
2     xrank = static_cast<float>( comm_world.rank() ),
3     xreduce;
4 // separate recv buffer
5 comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
6 // in place
7 comm_world.allreduce(mpl::plus<float>(), xrank);
```

Buffer:

```
1 // collectbuffer.cxx
2 float
3     xrank = static_cast<float>( comm_world.rank() );
4 vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
5 mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
6 comm_world.allreduce
7     (mpl::plus<float>(),
8      ↪rank2p2p1.data(),reduce2p2p1.data(),two_floats);
9 if ( iprint )
10    cout << "Got: " << reduce2p2p1.at(0) << ","
11           << reduce2p2p1.at(1) << endl;
```



```
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`



<https://youtu.be/aQYuwatlWME>



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

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

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.



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



# 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.



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         $x_0$                $x_0 \oplus x_1$                $\cdots$        $\oplus_{i=0}^{p-2} x_i$



Name	Param name	Explanation	C type	F type
MPI_Scan (				
	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	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)
	)			



Name	Param name	Explanation	C type	F type
MPI_Exscan (				
	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	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)
)				



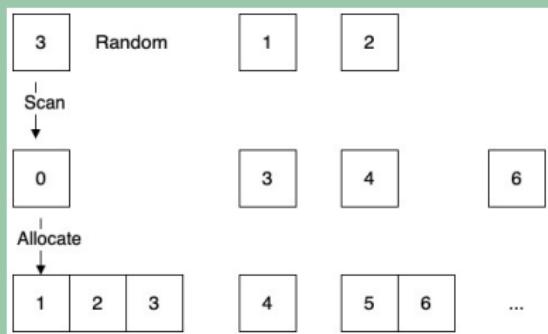
```
1 template<typename T , typename F >
2 void mpl::communicator::scan
3     ( F, const T &, T & ) const;
4     ( F, const T *, T *,
5         const contiguous_layout< T > & ) const;
6     ( F, T & ) const;
7     ( F, T *, const contiguous_layout< T > & ) const;
8 F : reduction function
9 T : type
```



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

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

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



# Gather/Scatter, Barrier, and others



Name	Param name	Explanation	C type	F type
MPI_Gather (				
sendbuf		starting address of send buffer	const void*	TYPE(*), DIMENSION(..)
sendcount		number of elements in send buffer	int	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	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)
)				



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



Name	Param name	Explanation	C type	F type
MPI_Scatter (				
	sendbuf	address of send buffer	const void*	TYPE(*), DIMENSION(..)
	sendcount	number of elements sent to each process	int	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	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)
	)			

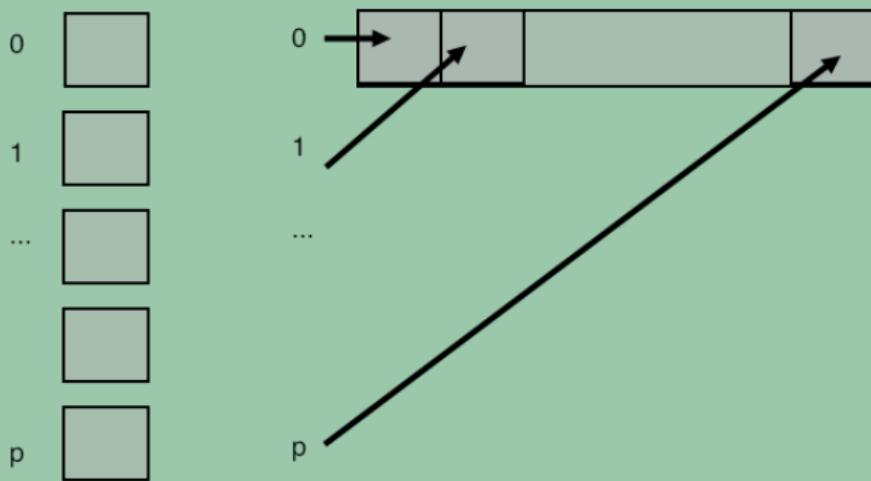


```
1 void mpl::communicator::scatter
2     ( int root_rank, const T * send_data, const layout< T > &
3       ↪sendl,
4       T * recv_data, const layout< T > & recvl ) const
5     ( int root_rank, const T * send_data,
6       T & recv_data ) const
7     // non-root versions:
8     ( int root_rank, T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



- 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.

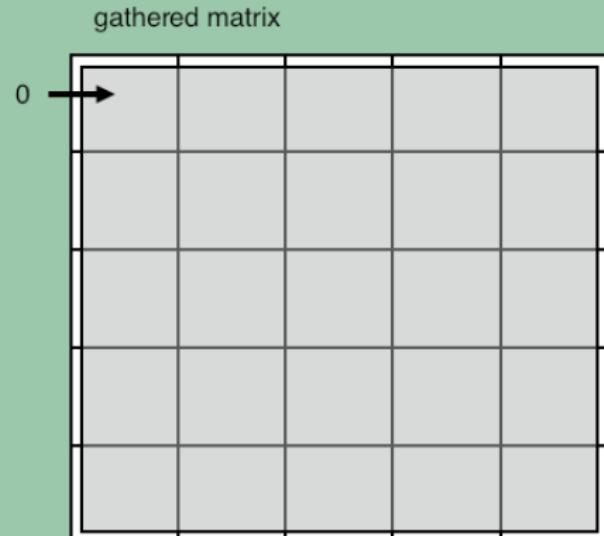




# Popular application of gather

TACC

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.

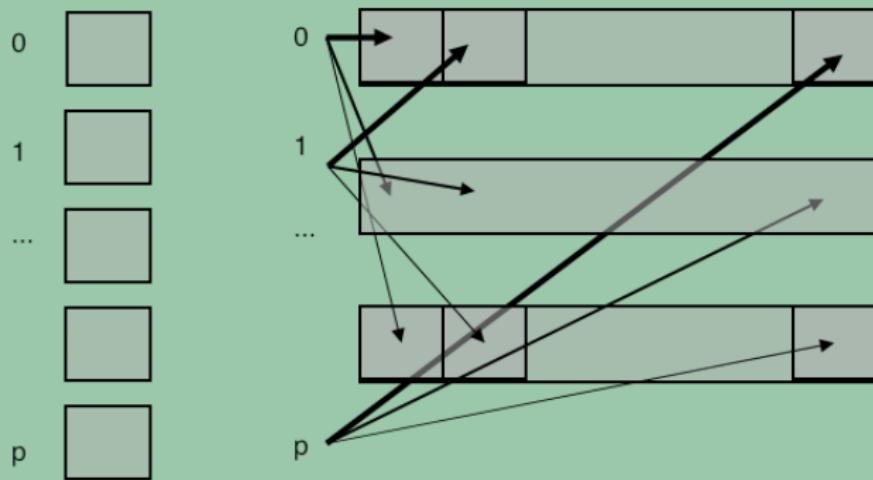


Name	Param name	Explanation	C type	F type
MPI_Allgather (				
sendbuf		starting address of send buffer	const void*	TYPE(*), DIMENSION(..)
sendcount		number of elements in send buffer	int	INTEGER
sendtype		datatype of send buffer elements	MPI_Datatype	TYPE(MPI_Datatype)
recvbuf		address of receive buffer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



```
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
```





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



Name	Param name	Explanation	C type	F type
MPI_Gatherv (				
sendbuf		starting address of send buffer	const void*	TYPE(*), DIMENSION(..)
sendcount		number of elements in send buffer	int	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[]	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[]	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)
)				



```
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
5       ↪&recvdispls) const
6     (int root_rank, const T *senddata, const layout<T> &sendl,
7      T *recvdata, const layouts<T> &recvls) const
7     (int root_rank, const T *senddata, const layout<T> &sendl ) const
```



Take the code from exercise 15 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?



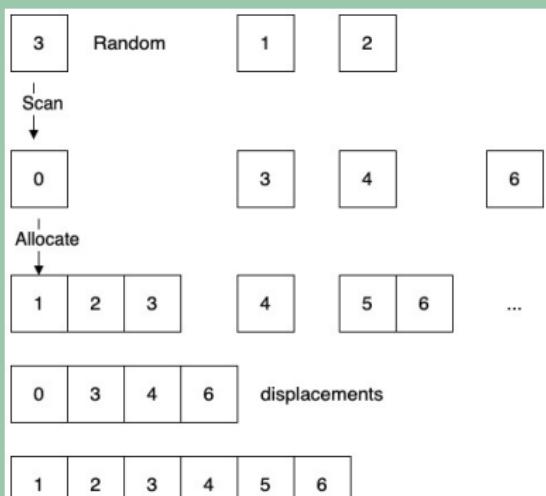
# Exercise 15 (scangather)

TACC

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

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

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



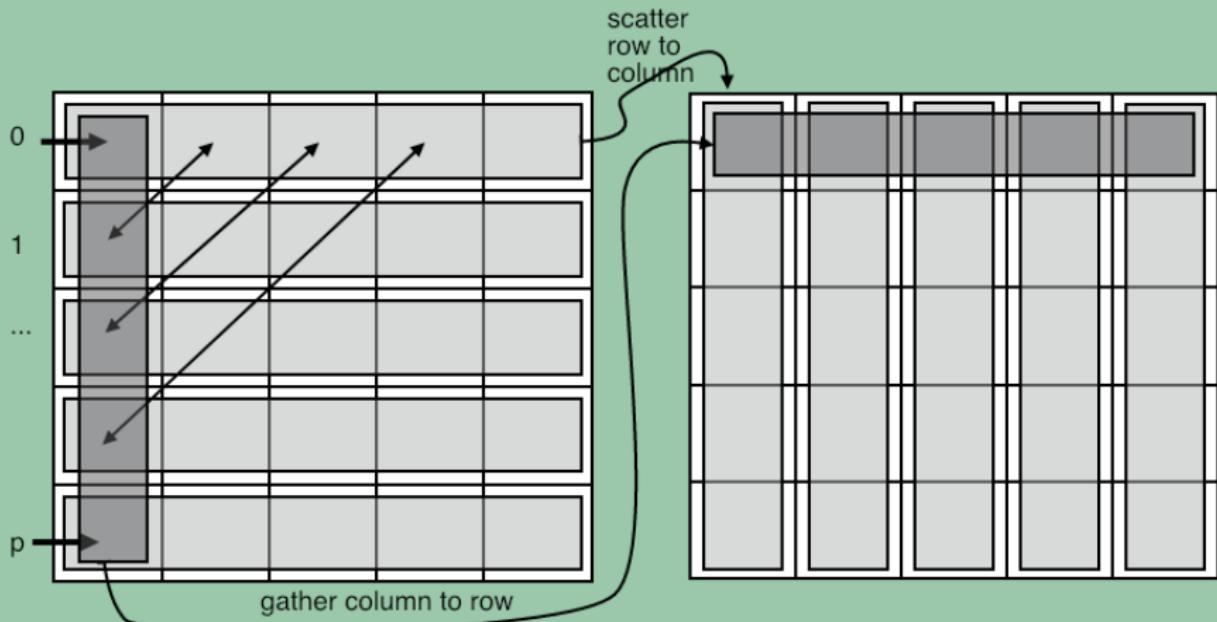
An **MPI\_Scatter** call puts the same data on each process

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



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





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



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



Sort 4 numbers on two processes:

array binary	proc0		proc1	
	2	5	7	1
	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



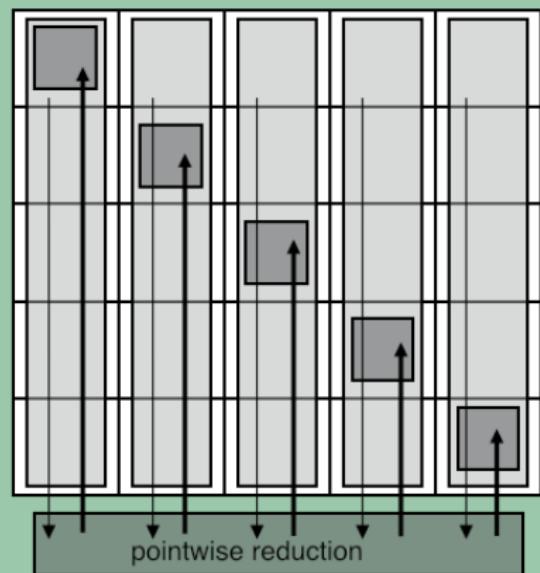
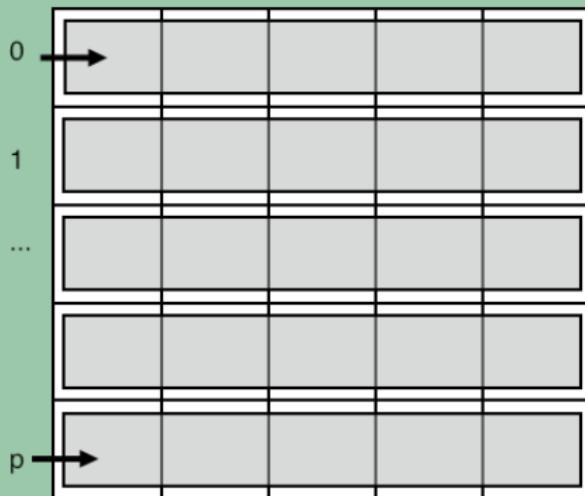
- Pointwise reduction (one element per process) followed by scatter
- Somewhat related to all-to-all: data transpose but reduced information, rather than gathered.
- Applications in both sparse and dense matrix-vector product.



# Example: sparse matrix setup

TACC

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



```
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



## 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**



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



Smallest nonzero:

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



Broadcast:



Single message:

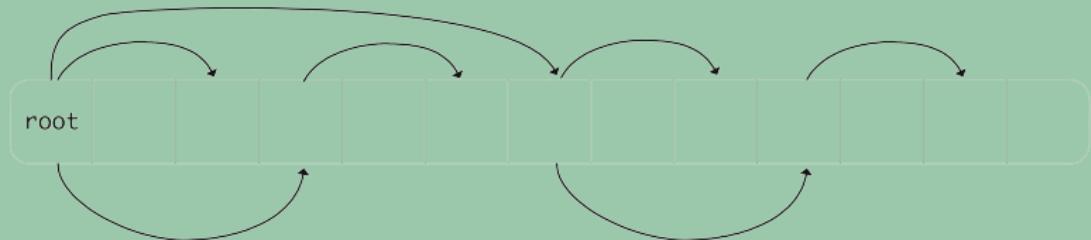
$$\alpha = \text{message startup} \approx 10^{-6} \text{s}, \quad \beta = \text{time per word} \approx 10^{-9} \text{s}$$

- Time for message of  $n$  words:

$$\alpha + \beta n$$

- Time for collective? Can you improve on that?





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



# Implementation of Reduce

TACC

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



# Implementation of Allreduce

TACC

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



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

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



## Reduction operators



Given a reduction function:

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

create a new operator:

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



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

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



# Part III

## Point-to-point communication



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**



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



A sends to B, B sends back to A

What is the code for A? For B?



Remember SPMD:

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



Name	Param name	Explanation	C type	F type
MPI_Send (				
buf		initial address of send buffer	const void*	TYPE(*), DIMENSION(..)
count		number of elements in send buffer	int	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)
)				



Name	Param name	Explanation	C type	F type
MPI_Recv (				
buf		initial address of receive buffer	void*	TYPE(*), DIMENSION(..)
count		number of elements in receive buffer	int	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)
)				



```
1 template<typename T >
2 void mpl::communicator::send
3     ( const T scalar&,int dest,tag = tag(0) ) const
4     ( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
5     ( iterT begin,iterT end,int dest,tag = tag(0) ) const
6 T : scalar type
7 begin : begin iterator
8 end : end iterator
```



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



Use `MPI_STATUS_IGNORE` unless ...

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

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



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

- Run multiple ping-pongs (say a thousand) and put the timer around the loop. The first run may take longer; try to discard it.
- Run your code with the two communicating processes first on the same node, then on different nodes. Do you see a difference?
- Then modify the program to use longer messages. How does the timing increase with message size?

For bonus points, can you do a regression to determine  $\alpha, \beta$ ?

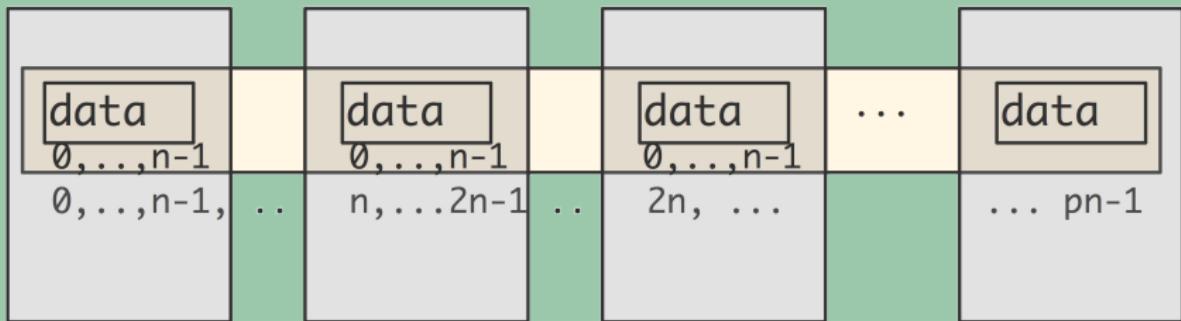


# Distributed data



Distributed array: each process stores disjoint local part

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



Local numbering  $0, \dots, n_{\text{local}}$ ;  
global numbering is ‘in your mind’.



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

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



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



If the distributed array is not perfectly divisible:

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

This gives a load balancing problem. Better solution?



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



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:  $u_i \equiv u(ih)$ .



## Equations

$$\begin{cases} -u_{i-1} + 2u_i - u_{i+1} &= h^2 f(x_i) \quad 1 < i < n \\ 2u_1 - u_2 &= h^2 f(x_1) + u_0 \\ 2u_n - u_{n-1} &= h^2 f(x_n) + u_{n+1}. \end{cases}$$
$$\begin{pmatrix} 2 & -1 & & \emptyset \\ -1 & 2 & -1 & \\ \emptyset & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} h^2 f_1 + u_0 \\ h^2 f_2 \\ \vdots \end{pmatrix} \quad (1)$$

So we are interested in sparse/banded matrices.



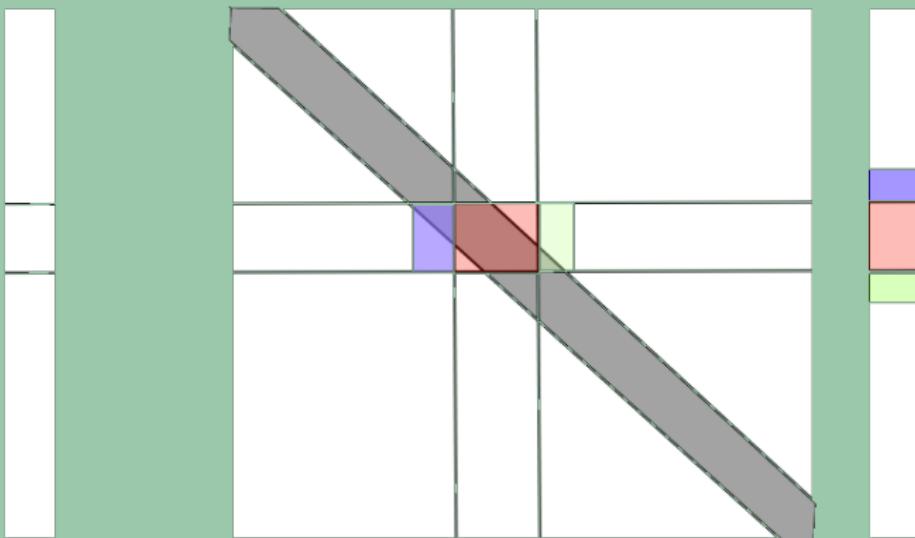
Most common operation: matrix vector product

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

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



We need a point-to-point mechanism:



each process with ones before/after it.

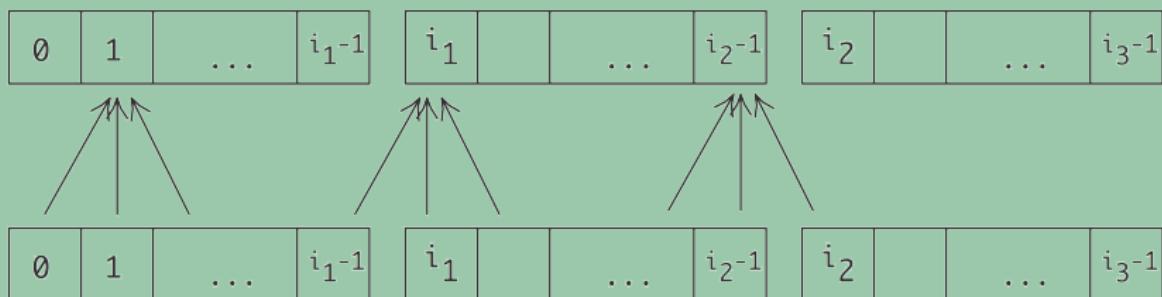


Array of numbers  $x_i : i = 0, \dots, N$   
compute

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

'owner computes'

This leads to communication:



so we need a point-to-point mechanism.

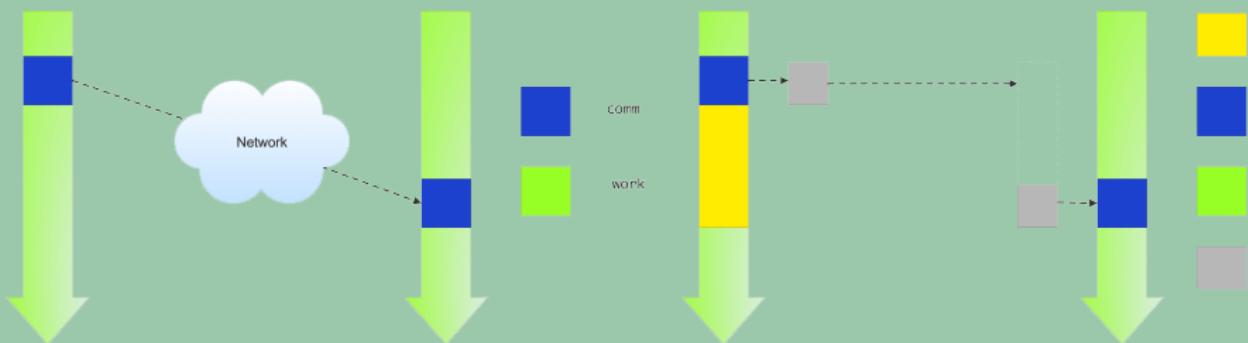


# Blocking communication



`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.



Exchange between two processes:

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

A subtlety.

This code may actually work:

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

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



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



(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.

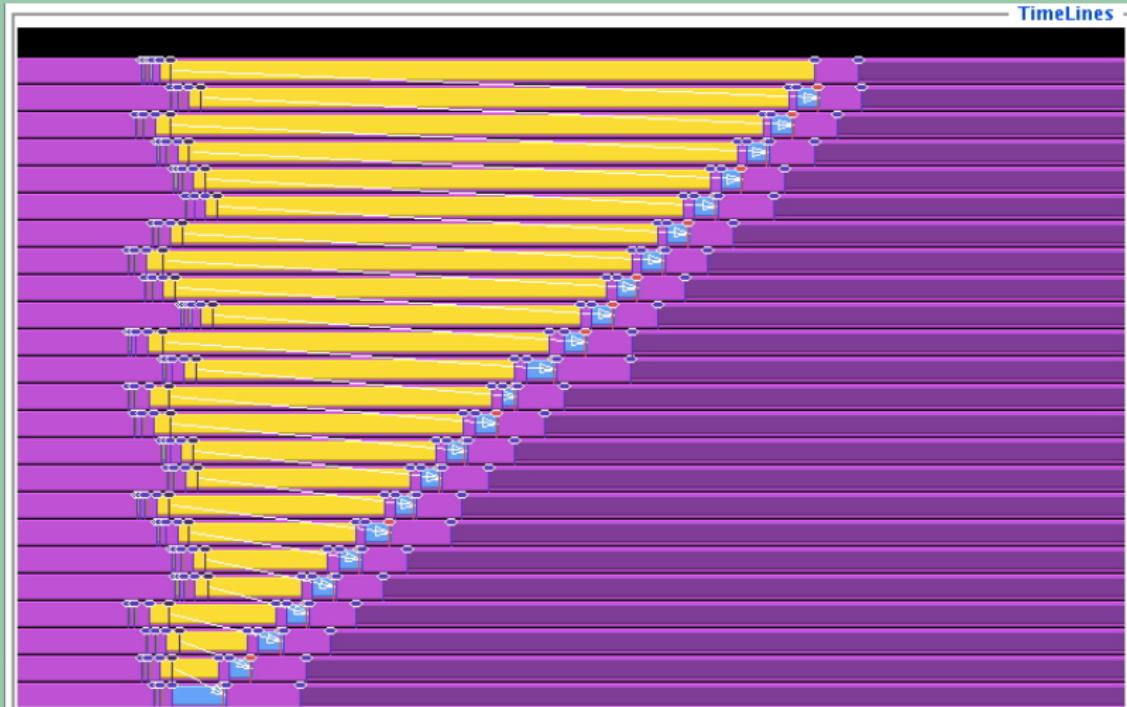
Including boundary conditions for first and last process, that becomes the following program:

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

TACC



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?



Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 159. If you don't have TAU, can you show this serialization behavior using timings, for instance running it on an increasing number of processes?

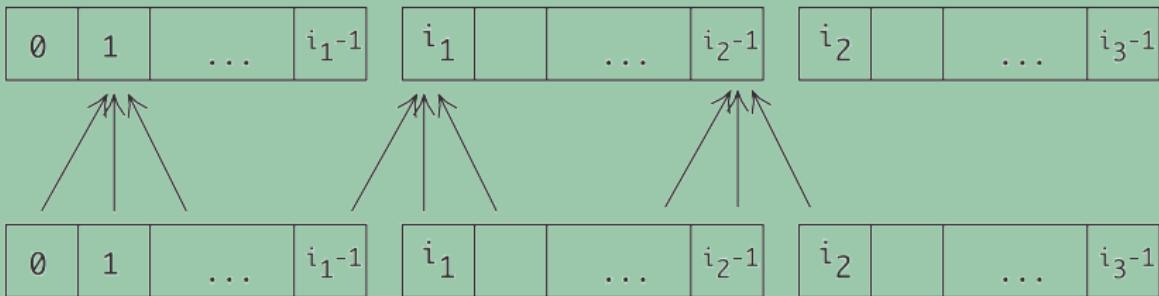


# Pairwise exchange



Take another look:

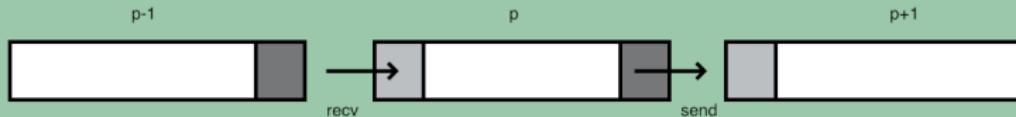
$$y_i = x_{i-1} + x_i + x_{i+1}: i = 1, \dots, N - 1$$



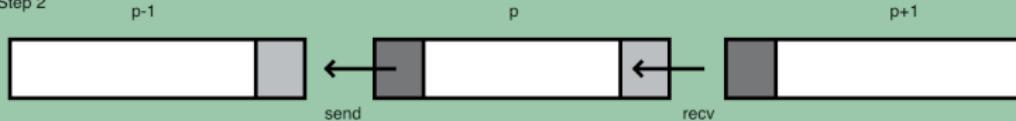
- One-dimensional data and linear process numbering;
- Operation between neighboring indices: communication between neighboring processes.



Step 1



Step 2



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!



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.)



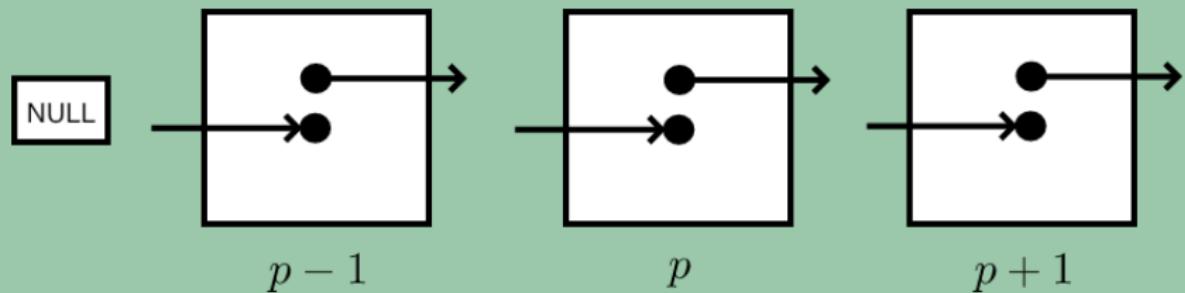
Name	Param name	Explanation	C type	F type
MPI_Sendrecv (				
sendbuf		initial address of send buffer	const void*	TYPE(*), DIMENSION(..)
sendcount		number of elements in send buffer	int	INTEGER
sendtype		type of elements in send buffer	MPI_Datatype	TYPE(MPI_Datatype)
dest		rank of destination	int	INTEGER
sendtag		send tag	int	INTEGER
recvbuf		initial address of receive buffer	void*	TYPE(*), DIMENSION(..)
recvcount		number of elements in receive buffer	int	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)
)				



```
1 template<typename T >
2 status mpl::communicator::sendrecv
3     ( const T & senddata, int dest,    tag sendtag,
4       T & recvdata, int source, tag recvtag
5     ) const
6     ( const T * senddata, const layout< T > & sendl, int dest,    tag
7       ↪sendtag,
8         T * recvdata, const layout< T > & recvl, int source, tag
9       ↪recvtag
10      ) const
11      ( iterT1 begin1, iterT1 end1, int dest,    tag sendtag,
12        iterT2 begin2, iterT2 end2, int source, tag recvtag
13      ) const
```



What does process  $p$  do?



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

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



Use `mpl::proc_null`.



The previous slide had:

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

Also possible:

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

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

But:

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



Revisit exercise 19 and solve it using `MPI_Sendrecv`.

If you have TAU installed, make a trace. Does it look different from the serialized send/recv code? If you don't have TAU, run your code with different numbers of processes and show that the runtime is essentially constant.



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

TACC



↔ transpose performed  
↔ no transpose needed

Odd-even transposition sort on 4 elements.



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 174.

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.



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.



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

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

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

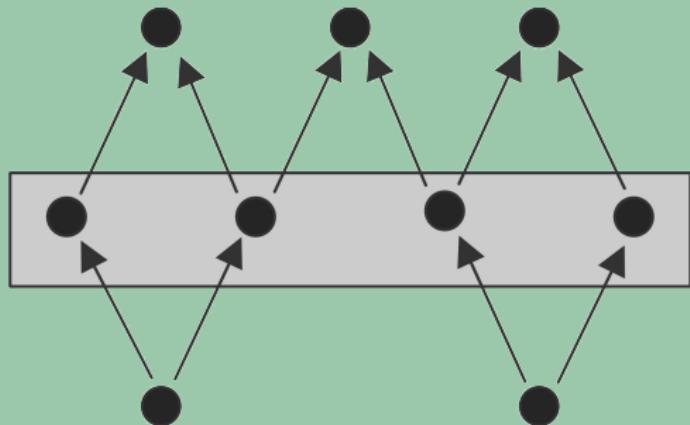
Food for thought: all quantities involved here are integers. Is it a good idea to use the integer datatype here?



# Irregular exchanges: non-blocking communication



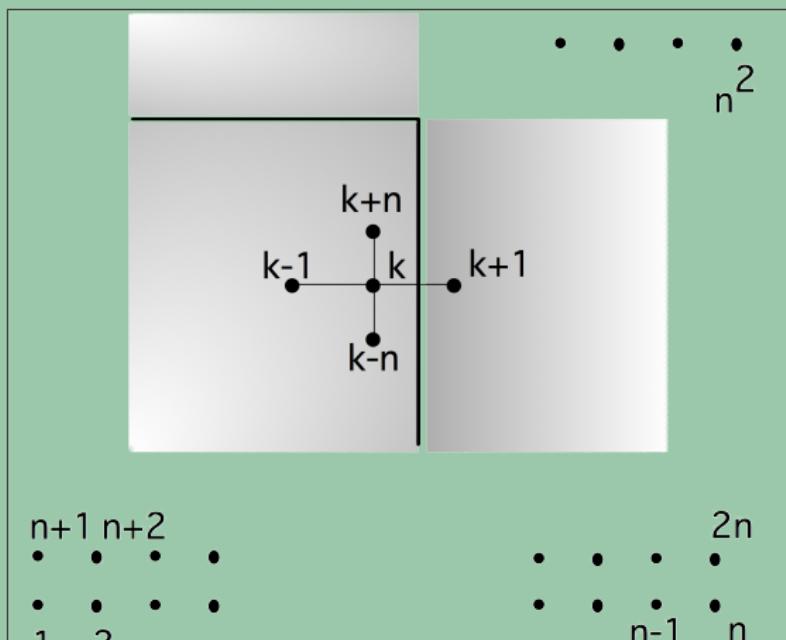
Graph operations:



## **Communicating other than in pairs**



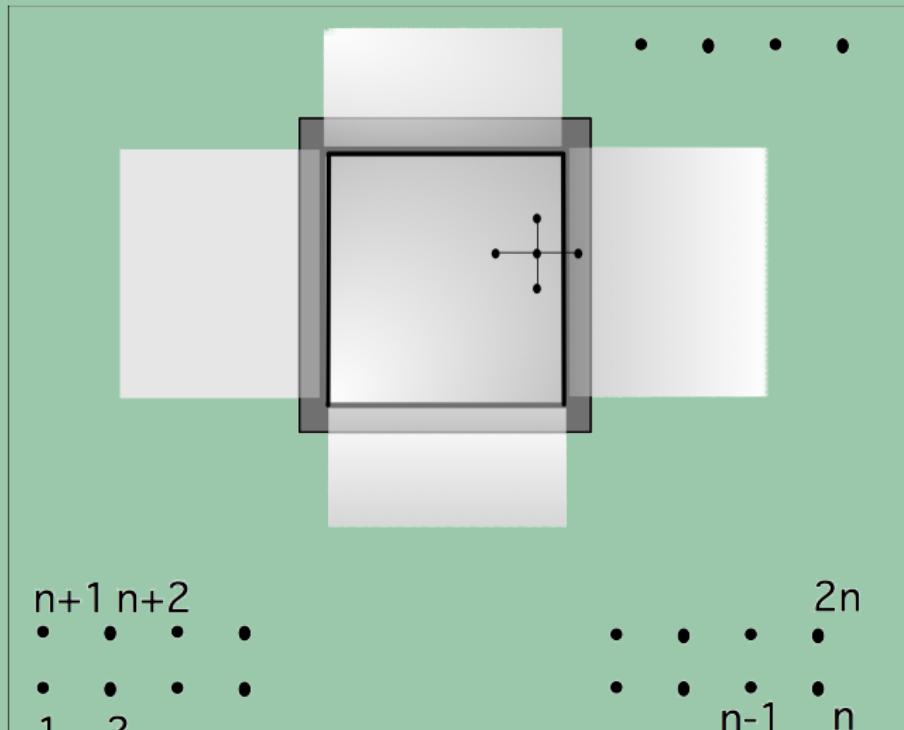
A difference stencil applied to a two-dimensional square domain, distributed over processes. A cross-process connection is indicated  $\Rightarrow$  complicated to express pairwise



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



The halo region of a process, induced by a stencil



- 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.



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

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



Very much like blocking MPI\_Send/MPI\_Recv:

```
1 int MPI_Isend(void *buf,
2     int count, MPI_Datatype datatype, int dest, int tag,
3     MPI_Comm comm, MPI_Request *request)
4 int MPI_Irecv(void *buf,
5     int count, MPI_Datatype datatype, int source, int tag,
6     MPI_Comm comm, MPI_Request *request)
```

Basic wait:

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

Most common way of waiting for completion:

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

- ignore status: MPI\_STATUSES\_IGNORE
- also MPI\_Wait, MPI\_Waitany, MPI\_Waitsome

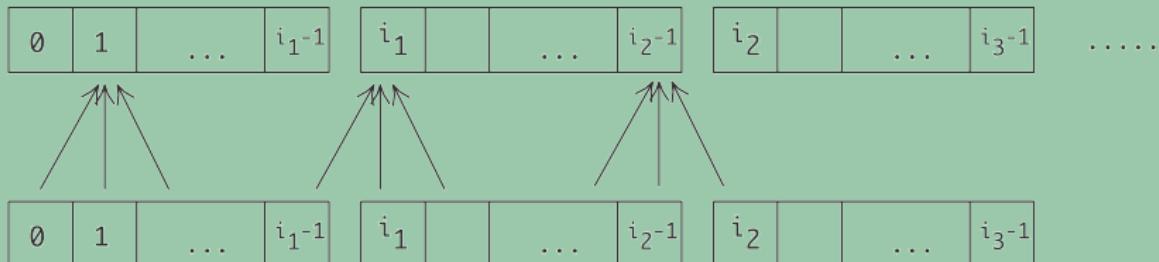


Now use nonblocking send/receive routines to implement the three-point averaging operation

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

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?)



- 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.



Blocking:

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

Non-blocking:

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

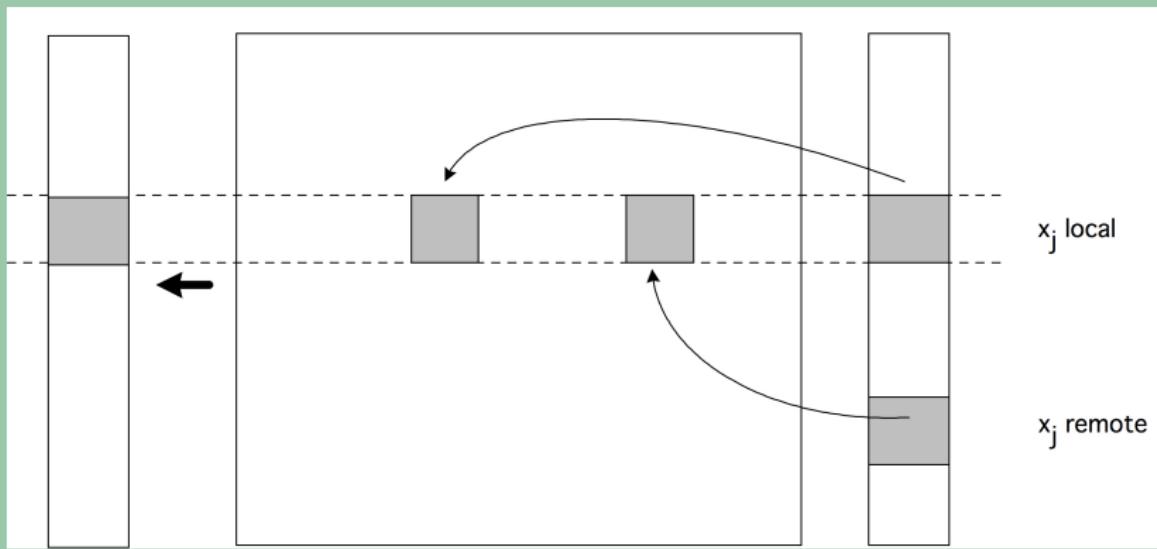


- Strictly one request/wait per `isend`/`irecv`:  
can not use one request for multiple simultaneous `isends`
- Some people argue:  
*Wait for the send is not necessary: if you wait for the receive,  
the message has arrived safely*  
This leads to memory leaks! The `wait` call deallocates the request  
object.

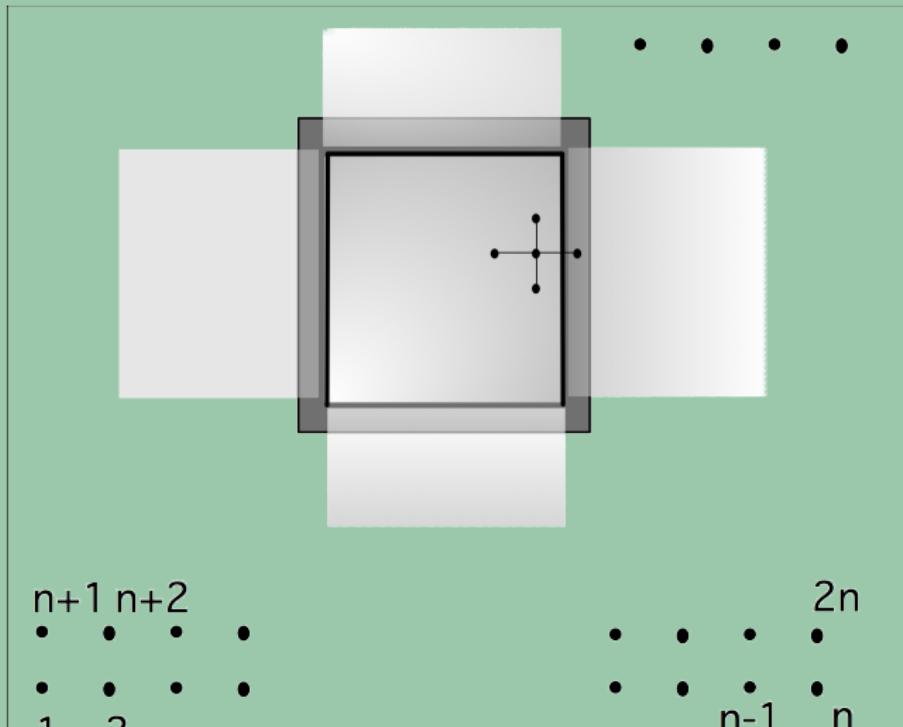


$$y \leftarrow Ax$$

and  $A, x, y$  all distributed:



Interior of a process domain can overlap with halo transfer:



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.



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

Write your code so that it can achieve latency hiding.



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

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



- 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



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



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



- `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.



Does this code deadlock?

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

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



Does this code deadlock?

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

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



Does this code deadlock?

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

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



Does this code deadlock?

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

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



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



# Intermediate topics



MPI basic concepts suffice for many applications. The Intermediate Topics section deals with more complicated data, process groups, file I/O, and the basics of one-sided communication.



# Part IV

## Derived Datatypes



In this section you will learn about derived data types.

Commands learned:

- `MPI_Type_contiguous`/vector/indexed/struct  
`MPI_Type_create_subarray`
- `MPI_Pack` / `MPI_Unpack`
- F90 types



# Discussion



All examples so far:

- contiguous buffer
- elements of single type

We need data structures with gaps, or heterogeneous types.

- Send real or imaginary parts out of complex array.
- Gather/scatter cyclicly.
- Send `struct` or `Type` data.

MPI allows for recursive construction of data types.



- Elementary types: built-in.
- Derived types: user-defined.
- Packed data: not really a datatype.



# Datatypes



C/C++	Fortran
MPI_CHAR	MPI_CHARACTER
MPI_UNSIGNED_CHAR	
MPI_SIGNED_CHAR	MPI_LOGICAL
MPI_SHORT	
MPI_UNSIGNED_SHORT	
MPI_INT	MPI_INTEGER
MPI_UNSIGNED	
MPI_LONG	
MPI_UNSIGNED_LONG	
MPI_FLOAT	MPI_REAL
MPI_DOUBLE	MPI_DOUBLE_PRECISION
MPI_LONG_DOUBLE	MPI_COMPLEX MPI_DOUBLE_COMPLEX



Create, commit, use, free:

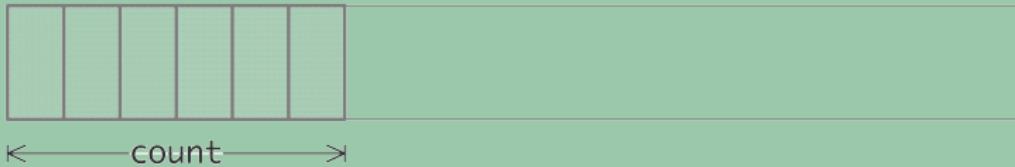
```
1 MPI_Datatype newtype;
2 MPI_Type_xxx( ... oldtype ... &newtype);
3 MPI_Type_commit ( &newtype );
4
5 // code using the new type
6
7 MPI_Type_free ( &newtype );

1 Type(MPI_Datatype) :: newtype ! F2008
2 Integer           :: newtype ! F90
```

The `oldtype` can be elementary or derived.  
Recursively constructed types.



```
1 int MPI_Type_contiguous(  
2     int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)
```



This one is indistinguishable from just sending `count` instances of the `old_type`.



# Example: non-contiguous data

TACC

Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

Logical:

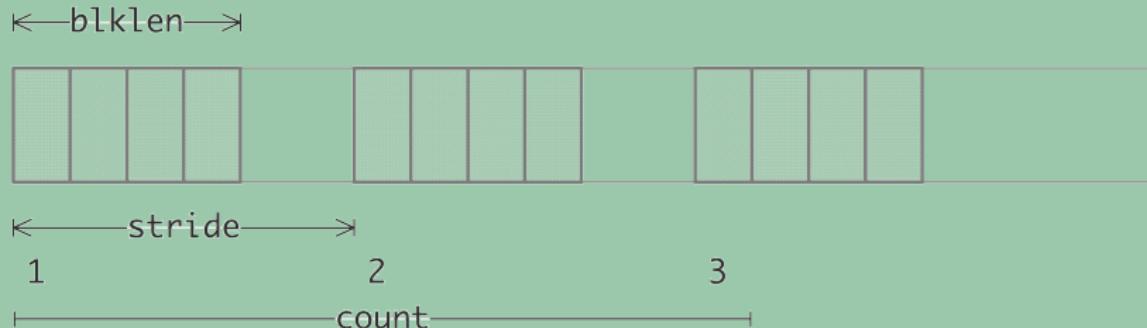
(1,1)	(1,2)	
(2,1)		
(3,1)		
▼	▼	

Physical:

(1,1)	(2,1)	(3,1)	...	(1,2)	...
-------	-------	-------	-----	-------	-----



```
1 int MPI_Type_vector(  
2     int count, int blocklength, int stride,  
3     MPI_Datatype old_type, MPI_Datatype *newtype_p  
4 );
```



Used to pick a regular subset of elements from an array.



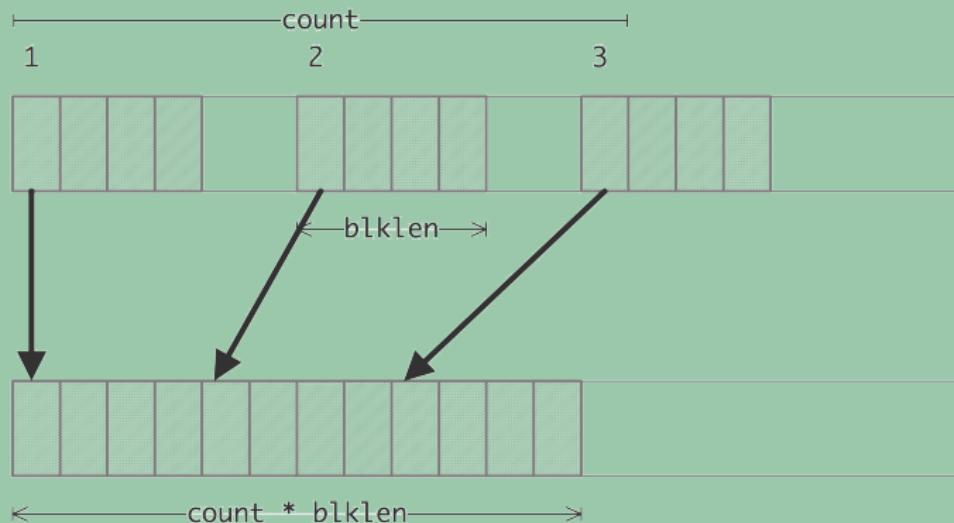
# Different send and receive types

TACC

Send and receive type can differ. Example:

Sender type: vector

receiver type: contiguous or elementary

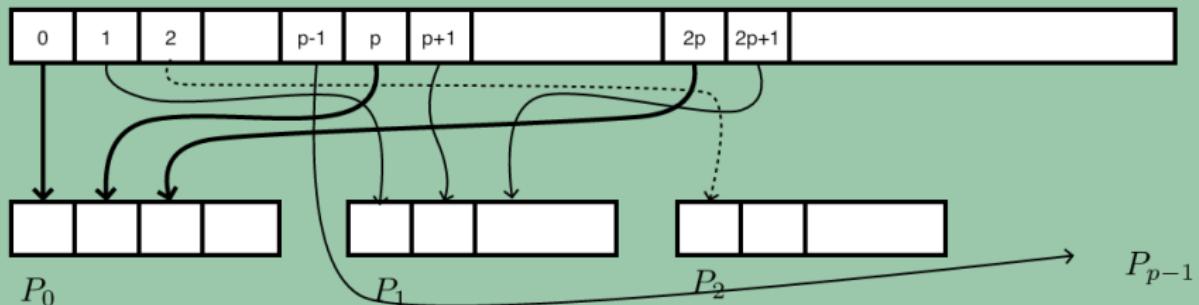


Receiver has no knowledge of the stride of the sender.



```
1 // vector.c
2 source = (double*) malloc(stride*count*sizeof(double));
3 target = (double*) malloc(count*sizeof(double));
4 MPI_Datatype newvectortype;
5 if (procno==sender) {
6     MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
7     MPI_Type_commit(&newvectortype);
8     MPI_Send(source,1,newvectortype,the_other,0,comm);
9     MPI_Type_free(&newvectortype);
10 } else if (procno==receiver) {
11     MPI_Status recv_status;
12     int recv_count;
13     MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
14             &recv_status);
15     MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
16     ASSERT(recv_count==count);
17 }
```





Sending strided data from process zero to all others



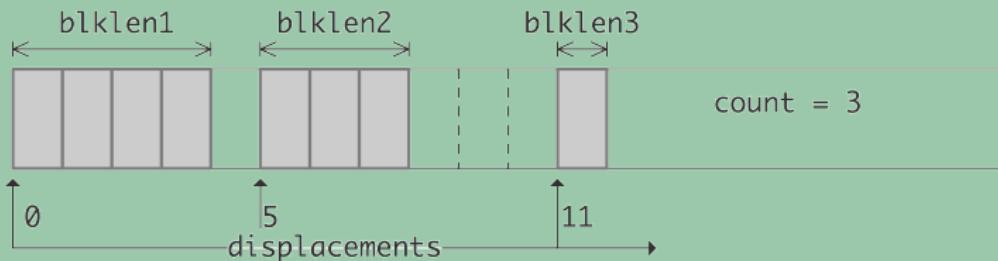
Let processor 0 have an array  $x$  of length  $10P$ , where  $P$  is the number of processors. Elements  $0, P, 2P, \dots, 9P$  should go to processor zero,  $1, P + 1, 2P + 1, \dots$  to processor 1, et cetera.

- Code this as a sequence of send/recv calls, using a vector datatype for the send, and a contiguous buffer for the receive.
- For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?
- For testing, define the array as  $x[i] = i$ .



Allocate a matrix on processor zero, using Fortran column-major storage.  
Using  $P$  sendrecv calls, distribute the rows of this matrix among the  
processors.





```
1 int MPI_Type_indexed(  
2     int count, int blocklens[], int displacements[],  
3     MPI_Datatype old_type, MPI_Datatype *newtype);
```



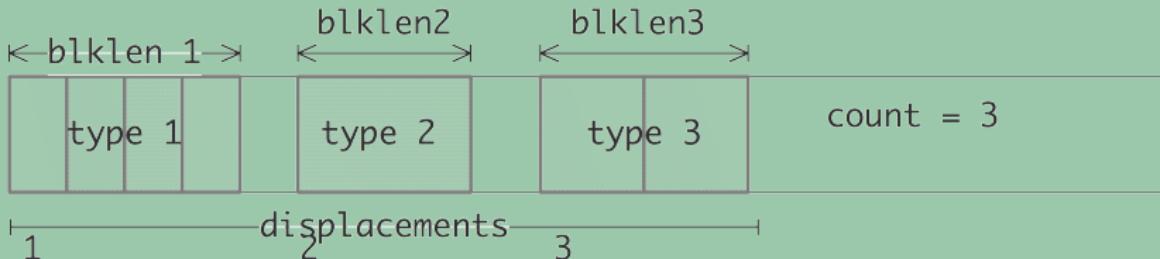
Similar to indexed but using byte offsets:  
explicit memory address.

Example usage scenario: send linked list.

Use `MPI_Get_address`



```
1 int MPI_Type_create_struct(  
2     int count, int blocklengths[], MPI_Aint displacements[],  
3     MPI_Datatype types[], MPI_Datatype *newtype);
```



This gets very tedious...



# Subarray type



Logical:



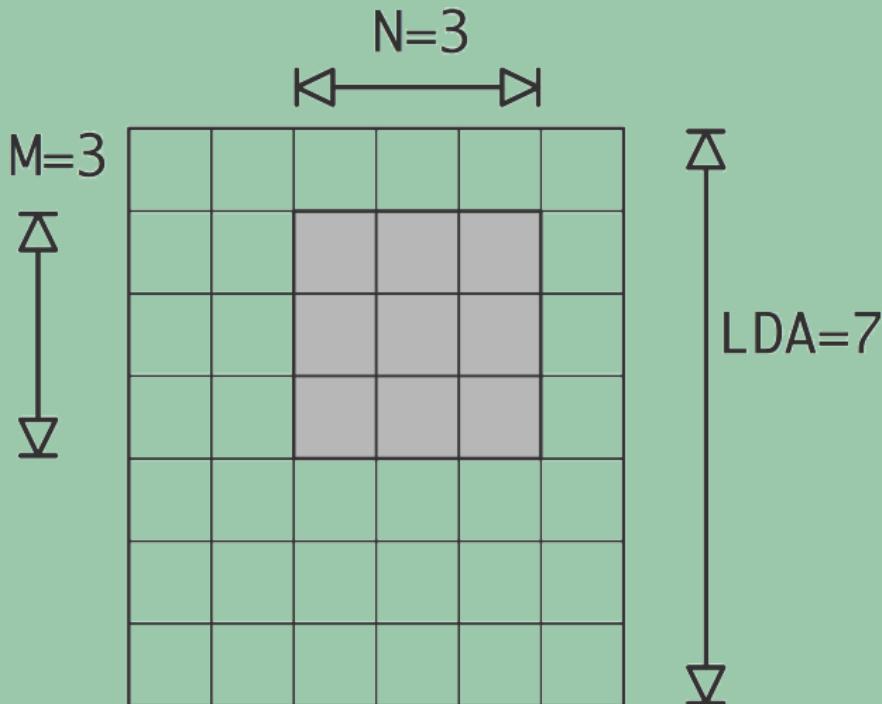
Physical:



- Location of first element
- Stride, blocksize



Three parameter description:



- Vector type is convenient for 2D subarrays,
- it gets tedious in higher dimensions.
- Better solution: `MPI_Type_create_subarray`

```
1 MPI_Type_create_subarray(  
2     ndims, array_of_sizes, array_of_subsizes,  
3     array_of_starts, order, oldtype, newtype)
```

Subtle: data does not start at the buffer start



Assume that your number of processors is  $P = Q^3$ , and that each process has an array of identical size. Use `MPI_Type_create_subarray` to gather all data onto a root process. Use a sequence of send and receive calls; `MPI_Gather` does not work here.

If you haven't started `idev` with the right number of processes, use

```
ibrun -np 27 cubegather
```

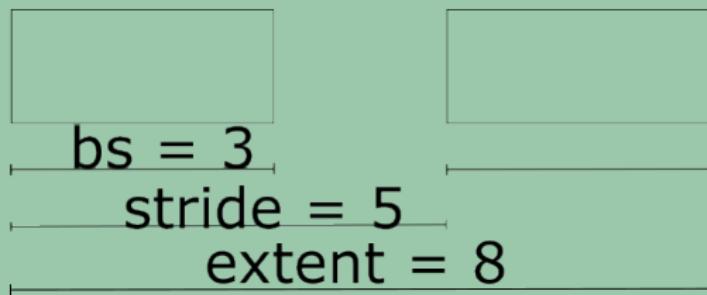
Normally you use `ibrun` without process count argument.



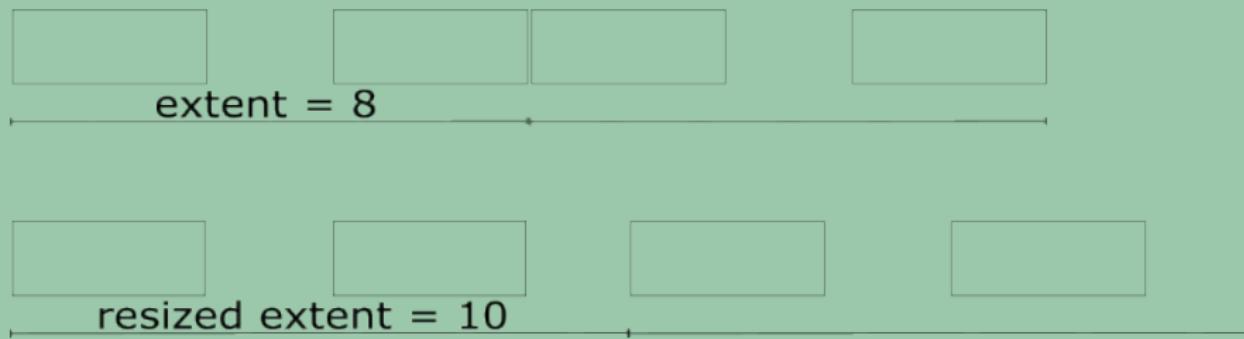
# Extent and resizing



Extent: ‘size’ of a type,  
especially useful for derived types.



Multiple derived types may not be what you intended  
extent resizing makes it artificially larger:



Use `MPI_Type_get_extent` to query extent  
note: parameters are measured in bytes.

```
1 MPI_Aint lb,asize;
2 MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
3 MPI_Type_commit(&newtype);
4 MPI_Type_get_extent(newtype,&lb,&asize);
5 ASSERT( lb==0 );
6 ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
7 MPI_Type_free(&newtype);
```



Send multiple derived types from

0 1 2 3 4 5 6 7 8 9 10

```
1 // vectorpadsend.c
2 for (int i=0; i<max_elements; i++) sendbuffer[i] = i;
3 MPI_Type_vector(count,blocklength,stride,MPI_INT,&stridetype);
4 MPI_Type_commit(&stridetype);
5 MPI_Send( sendbuffer,ntypes,stridetype, receiver,0, comm );
```

Receive as single block:

```
1 MPI_Recv( recvbuffer,max_elements,MPI_INT, sender,0, comm,&status );
2 int count; MPI_Get_count(&status,MPI_INT,&count);
3 printf("Receive %d elements:",count);
4 for (int i=0; i<count; i++) printf(" %d",recvbuffer[i]);
5 printf("\n");
```

giving an output of:

Receive 6 elements: 0 2 4 5 7 9



Extend the vector type with padding:

```
1 MPI_Type_get_extent(stridetype,&l,&e);
2 printf("Stride type l=%ld e=%ld\n",l,e);
3 e += ( stride-blocklength ) * sizeof(int);
4 MPI_Type_create_resized(stridetype,l,e,&paddedtype);
5 MPI_Type_get_extent(paddedtype,&l,&e);
6 printf("Padded type l=%ld e=%ld\n",l,e);
7 MPI_Type_commit(&paddedtype);
8 MPI_Send( sendbuffer,ntypes,paddedtype, receiver,0, comm );
```

giving:

Strided type l=0 e=20

Padded type l=0 e=24

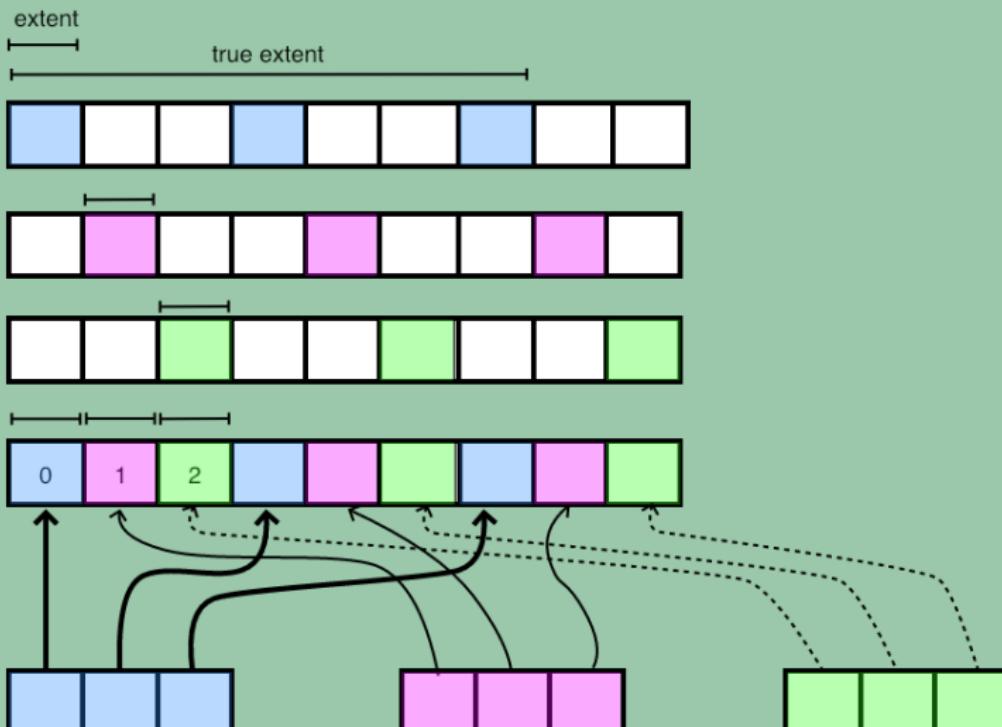
Receive 6 elements: 0 2 4 6 8 10



# Extent resizing: shrinking

TACC

Elements are placed at distance equal to extent:



Change the `stridesend` code to use a scatter call, rather than a sequence of sends.



The ‘subarray’ type:  
data does not start at the start of the type.

`MPI_Type_get_true_extent` returns non-zero lower bound.

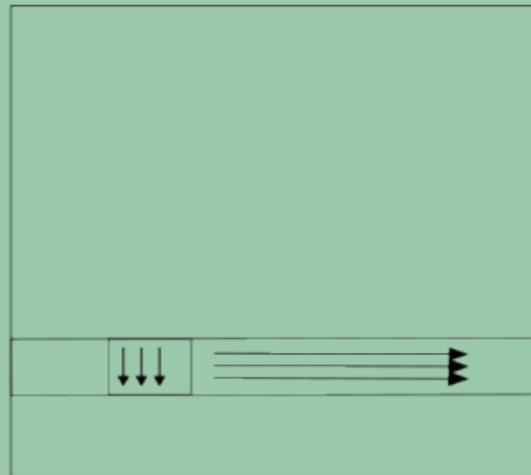
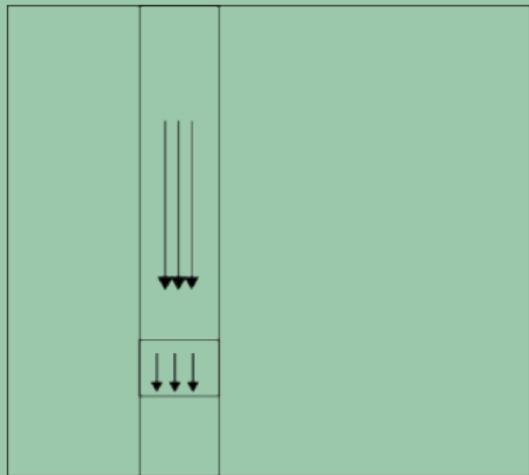


- Basic block of FFT
- Before: Each process stores a block column,
- After: Each process stores a block row



in a picture

TACC



Fill in the missing bits of the skeleton code.



# Packed data



```
1 int MPI_Pack(
2     void *inbuf, int incount, MPI_Datatype datatype,
3     void *outbuf, int outcount, int *position,
4     MPI_Comm comm);
5
6 int MPI_Unpack(
7     void *inbuf, int insize, int *position,
8     void *outbuf, int outcount, MPI_Datatype datatype,
9     MPI_Comm comm);
```



# Example

TACC

```
1 if (procno==sender) {  
2     position = 0;  
3     MPI_Pack(&nse...nsends,1,MPI_INT,  
4                 buffer,buflen,&position,comm);  
5     for (int i=0; i<nse...nsends; i++) {  
6         double value = rand()/(double)RAND_MAX;  
7         printf(" [%d] pack %e\n",procno,value);  
8         MPI_Pack(&value,1,MPI_DOUBLE,  
9                 buffer,buflen,&position,comm);  
10    }  
11    MPI_Pack(&nse...nsends,1,MPI_INT,  
12                 buffer,buflen,&position,comm);  
13    MPI_Send(buffer,position,MPI_PACKED,other,0,comm);  
14 } else if (procno==receiver) {  
15     int irecv_value;  
16     double xrecv_value;  
17     MPI_Recv(buffer,buflen,MPI_PACKED,other,0,  
18                 comm,MPI_STATUS_IGNORE);  
19     position = 0;  
20     MPI_Unpack(buffer,buflen,&position,  
21                 &nse...nsends,1,MPI_INT,comm);  
22     for (int i=0; i<nse...nsends; i++) {  
23         MPI_Unpack(buffer.buflen.
```



# Part V

## Communicator manipulations



In this section you will learn about various subcommunicators.

Commands learned:

- `MPI_Comm_dup`, discussion of library design
- `MPI_Comm_split`
- discussion of groups
- discussion of inter/intra communicators.



Simultaneous groups of processes, doing different tasks, but loosely interacting:

- Simulation pipeline: produce input data, run simulation, post-process.
- Climate model: separate groups for air, ocean, land, ice.
- Quicksort: split data in two, run quicksort independently on the halves.
- Process grid: do broadcast in each column.

New communicators are formed recursively from `MPI_COMM_WORLD`.



Simplest new communicator: identical to a previous one.

```
1 int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

This is useful for library writers:

```
1 MPI_Isend(...); MPI_Irecv(...);
2 // library call
3 MPI_Waitall(...);
```

- Naively, the library can 'catch' the user messages.
- With a duplicate communicator there is no confusion:  
user and library both have their own 'context' for their messages.



**missing snippet catchmain**



Library internally has messages

TACC

**missing snippet catchcalls**



**missing snippet wrongcatchlib**



**missing snippet rightcatchlib**



Split a communicator in multiple disjoint others.

Give each process a ‘color’, group processes by color:

```
1 int MPI_Comm_split(MPI_Comm comm, int color, int key,  
2 MPI_Comm *newcomm)
```

(key determines ordering: use rank unless you want special effects)



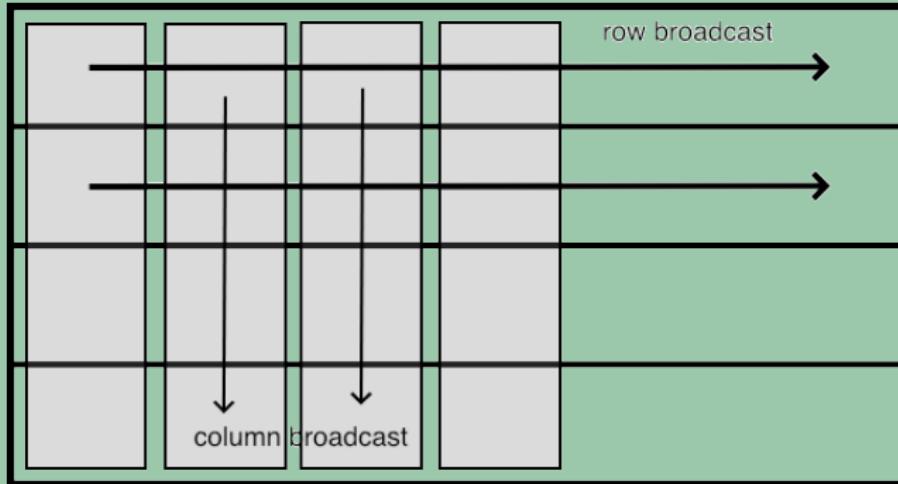
Simulate a process grid

create subcommunicator per column (or row)  
assume processes numbered by rows

```
1 MPI_Comm_rank( MPI_COMM_WORLD, &procno );
2 proc_i = procno / proc_column_length;
3 proc_j = procno % proc_column_length;
4
5 MPI_Comm column_comm;
6 MPI_Comm_split( MPI_COMM_WORLD, proc_j, procno, &column_comm );
7
8 MPI_Bcast( data, ... column_comm );
```

Food for thought: there are many columns, but only one column\_comm variable. Why?





Row and column broadcasts in subcommunicators



Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.

In the row and column communicator, compute the rank. For instance, on a  $2 \times 3$  processor grid you should find:

Global ranks: Ranks in row: Ranks in column:

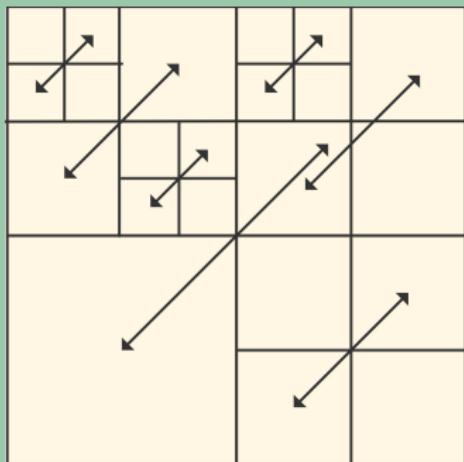
0	1	2	0	1	2	0	0	0
3	4	5	0	1	2	1	1	1

Check that the rank in the row communicator is the column number, and the other way around.

Run your code on different number of processes, for instance a number of rows and columns that is a power of 2, or that is a prime number. This is one occasion where you could use `ibrun -np 9`; normally you would *never* put a processor count on `ibrun`.



Implement a recursive algorithm for matrix transposition:



- Swap blocks (1, 2) and (2, 1); then
- Divide the processors into four subcommunicators, and apply this algorithm recursively on each;
- If the communicator has only one process, transpose the matrix in place.



- `MPI_Comm_split_type` splits into communicators of same type.
- MPI-3: only `MPI_COMM_TYPE_SHARED` splitting by shared memory.
- MPI-4: `MPI_COMM_TYPE_HW_GUIDED` split using an info value from `MPI_Get_hw_resource_types`.

```
1 // commssplittype.c
2 MPI_Info info;
3 MPI_Comm_split_type
4     (MPI_COMM_WORLD,
5      MPI_COMM_TYPE_SHARED,
6      procno,info,&sharedcomm);
7 MPI_Comm_size
8     (sharedcomm,&new_nprocs);
9 MPI_Comm_rank
10    (sharedcomm,&new_procno);
```



- Communicators so far are of *intra-communicator* type.
- Bridge between two communicators: *inter-communicator*.
- Example: communicator with newly spawned processes





Illustration of ranks in an inter-communicator setup

```
1 // intercomm.c
2 MPI_Comm intercomm;
3 MPI_Intercomm_create
4     /* local_comm:          */ split_half_comm,
5     /* local_leader:        */ local_leader_in_inter_comm,
6     /* peer_comm:           */ MPI_COMM_WORLD,
7     /* remote_peer_rank:   */ global_rank_of_other_leader,
8     /* tag:                 */ inter_tag,
9     /* newintercomm:         */ &intercomm );
```



- Two local communicators
- The ‘peer’ communicator that contains them
- Leaders in each of them
- An inter-communicator over the leaders.



- `MPI_Intercomm_create`: create
- `MPI_Comm_get_parent`: the other leader (see process management)
- `MPI_Comm_remote_size`, `MPI_Comm_remote_group`: query the other communicator
- `MPI_Comm_test_inter`: is this an inter or intra?



- Non-disjoint subcommunicators through process groups.
- Process topologies: cartesian and graph.  
There will also be a section about this, later.



# Cartesian topologies



# Cartesian decomposition

TACC

## Code:

```
1 // cartdims.c
2 int *dimensions = (int*)
     ↪malloc(dim*sizeof(int));
3 for (int idim=0; idim<dim; idim++)
4     dimensions[idim] = 0;
5 MPI_Dims_create(nprocs, dim, dimensions);
```

## Output:

```
1 mpicc -o cartdims
         ↪cartdims.o
2 Cartesian grid size: 3
         ↪dim: 1
3   3
4   Cartesian grid size: 3
         ↪dim: 2
5   3 x 1
6   Cartesian grid size: 4
         ↪dim: 1
7   4
8   Cartesian grid size: 4
         ↪dim: 2
9   2 x 2
10  Cartesian grid size: 4
         ↪dim: 3
11  2 x 2 x 1
12  Cartesian grid size:
         ↪12 dim: 1
13  12
14  Cartesian grid size:
```



```
1 MPI_Comm cart_comm;
2 int *periods = (int*) malloc(dim*sizeof(int));
3 for ( int id=0; id<dim; id++ ) periods[id] = 0;
4 MPI_Cart_create
5   ( comm,dim,dimensions,periods,
6     0,&cart_comm );
7
8 int dim;
9 MPI_Cartdim_get( cart_comm,&dim );
10 int *dimensions = (int*) malloc(dim*sizeof(int));
11 int *periods    = (int*) malloc(dim*sizeof(int));
12 int *coords     = (int*) malloc(dim*sizeof(int));
13 MPI_Cart_get( cart_comm,dim,dimensions,periods,coords );
```



```
1 // cart.c
2 MPI_Comm comm2d;
3 int periodic[ndim]; periodic[0] = periodic[1] = 0;
4 MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
5 if (comm2d==MPI_COMM_NULL) {
6     printf("Process %d not included\n",procno);
7 } else {
8     MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
9     MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
10    printf("I am %d: (%d,%d); originally %d\n",
11           rank_2d,coord_2d[0],coord_2d[1],procno);
```



```
1 // cartcoord.c
2 for ( int id=0; id<dim; id++)
3     periods[id] = id==0 ? 1 : 0;
4 MPI_Cart_create
5   ( comm, dim, dimensions, periods,
6     0, &period_comm );
```

Code:

```
1 int pred,succ;
2 MPI_Cart_shift
3   (period_comm,/* dim: */ 0,/* up:
4     ↪*/ 1,
5      &pred,&succ);
5 printf("periodic dimension 0:\n  src=%d,
6     ↪tgt=%d\n",
7      pred,succ);
7 MPI_Cart_shift
8   (period_comm,/* dim: */ 1,/* up:
9     ↪*/ 1,
10    &pred,&succ);
10 printf("non-periodic dimension 1:\n
11     ↪src=%d, tgt=%d\n",
12     -1, -1);
```

Output:

```
1 Grid of size 6 in 3
2   ↪dimensions:
3     3 x 2 x 1
4 Shifting process 0.
5 periodic dimension 0:
6   src=4, tgt=2
7 non-periodic dimension
8   ↪1:
9   src=-1, tgt=1
```



## Code:

```
1 MPI_Cart_sub(
2     ↪period_comm,remain,&hyperplane );
3 if (procno==0) {
4     MPI_Topo_test( hyperplane,&topo_type );
5     MPI_Cartdim_get( hyperplane,&hyperdim );
6     printf("hyperplane has dimension %d,
7         ↪type %d\n",
8             hyperdim,topo_type);
9     MPI_Cart_get(
10        ↪hyperplane,dim,dims,period,coords );
11     printf(" periodic: ");
12     for (int id=0; id<2; id++)
13         printf("%d,",period[id]);
14     printf("\n");
```

## Output:

```
1 Grid of size 6 in 3
2     ↪dimensions:
3         3 x 2 x 1
4     hyperplane has
5         ↪dimension 2,
6             ↪type 2
7     periodic: 1,0,
```



# Part VI

## MPI File I/O



This section discusses parallel I/O. What is the problem with regular I/O in parallel?

Commands learned:

- `MPI_File_open`/`write`/`close` and variants
- parallel file pointer routines: `MPI_File_set_view`/`write_at`



- Multiple process reads from one file: no problem.
- Multiple writes to one file: big problem.
- Everyone writes to separate file: stress on the file system, and requires post-processing.



- Part of MPI since MPI-2
- Joint creation of one file from bunch of processes.
- You could also use hdf5, netcdf, silo ...



```
1 MPI_File mpifile;
2 MPI_File_open(comm,"blockwrite.dat",
3                 MPI_MODE_CREATE | MPI_MODE_WRONLY,MPI_INFO_NULL,
4                 &mpifile);
5 if (procno==0) {
6     MPI_File_write
7         (mpifile,output_data,nwords,MPI_INT,MPI_STATUS_IGNORE);
8 }
9 MPI_File_close(&mpifile);

1 type(MPI_File) :: mpifile ! F08
2 integer          :: mpifile ! F90
```



```
1 MPI_File_write_at  
2     (mpifile,offset,output_data,nwords,  
3      MPI_INT,MPI_STATUS_IGNORE);
```

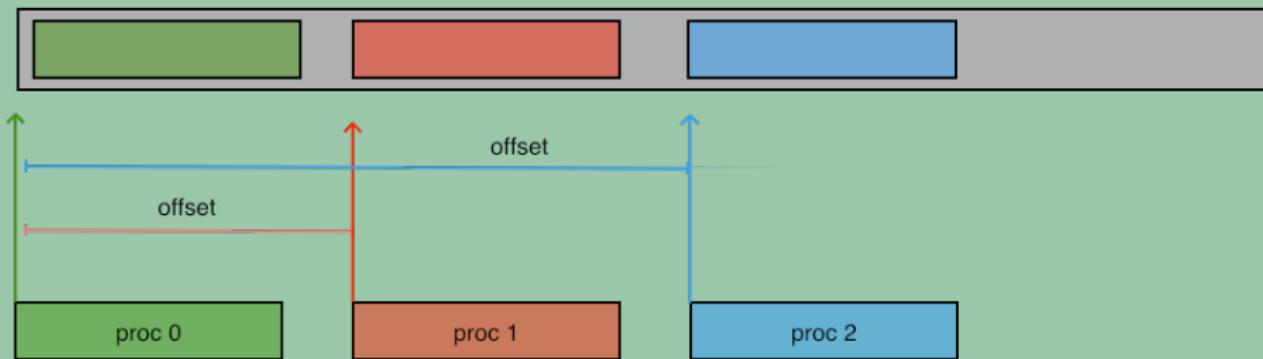
or

```
1 MPI_File_set_view  
2     (mpifile,  
3      offset,datatype,  
4      MPI_INT,"native",MPI_INFO_NULL);  
5 MPI_File_write // no offset, we have a view  
6     (mpifile,output_data,nwords,MPI_INT,MPI_STATUS_IGNORE);
```

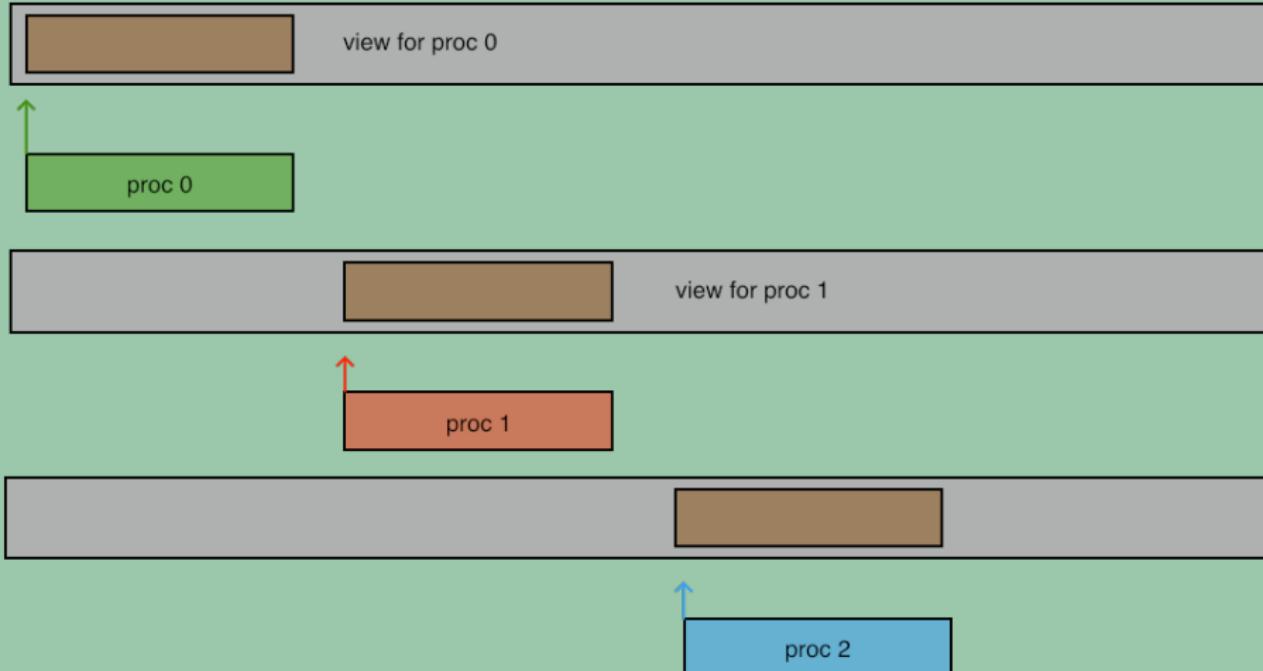


# Write at an offset

TACC

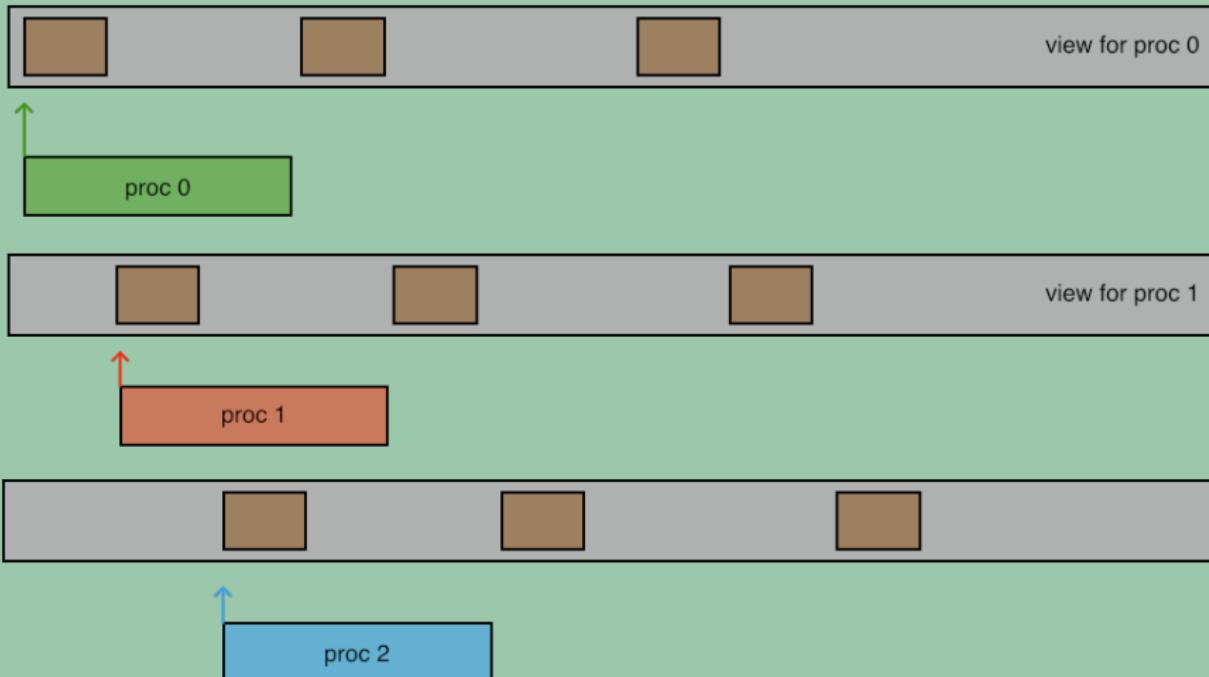


# Write to a view



# Write to a view

TACC



Made



The given code works for one writing process. Compute a unique offset for each process (in bytes!) so that all the local arrays are placed in the output file in sequence.



Solve the previous exercise by using `MPI_File_write` (that is, without offset), but by using `MPI_File_set_view` to specify the location.



Now write the local arrays cyclically to the file: with 5 processes and 3 elements per process the file should contain

```
1 4 7 10 13 | 2 5 8 11 14 | 3 6 9 12 15
```

Do this by defining a vector derived type and setting that as the file view.



## Part VII

### One-sided communication



This section concerns one-sided operations, which allows ‘shared memory’ type programming. (Actual shared memory later.)

Commands learned:

- `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
- Window commands: `MPI_Win_create`, `MPI_Win_allocate`
- Active target synchronization `MPI_Win_fence`
- `MPI_Win_post`/`wait`/`start`/`complete`
- Passive target synchronization `MPI_Win_lock` / `MPI_Win_unlock`
- Atomic operations: `MPI_Fetch_and_op`



# Basic mechanisms



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

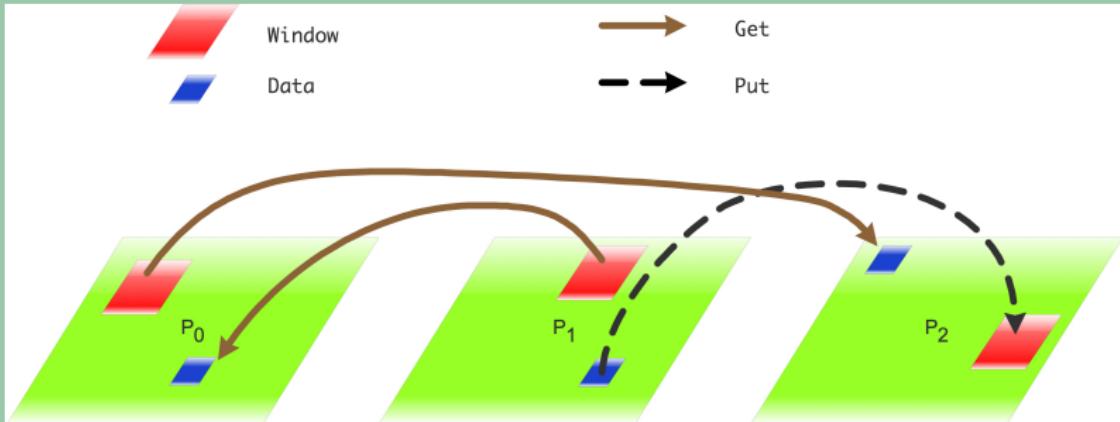
- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbors. Uncertain when this happens.
- Other irregular data structures: distributed hash tables.



```
1  x = f();  
2  p = hash(x);  
3  MPI_Send( x, /* to: */ p );
```

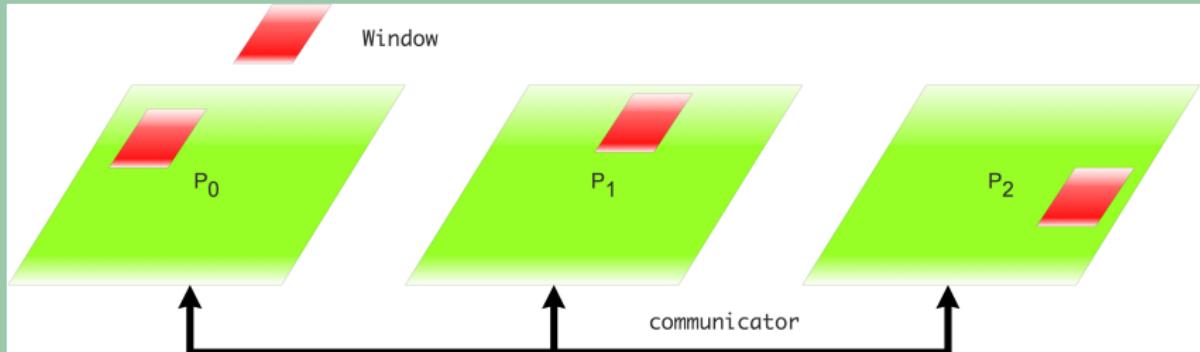
Problem: how does *p* know to post a receive,  
and how does everyone else know not to?





- 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.





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

- `size`: in bytes
- `disp_unit`: `sizeof(type)`

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



Instead of passing buffer, let MPI allocate with `MPI_Win_allocate` and return the buffer pointer:

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

can use dedicated fast memory.



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

```
1 MPI_Win_fence(MPI_MODE_NOPRECEDE, win);  
2 if (procno==producer)  
3     MPI_Put( /* operands */, win);  
4 MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Second fence indicates that one-sided communication is concluded:  
target knows that data has been put.

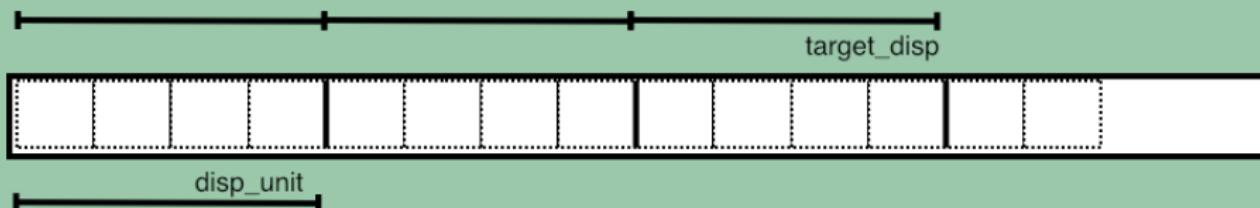


Name	Param name	Explanation	C type	F type
MPI_Put (				
MPI_Put_c (				
origin_addr		initial address of origin buffer	const void*	TYPE(*), DIMENSION(..)
origin_count		number of entries in origin buffer	[ int MPI_Count ]	INTEGER
origin_datatype		datatype of each entry in origin buffer	MPI_Datatype	TYPE(MPI_Datatype)
target_rank		rank of target	int	INTEGER
target_disp		displacement from start of window to target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
target_count		number of entries in target buffer	[ int MPI_Count ]	INTEGER
target_datatype		datatype of each entry in target buffer	MPI_Datatype	TYPE(MPI_Datatype)
win		window object used for communication	MPI_Win	TYPE(MPI_Win)
)				



Location to write:

$$\text{window\_base} + \text{target\_disp} \times \text{disp\_unit}.$$



Revisit exercise 19 and solve it using `MPI_Put`.



Write code where:

- process 0 computes a random number  $r$
- if  $r < .5$ , zero writes in the window on 1;
- if  $r \geq .5$ , zero writes in the window on 2.



Replace `MPI_Win_create` by `MPI_Win_allocate`.



- **MPI\_Get** is converse of **MPI\_Put**. Like Recv, but no status argument.
- **MPI\_Accumulate** is a Put plus a reduction on the result: multiple accumulate calls in one epoch well-defined.  
Can use any predefined **MPI\_Op** (not user-defined) or **MPI\_REPLACE**.



Name	Param name	Explanation	C type	F type
MPI_Get (				
MPI_Get_c (				
origin_addr		initial address of origin buffer	void*	TYPE(*), DIMENSION(..)
origin_count		number of entries in origin buffer	[ int MPI_Count	INTEGER
origin_datatype		datatype of each entry in origin buffer	MPI_Datatype	TYPE(MPI_Datatype)
target_rank		rank of target	int	INTEGER
target_disp		displacement from window start to the beginning of the target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
target_count		number of entries in target buffer	[ int MPI_Count	INTEGER
target_datatype		datatype of each entry in target buffer	MPI_Datatype	TYPE(MPI_Datatype)
win		window object used for communication	MPI_Win	TYPE(MPI_Win)
)				



Name	Param name	Explanation	C type	F type
MPI_Accumulate (				
MPI_Accumulate_c (				
origin_addr		initial address of buffer	const void*	TYPE(*), DIMENSION(..)
origin_count		number of entries in buffer	[ int MPI_Count ]	INTEGER
origin_datatype		datatype of each entry	MPI_Datatype	TYPE(MPI_Datatype)
target_rank		rank of target	int	INTEGER
target_disp		displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
target_count		number of entries in target buffer	[ int MPI_Count ]	INTEGER
target_datatype		datatype of each entry in target buffer	MPI_Datatype	TYPE(MPI_Datatype)
op		reduce operation	MPI_Op	TYPE(MPI_Op)
win		window object	MPI_Win	TYPE(MPI_Win)
)				



# Ordering and synchronization



Already mentioned active target synchronization:  
the target indicates the start/end of an epoch.

Simplest mechanism: `MPI_Win_fence`, collective.

After the closing fence, buffers have been sent / windows have been updated.



Ordering is often undefined:

- No ordering of Get and Put/Accumulate operations
- No ordering of multiple Puts. Use Accumulate.

The following operations are well-defined inside one epoch:

- Instead of multiple Put operations, use Accumulate with `MPI_REPLACE`.
- `MPI_Get_accumulate` with `MPI_NO_OP` is safe.
- Multiple Accumulate operations from one origin are ordered by default.



Implement a shared counter:

- One process maintains a counter;
- Iterate: all others at random moments update this counter.
- When the counter is no longer positive, everyone stops iterating.

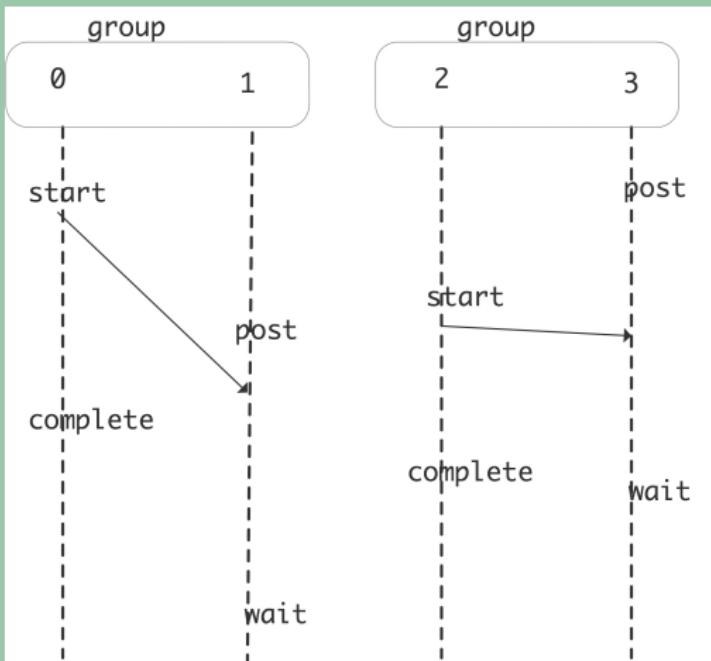
The problem here is data synchronization: does everyone see the counter the same way?



# A second active synchronization

TACC

Use `MPI_Win_post`, `MPI_Win_wait`, `MPI_Win_start`, `MPI_Win_complete` calls



More fine-grained than fences



# Passive target synchronization



Lock a window on the target:

```
1 MPI_Win_lock  
2     (int locktype, int rank, int assert, MPI_Win win)  
3 MPI_Win_unlock  
4     (int rank, MPI_Win win)
```

with types: MPI\_LOCK\_SHARED MPI\_LOCK\_EXCLUSIVE



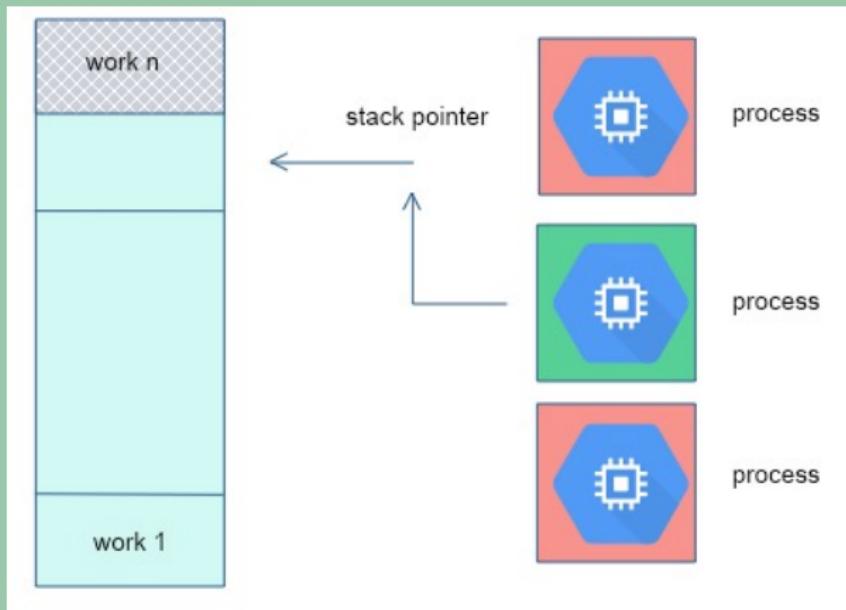
MPI-1/2 lacked tools for race condition-free one-sided communication.  
These have been added in MPI-3.



# Emulating shared memory with one-sided communication

- One process stores a table of work descriptors, and a ‘stack pointer’ stating how many there are.
- Each process reads the pointer, reads the corresponding descriptor, and decrements the pointer; and
- A process that has read a descriptor then executes the corresponding task.
- Non-collective behavior: processes only take a descriptor when they are available.





- One process has a counter, which models the shared memory;
- Each process, if available, reads the counter; and
- ... decrements the counter.
- No actual work: random decision if process is available.



# Shared memory problems: what is a race condition?

Race condition: outward behavior depends on timing/synchronization of low-level events.  
In shared memory associated with shared data.

Example:

```
Init: I=0
process 1: I=I+2
process 2: I=I+3
```

scenario 1.	scenario 2.	scenario 3.
$I = 0$		
read I = 0 local I = 2 write I = 2	read I = 0 local I = 3 write I = 3	read I = 0 local I = 2 write I = 3
		read I = 2 local I = 5 write I = 5
$I = 3$	$I = 2$	$I = 5$

(In MPI, the read/write would be **MPI\_Get** / **MPI\_Put** calls)



```
1 // countdownput.c
2 MPI_Win_fence(0,the_window);
3 int counter_value;
4 MPI_Get( &counter_value,1,MPI_INT,
5          counter_process,0,1,MPI_INT,
6          the_window);
7 MPI_Win_fence(0,the_window);
8 if (i_am_available) {
9     int decrement = -1;
10    counter_value += decrement;
11    MPI_Put
12      ( &counter_value,    1,MPI_INT,
13        counter_process,0,1,MPI_INT,
14        the_window);
15 }
16 MPI_Win_fence(0,the_window);
```



- The multiple `MPI_Put` calls conflict.
- Code is correct if in each iteration there is only one writer.
- Question: In that case, can we take out the middle fence?
- Question: what is wrong with

```
1 MPI_Win_fence(0,the_window);
2 if (i_am_available) {
3     MPI_Get( &counter_value, ... )
4     MPI_Win_fence(0,the_window);
5     MPI_Put( ... )
6 }
7 MPI_Win_fence(0,the_window);

?
```



```
1 // countdownacc.c
2 MPI_Win_fence(0,the_window);
3 int counter_value;
4 MPI_Get( &counter_value,1,MPI_INT,
5         counter_process,0,1,MPI_INT,
6         the_window);
7 MPI_Win_fence(0,the_window);
8 if (i_am_available) {
9     int decrement = -1;
10    MPI_Accumulate
11      ( &decrement,           1,MPI_INT,
12        counter_process,0,1,MPI_INT,
13        MPI_SUM,
14        the_window);
15 }
16 MPI_Win_fence(0,the_window);
```



- `MPI_Accumulate` is atomic, so no conflicting writes.
- What is the problem?
- Answer: Processes are not reading unique `counter_value` values.
- Conclusion: Read and update need to come together:  
read unique value and immediately update.

Atomic ‘get-and-set-with-no-one-coming-in-between’:

`MPI_Fetch_and_op` / `MPI_Get_accumulate`.

Former is simple version: scalar only.



Name	Param name	Explanation	C type	F type
MPI_Fetch_and_op (				
origin_addr		initial address of buffer	const void*	TYPE(*), DIMENSION(..)
result_addr		initial address of result buffer	void*	TYPE(*), DIMENSION(..)
datatype		datatype of the entry in origin, result, and target buffers	MPI_Datatype	TYPE(MPI_Datatype)
target_rank		rank of target	int	INTEGER
target_disp		displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
op		reduce operation	MPI_Op	TYPE(MPI_Op)
win		window object	MPI_Win	TYPE(MPI_Win)
)				



# Case study in shared memory: 3, good

TACC

```
1 MPI_Win_fence(0, the_window);
2 int
3     counter_value;
4 if (i_am_available) {
5     int
6         decrement = -1;
7     total_decrement++;
8     MPI_Fetch_and_op
9         ( /* operate with data from origin: */ &decrement,
10           /* retrieve data from target: */ &counter_value,
11             MPI_INT, counter_process, 0, MPI_SUM,
12             the_window);
13 }
14 MPI_Win_fence(0, the_window);
15 if (i_am_available) {
16     my_counter_values[n_my_counter_values++] = counter_value;
17 }
```



MPI type	meaning	applies to
<code>MPI_MAX</code>	maximum	integer, floating point
<code>MPI_MIN</code>	minimum	
<code>MPI_SUM</code>	sum	integer, floating point, complex, multilanguage
<code>MPI_REPLACE</code>	overwrite	
<code>MPI_NO_OP</code>	no change	
<code>MPI_PROD</code>	product	
<code>MPI_LAND</code>	logical and	C integer, logical
<code>MPI_LOR</code>	logical or	
<code>MPI_LXOR</code>	logical xor	
<code>MPI_BAND</code>	bitwise and	integer, byte, multilanguage types
<code>MPI_BOR</code>	bitwise or	
<code>MPI_BXOR</code>	bitwise xor	
<code>MPI_MAXLOC</code>	max value and location	<code>MPI_DOUBLE_INT</code> and such
<code>MPI_MINLOC</code>	min value and location	

No user-defined operators.



We are using fences, which are collective.  
What if a process is still operating on its local work?

Better (but more tricky) solution:  
use passive target synchronization and locks.



```
1 if (rank == 0) {  
2     MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);  
3     MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);  
4     MPI_Win_unlock (1, win);  
5 }
```

No action on the target required!



Investigate atomic updates using passive target synchronization. Use `MPI_Win_lock` with an exclusive lock, which means that each process only acquires the lock when it absolutely has to.

- All processes but one update a window:

```
1 int one=1;
2 MPI_Fetch_and_op(&one, &readout,
3 MPI_INT, repo, zero_disp, MPI_SUM,
4 the_win);
```

- while the remaining process spins until the others have performed their update.

Use an atomic operation for the latter process to read out the shared value.

Can you replace the exclusive lock with a shared one?



As exercise 43, but now use a shared lock: all processes acquire the lock simultaneously and keep it as long as is needed.

The problem here is that coherence between window buffers and local variables is now not forced by a fence or releasing a lock. Use `MPI_Win_flush_local` to force coherence of a window (on another process) and the local variable from `MPI_Fetch_and_op`.



# Part VIII

## Big data communication



This section discusses big messages.

Commands learned:

- `MPI_Send_c`, `MPI_Allreduce_c`, `MPI_Get_count_c` etc. (MPI-4)



- There is no problem allocating large buffers:

```
1 size_t bigsize = 1<<33;  
2 double *buffer =  
3     (double*) malloc(bigsize*sizeof(double));
```

- But you can not tell MPI how big the buffer is:

```
1 MPI_Send(buffer,bigsize,MPI_DOUBLE,...) // WRONG
```

because the size argument has to be *int*.



Count type since MPI 3

C:

```
1 MPI_Count count;
```

Fortran:

```
1 Integer(kind=MPI_COUNT_KIND) :: count
```

Big enough for

- `int`;
- `MPI_Aint`, used in one-sided (and therefore big enough for `intptr_t` and `ptrdiff_t`);
- `MPI_Offset`, used in file I/O.

However, this type could not be used in MPI-3 to describe send buffers.



C: routines with `_c` suffix

```
1 MPI_Count count;  
2 MPI_Send_c( buff,count,MPI_INT, ... );
```

also `MPI_Reduce_c`, `MPI_Get_c`, ... (some 190 routines in all)

Fortran: polymorphism rules

```
1 Integer(kind=MPI_COUNT_KIND) :: count  
2 call MPI_Send( buff,count, MPI_INTEGER, ... )
```



# Big count example

TACC

```
1 // pingpongbig.c
2 assert( sizeof(MPI_Count)>4 );
3 for ( int power=3; power<=10; power++ ) {
4     MPI_Count length=pow(10,power);
5     buffer = (double*)malloc( length*sizeof(double) );
6     MPI_Ssend_c
7         (buffer,length,MPI_DOUBLE,
8          processB,0,comm);
9     MPI_Recv_c
10        (buffer,length,MPI_DOUBLE,
11          processB,0,comm,MPI_STATUS_IGNORE);
```



```
1 !! pingpongbig.F90
2     integer :: power,countbytes
3     Integer(KIND=MPI_COUNT_KIND) :: length
4     call MPI_Sizeof(length,countbytes,ierr)
5     if (procno==0) &
6         print *, "Bytes in count:",countbytes
7         length = 10**power
8         allocate( senddata(length),recvdata(length) )
9         call MPI_Send(senddata,length,MPI_DOUBLE_PRECISION, &
10                      processB,0, comm)
11         call MPI_Recv(recvdata,length,MPI_DOUBLE_PRECISION, &
12                      processB,0, comm,MPI_STATUS_IGNORE)
```



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)
)				



C:

```
1 MPI_Count count;
2 MPI_Get_count_c( &status,MPI_INT, &count );
3 MPI_Get_elements_c( &status,MPI_INT, &count );
```

Fortran:

```
1 Integer(kind=MPI_COUNT_KIND) :: count
2 call MPI_Get_count( status,MPI_INTEGER,count )
3 call MPI_Get_elements( status,MPI_INTEGER,count )
```



MPI-3 mechanism, deprecated in MPI-4.1:  
send a number of contiguous types:

```
1 MPI_Datatype blocktype;
2 MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
3 MPI_Type_commit(&blocktype);
4 if (procno==sender) {
5     MPI_Send(source,nblocks,blocktype,receiver,0,comm);
```

By composing types you can make a ‘big type’. Use

`MPI_Type_get_extent_x`, `MPI_Type_get_true_extent_x`, `MPI_Get_elements_x`  
to query.

```
1 MPI_Count recv_count;
2 MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```



# Advanced (MPI-3/4) topics



Recent additions to the MPI standard allow your code to deal with unusual scenarios or very large scale runs.



## Part IX

### Advanced collectives



- Collectives are blocking.
- Compare blocking/non-blocking sends:

`MPI_Send` → `MPI_Isend`

immediate return of control, produce request object.

- Non-blocking collectives:

`MPI_Bcast` → `MPI_Ibcast`

Same:

```
1 MPI_Isomething( <usual arguments>, MPI_Request *req);
```

- Considerations:

- Calls return immediately;
- the usual story about buffer reuse
- Requires `MPI_Wait...` for completion.
- Multiple collectives can complete in any order

- Why?

- Use for overlap communication/computation
- Imbalance resilience
- Allows pipelining



Name	Param name	Explanation	C type	F type
MPI_Ibcast (				
MPI_Ibcast_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)
request		communication request	MPI_Request*	TYPE(MPI_Request)
)				



Independent collective and local operations:

$$y \leftarrow Ax + (x^t x)y$$

```
1 MPI_Iallreduce( .... x ...., &request);  
2 // compute the matrix vector product  
3 MPI_Wait(request);  
4 // do the addition
```



Do two reductions (on the same communicator) with different operators simultaneously:

$$\begin{aligned}\alpha &\leftarrow x^t y \\ \beta &\leftarrow \|z\|_\infty\end{aligned}$$

which translates to:

```
1 MPI_Request reqs[2];
2 MPI_Iallreduce
3   ( &local_xy, &global_xy, 1, MPI_DOUBLE, MPI_SUM, comm,
4     &(reqs[0]) );
5 MPI_Iallreduce
6   ( &local_xinf, &global_xin, 1, MPI_DOUBLE, MPI_MAX, comm,
7     &(reqs[1]) );
8 MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```



Blocking and non-blocking collectives don't match:  
either all processes call the non-blocking or all call the blocking one.  
Thus the following code is incorrect:

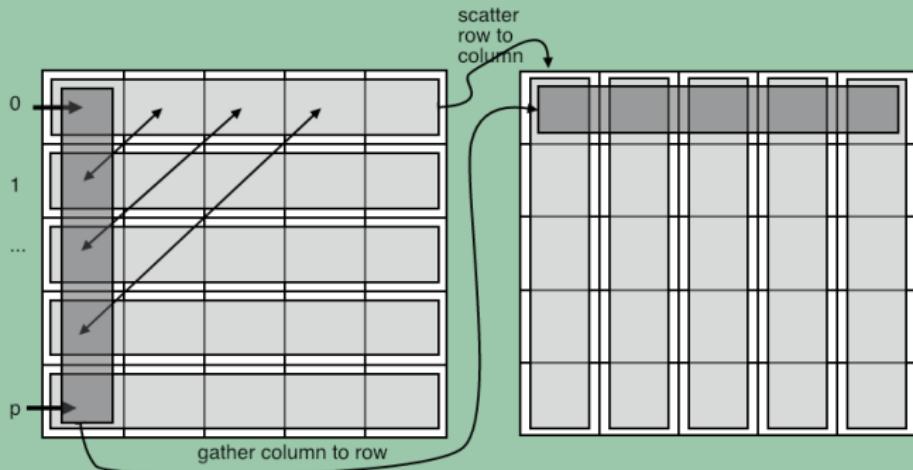
```
1  if (rank==root)
2      MPI_Reduce( &x /* ... */ root,comm );
3  else
4      MPI_Ireduce( &x /* ... */ root,comm,&req);
```

This is unlike the point-to-point behavior of non-blocking calls: you can catch a message with `MPI_Irecv` that was sent with `MPI_Send`.



# Transpose as gather/scatter

TACC



Every process needs to do a scatter or gather.



Transpose matrix by scattering all rows simultaneously.

Each scatter involves all processes, but with a different spanning tree.

```
1 MPI_Request scatter_requests[nprocs] ;
2 for (int iproc=0; iproc<nprocs; iproc++) {
3     MPI_Iscatter( regular,1,MPI_DOUBLE,
4                   &(transpose[iproc]),1,MPI_DOUBLE,
5                   iproc,comm,scatter_requests+iproc) ;
6 }
7 MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE) ;
```



## Persistent collectives



Similar to persistent send/recv:

```
1  double *buffer;
2  MPI_Allreduce_init( buffer ...., &request );
3  for ( ... ) {
4      // fill buffer
5      MPI_Start( request );
6      // possibly other activities
7      MPI_Wait( &request );
8  }
9  MPI_Request_free( &request );
```

Available for all collectives and neighborhood collectives.



# Example

TACC

```
1 // powerpersist1.c
2 double localnorm,globalnorm=1.;
3 MPI_Request reduce_request;
4 MPI_Allreduce_init
5   ( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
6     comm,MPI_INFO_NULL,&reduce_request);
7 for (int it=0; ; it++) {
8   /*
9    * Matrix vector product
10   */
11   matmult(indata,outdata,buffersize);
12
13 // start computing norm of output vector
14 localnorm = local_l2_norm(outdata,buffersize);
15 double old_globalnorm = globalnorm;
16 MPI_Start( &reduce_request );
17
18 // end computing norm of output vector
19 MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
20 globalnorm = sqrt(globalnorm);
21 // now 'globalnorm' is the L2 norm of 'outdata'
22 scale(outdata,indata,buffersize,1./globalnorm);
23 }
```



Both request-based.

- Non-blocking is ‘ad hoc’: buffer info not known before the collective call.
- Persistent allows ‘planning ahead’: management of internal buffers and such.

Request handling:

- Non-blocking: wait deallocates the request
- Persistent: wait deactivates the request, still requires `MPI_Request_free`.



## Non-blocking barrier



- Barrier is not *time* synchronization but *state* synchronization.
- Test on non-blocking barrier: 'has everyone reached some state'



- Some processes decide locally to alter their structure
- ... need to communicate that to neighbors
- Problem: neighbors don't know whether to expect update calls, if at all.
- Solution:
  - send update msgs, if any;
  - then post barrier.
  - Everyone probe for updates, test for barrier.



- Distributed termination detection (Matocha and Kamp, 1998): draw a global conclusion with local operations
- Everyone posts the barrier when done;
- keeps doing local computation while testing for the barrier to complete



Name	Param name	Explanation	C type	F type
MPI_Ibarrier (				
comm		communicator	MPI_Comm	TYPE(MPI_Comm)
request		communication request	MPI_Request*	TYPE(MPI_Request)
)				



Do sends, post barrier.

```
1 // ibarrierprobe.c
2 if (i_do_send) {
3     /*
4      * Pick a random process to send to,
5      * not yourself.
6      */
7     int receiver = rand()%nprocs;
8     MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
9 }
10 /*
11  * Everyone posts the non-blocking barrier
12  * and gets a request to test/wait for
13  */
14 MPI_Request barrier_request;
15 MPI_Ibarrier(comm,&barrier_request);
```



## Poll for barrier and messages

```
1  for ( ; ; step++) {  
2      int barrier_done_flag=0;  
3      MPI_Test(&barrier_request,&barrier_done_flag,  
4                  MPI_STATUS_IGNORE);  
5      //stop if you're done!  
6      if (barrier_done_flag) {  
7          break;  
8      } else {  
9          // if you're not done with the barrier:  
10         int flag; MPI_Status status;  
11         MPI_Iprobe  
12         ( MPI_ANY_SOURCE,MPI_ANY_TAG,  
13             comm, &flag, &status );  
14         if (flag) {  
15             // absorb message!
```



# Part X

## Shared memory



Myth:

*MPI processes use network calls, whereas OpenMP threads access memory directly, therefore OpenMP is more efficient for shared memory.*

Truth:

*MPI implementations use copy operations when possible, whereas OpenMP has thread overhead, and affinity/coherence problems.*

Main problem with MPI on shared memory: data duplication.



- Shared memory access: two processes can access each other's memory through `double*` (and such) pointers, if they are on the same shared memory.
- Limitation: only window memory.
- Non-use case: remote update. This has all the problems of traditional shared memory (race conditions, consistency).
- Good use case: every process needs access to large read-only dataset  
Example: ray tracing.



- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers ‘fake shared memory’: you can access another process’ data, but only through function calls.
- MPI-3 shared memory gives you a pointer to another process’ memory,  
*if that process is on the same shared memory.*





- Cluster node has shared memory
- Memory is attached to specific socket
- beware Non-Uniform Memory Access (NUMA) effects



Here is the high level overview; details next.

- 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  
(MPI-4.1: other window creation calls allowed, but the burden is on you!)
- Use `MPI_Win_shared_query` to get pointer to another process' window data.
- You can now use `memcpy` instead of `MPI_Put`.



- `MPI_Comm_split_type` splits into communicators of same type.
- Use type: `MPI_COMM_TYPE_SHARED` splitting by shared memory.

## Code:

```
1 // commssplittype.c
2 MPI_Info info;
3 MPI_Comm_split_type
4     (MPI_COMM_WORLD,
5      MPI_COMM_TYPE_SHARED,
6      procno, info, &sharedcomm);
7 MPI_Comm_size
8     (sharedcomm, &new_nprocs);
9 MPI_Comm_rank
10    (sharedcomm, &new_procno);
```

## Output:

```
1 make[3]: `commssplittype' is up to date.
2 TACC: Starting up job 4356245
3 TACC: Starting parallel tasks...
4 There are 10 ranks total
5 [0] is processor 0 in a shared group of 5,
   ↪running on
   ↪c209-010.frontera.tacc.utexas.edu
6 [5] is processor 0 in a shared group of 5,
   ↪running on
   ↪c209-011.frontera.tacc.utexas.edu
7 TACC: Shutdown complete. Exiting.
```



Write a program that uses `MPI_Comm_split_type` to analyze for a run

1. How many nodes there are;
2. How many processes there are on each node.

If you run this program on an unequal distribution, say 10 processes on 3 nodes, what distribution do you find?

```
1  Nodes: 3; processes: 10
2  TACC: Starting up job 4210429
3  TACC: Starting parallel tasks...
4  There are 3 nodes
5  Node sizes: 4 3 3
6  TACC: Shutdown complete. Exiting.
```



Use `MPI_Win_allocate_shared` to create a window that can be shared;

- Has to be on a communicator on shared memory
- Example: window is one double.

```
1 // sharedbulk.c
2 MPI_Win node_window;
3 MPI_Aint window_size; double *window_data;
4 if (onnode_procid==0)
5     window_size = sizeof(double);
6 else window_size = 0;
7 MPI_Win_allocate_shared
8     ( window_size,sizeof(double),MPI_INFO_NULL,
9         nodecomm,
10         &window_data,&node_window);
```



Use `MPI_Win_shared_query`:

```
1 MPI_Aint window_size0; int window_unit; double *win0_addr;  
2 MPI_Win_shared_query  
3   ( node_window,0,  
4     &window_size0,&window_unit, &win0_addr );
```



Name	Param name	Explanation	C type	F type
MPI_Win_shared_query (				
MPI_Win_shared_query_c (				
win		shared memory window object	MPI_Win	TYPE(MPI_Win)
rank		rank in the group of window win or MPI_PROC_NULL	int	INTEGER
size		size of the window segment	MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)
disp_unit		local unit size for displacements, in bytes	[ int* MPI_Aint* ]	INTEGER
baseptr		address for load/store access to window segment	void*	TYPE(C_PTR)
)				



Memory will be allocated contiguously  
convenient for address arithmetic,  
not for NUMA: set `alloc_shared_noncontig` true in `MPI_Info` object.

Example: each window stores one double. Measure distance in bytes:

Strategy: default behavior of  
shared window allocation

Distance 1 to zero: 8  
Distance 2 to zero: 16

Strategy: allow non-contiguous  
shared window allocation

Distance 1 to zero: 4096  
Distance 2 to zero: 8192

Question: what is going on here?



- Application: ray tracing:  
large read-only data structure describing the scene
- traditional MPI would duplicate:  
excessive memory demands
- Better: allocate shared data on process 0 of the shared  
communicator
- Everyone else points to this object.



MPI-4: split by other hardware features through  
`MPI_COMM_TYPE_HW_GUIDED` and `MPI_Get_hw_resource_types()`



## Part XI

# Process management



This section discusses processes management; intra communicators.

Commands learned:

- `MPI_Comm_spawn`, `MPI_UNIVERSE_SIZE`
- `MPI_Comm_get_parent`, `MPI_Comm_remote_size`



- PVM was a precursor of MPI: could dynamically create new processes.
- It took MPI a while to catch up.
- Use `MPI_Attr_get` to retrieve `MPI_UNIVERSE_SIZE` attribute indicating space for creating more processes outside `MPI_COMM_WORLD`.
- New processes have their own `MPI_COMM_WORLD`.
- Communication between the two communicators: ‘inter communicator’  
(the old type is ‘intra communicator’)



Probably a machine dependent component.

Suggested standard:

```
mpiexec -n 4 -usize 8 spawn_manager
```

Intel MPI at TACC:

```
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawn_manager
```

Discover size of the universe:

```
1 MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,  
2     (void*)&universe_sizep, &flag);
```



```
1 int universe_size, *universe_size_attr,uflag;
2 MPI_Comm_get_attr
3     (comm_world,MPI_UNIVERSE_SIZE,
4      &universe_size_attr,&uflag);
5 if (uflag) {
6     universe_size = *universe_size_attr;
7 } else {
8     printf("This MPI does not support UNIVERSE_SIZE.\nUsing world
9         ←size");
10    universe_size = world_n;
11 }
12 int work_n = universe_size - world_n;
13 if (world_p==0) {
14     printf("A universe of size %d leaves room for %d workers\n",
15            universe_size,work_n);
16     printf(.. spawning from %s\n",procname);
17 }
```



```
1 const char *workerprogram = "./spawnapp";
2 MPI_Comm_spawn(workerprogram,MPI_ARGV_NULL,
3                  work_n,MPI_INFO_NULL,
4                  0,comm_world,&comm_inter,NULL);
```



```
1 // spawnworker.c
2 MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
3 MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
4 MPI_Comm_get_parent(&parent);
```



```
1 // spawnapp.c
2 MPI_Comm comm_parent;
3 MPI_Comm_get_parent(&comm_parent);
4 int is_child = (comm_parent!=MPI_COMM_NULL);
5 if (is_child) {
6     int nworkers,workerno;
7     MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
8     MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
9     printf("I detect I am worker %d/%d running on %s\n",
10           workerno,nworkers,procname);
```



# Part XII

## Process topologies



This section discusses topologies:

- Cartesian topology
- MPI-1 Graph topology
- MPI-3 Graph topology

Commands learned:

- `MPI_Dist_graph_create`, `MPI_DIST_GRAPH`,  
`MPI_Dist_graph_neighbors_count`
- `MPI_Neighbor_allgather` and such



- Processes don't communicate at random
- Example: Cartesian grid, each process 4 (or so) neighbors
- Express operations in terms of topology
- Elegance of expression
- MPI can optimize



- Consecutive process numbering often the best:  
divide array by chunks
- Not optimal for grids or general graphs:
- MPI is allowed to renumber ranks
- Graph topology gives information from which MPI can deduce  
renumbering



- Cartesian topology
- Graph topology, globally specified.  
Not scalable, do not use!



- Graph topologies locally specified: scalable!  
Limit cases: each process specifies its own connectivity one process specifies whole graph.
- Neighborhood collectives:  
expression close to the algorithm.



## Graph topologies

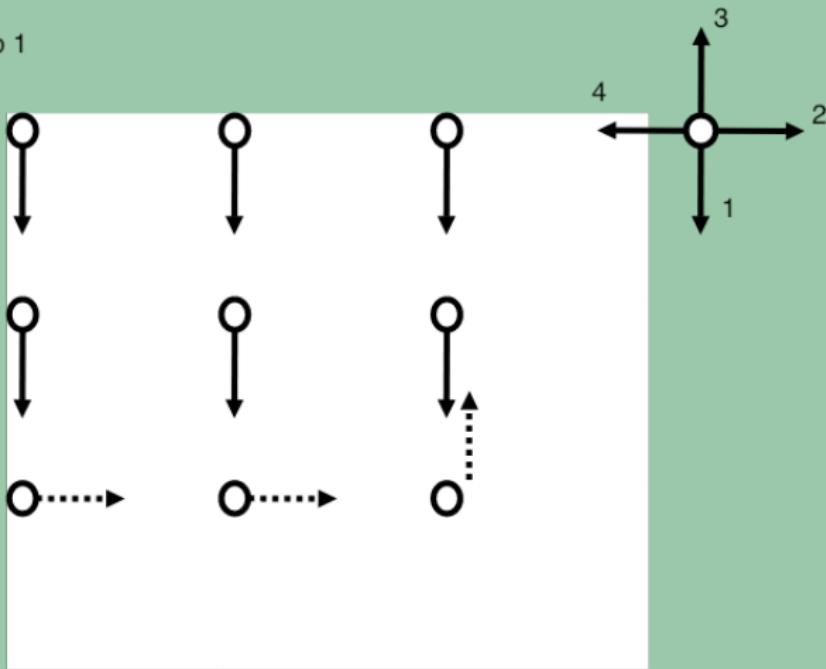


Neighbor exchange, spelled out:

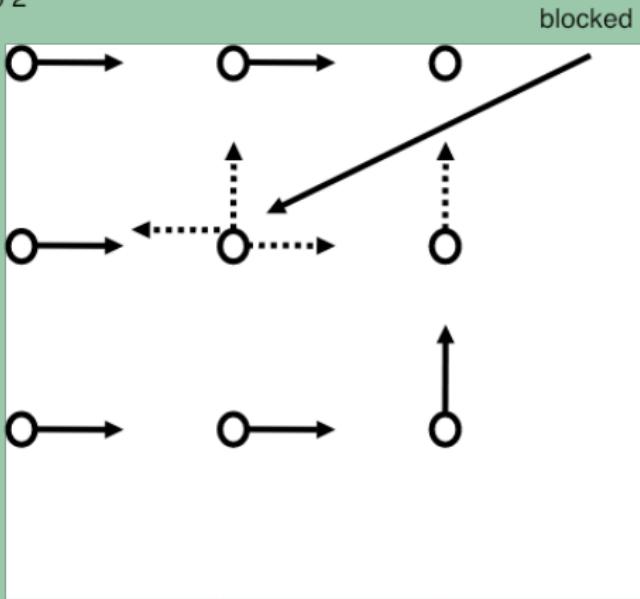
- Each process communicates down/right/up/left
- Send and receive at the same time.
- Can optimally be done in four steps



Step 1



Step 2

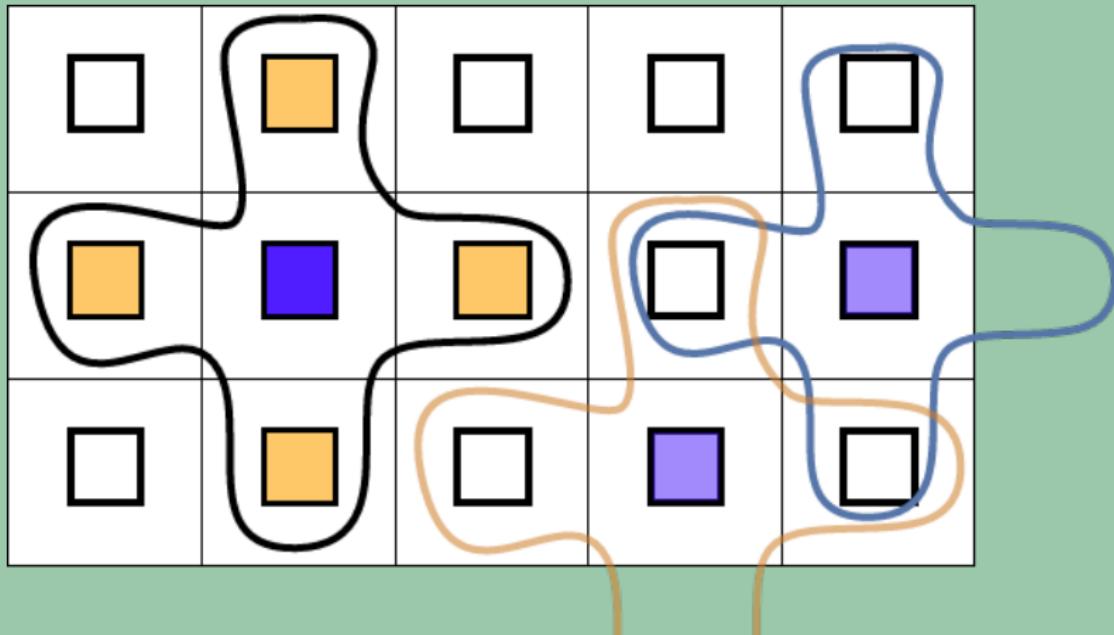


The middle node is blocked because all its targets are already receiving  
or a channel is occupied:  
one missed turn



This is really a 'local gather':  
each node does a gather from its neighbors in whatever order.

`MPI_Neighbor_allgather`



- Using `MPI_Isend` / `MPI_Irecv` is like spelling out a collective, imposes order;
- Collectives can use pipelining as opposed to sending a whole buffer;
- Collectives can use spanning trees as opposed to direct connections.



```
1 int MPI_Dist_graph_create
2     (MPI_Comm comm_old, int nsources, const int sources[],
3      const int degrees[], const int destinations[],
4      const int weights[], MPI_Info info, int reorder,
5      MPI_Comm *comm_dist_graph)
```

- *nsources* how many source nodes described? (Usually 1)
- *sources* the processes being described (Usually `MPI_Comm_rank` value)
- *degrees* how many processes to send to
- *destinations* their ranks
- *weights*: usually set to `MPI_UNWEIGHTED`.
- *info*: `MPI_INFO_NULL` will do
- *reorder*: 1 if dynamically reorder processes



```
1 int MPI_Neighbor_allgather
2     (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
3      void *recvbuf, int recvcount, MPI_Datatype recvtype,
4      MPI_Comm comm)
```

Like an ordinary `MPI_Allgather`, but  
the receive buffer has a length enough for `degree` messages  
(instead of comm size).



After `MPI_Neighbor_allgather` data in the buffer is *not* in normal rank order.

- `MPI_Dist_graph_neighbors_count` gives actual number of neighbors.  
(Why do you need this?)
- `MPI_Dist_graph_neighbors` lists neighbor numbers.



Name	Param name	Explanation	C type	F type
MPI_Dist_graph_neighbors_count				
	comm	communicator with distributed graph topology	MPI_Comm	TYPE(MPI_Comm)
	indegree	number of edges into this process	int*	INTEGER
	outdegree	number of edges out of this process	int*	INTEGER
	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise	int*	LOGICAL
	)			



Name	Param name	Explanation	C type	F type
MPI_Dist_graph_neighbors (				
comm		communicator with distributed graph topology	MPI_Comm	TYPE(MPI_Comm)
maxindegree		size of sources and sourceweights arrays	int	INTEGER
sources		processes for which the calling process is a destination	int[]	INTEGER(maxindegree)
sourceweights		weights of the edges into the calling process	int[]	INTEGER(*)
maxoutdegree		size of destinations and destweights arrays	int	INTEGER
destinations		processes for which the calling process is a source	int[]	INTEGER(maxoutdegree)
destweights		weights of the edges out of the calling process	int[]	INTEGER(*)
)				

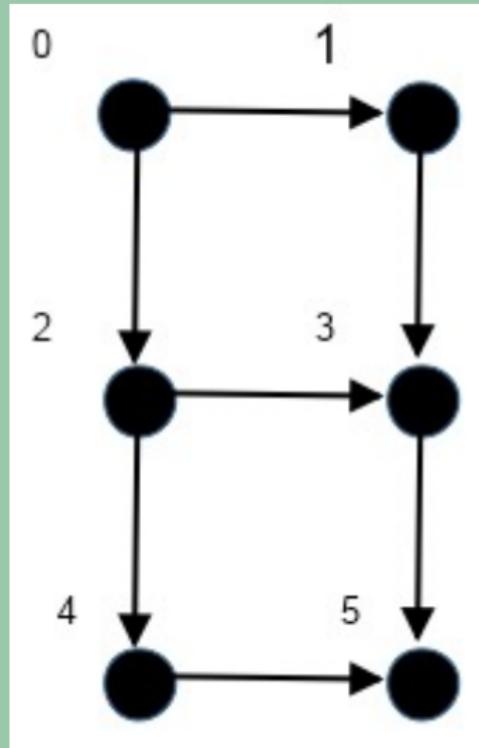


# Example: Systolic graph

TACC

Code:

```
1 // graph.c
2 for ( int i=0; i<=1; i++ ) {
3     int neighb_i = proci+i;
4     if (neighb_i<0 || neighb_i>=idim)
5         continue;
6     int j = 1-i;
7     int neighb_j = procj+j;
8     if (neighb_j<0 || neighb_j>=jdim)
9         continue;
10    destinations[ degree++ ] =
11        PROC(neighb_i,neighb_j,idim,jdim);
12 }
13 MPI_Dist_graph_create
14     (comm,
15     /* I specify just one proc: me */ 1,
16     &procno,&degree,destinations,weights,
17     MPI_INFO_NULL,0,
18     &comm2d
19 );
```



# Output

TACC

## Code:

```
1 int indegree,outdegree,  
2     weighted;  
3 MPI_Dist_graph_neighbors_count  
4     (comm2d,  
5      &indegree,&outdegree,  
6      &weighted);  
7 int  
8     my_ij[2] = {proci,procj},  
9     other_ij[4][2];  
10 MPI_Neighbor_allgather  
11   ( my_ij,2,MPI_INT,  
12     other_ij,2,MPI_INT,  
13       comm2d );
```

## Output:

```
1 [ 0 = (0,0)] has 2 outbound: 1, 2,  
2     0 inbound:  
3 [ 1 = (0,1)] has 1 outbound: 3,  
4     1 inbound: (0,0)=0  
5 [ 2 = (1,0)] has 2 outbound: 3, 4,  
6     1 inbound: (0,0)=0  
7 [ 3 = (1,1)] has 1 outbound: 5,  
8     2 inbound: (0,1)=1 (1,0)=2  
9 [ 4 = (2,0)] has 1 outbound: 5,  
10    1 inbound: (1,0)=2  
11 [ 5 = (2,1)] has 0 outbound:  
12     2 inbound: (1,1)=3 (2,0)=4
```

Note that the neighbors are listed in correct order. This need not be the case.



Explicit query of neighbor process ranks.

Code:

```
1 int indegree,outdegree,  
2     weighted;  
3 MPI_Dist_graph_neighbors_count  
4     (comm2d,  
5      &indegree,&outdegree,  
6      &weighted);  
7 int  
8     my_ij[2] = {proci,procj},  
9     other_ij[4][2];  
10    MPI_Neighbor_allgather  
11    ( my_ij,2,MPI_INT,  
12      other_ij,2,MPI_INT,  
13      comm2d );
```

Output:

```
1      0 inbound:  
2      1 inbound: 0  
3      1 inbound: 0  
4      2 inbound: 1 2  
5      1 inbound: 2  
6      2 inbound: 4 3
```

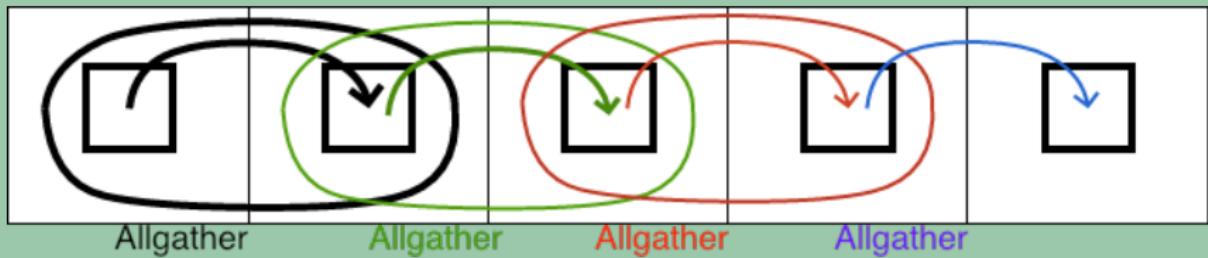


▶ Earlier rightsend exercise

Revisit exercise 19 and solve it using `MPI_Dist_graph_create`. Use figure 398 for inspiration.

Use a degree value of 1.





Solving the right-send exercise with neighborhood collectives



Two approaches:

1. Declare just one source: the previous process. Do this! Or:
2. Declare two sources: the previous and yourself. In that case bear in mind slide 391.



- Heterogeneous: `MPI_Neighbor_alltoallw`.
- Non-blocking: `MPI_Ineighbor_allgather` and such
- Persistent: `MPI_Neighbor_allgather_init`,  
`MPI_Neighbor_allgatherv_init`.



# Other



## Part XIII

Tracing, performance, and such



We briefly touch on peripheral issues issues to MPI.



# CMake



- . MPI is discoverable by cmake.



```
1 cmake_minimum_required( VERSION 3.12 )
2 project( ${PROJECT_NAME} VERSION 1.0 )
3
4 # https://cmake.org/cmake/help/latest/module/FindMPI.html
5 find_package( MPI )
6
7 add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
8 target_compile_features( ${PROJECT_NAME} PRIVATE cxx_std_20 )
9 target_include_directories(
10     ${PROJECT_NAME} PUBLIC
11     ${MPI_CXX_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
12 target_link_libraries(
13     ${PROJECT_NAME} PUBLIC
14     ${MPI_CXX_LIBRARIES} )
15
16 install( TARGETS ${PROJECT_NAME} DESTINATION . )
```



# Errors



Default: global termination.

```
1 MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_ARE_FATAL);
```

**MPI-4:** *Only terminate on communicator: MPI\_ERRORS\_ABORT.*

Local handling: MPI\_ERRORS\_RETURN



Associate error handler with communicator:

`MPI_Comm_set_errhandler MPI_Comm_get_errhandler`

Other:

- `MPI_File_set_errhandler, MPI_File_call_errhandler,`  
**MPI-4:** *MPI\_Session\_set\_errhandler,*  
*MPI\_Session\_call\_errhandler,*  
`MPI_Win_set_errhandler, MPI_Win_call_errhandler.`



```
1 char errtxt[MPI_MAX_ERROR_STRING] ;
2 int err = status.MPI_ERROR;
3 int len=MPI_MAX_ERROR_STRING;
4 MPI_Error_string(err,errtxt,&len);
5 printf("Waitall error: %d %s\n",err,errtxt);
```



```
1 int nonzero_code;
2 MPI_Add_error_code(nonzero_class,&nonzero_code);
3 MPI_Add_error_string(nonzero_code,"Attempting to send zero buffer");
```



# Performance measurement



MPI has a *wall clock* timer: `MPI_Wtime` which gives the number of seconds from a certain point in the past.

The timer has a resolution of `MPI_Wtick`

Timers can be global

```
1 int *v,flag;  
2 MPI_Attr_get( comm, MPI_WTIME_IS_GLOBAL, &v, &flag );  
3 if (mytid==0) printf("Time synchronized? %d->%d\n",flag,*v);
```

but probably aren't.



```
1 // pingpong.c
2 if (procno==processA) {
3     t = MPI_Wtime();
4     for (int n=0; n<NEXPERIMENTS; n++) {
5         MPI_Send(send,1,MPI_DOUBLE,
6         MPI_Recv(recv,1,MPI_DOUBLE,
7     }
8     t = MPI_Wtime()-t; t /= NEXPERIMENTS;
```



Processes don't start/end simultaneously. What does a timing result mean overall? Take average or maximum?

Alternative:

```
1      MPI_Barrier(comm)
2      t = MPI_Wtime();
3      // something happens here
4      MPI_Barrier(comm)
5      t = MPI_Wtime()-t;
```



See other lecture: MPIP, TAU, et cetera.



Every routine `MPI_Something` calls a routine `PMPI_Something` that does the actual work. You can now write your `MPI_...` routine which calls `PMPI_...`, and inserting your own profiling calls.

```
main() {  
    MPI_Send ( buffer,ct,tp, ... );  
}
```

call

```
int MPI_Send ( buffer,ct,tp, ... ) {  
    PMPI_Send( buffer,ct,tp, ... );
```



# Programming for performance



- Optimization for small messages: bypass rendez-vous protocol (slide 157)
- Cross-over point: ‘Eager limit’.
- Force efficient messages by increasing the eager limit.
- Beware: decreasing payoff for large messages, and
- Beware: buffers for eager send eat into your available memory.



- For Intel MPI: I\_MPI\_EAGER\_THRESHOLD
- mvapich2: MV2\_IBA\_EAGER\_THRESHOLD
- OpenMPI: OpenMPI has the --mca options btl\_openib\_eager\_limit and btl\_openib\_rndv\_eager\_limit.



- Non-blocking sends `MPI_Isend` / `MPI_Irecv` can be more efficient than blocking
- Also: allow overlap computation/communication (latency hiding)
- However: can usually not be considered a replacement.



MPI is not magically active in the background, so latency hiding is not automatic. Same for passive target synchronization and non-blocking barrier completion.

- Dedicated communications processor or thread.  
This is implementation dependent; for instance, Intel MPI:  
`I_MPI_ASYNC_PROGRESS_...` variables.
- Force progress by occasional calls to a polling routine such as  
`MPI_Iprobe`.



If a communication between the same pair of processes, involving the same buffer, happens regularly, it is possible to set up a *persistent communication*.

- `MPI_Send_init`
- `MPI_Recv_init`
- `MPI_Start`



- MPI has internal buffers: copying costs performance
- Use your own buffer:
  - `MPI_Buffer_attach`
  - `MPI_Bsend`
- Copying is also a problem for derived datatypes.



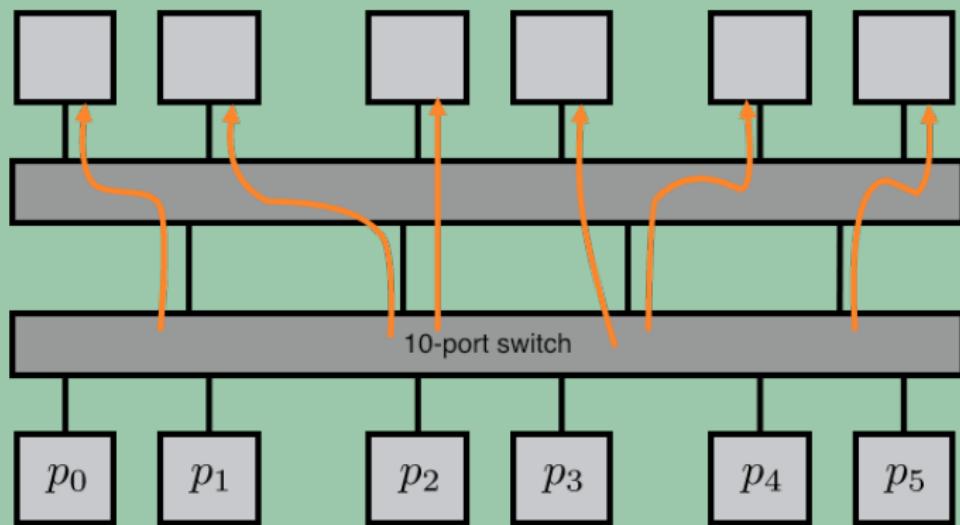
- Mapping problem to architecture sometimes not trivial
- Load balancers: *ParMetis, Zoltan*
- Graph topologies: *MPI\_Dist\_graph\_adjacent*: allowed to reorder ranks for proximity
- Neighborhood collectives allow MPI to schedule optimally.
  - `MPI_Neighbor_allgather` (and `MPI_Neighbor_allgather_v`)
  - `MPI_Neighbor_alltoall`

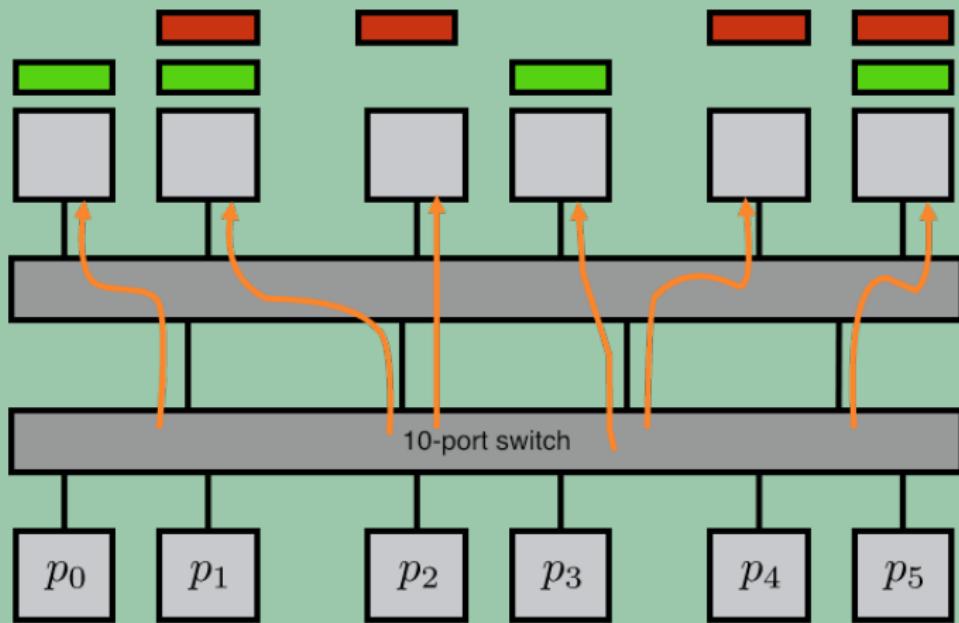


Network contention means that

- Your messages can collide with other jobs
- messages within your job can collide







- Network cards can offer assistance
- Mellanox: off-loading
  - limited repertoire of scenarios where it helps
- Intel disagrees: on-loading
- Either way, investigate the capabilities of your network.

