



WWW.TACC.UTEXAS.EDU



Tutorial on MPI programming  
Victor Eijkhout [eijkhout@tacc.utexas.edu](mailto:eijkhout@tacc.utexas.edu)  
TACC/XSEDE MPI training 2021

# Code of Conduct

XSEDE has an external code of conduct for XSEDE sponsored events which represents XSEDE's commitment to providing an inclusive and harassment-free environment in all interactions regardless of gender, sexual orientation, disability, physical appearance, race, or religion. The code of conduct extends to all XSEDE-sponsored events, services, and interactions.

Code of Conduct: <https://www.xsede.org/codeofconduct>

Contact:

- Teacher: Victor Eijkhout [eijkhout@tacc.utexas.edu](mailto:eijkhout@tacc.utexas.edu)
- Event organizer: Jason Allison [jasona@tacc.utexas.edu](mailto:jasona@tacc.utexas.edu)

XSEDE ombudspersons:

- Linda Akli, Southeastern Universities Research Association [akli@sura.org](mailto:akli@sura.org)
- Lizanne Destefano, Georgia Tech [lizanne.destefano@ceismc.gatech.edu](mailto:lizanne.destefano@ceismc.gatech.edu)
- Ken Hackworth, Pittsburgh Supercomputing Center [hackworth@psc.edu](mailto:hackworth@psc.edu)
- Bryan Snead, Texas Advanced Computing Center [jbsnead@tacc.utexas.edu](mailto:jbsnead@tacc.utexas.edu)

# Justification

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

# Table of Contents

# Basics

# Part I

## The SPMD model

## 2. Overview

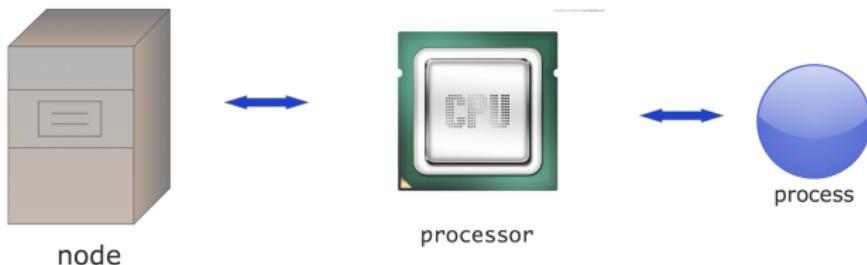
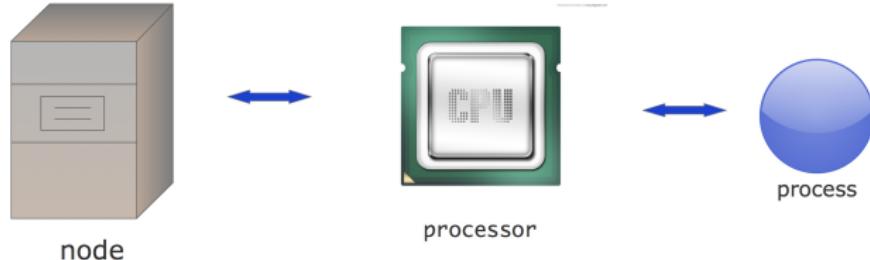
In this section you will learn how to think about parallelism in MPI.

Commands learned:

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

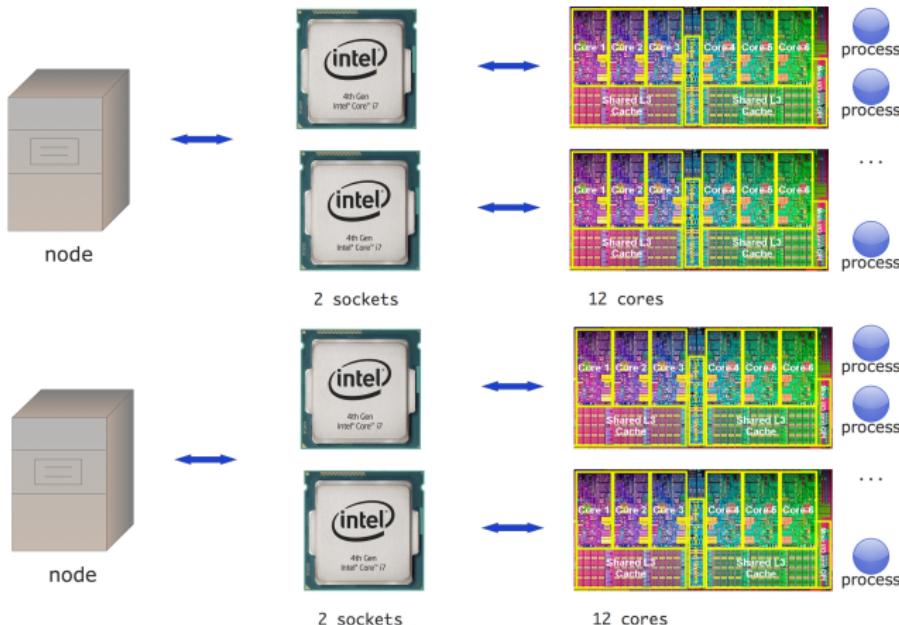
# The MPI worldview: SPMD

### 3. Computers when MPI was designed



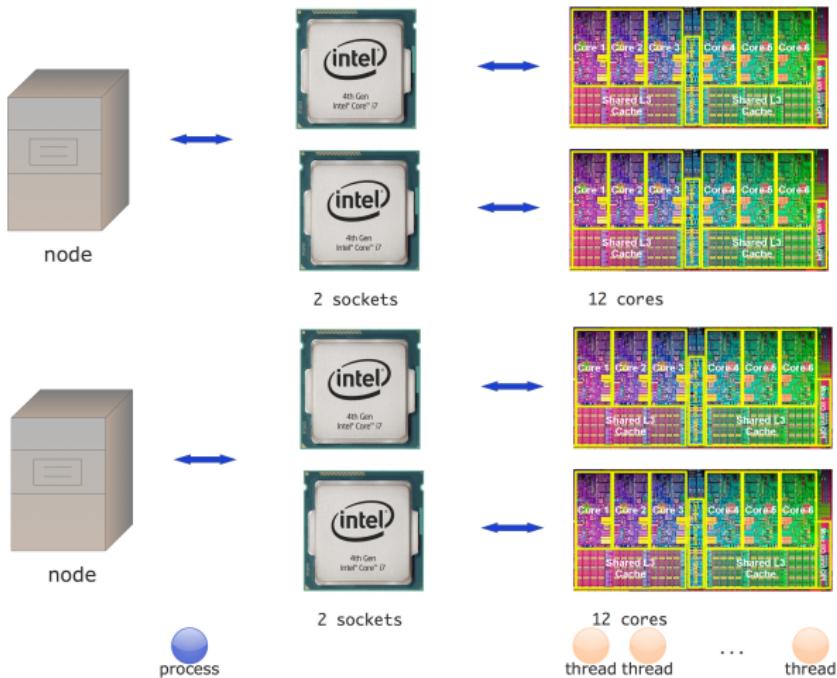
One processor and one process per node;  
all communication goes through the network.

## 4. Pure MPI



A node has multiple sockets, each with multiple cores.  
Pure MPI puts a process on each core: pretend shared memory doesn't exist.

## 5. Hybrid programming



Hybrid programming puts a process per node or per socket;  
further parallelism comes from threading.  
Not in this course...

## 6. Terminology

‘Processor’ is ambiguous: is that a chip or one independent instruction processing unit?

- Socket: the processor chip
- Processor: we don’t use that word
- Core: one instruction-stream processing unit
- Process: preferred terminology in talking about MPI.

## 7. SPMD

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

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

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

# Practicalities

## 8. Compiling and running

MPI compilers are usually called mpicc, mpif90, mpicxx.

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

Running your program: At TACC:

ibrun yourprog

the number of processes is determined by SLURM. general case:

mpiexec -n 4 hostfile ... yourprogram arguments

mpirun -np 4 hostfile ... yourprogram arguments

## 9. Do I need a supercomputer?

- With mpiexec and such, you start a bunch of processes that execute your MPI program.
- Does that mean that you need a cluster or a big multicore?
- No! You can start a large number of MPI processes, even on your laptop. The OS will use ‘time slicing’.
- Of course it will not be very efficient...

# 10. Cluster setup

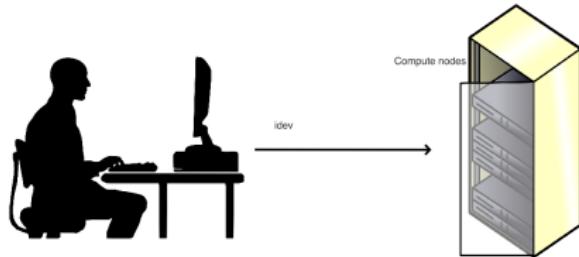
Typical cluster:

- Login nodes, where you ssh into; usually shared with 100 (or so) other people. You don't run your parallel program there!
- Compute nodes: where your job is run. They are often exclusive to you: no other users getting in the way of your program.

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

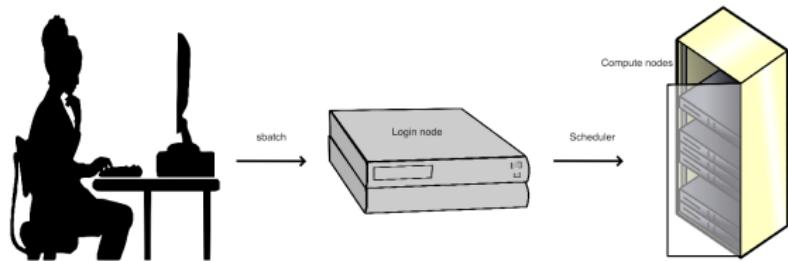
# 11. Interactive run

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



## 12. Batch run

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



## 13. Lab setup

- Open a terminal window on a TACC cluster.
- Type `idev -N 2 -n 32 -t 4:0:0` which gives you an interactive session of 2 nodes, 32 cores, for the next 4 hours.
- (After this course, for serious work, you would write a batch script. The `idev` sessions are strictly limited in time and resources.)
- See the handout for reservations, project IDs, and location of training materials.
- Next slide for how to make and run exercises.

## 14. How to make exercises

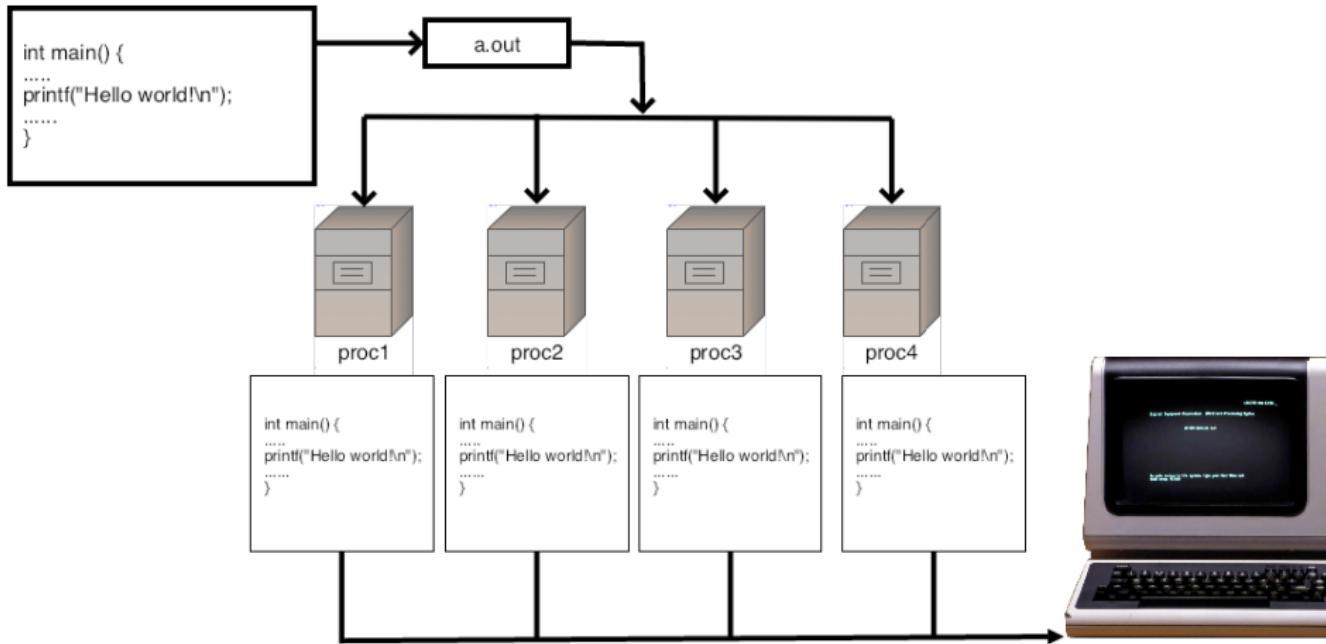
- Directory: exercises-mpi-c or cxx or f or f08 or p or mpl
- If a slide has a (exercisename) over it, there will be a template program exercisename.c (or F90 or py).
- Type make exercisename to compile it
- Python: setup once per session  
`module load python3 # or python2`  
No compilation needed. Run:  
`ibrun python3 yourprogram.py # or python2`
- Add an exercise of your own to the makefile: add the name to the EXERCISES

## Exercise 1 (hello)

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

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

## 15. In a picture



We start learning MPI!

## 16. MPI definitions

You need an include file:

```
#include "mpi.h" // for C
```

```
use mpi_f08 ! for Fortran2008
```

```
use mpi      ! for legacy Fortran90
```

```
from mpi4py import MPI
```

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

- There are no real C++ bindings, but see MPL.
- True Fortran bindings as of the 2008 standard. Provided in Intel compiler:

```
module load intel/18.0.2
```

or newer. Not in gcc7. (Legacy bindings are far inferior)

## 17. MPI Init / Finalize

Then put these calls around your code:

```
|| MPI_Init(&argc,&argv); // zeros allowed  
|| // your code  
|| MPI_Finalize();
```

and for Fortran:

```
|| call MPI_Init()      ! F08 style  
|| ! your code  
|| call MPI_Finalize()  
  
|| call MPI_Init(ierr) ! F90 style  
|| ! your code  
|| call MPI_Finalize(ierr)
```

## 18. About error codes

MPI routines return an integer error code (slide ??)

- In C: function result. Can be ignored; is ignored in this course.

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

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

There's actually not a lot you can do with an error code:  
very hard to recover from errors in parallel.

By default code bombs with (hopefully informative) message.

## 19. Python bindings

```
module load python3
```

```
from mpi4py import MPI
```

Run:

```
ibrun python3 yourprogram.py
```

Your local installation:

```
mpiexec -n 15 python yourprogram.py
```

No initialization needed.

## 20. C++ bindings

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 C++ bindings in Boost. No longer developed?
- Try MPL: <https://github.com/rabauke/mpl>
  - Very modern OO.
  - Exercises in mpl directory.
  - Caution: not a full MPI implementation  
(I/O and one-sided mostly missing)

## Exercise 2 (hello)

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.

Run your program on a large scale, using a batch job. Where does the output go? Experiment with

```
MY_MPIRUN_OPTIONS="-prepend-rank" ibrun yourprogram
```

## 21. Process identification

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

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

Lowest number is always zero.

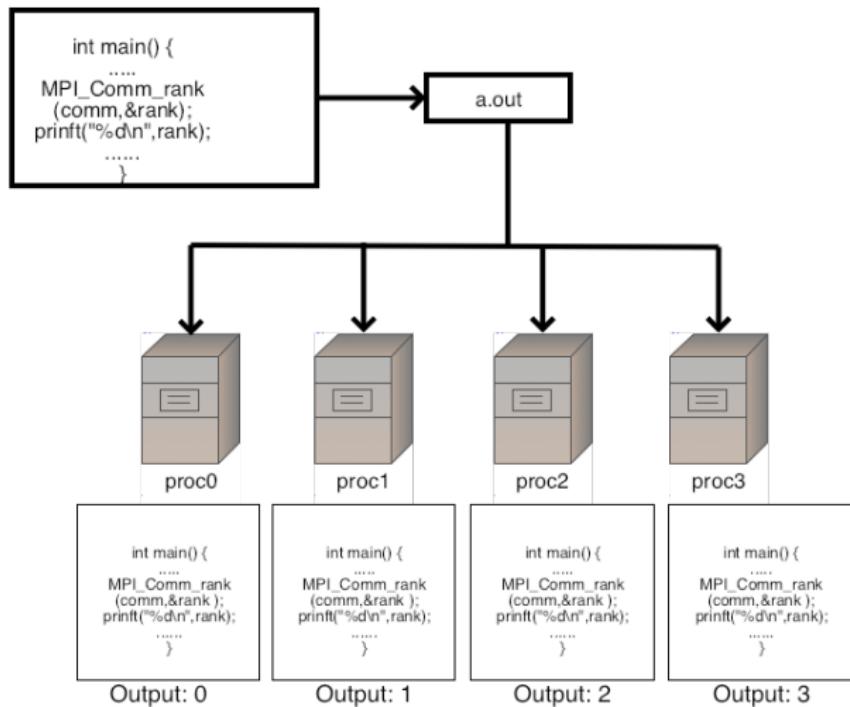
This is a logical view of parallelism: mapping to physical processors/cores is invisible here.

For now, the communicator will be `MPI_COMM_WORLD`.

```
|| MPI_Comm comm = MPI_COMM_WORLD
```

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

## 22. Illustration



## 23. About routine prototypes: C/C++

Prototype:

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

Use:

```
|| MPI_Comm comm = MPI_COMM_WORLD;  
|| int nprocs;  
|| int errorcode;  
|| errorcode = MPI_Comm_size( comm,&nprocs );
```

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

## 24. About routine prototypes: Fortran2008

### Prototype

```
|| MPI_Comm_size(comm, size, ierror)
|| Type(MPI_Comm), INTENT(IN) :: comm
|| INTEGER, INTENT(OUT) :: size
|| INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Use:

```
|| Type(MPI_Comm) :: comm = MPI_COMM_WORLD
|| integer :: size
|| CALL MPI_Comm_size( comm, size ) ! F2008 style
```

- final parameter optional.
- MPI\_... types are Type.

## 25. About routine prototypes: Fortran90

### Prototype

```
|| MPI_Comm_size(comm, size, ierror)
|| Integer, Intent(in) :: comm
|| Integer, Intent(out) :: ierror
```

### Use:

```
|| Integer :: comm = MPI_COMM_WORLD
|| Integer :: size,ierr
|| CALL MPI_Comm_size( comm, size, ierr ) ! F90 style
```

- Final parameter always error parameter. Do not forget!
- MPI\_... types are INTEGER.

## 26. About routine prototypes: Python

Prototype:

```
|| # object method
|| MPI.Comm.Send(self, buf, int dest, int tag=0)
|| # class method
|| MPI.Request.Waitall(type cls, requests, statuses=None)
```

Use:

```
|| from mpi4py import MPI
|| comm = MPI.COMM_WORLD
|| comm.Send(sendbuf, dest=other)
|| MPI.Request.Waitall(requests)
```

Note: most functions are methods of the MPI.Comm class.  
(Sometimes of MPI, sometimes other.)

## 27. Processor name

Processes (can) run on physically distinct locations.

```
// procname.c
int name_length = MPI_MAX_PROCESSOR_NAME;
char proc_name[name_length];
MPI_Get_processor_name(proc_name,&name_length);
printf("This process is running on node <<%s>>\n",proc_name);
```

## MPI\_Get\_processor\_name

---

Name	Param name	C type	F type	inout
MPI_Get_processor_name (				
name	char*	CHARACTER		
		A unique specifier for the actual (as opposed to virtual) node.		
resultlen	int*	INTEGER		
		Length (in printable characters) of the result returned in name		

---

## Exercise 3

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!

## 28. In a picture

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
```

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

c111.tacc.utexas.edu

c222.tacc.utexas.edu

## 29. Processor identification

- Processors are organized in ‘communicators’.
- For now only:

```
|| MPI_Comm comm = MPI_COMM_WORLD;
```

- Each process has a ‘rank’ wrt the communicator.

# MPI\_Comm\_size

---

Name	Param name	C type	F type	inout
MPI_Comm_size (				

comm                    MPI\_Comm     TYPE(MPI\_Comm)  
communicator

size                  int\*         INTEGER  
number of processes in the group of comm

---

# MPI\_Comm\_rank

---

Name	Param name	C type	F type	inout
MPI_Comm_rank (				

comm                    MPI\_Comm     TYPE(MPI\_Comm)  
communicator

rank                  int\*         INTEGER  
rank of the calling process in group of comm

---

## 30. Have you been paying attention?

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.

## Exercise 4 (commrank)

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.

## Exercise 5 (commrank)

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

# A practical example

## 31. Functional Parallelism

Parallelism by letting each process do a different thing.

Example: divide up a search space.

Each process knows its rank, so it can find its part of the search space.

## Exercise 6 (prime)

Is the number  $N = 2,000,000,111$  prime? Let each process test a 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?

## Exercise 7

Allocate on each process an array:

```
|| int my_ints[10];
```

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

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

## Part II

### Collectives

## 32. Overview

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

Commands learned:

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

### 33. Technically

Routines can be ‘collective on a communicator’:

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

# Concepts

## 34. Collectives

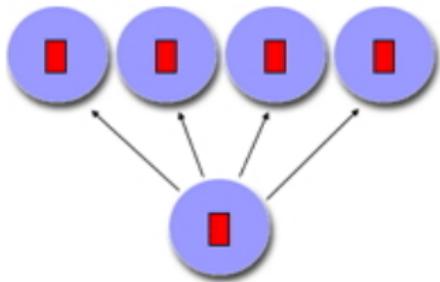
Gathering and spreading information:

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

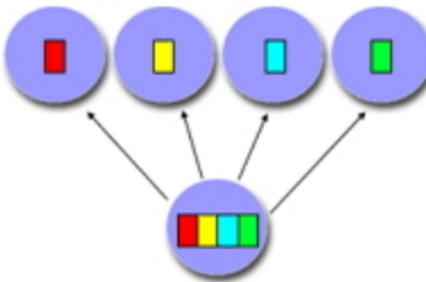
Root process: the one doing the collecting or disseminating.

Basic cases:

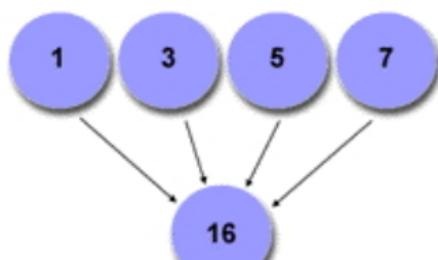
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



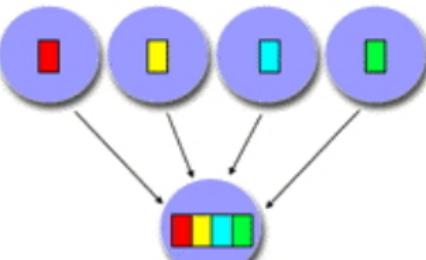
broadcast



scatter



reduction



gather

## Exercise 8

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.

## 35. More collectives

- Instead of a root, collect to all: `MPI_All...`
- Scatter/Gather individual data, but also individual size:  
`MPI_Scatterv`, `MPI_Allgatherv` et cetera.
- Everyone broadcasts: `MPI_Alltoall`
- Scan: like a reduction, but with partial results
- Non-blocking collectives.

# Basic collectives

## 36. Allreduce: reduce-to-all

- `MPI_Allreduce` does the same as:  
`MPI_Reduce` (reduction) followed by `MPI_Bcast` (broadcast)
- One line less code
- Same running time as either, half of reduce-followed-by-broadcast
- (And you don't have to think about who is the root)

## 37. Motivation for allreduce

Example: normalizing a vector

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

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

## 38. Another Allreduce

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 processor stores just one  $x_i$  value.

How do we compute this?

## 39. Allreduce syntax

```
|| int MPI_Allreduce(  
||     const void* sendbuf,  
||     void* recvbuf, int count, MPI_Datatype datatype,  
||     MPI_Op op, MPI_Comm comm)
```

- All processes have send and recv buffer
- (No root argument)
- count is number of items in the buffer: 1 for scalar.  
  > 1: pointwise application of the operator
- MPI\_Datatype is MPI\_INT, MPI\_REAL8 et cetera.
- MPI\_Op is MPI\_SUM, MPI\_MAX et cetera.

## 40. Elementary datatypes

C	Fortran	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	
		internal use
		MPI_Aint MPI_Offset

A bunch more.

## 41. MPI operators

MPI operator	description
<code>MPI_MAX</code>	maximum
<code>MPI_MIN</code>	minimum
<code>MPI_SUM</code>	sum
<code>MPI_PROD</code>	product
<code>MPI LAND</code>	logical and
<code>MPI_BAND</code>	bitwise and
<code>MPI_LOR</code>	logical or
<code>MPI_BOR</code>	bitwise or
<code>MPI_LXOR</code>	logical xor
<code>MPI_BXOR</code>	bitwise xor

A couple more.

## 42. Buffers in C

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

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

## 43. Buffers in Fortran

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

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

## 44. Buffers in C++

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

```
|| vector<float> xx(25);
|| MPI_Send( xx.data(),25,MPI_FLOAT, .... );
|| MPI_Send( &xx[0],25,MPI_FLOAT, .... );
```

## 45. Buffers in Python

For many routines there are two variants:

- lowercase: can send Python objects;  
output is return result

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

this uses pickle: slow.

- uppercase: communicates numpy objects;  
input and output are function argument.

```
result = np.empty(.....)
```

```
comm.Recv(result, ...)
```

basically wrapper around C code: fast

## Exercise 9 (randommax)

Let each process compute a random number, and compute the sum of these numbers using the **MPI\_Allreduce** routine.

$$\xi = \sum_i x_i$$

Each process then scales its value by this sum.

$$x'_i \leftarrow x_i / \xi$$

Compute the sum of the scaled numbers

$$\xi' = \sum_i x'_i$$

and check that it is 1.

## 46. Random numbers in C

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

## 47. Random numbers in Fortran

```
integer :: randsize
integer,allocatable,dimension(:) :: randseed
real :: random_value

call random_seed(size=randsize)
allocate(randseed(randsize))
randseed(:) = 1023*procno
call random_seed(put=randseed)

call random_number(random_value)
```

## 48. Random numbers in Python

```
|| import random  
||  
|| random.seed(procno)  
||  
|| random_value = random.random()
```

## 49. Inner product calculation

Given vectors  $x, y$ :

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

Start out with distributed vectors  $x, y$ ,  
assume same distribution.

Proposed solution:

`MPI_Gather` or `MPI_Allgather` and calculate locally.

Comments?

## 50. Inner product calculation another way

What are (at least two) problems with:

```
|| MPI_Allreduce( &local_inprod, &global_inprod,  
                  localsize,MPI_DOUBLE,MPI_SUM,comm )
```

## 51. Inner product calculation: the right way

Compute local part, then collect local sums.

```
|| local_inprod = 0;  
|| for (i=0; i<localsize; i++)  
||   local_inprod += x[i]*y[i];  
|| MPI_Allreduce( &local_inprod, &global_inprod,  
||                 1,MPI_DOUBLE,MPI_SUM,comm )
```

## Exercise (optional) 10

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

## 52. Reduction to single process

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

## 53. Reduction to root

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

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

## 54. Broadcast

```
|| int MPI_Bcast(  
||     void *buffer, int count, MPI_Datatype datatype,  
||     int root, MPI_Comm comm )
```

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

## 55. Gauss-Jordan elimination

<https://youtu.be/aQYuwatlWME>

## Exercise 11 (jordan)

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

for pivot  $k = 1, \dots, n$

    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.

## Exercise (optional) 12

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

# Scan

## 56. 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.

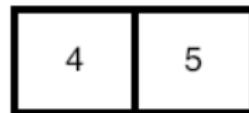
## 57. In vs Exclusive

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

# MPI\_Scan

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

58. For the next exercise



## Exercise 13 (scangather)

- 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 58.)

## Gather/Scatter, Barrier, and others

# MPI\_Gather

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

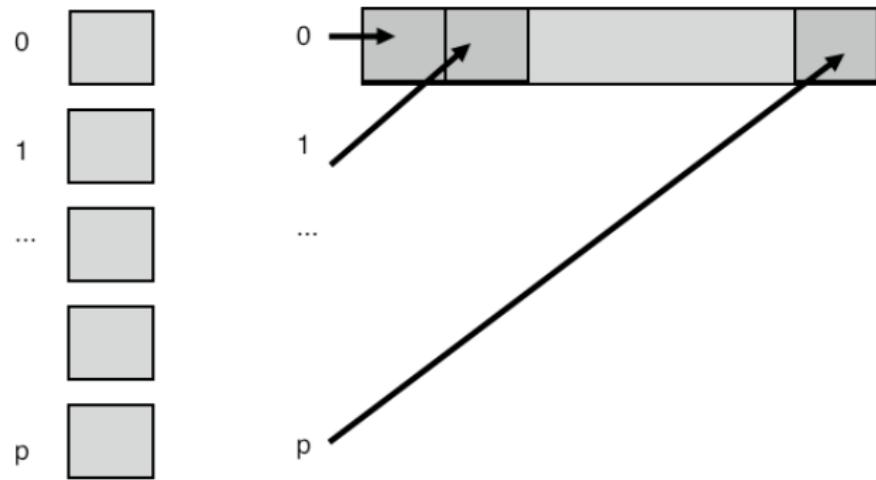
# MPI\_Scatter

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

## 59. Gather/Scatter

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

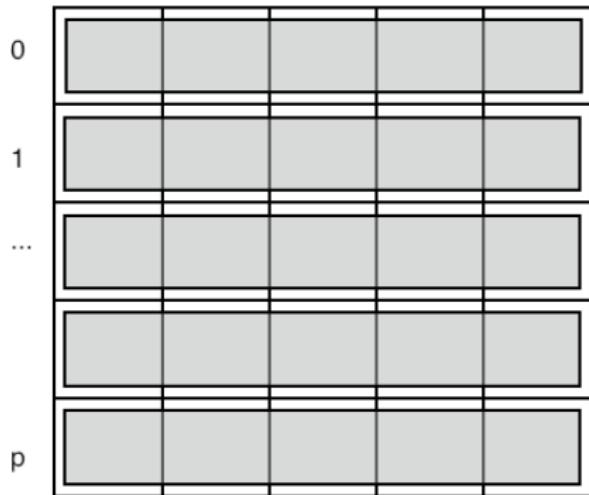
## 60. Gather pictured



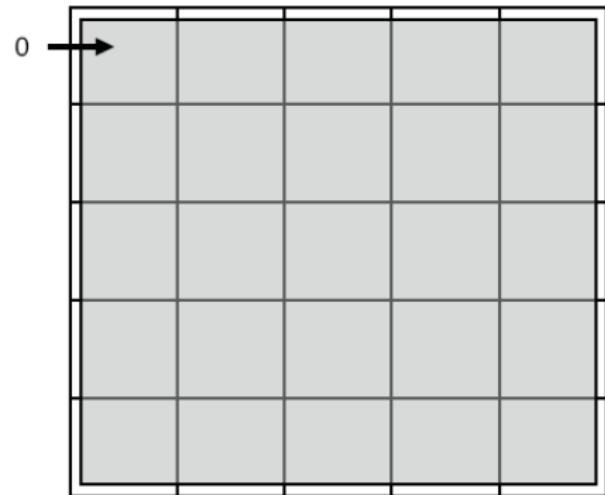
## 61. Popular application of gather

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

distributed matrix



gathered matrix

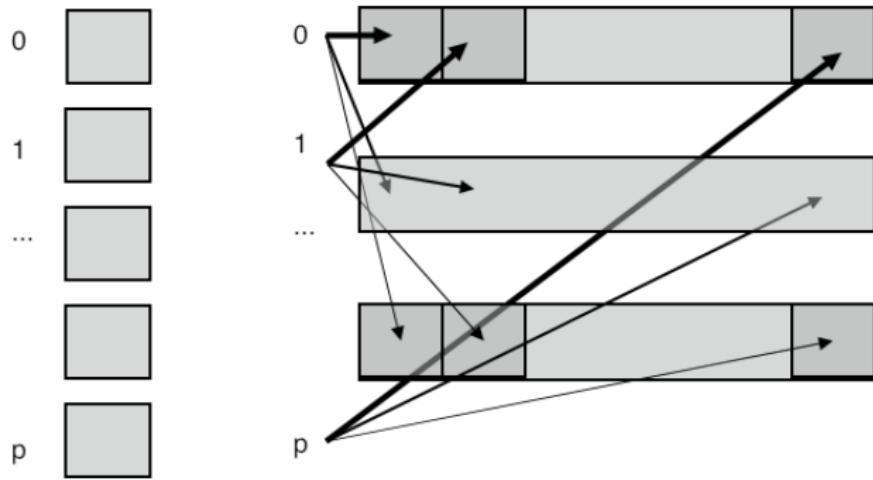


This is not efficient in time or space. Do this only when strictly necessary. Remember SPMD: try to keep everything symmetrically parallel.

# MPI\_Allgather

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

## 62. Allgather pictured



## 63. V-type collectives

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

# MPI\_Gatherv

Name	Param name	C type	F type	inout
	MPI_Gatherv (			
	MPI_Gatherv_c (			
	sendbuf	void*	TYPE(*), DIMENSION(..)	IN
	starting address of send buffer			
(big)	sendcount	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
	number of elements in send buffer			
	sendtype	MPI_Datatype	TYPE(MPI_Datatype)	IN
	datatype of send buffer elements			
	recvbuf	void*	TYPE(*), DIMENSION(..)	OUT
	address of receive buffer			
(big)	recvcounts	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
	non-negative integer array (of length group size) containing the number of ele			
(big)	displs	int MPI_Aint	INTEGER INTEGER(KIND=MPI_ADDRESS_KIND)	IN
	integer array (of length group size). Entry i specifies the displacement relative			
	recvtype	MPI_Datatype	TYPE(MPI_Datatype)	IN
	datatype of recv buffer elements			
	root	int	INTEGER	IN
	rank of receiving process			
	comm	MPI_Comm	TYPE(MPI_	

## Exercise 14 (scangather)

Take the code from exercise 13 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires [MPI\\_Gatherv](#).

How do you construct the lengths and displacements arrays?

# Review 1

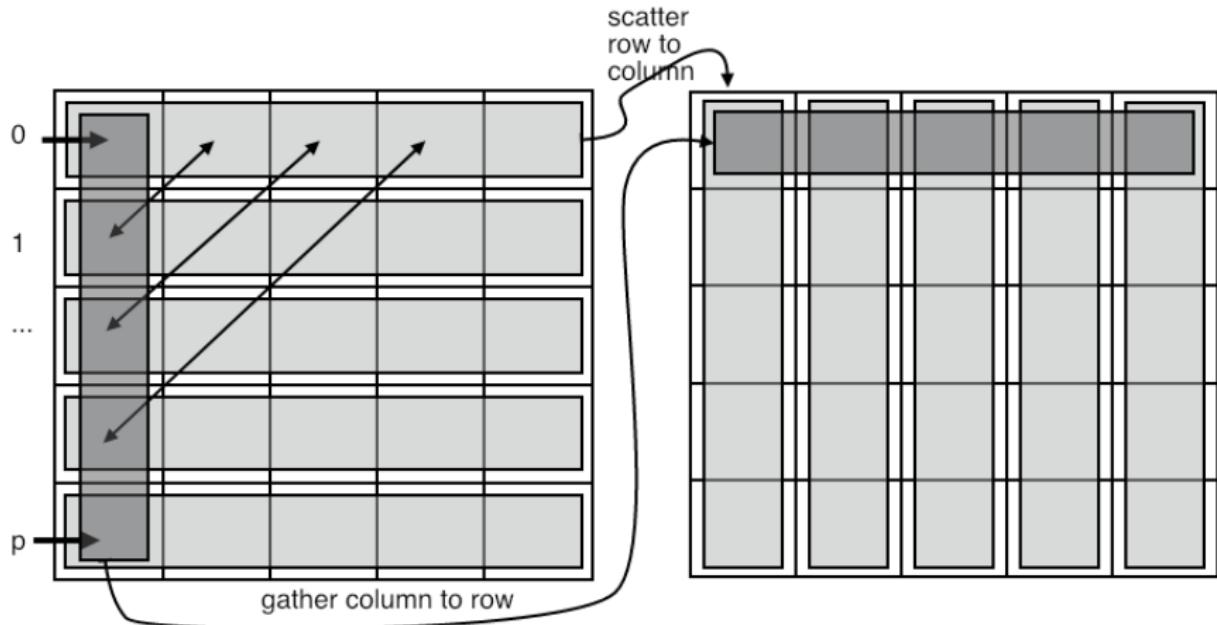
An **MPI\_Scatter** call puts the same data on each process

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

## 64. All-to-all

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

## 65. Data transposition



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

## 66. All-to-allv

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

## 67. Radix sort

Sort 4 numbers on two processes:

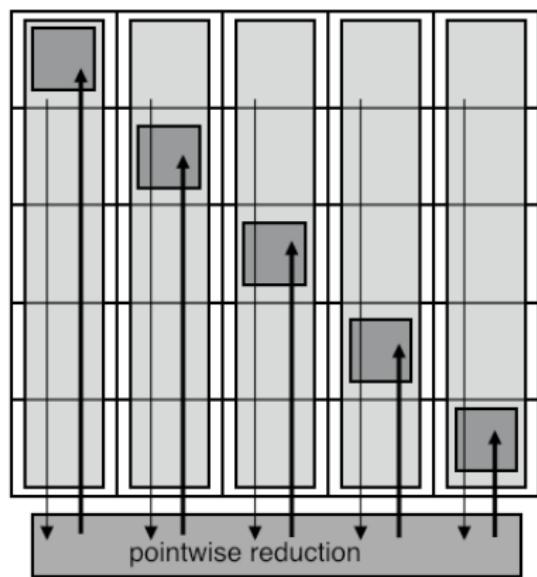
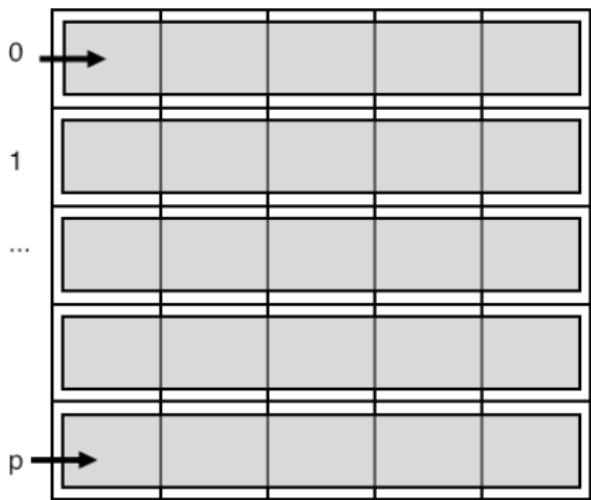
	proc0		proc1	
array	2	5	7	1
binary	010	101	111	001
stage 1				
last digit	0	1	1	1
	(this serves as bin number)			
sorted	010		101	111 001
stage 2				
next digit	1		0	1 0
	(this serves as bin number)			
sorted	101	001	010	111
stage 3				
next digit	1	0	0	1
	(this serves as bin number)			
sorted	001	010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

## 68. Reduce-scatter

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

## 69. Example: sparse matrix setup

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



## 70. Barrier

```
|| 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

# 71. MPI Operators

Define your own reduction operator

- Define operator between partial result and new operand

```
|| typedef void MPI_User_function
```

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

```
|| FUNCTION user_function( invec(*), inoutvec(*), length, mpitype)
```

```
|| <fortranstype> :: invec(length), inoutvec(length)
```

```
|| INTEGER :: length, mpitype
```

- Don't forget to free:

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

- Make your own reduction scheme `MPI_Reduce_local`

# MPI\_Op\_create

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

## 72. Example

Smallest nonzero:

```
// reductpositive.c
void reduce_without_zero(void *in,void *inout,int *len,MPI_Datatype *type)
    ↪ {
// r is the already reduced value, n is the new value
    int n = *(int*)in, r = *(int*)inout;
    int m;
    if (n==0) { // new value is zero: keep r
        m = r;
    } else if (r==0) {
        m = n;
    } else if (n<r) { // new value is less but not zero: use n
        m = n;
    } else { // new value is more: use r
        m = r;
    };
    *(int*)inout = m;
}
```

## Review 2

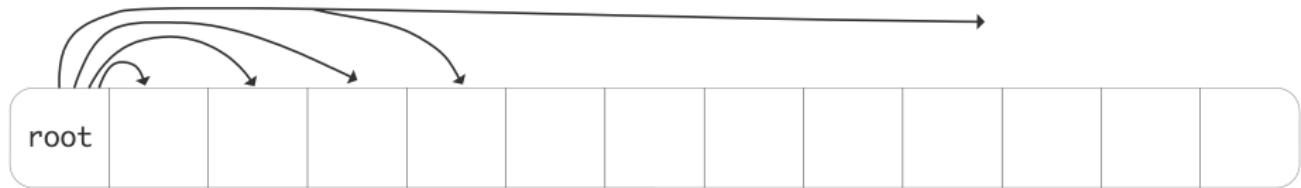
The  $\|\cdot\|_2$  norm (sum of squares) needs a custom operator.

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

# Performance of collectives

## 73. Naive realization of collectives

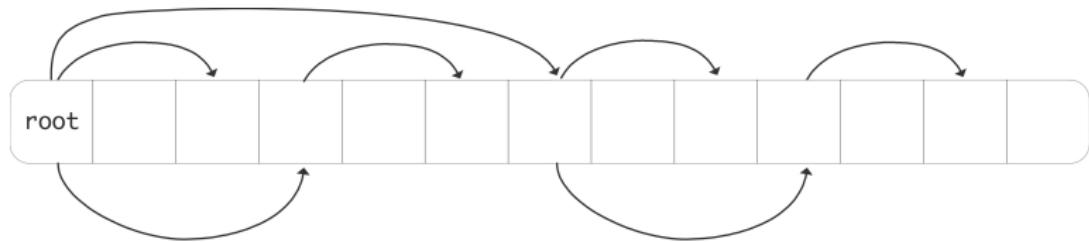
Broadcast:



Single message:

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

## 74. Better implementation of collective



- What is the running time now?

## Review 3

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

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

# Part III

Point-to-point communication

## 75. Overview

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

Commands learned:

- MPI\_Send, MPI\_Recv, MPI\_Sendrecv, MPI\_Isend, MPI\_Irecv
- MPI\_Wait...
- Mention of MPI\_Test, MPI\_Bsend/Ssend/Rsend.

# Point-to-point communication

## 76. MPI point-to-point mechanism

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

## 77. Ping-pong

A sends to B, B sends back to A

What is the code for A? For B?

## 78. Ping-pong in MPI

Remember SPMD:

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

# MPI\_Send

Name	Param name	C type	F type	inout
	MPI_Send (			
	MPI_Send_c (			
	buf	void*	TYPE(*), DIMENSION(..)	IN
		initial address of send buffer		
(big)	count	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
		number of elements in send buffer		
	datatype	MPI_Datatype	TYPE(MPI_Datatype)	IN
		datatype of each send buffer element		
	dest	int	INTEGER	IN
		rank of destination		
	tag	int	INTEGER	IN
		message tag		
	comm	MPI_Comm	TYPE(MPI_Comm)	IN
		communicator		

# MPI\_Recv

Name	Param name	C type	F type	inout
	MPI_Recv (			
	MPI_Recv_c (			
	buf	void*	TYPE(*), DIMENSION(..)	OUT
	initial address of receive buffer			
(big)	count	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
	number of elements in receive buffer			
	datatype	MPI_Datatype	TYPE(MPI_Datatype)	IN
	datatype of each receive buffer element			
	source	int	INTEGER	IN
	rank of source or MPI_ANY_SOURCE			
	tag	int	INTEGER	IN
	message tag or MPI_ANY_TAG			
	comm	MPI_Comm	TYPE(MPI_Comm)	IN
	communicator			
	status	MPI_Status*	TYPE(MPI_Status)	OUT
	status object			

## 79. Status object

Use `MPI_STATUS_IGNORE` unless ...

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

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

## Exercise 15 (pingpong)

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

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

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

# MPI\_Wtime

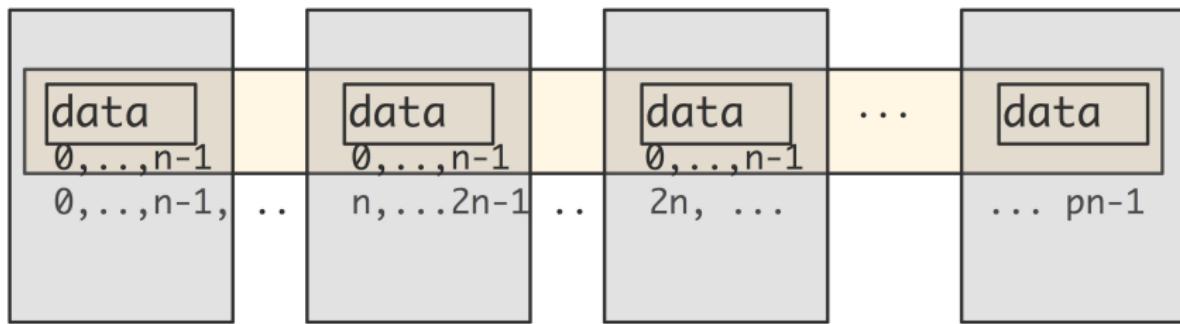
Name	Param name	C type	F type	inout
MPI_Wtime	(			

# Distributed data

## 80. 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’.

## 81. Local and global indexing

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

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

## Exercise (optional) 16

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

## 82. Load balancing

If the distributed array is not perfectly divisible:

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

This gives a load balancing problem. Better solution?

### 83. (for future reference)

Let

$$f(i) = \lfloor iN/p \rfloor$$

and give process  $i$  the points  $f(i)$  up to  $f(i + 1)$ .

Result:

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

# Local information exchange

## 84. Motivation

Partial differential equations:

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

Simple case:

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

Finite difference approximation:

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

## 85. Motivation (continued)

Equations

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

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

So we are interested in sparse/banded matrices.

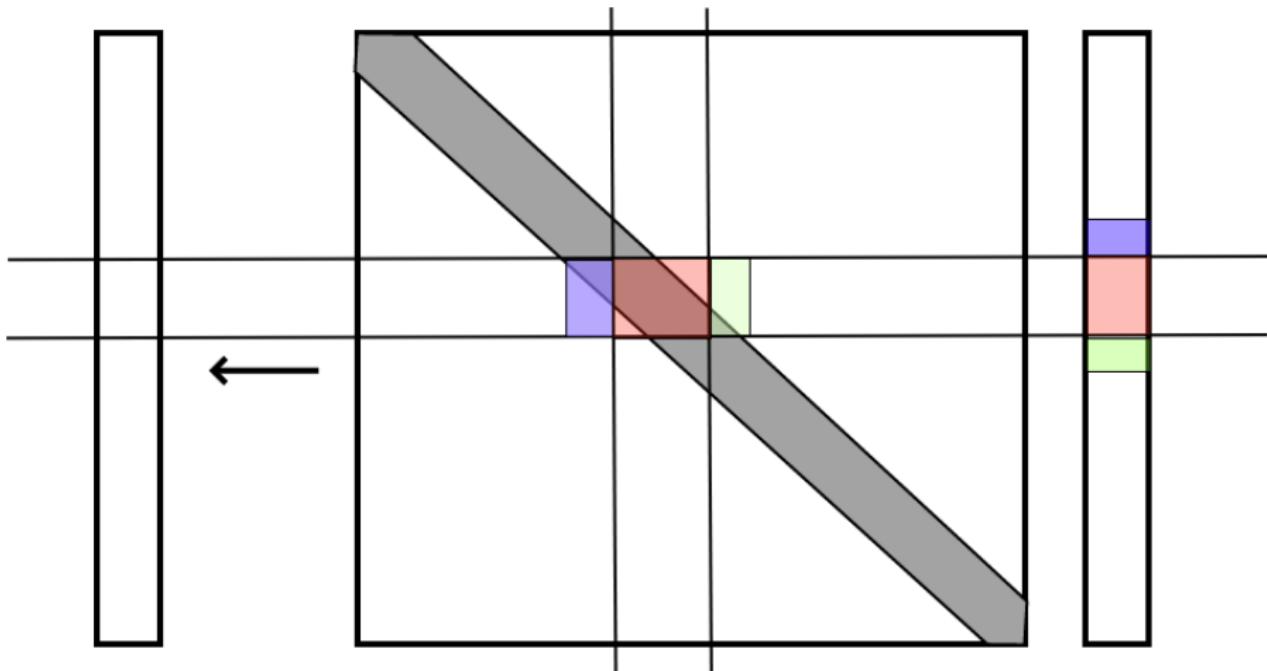
## 86. Matrix vector product

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
- So we need a point-to-point mechanism

## 87. In a picture



## 88. Operating on distributed data

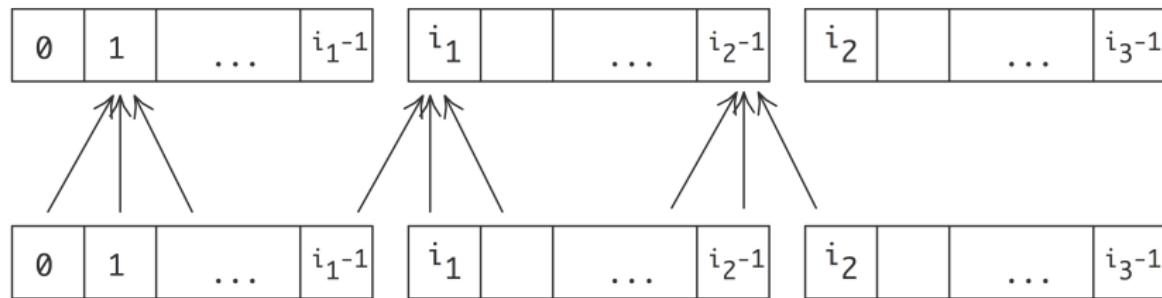
Array of numbers  $x_i : i = 0, \dots, N$

compute

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

'owner computes'

This leads to communication:



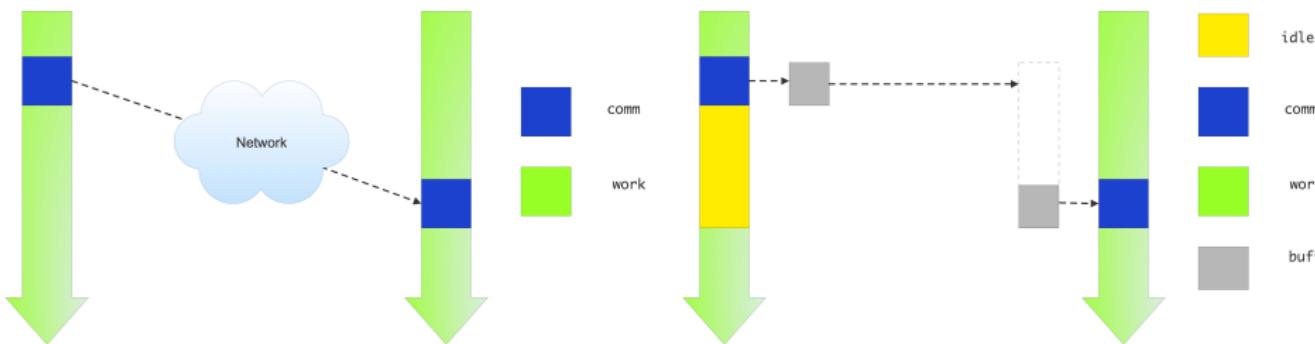
so we need a point-to-point mechanism.

# Blocking communication

## 89. Blocking send/recv

`MPI_Send` and `MPI_Recv` are blocking operations:

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



Ideal vs actual send/recv behaviour.

## 90. Deadlock

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

A subtlety.

This code may actually work:

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

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

## 91. Protocol

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

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

Small messages bypass this: ‘eager’ send.

Definition of ‘small message’ controlled by environment variables:

`I_MPI_EAGER_THRESHOLD MV2_IBA_EAGER_THRESHOLD`

## Exercise 17

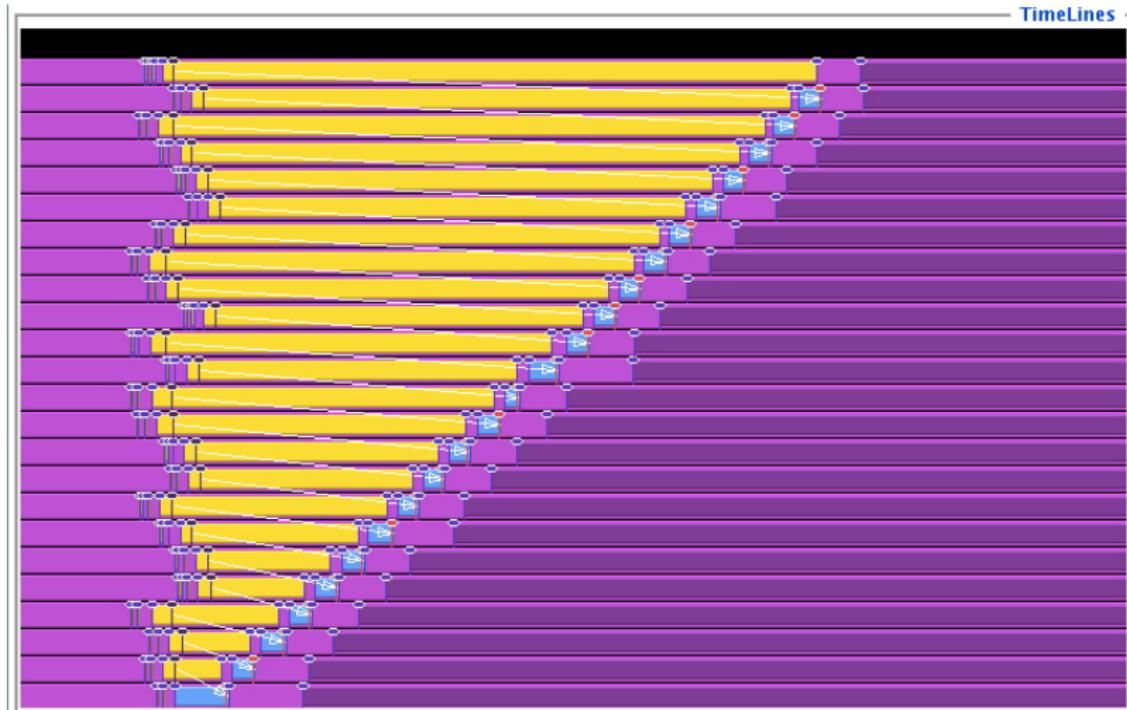
(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

- 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.

## 92. TAU trace: serialization



## 93. The problem here...

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

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

Food for thought: what happens if you flip the send and receive call?

## Exercise (optional) 18

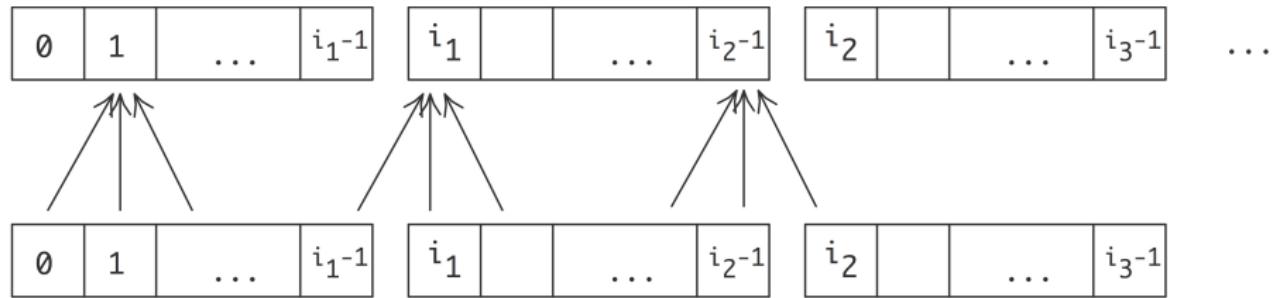
Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls.  
Run the code, and use TAU to reproduce the trace output of figure 92.  
If you don't have TAU, can you show this serialization behavior using  
timings, for instance running it on an increasing number of processes?

# Pairwise exchange

## 94. Operating on distributed data

Take another look:

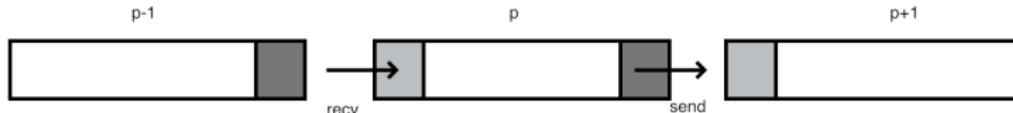
$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \dots, N - 1$$



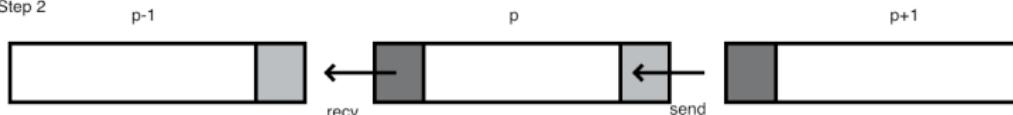
- One-dimensional data and linear process numbering;
- Operation between neighboring indices: communication between neighboring processes.

## 95. Two steps

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!

## 96. Sendrecv

Instead of separate send and receive: use

### MPI\_Sendrecv

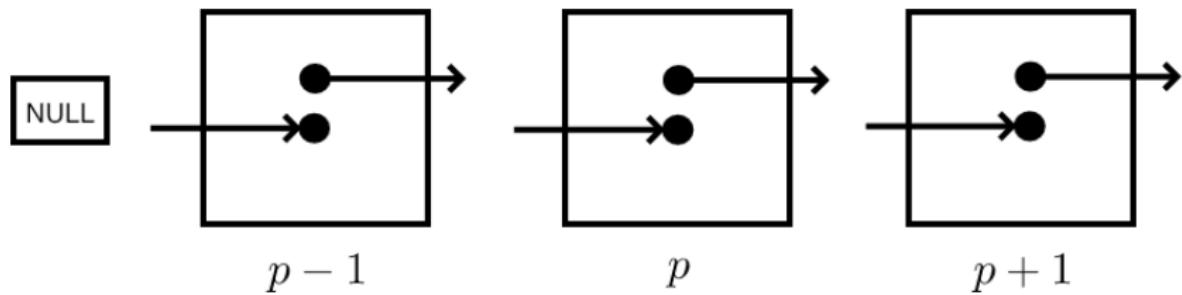
Combined calling sequence of send and receive;  
execute such that no deadlock or sequentialization.

# MPI\_Sendrecv

Name	Param name	C type	F type	inout
	MPI_Sendrecv (			
	MPI_Sendrecv_c (			
	sendbuf	void*	TYPE(*), DIMENSION(..)	IN
	initial address of send buffer			
(big)	sendcount	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
	number of elements in send buffer			
	sendtype	MPI_Datatype	TYPE(MPI_Datatype)	IN
	type of elements in send buffer			
	dest	int	INTEGER	IN
	rank of destination			
	sendtag	int	INTEGER	IN
	send tag			
	recvbuf	void*	TYPE(*), DIMENSION(..)	OUT
	initial address of receive buffer			
(big)	recvcount	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
	number of elements in receive buffer			
	recvtype	MPI_Datatype	TYPE(MPI_Datatype)	IN
	type of elements receive buffer element			
	source	int	INTEGER	
	rank of source or MPI_ANY_SOURCE			

## 97. SPMD picture

What does process  $p$  do?



## 98. Sendrecv with incomplete pairs

```
MPI_Comm_rank( .... &procno );
if ( /* I am not the first process */
    predecessor = procno-1;
else
    predecessor = MPI_PROC_NULL;

if ( /* I am not the last process */
    successor = procno+1;
else
    successor = MPI_PROC_NULL;

sendrecv(from=predecessor,to=successor);
```

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

## 99. A point of programming style

The previous slide had:

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

Also possible:

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

But:

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

## Exercise (optional) 19 (rightsend)

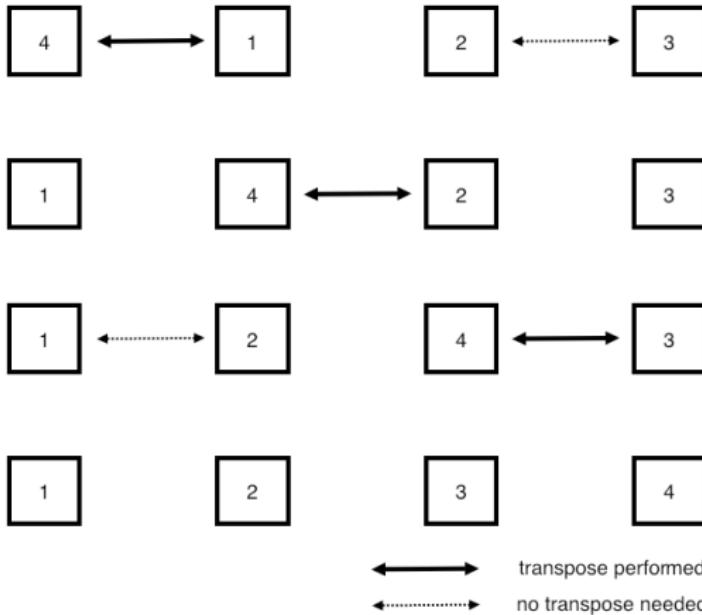
Revisit exercise 17 and solve it using `MPI_Sendrecv`.

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

## Exercise 20 (sendrecv)

Implement the above three-point combination scheme using `MPI_Sendrecv`; every processor only has a single number to send to its neighbor.

## 100. Odd-even transposition sort



Odd-even transposition sort on 4 elements.

## Exercise (optional) 21

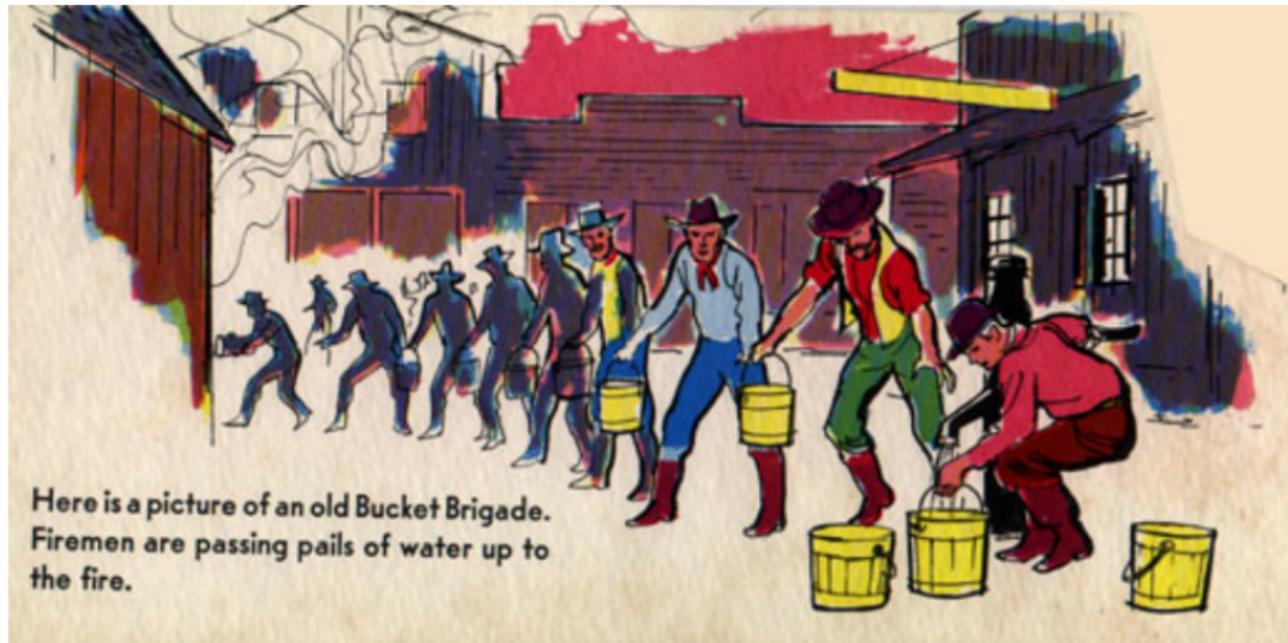
A very simple sorting algorithm is swap sort or odd-even transposition sort: pairs of processors compare data, and if necessary exchange. The elementary step is called a compare-and-swap: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the maximum.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

The exchange sort algorithm is split in even and odd stages, where in the even stage, processors  $2i$  and  $2i+1$  compare and swap data, and in the odd stage, processors  $2i+1$  and  $2i+2$  compare and swap. You need to repeat this  $P/2$  times, where  $P$  is the number of processors; see figure 100.

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.

## 101. Bucket brigade

Sometimes you really want to pass information from one process to the next: ‘bucket brigade’



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

## Exercise 22 (bucketblock)

Take the code of exercise 18 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.

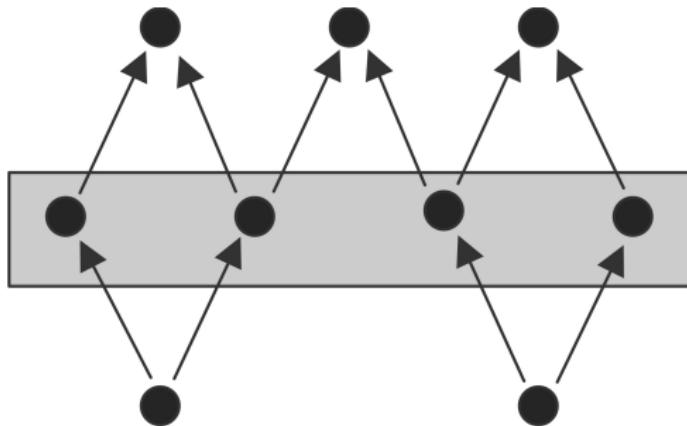
Food for thought: all quantities involved here are integers. Is it a good idea to use the integer datatype here?

Question: could you have done this with a collective call?

## Irregular exchanges: non-blocking communication

## 102. Sending with irregular connections

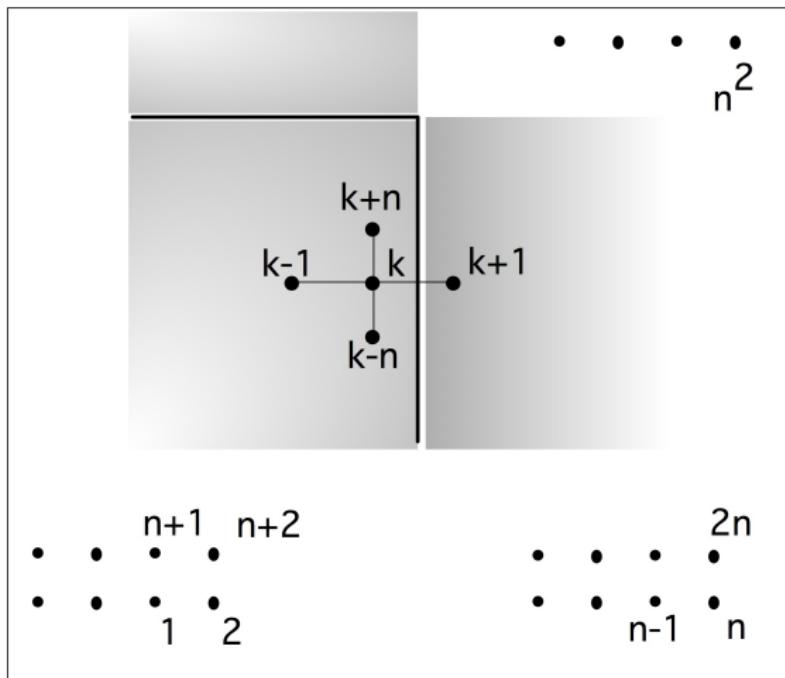
Graph operations:



# Communicating other than in pairs

## 103. PDE, 2D case

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

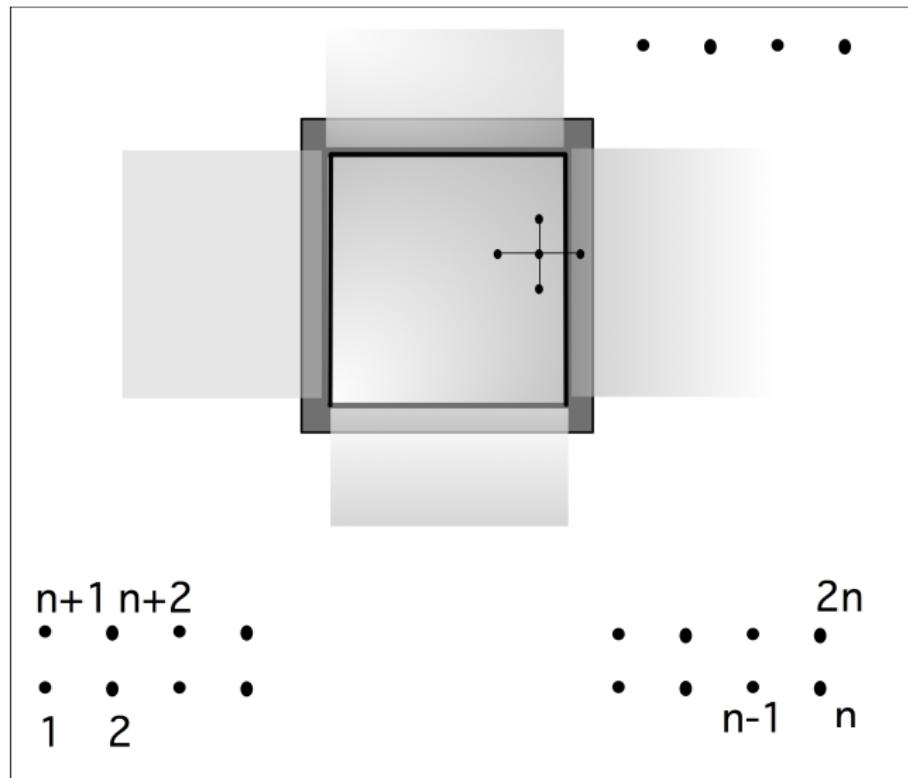


## 104. PDE matrix

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

## 105. Halo region

The halo region of a process, induced by a stencil



## 106. How do you approach this?

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

Instead:

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

## 107. Non-blocking send/recv

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

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

## 108. Syntax

Very much like blocking `MPI_Send/MPI_Recv`:

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

Most common way of waiting for completion:

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

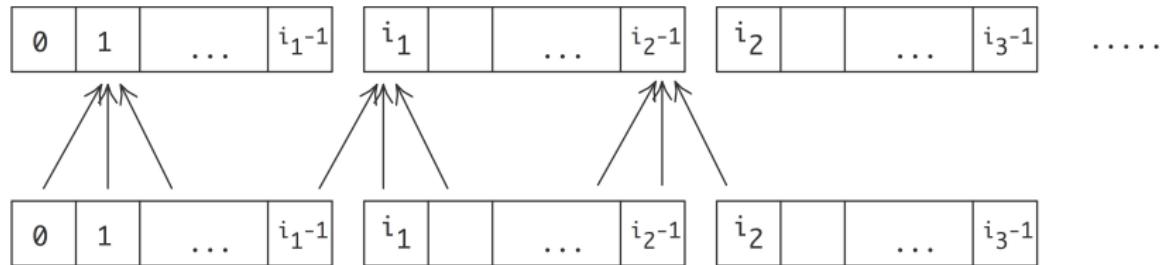
- ignore status: `MPI_STATUSES_IGNORE`
- also `MPI_Wait`, `MPI_Waitany`, `MPI_Waitsome`

## Exercise 23 (isendrecv)

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



(Can you think of a different way of handling the end points?)

## 109. Comparison

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

## 110. Buffer use in blocking/non-blocking case

Blocking:

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

Non-blocking:

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

# 111. Latency hiding

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

Also known as ‘latency hiding’.

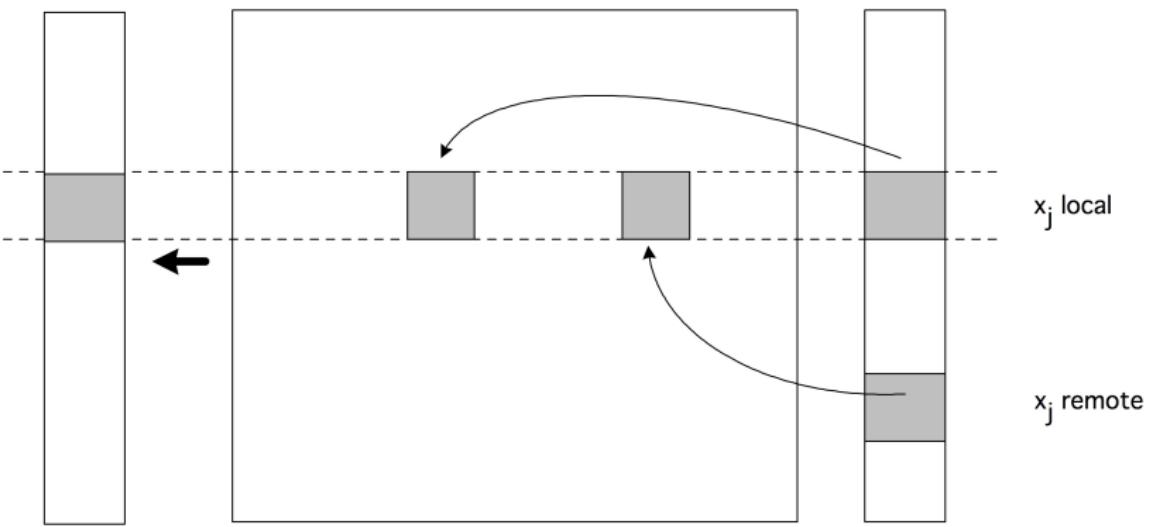
Example: three-point combination operation (see above):

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

## 112. Matrices in parallel

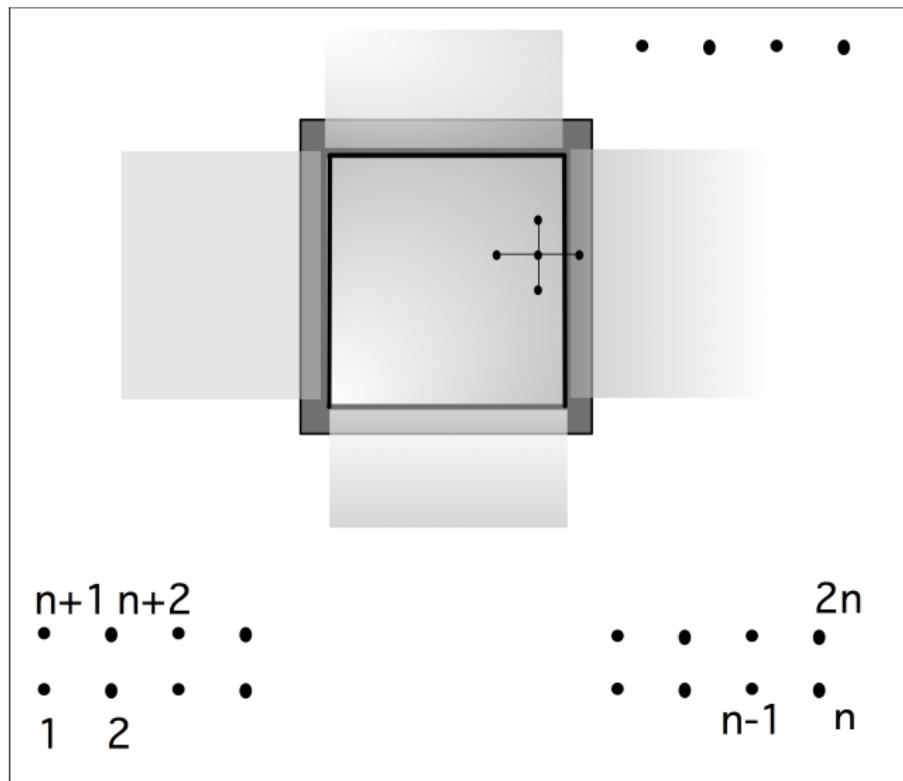
$$y \leftarrow Ax$$

and A, x, y all distributed:



## 113. Hiding the halo

Interior of a process domain can overlap with halo transfer:



## Exercise 24 (isendirecvarray)

Take your code of exercise 23 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.

## 114. Test: non-blocking wait

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

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

## 115. The Pipeline Pattern

- Remember the bucket brigade: data propagating through processes
- If you have many buckets being passed: pipeline
- This is very parallel: only filling and draining the pipeline is not completely parallel
- Application to long-vector broadcast: pipelining gives overlap

## Exercise (optional) 25 (bucketpipenonblock)

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

## Exercise 26 (setdiff)

Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.

How could you solve this with an `MPI_Allgather` call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.

## 116. The wheel of reinvention

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

## 117. More sends and receive

- `MPI_Bsend`, `MPI_Ibsend`: buffered send
- `MPI_Ssend`, `MPI_Issend`: synchronous send
- `MPI_Rsend`, `MPI_Irsend`: ready send
- Persistent communication: repeated instance of same proc/data description.

MPI-4:

- Partitioned sends.

too obscure to go into.

# Review 4

Does this code deadlock?

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

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

# Review 5

Does this code deadlock?

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

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

# Review 6

Does this code deadlock?

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

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

# Review 7

Does this code deadlock?

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

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

# Where to go from here...

- 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

# Justification

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

## 118. Overview

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

## 119. Motivation: datatypes in MPI

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.

## 120. Datatype topics

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

# Datatypes

## 121. Elementary 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

## 122. How to use derived types

Create, commit, use, free:

```
|| MPI_Datatype newtype;  
|| MPI_Type_xxx( ... oldtype ... &newtype);  
|| MPI_Type_commit ( &newtype );  
  
|| // code using the new type  
  
|| MPI_Type_free ( &newtype );  
  
|| Type(MPI_Datatype) :: newtype ! F2008  
|| Integer          :: newtype ! F90
```

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

## 123. Contiguous type

```
|| int MPI_Type_contiguous(  
||   int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)
```



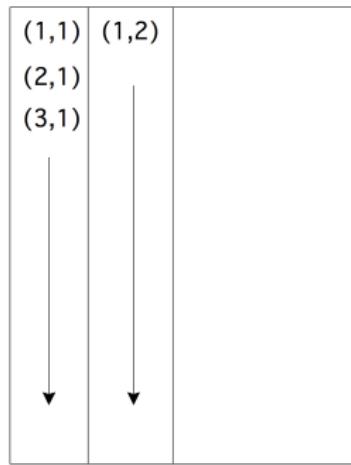
This one is indistinguishable from just sending count instances of the old\_type.

## 124. Example: non-contiguous data

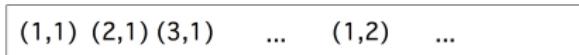
Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

Logical:



Physical:



## 125. Vector type

```
|| int MPI_Type_vector(  
||   int count, int blocklength, int stride,  
||   MPI_Datatype old_type, MPI_Datatype *newtype_p  
|| );
```



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

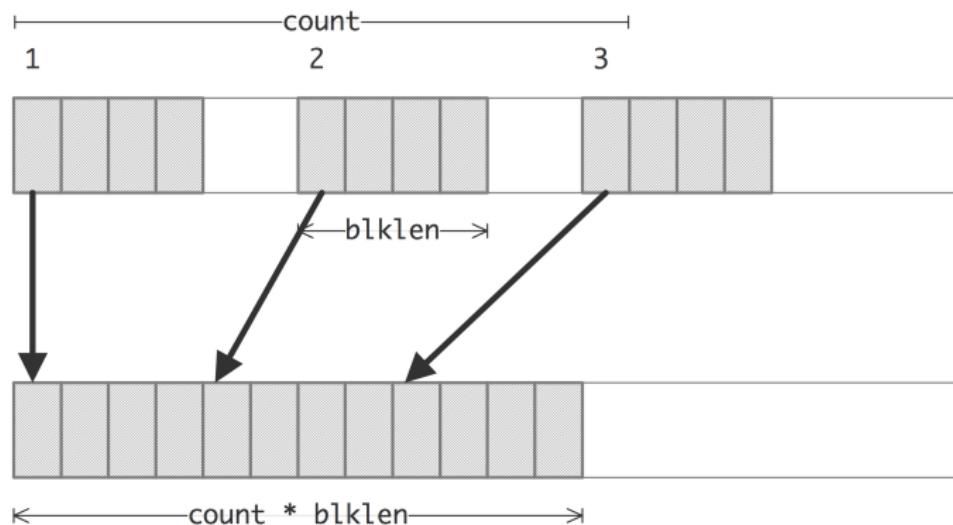
```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}
```

## 127. Different send and receive types

Send and receive type can differ. Example:

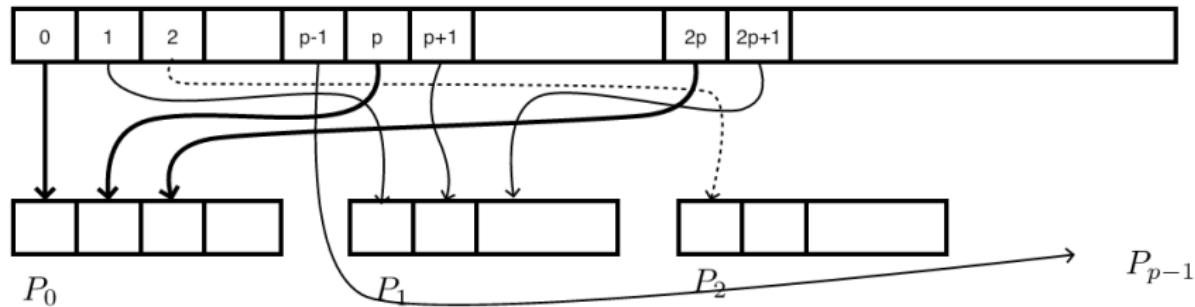
Sender type: vector

receiver type: contiguous or elementary



Receiver has no knowledge of the stride of the sender.

## 128. Illustration of the next exercise



Sending strided data from process zero to all others

## Exercise 27 (stridesend)

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.

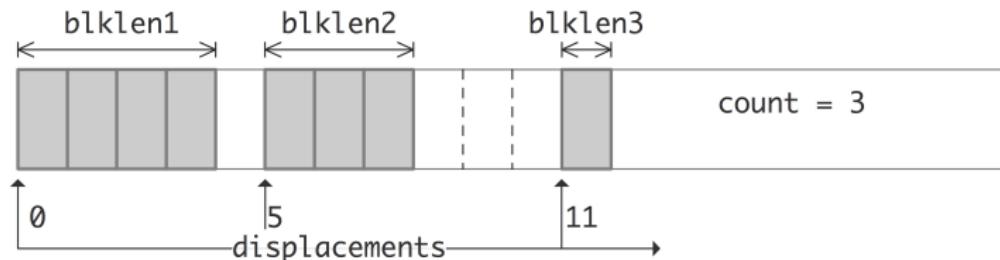
For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?

For testing, define the array as  $x[i] = i$ .

## Exercise 28

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

## 129. Indexed type



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

## 130. Hindexed type

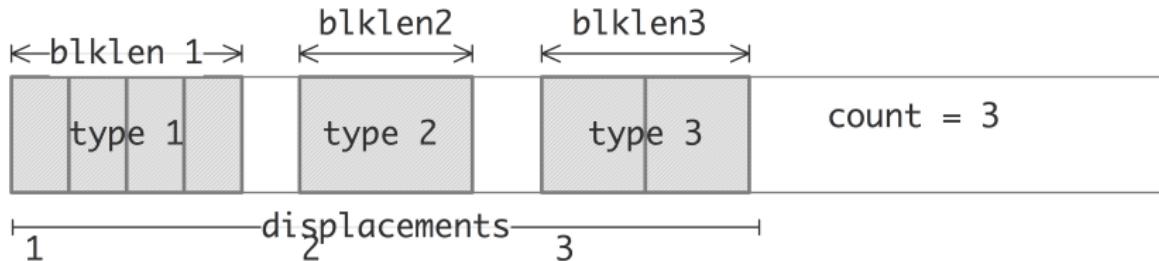
Similar to indexed but using byte offsets:  
explicit memory address.

Example usage scenario: send linked list.

Use `MPI_Get_address`

## 131. Heterogeneous: Structure type

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

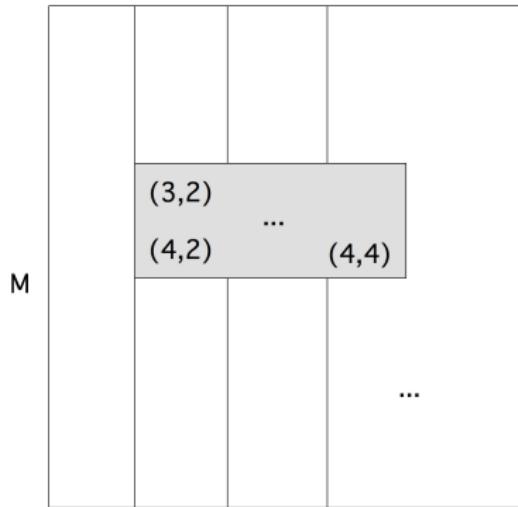


This gets very tedious...

## Subarray type

## 132. Submatrix storage

Logical:



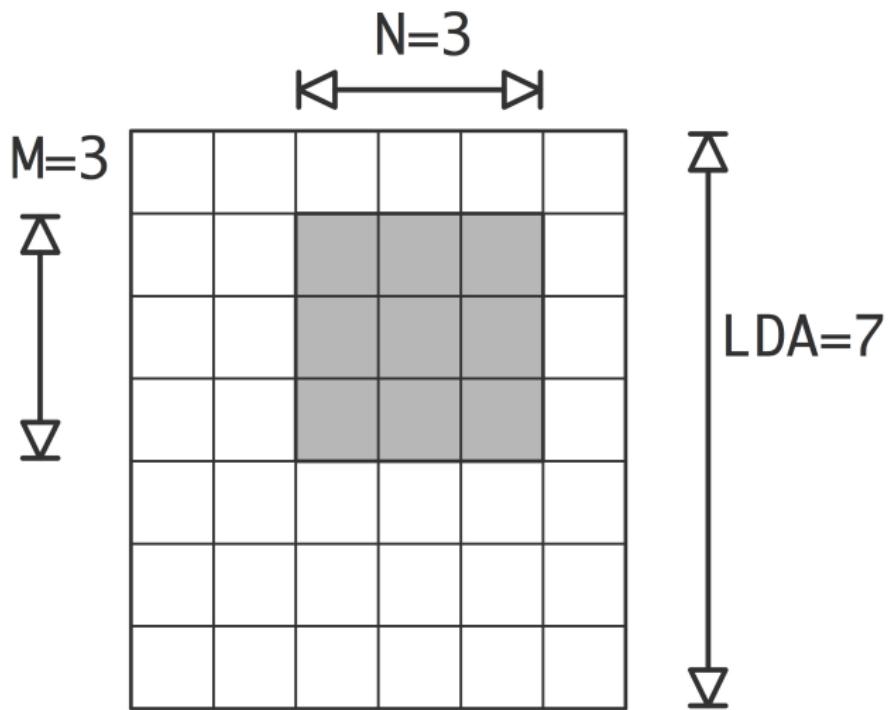
Physical:



- Location of first element
- Stride, blocksize

## 133. BLAS/Lapack storage

Three parameter description:



How about as a ‘block within a block’?  
Eijkhout: MPI course

## 134. Subarray type

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

```
|| MPI_Type_create_subarray(  
||   ndims, array_of_sizes, array_of_subsizes,  
||   array_of_starts, order, oldtype, newtype)
```

Subtle: data does not start at the buffer start

## Exercise 29 (cubegather)

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.

## 135. Fortran ‘kind’ types

Check out `MPI_Type_create_f90_integer`, `MPI_Type_create_f90_real`,  
`MPI_Type_create_f90_complex`

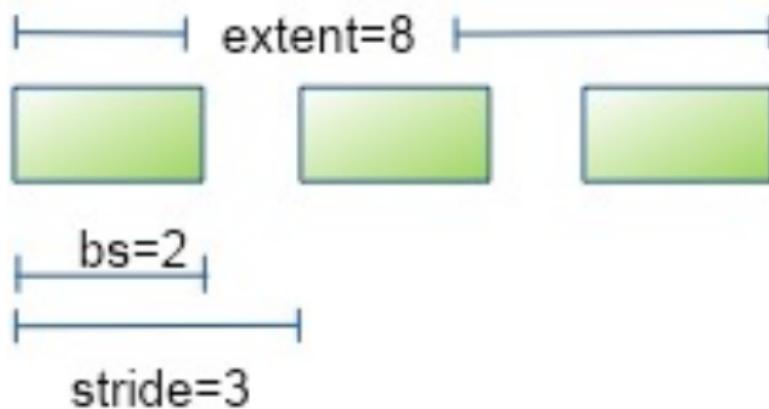
Example:

```
|| REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
|| DIMENSION(100) :: array
|| Type(MPI_Datatype) :: realltype
|| CALL MPI_Type_create_f90_real( 15 , 300 , realltype , error )
```

# Extent and resizing

## 136. Extent

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



## 137. Extent computation

```
|| MPI_Aint lb,asize;
|| MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
|| MPI_Type_commit(&newtype);
|| MPI_Type_get_extent(newtype,&lb,&asize);
|| ASSERT( lb==0 );
|| ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
|| MPI_Type_free(&newtype);
```

## 138. Extent of subarray type

The ‘subarray’ type:

data does not start at the start of the type.

`MPI_Type_get_true_extent` returns non-zero lower bound.

## 139. Extent resizing: enlarging

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



## 140. Naive code

Send:

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

Recv:

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

giving an output of:

Receive 6 elements: 0 2 4 5 7 9

## 141. Resizing code

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

giving:

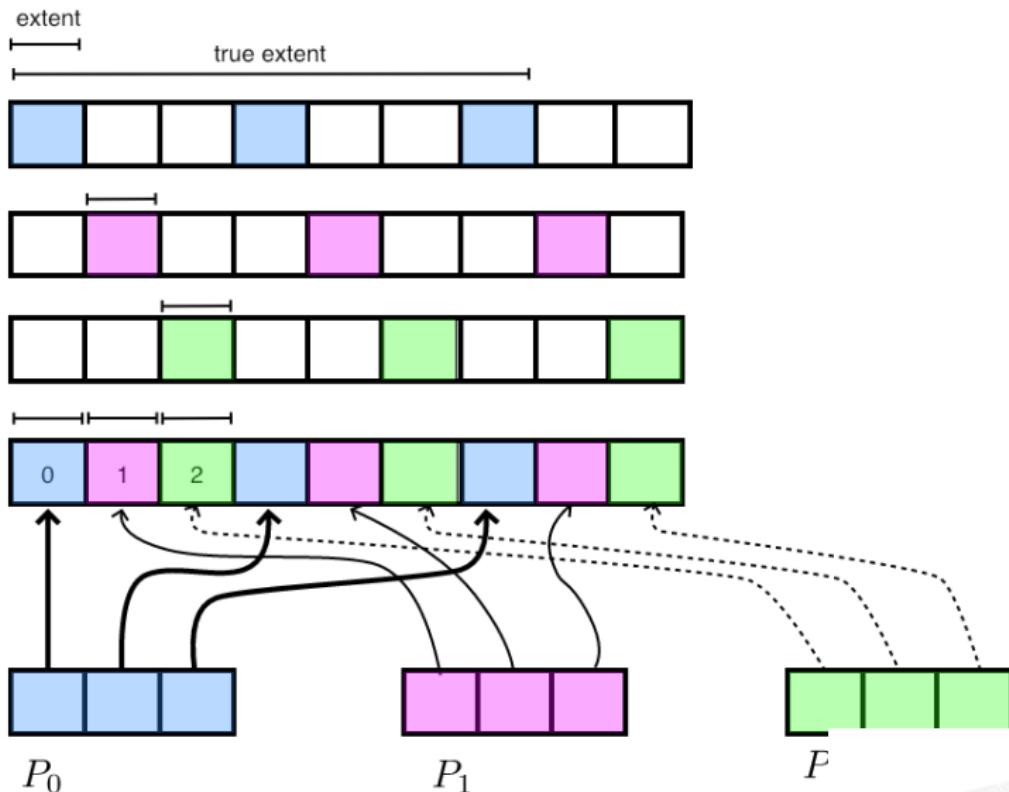
Strided type l=0 e=20

Padded type l=0 e=24

Receive 6 elements: 0 2 4 6 8 10

## 142. Extent resizing: shrinking

Elements are placed at distance equal to extent:



## Exercise 30 (stridesend)

Rewrite exercise 27 to use a gather, rather than individual messages.

# Packed data

## 143. Packing into buffer

```
int MPI_Pack(  
    void *inbuf, int incount, MPI_Datatype datatype,  
    void *outbuf, int outcount, int *position,  
    MPI_Comm comm);  
  
int MPI_Unpack(  
    void *inbuf, int insize, int *position,  
    void *outbuf, int outcount, MPI_Datatype datatype,  
    MPI_Comm comm);
```

## 144. Example

```
// pack.c
if (procno==sender) {
    MPI_Pack(&nstarts,1,MPI_INT,buffer,buflen,&position,comm);
    for (int i=0; i<nstarts; i++) {
        double value = rand()/(double)RAND_MAX;
        MPI_Pack(&value,1,MPI_DOUBLE,buffer,buflen,&position,comm);
    }
    MPI_Pack(&nstarts,1,MPI_INT,buffer,buflen,&position,comm);
    MPI_Send(buffer,position,MPI_PACKED,other,0,comm);
} else if (procno==receiver) {
    int irecv_value;
    double xrecv_value;
    MPI_Recv(buffer,buflen,MPI_PACKED,other,0,comm,
              →MPI_STATUS_IGNORE);
    MPI_Unpack(buffer,buflen,&position,&nstarts,1,MPI_INT,comm);
    for (int i=0; i<nstarts; i++) {
        MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,MPI_DOUBLE,
                   →comm);
    }
    MPI_Unpack(buffer,buflen,&position,&irecv_value 1 MPI INT comm).
    ASSERT(irecv_value==nstarts);
}
```

## Part V

### Communicator manipulations

## 145. Overview

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.

## 146. Sub-computations

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`.

## 147. Communicator duplication

Simplest new communicator: identical to a previous one.

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

This is useful for library writers:

```
|| MPI_Isend(...); MPI_Irecv(...);  
|| // library call  
|| 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.

## 148. Interleaved library and user code

```
library my_library(comm);
MPI_Isend(&senddata,1,MPI_INT,other,1,comm,&(request[0]));
my_library.communication_start();
MPI_Irecv(&recvdata,1,MPI_INT,other,MPI_ANY_TAG,
            comm,&(request[1]));
MPI_Waitall(2,request,status);
my_library.communication_end();
```

## 149. Library internally has messages

```
int library::communication_start() {
    int sdata=6,rdata;
    MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
    MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
              comm,&(request[1]));
    return 0;
}

int library::communication_end() {
    MPI_Status status[2];
    MPI_Waitall(2,request,status);
    return 0;
}
```

## 150. Wrong way of setting up the library

```
// commdupwrong.cxx
class library {
private:
    MPI_Comm comm;
    int procno,nprocs,other;
    MPI_Request request[2];
public:
    library(MPI_Comm incomm) {
        comm = incomm;
        MPI_Comm_rank(comm,&procno);
        other = 1-procno;
    };
    int communication_start();
    int communication_end();
};
```

## 151. Right way of setting up the library

```
// commdupright.cxx
class library {
private:
    MPI_Comm comm;
    int procno,nprocs,other;
    MPI_Request request[2];
public:
    library(MPI_Comm incomm) {
        MPI_Comm_dup(incomm,&comm);
        MPI_Comm_rank(comm,&procno);
        other = 1-procno;
    };
    ~library() {
        MPI_Comm_free(&comm);
    }
    int communication_start();
    int communication_end();
};
```

## 152. Disjoint splitting

Split a communicator in multiple disjoint others.

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

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

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

## 153. Row/column example

Simulate a processor grid  
create subcommunicator per column (or row):

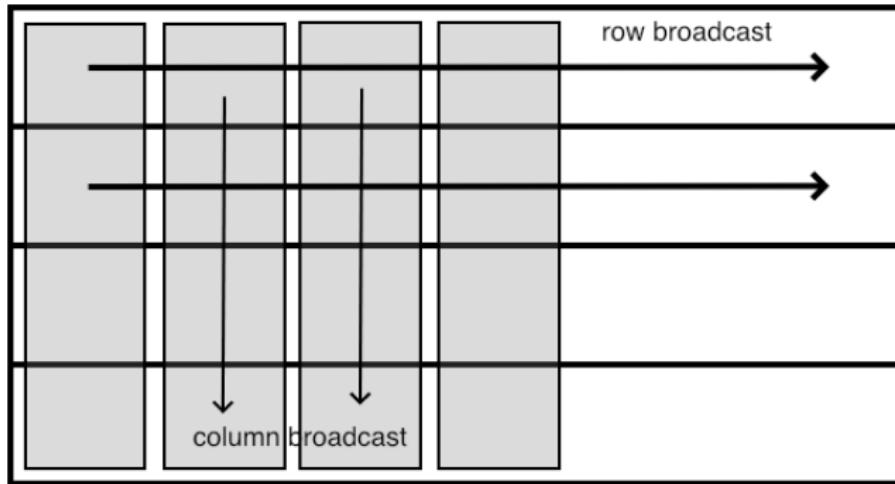
```
MPI_Comm_rank( MPI_COMM_WORLD, &procno );
proc_i = procno % proc_column_length;
proc_j = procno / proc_column_length;

MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, procno, &column_comm
    → );

MPI_Bcast( data, ... column_comm );
```

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

## 154. Row and column communicators



Row and column broadcasts in subcommunicators

## Exercise 31 (procgrid)

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 colum:

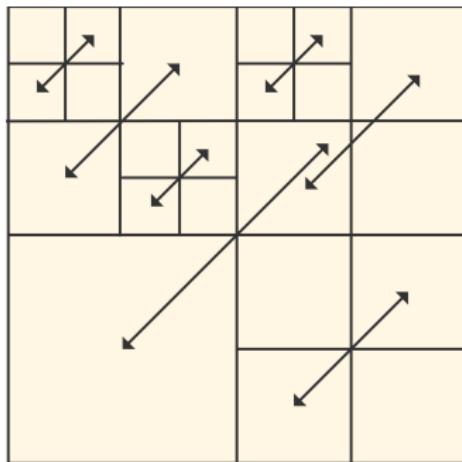
0 1 2	0 1 2	0 0 0
3 4 5	0 1 2	1 1 1

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`.

## Exercise 32

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.

## 155. Splitting by shared memory

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

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type(MPI_COMM_WORLD,
    ↪MPI_COMM_TYPE_SHARED,procno,info,&sharedcomm);
MPI_Comm_size(sharedcomm,&new_nprocs);
MPI_Comm_rank(sharedcomm,&new_procno);
```

## 156. Inter-communicators

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

## 157. In a picture

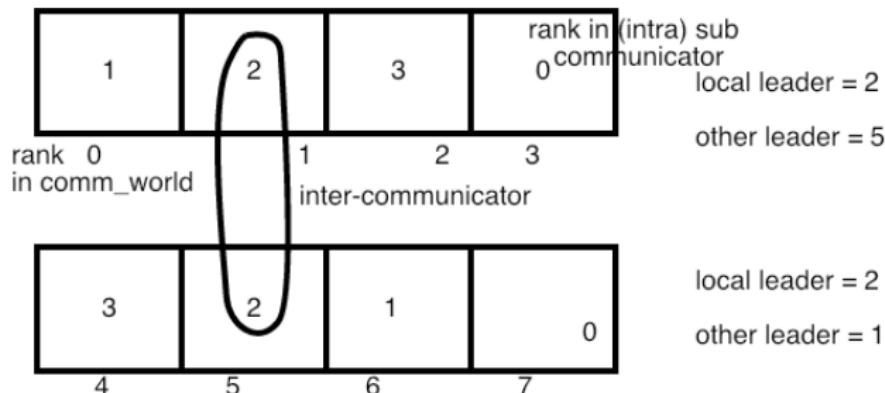


Illustration of ranks in an inter-communicator setup

```
// intercomm.c
MPI_Comm intercomm;
MPI_Intercomm_create
/* local_comm: */ split_half_comm,
/* local_leader: */ local_leader_in_inter_comm,
/* peer_comm: */ MPI_COMM_WORLD,
/* remote_peer_rank: */ global_rank_of_other_leader,
/* tag: */ inter_tag,
/* newintercomm: */ &intercomm );
```

## 158. Concepts

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

## 159. Routines

- `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?

## 160. More

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

# Part VI

## MPI File I/O

# 161. Overview

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

## 162. The trouble with parallel I/O

- 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.

## 163. MPI I/O

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

## 164. The usual bits

```
|| MPI_File mpifile;
|| MPI_File_open(comm,"blockwrite.dat",
||   MPI_MODE_CREATE | MPI_MODE_WRONLY,MPI_INFO_NULL
||   ↗,
||   &mpifile);
|| if (procno==0) {
||   MPI_File_write
||     (mpifile,output_data,nwords,MPI_INT,MPI_STATUS_IGNORE);
|| }
|| MPI_File_close(&mpifile);

|| type(MPI_File) :: mpifile ! F08
|| integer       :: mpifile ! F90
```

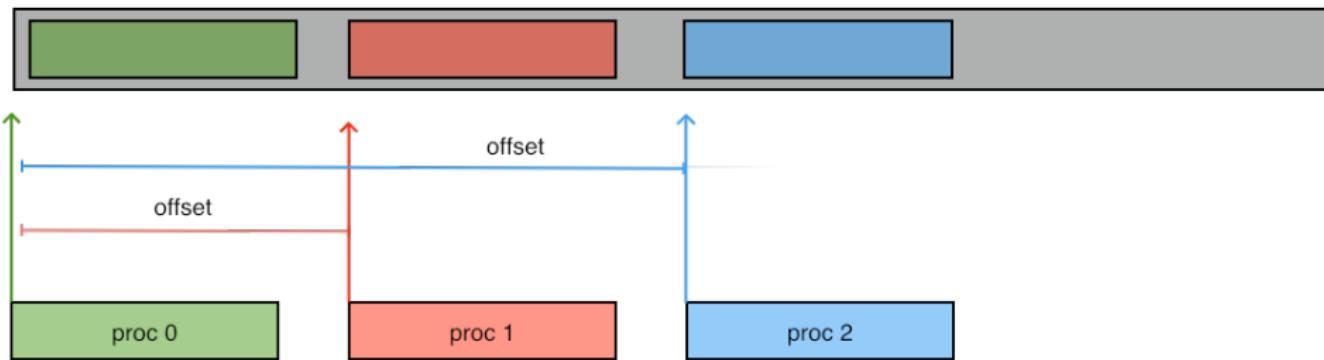
## 165. How do you make it unique for a process?

```
|| MPI_File_write_at  
||   (mpifile,offset,output_data,nwords,  
||     MPI_INT,MPI_STATUS_IGNORE);
```

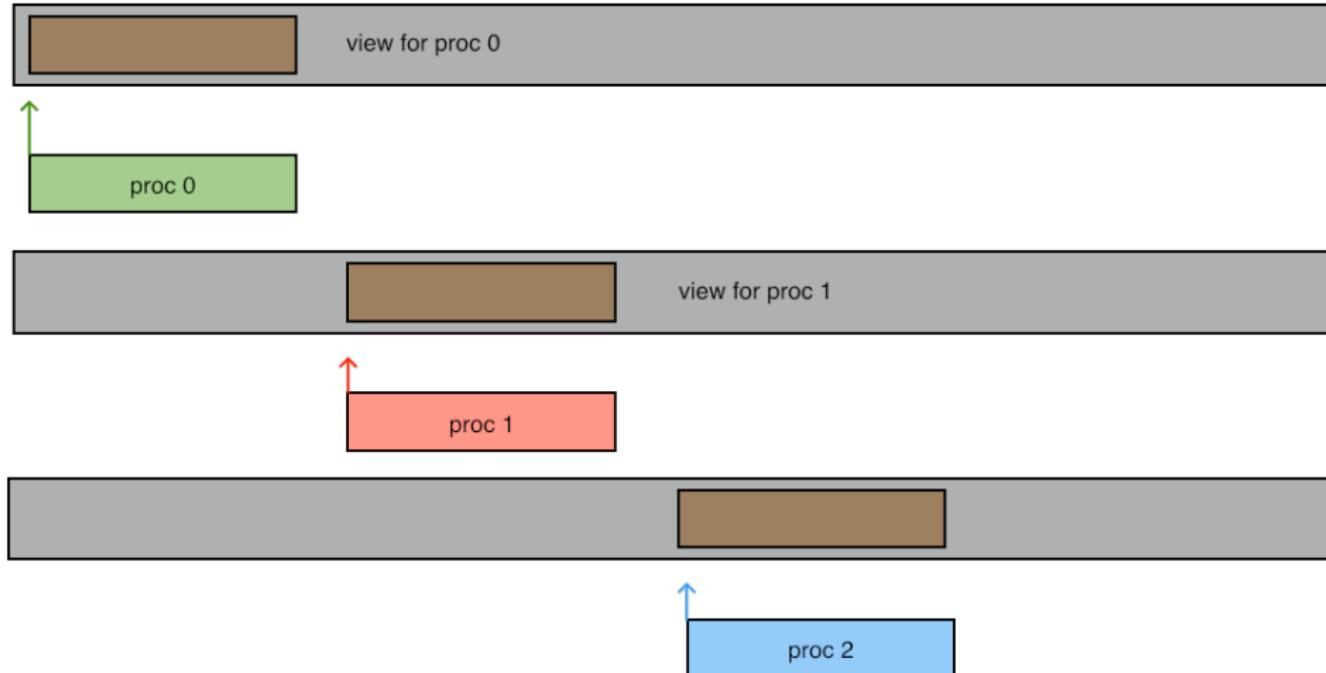
or

```
|| MPI_File_set_view  
||   (mpifile,  
||     offset,datatype,  
||     MPI_INT,"native",MPI_INFO_NULL);  
|| MPI_File_write // no offset, we have a view  
||   (mpifile,output_data,nwords,MPI_INT,MPI_STATUS_IGNORE);
```

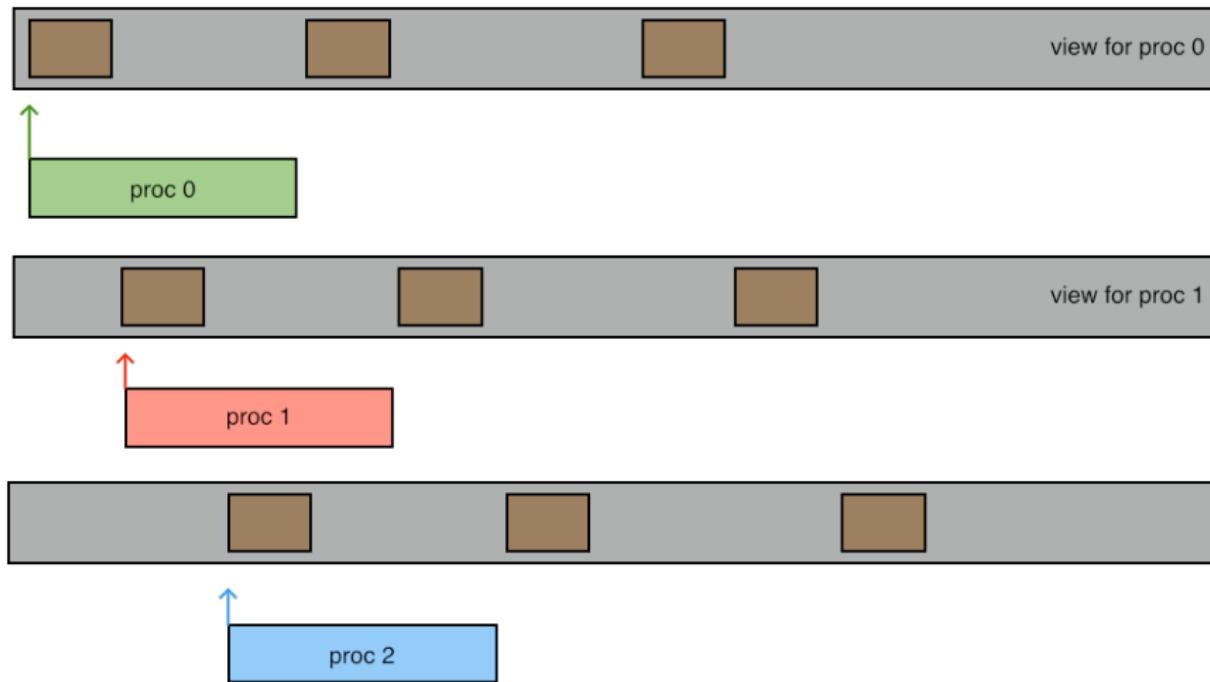
## 166. Write at an offset



## 167. Write to a view



## 168. Write to a view



Made with Vex

## Exercise 33 (blockwrite)

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.

## Exercise 34 (viewwrite)

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

## Exercise 35 (scatterwrite)

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

## 169. Overview

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

## 170. Motivation

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

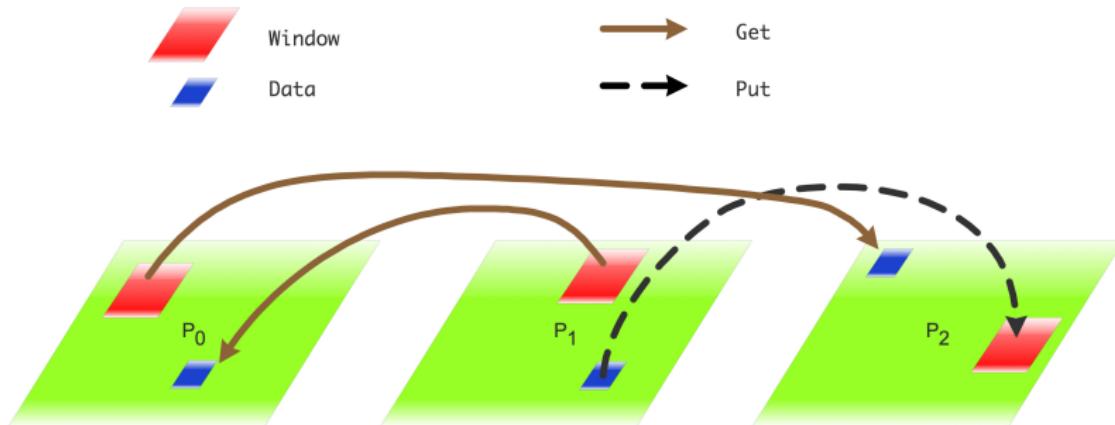
- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbors. Uncertain when this happens.
- Other irregular data structures: distributed hash tables.

## 171. Dynamic data

```
|| x = f();  
|| p = hash(x);  
|| MPI_Send( x, /* to: */ p );
```

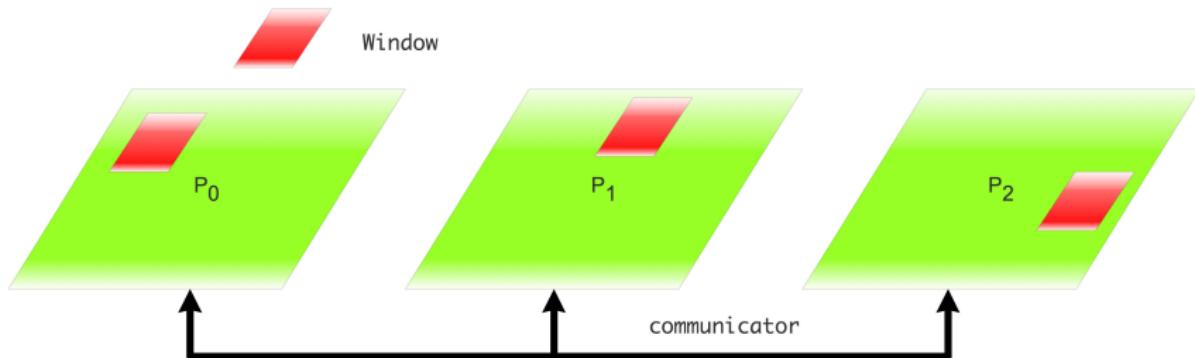
Problem: how does p know to post a receive,  
and how does everyone else know not to?

## 172. One-sided concepts



- A process has a window that other processes can access.
- origin: process doing a one-sided call  
target: process being accessed.
- One-sided calls: `MPI_Put`, `MPI_Get`, `MPI_Accumulate`.
- Various synchronization mechanisms.

## 173. Window creation



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

- **size**: in bytes
- **disp\_unit**: `sizeof(type)`

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

## 174. Window allocation

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

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

can use dedicated fast memory.

## 175. Active target synchronization

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

```
|| MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
|| if (procno==producer)
||   MPI_Put( /* operands */, win);
|| MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

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

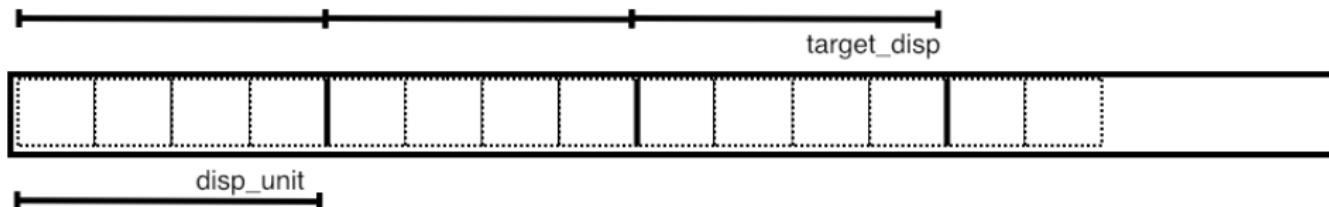
# MPI\_Put

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

## 176. Location in the window

Location to write:

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



## Exercise 36 (rightput)

Revisit exercise 17 and solve it using `MPI_Put`.

## Exercise 37 (randomput)

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.

## Exercise (optional) 38 (randomput)

Replace `MPI_Win_create` by `MPI_Win_allocate`.

## 177. Remaining simple routines: Get, Accumulate

- `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`.

# MPI\_Get

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

# MPI\_Accumulate

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

# Ordering and synchronization

## 178. Fence 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.

## 179. Ordering of operations

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.

## Exercise (optional) 39 (countdown)

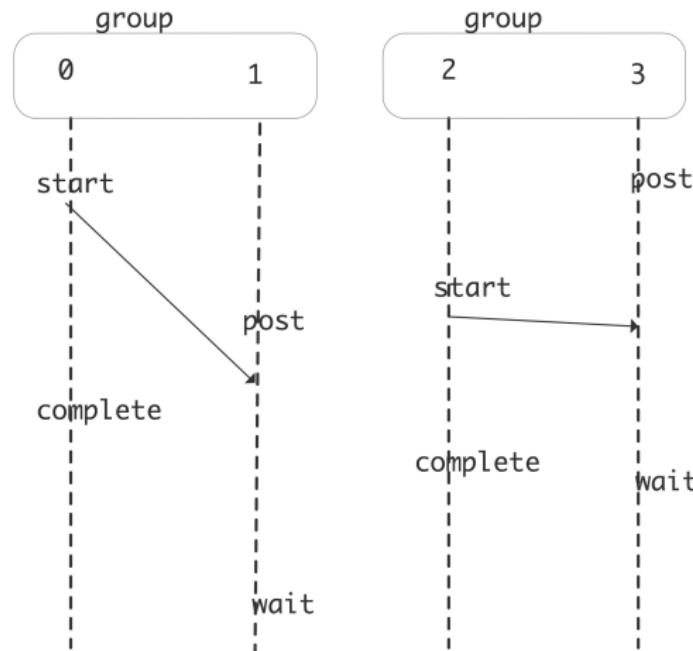
Implement a shared counter:

- 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?

## 180. A second active synchronization

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



More fine-grained than fences.

Eijkhout: MPI course

# Passive target synchronization

## 181. Passive target synchronization

Lock a window on the target:

```
|| MPI_Win_lock  
||   (int locktype, int rank, int assert, MPI_Win win)  
|| MPI_Win_unlock  
||   (int rank, MPI_Win win)
```

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

## 182. Justification

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

## 183. 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.

## 184. Simplified model

- 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.

## 185. 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:

process 1:  $I = I + 2$

process 2:  $I = I + 3$

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

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

## 186. Case study in shared memory: 1, wrong

```
// countdownput.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
        counter_process,0,1,MPI_INT,
        the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
    my_counter_values[ n_my_counter_values++ ] = counter_value;
    total_decrement++;
    int decrement = -1;
    counter_value += decrement;
    MPI_Put
        ( &counter_value, 1,MPI_INT,
          counter_process,0,1,MPI_INT,
          the_window);
}
MPI_Win_fence(0,the_window);
```

## 187. Discussion

- 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

```
|| MPI_Win_fence(0,the_window);
|| if (i_am_available) {
||   MPI_Get( &counter_value, ... )
||   MPI_Win_fence(0,the_window);
||   MPI_Put( ... )
|| }
|| MPI_Win_fence(0,the_window);
```

?

## 188. Case study in shared memory: 2, hm

```
// countdownacc.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
        counter_process,0,1,MPI_INT,
        the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
    total_decrement++;
    int decrement = -1;
    MPI_Accumulate
        ( &decrement,      1,MPI_INT,
          counter_process,0,1,MPI_INT,
          MPI_SUM,
          the_window);
}
MPI_Win_fence(0,the_window);
```

## 189. Discussion: need for atomics

- `MPI_Accumulate` is atomic, so no conflicting writes.
- What is the problem?
- Processes are not reading unique `counter_value` values.
- 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 syntactic sugar around the latter.

# MPI\_Fetch\_and\_op

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

## 190. Case study in shared memory: 3, good

```
MPI_Win_fence(0,the_window);
int
counter_value;
if (i_am_available) {
    int
        decrement = -1;
    total_decrement++;
    MPI_Fetch_and_op
        ( /* operate with data from origin: */ &decrement,
         /* retrieve data from target: */ &counter_value,
         MPI_INT, counter_process, 0, MPI_SUM,
         the_window);
}
MPI_Win_fence(0,the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
}
```

## 191. Allowable operators. (Hint!)

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
<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 t
<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.

## 192. Problem

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.

## Exercise 40 (lockfetch)

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:

```
|| int one=1;  
|| MPI_Fetch_and_op(&zone, &readout,  
|| MPI_INT, repo, zero_disp, MPI_SUM,  
|| 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?

## Exercise 41 (lockfetchshared)

As exercise ??, 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

## 193. Overview

This section discusses big messages.

Commands learned:

- `MPI_Send_c`, `MPI_Allreduce_c`, `MPI_Get_count_c` (MPI-4)
- `MPI_Get_elements_x`, `MPI_Type_get_extent_x`,  
`MPI_Type_get_true_extent_x` (MPI-3)

## 194. The problem with large messages

- There is no problem allocating large buffers:

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

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

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

because the size argument has to be int.

## 195. MPI 3 count type

Count type since MPI 3

C:

```
|| MPI_Count count;
```

Fortran:

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

Big enough for `int`, `MPI_Aint`, `MPI_Offset`.

However, this type could not be used to describe send buffers.

## 196. MPI 4 large count routines

C: routines with \_c suffix

```
|| MPI_Count count;  
|| MPI_Send_c( buff,count,MPI_INT, ... );
```

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

Fortran: polymorphism rules

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

# MPI\_Send

Name	Param name	C type	F type	inout
	MPI_Send (			
	MPI_Send_c (			
	buf	void*	TYPE(*), DIMENSION(..)	IN
		initial address of send buffer		
(big)	count	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
		number of elements in send buffer		
	datatype	MPI_Datatype	TYPE(MPI_Datatype)	IN
		datatype of each send buffer element		
	dest	int	INTEGER	IN
		rank of destination		
	tag	int	INTEGER	IN
		message tag		
	comm	MPI_Comm	TYPE(MPI_Comm)	IN
		communicator		

## 197. MPI 4 large count querying

C:

```
|| MPI_Count count;  
|| MPI_Get_count_c( &status,MPI_INT, &count );  
|| MPI_Get_elements_c( &status,MPI_INT, &count );
```

Fortran:

```
|| Integer(kind=MPI_COUNT_KIND) :: count  
|| call MPI_Get_count( status,MPI_INTEGER,count )  
|| call MPI_Get_elements( status,MPI_INTEGER,count )
```

## 198. MPI 3 kludge: use semi-large types

Make a derived datatype, and send a couple of those:

```
|| MPI_Datatype blocktype;
|| MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
|| MPI_Type_commit(&blocktype);
|| if (procno==sender) {
||   MPI_Send(source,nblocks,blocktype,receiver,0,comm);
```

You can even receive them:

```
|| } else if (procno==receiver) {
||   MPI_Status recv_status;
||   MPI_Recv(target,nblocks,blocktype, sender,0,comm,
||             &recv_status);
```

## 199. So what's the problem?

- Remember that in a receive call the buffer size is an upper bound.
- You use `MPI_Get_count` to ask how much you got
  - ... that tells you how many semi-big objects there were
- There is also `MPI_Get_elements` that counts the underlying type
  - ... and returns an int.

## 200. Large int counting

New to MPI-3:

- Datatype `MPI_Count`
- Routines such as `MPI_Get_elements_x`

```
|| MPI_Count recv_count;  
|| MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```

MPI-4 will use `MPI_Count` parameters everywhere,  
in Fortran through polymorphism, in C through `_c` suffix.

## 201. Big types

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

- `MPI_Type_get_extent_x`, `MPI_Type_get_true_extent_x`

to query. These return `MPI_Count` instead of int.

## 202. For your amusement

What do you get if you print the following:

```
// getx.c
int gig = 1<<30;
int nblocks = 8;
size_t big1 = gig * nblocks * sizeof(double);
size_t big2 = (size_t)1 * gig * nblocks * sizeof(double);
size_t big3 = (size_t) gig * nblocks * sizeof(double);
size_t big4 = gig * nblocks * (size_t) ( sizeof(double) );
size_t big5 = sizeof(double) * gig * nblocks;
;
```

?

## 203. Big data in python

The mpi4py interface uses Python integers,  
which are 8 bytes. So the `_x` routines are not needed.

# Advanced (MPI-3/4) topics

# Justification

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

# Part IX

More about collectives

204. [

containsverbatim]User-defined operators Given a reduction function:

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

create a new operator:

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

## Exercise 42 (onenorm)

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

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

## 205. Non-blocking collectives

- Collectives are blocking.
- Compare blocking/non-blocking sends:  
`MPI_Send` → `MPI_Isend`
- Non-blocking collectives:  
`MPI_Bcast` → `MPI_Ibcast`
- Use for overlap communication/computation
- Imbalance resilience
- Allows pipelining

## 206. Use of non-blocking collectives

- Similar calls, but output a request object:

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

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

# MPI\_Ibcast

Name	Param name	C type	F type	inout
	MPI_Ibcast (			
	MPI_Ibcast_c (			
	buffer	void*	TYPE(*), DIMENSION(..)	INOUT
	starting address of buffer			
(big)	count	int MPI_Count	INTEGER INTEGER(KIND=MPI_COUNT_KIND)	IN
	number of entries in buffer			
	datatype	MPI_Datatype	TYPE(MPI_Datatype)	IN
	datatype of buffer			
	root	int	INTEGER	IN
	rank of broadcast root			
	comm	MPI_Comm	TYPE(MPI_Comm)	IN
	communicator			
	request	MPI_Request*	TYPE(MPI_Request)	OUT
	communication request			

## 207. Overlapping collectives

Independent collective and local operations:

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

```
|| MPI_Iallreduce( .... x ..., &request);
|| // compute the matrix vector product
|| MPI_Wait(request);
|| // do the addition
```

## 208. Simultaneous reductions

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

$$\alpha \leftarrow x^T y$$
$$\beta \leftarrow \|z\|_\infty$$

which translates to:

```
|| MPI_Request reqs[2];
|| MPI_Iallreduce
|| ( &local_xy, &global_xy, 1, MPI_DOUBLE, MPI_SUM, comm,
||   &(reqs[0]) );
|| MPI_Iallreduce
|| ( &local_xinf, &global_xin, 1, MPI_DOUBLE, MPI_MAX, comm,
||   &(reqs[1]) );
|| MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```

## Exercise 43 (procgridnonblock)

► Earlier procgrid exercise

Revisit exercise ???. Let only the first row and first column have certain data, which they broadcast through columns and rows respectively. Each process is now involved in two simultaneous collectives. Implement this with nonblocking broadcasts, and time the difference between a blocking and a nonblocking solution.

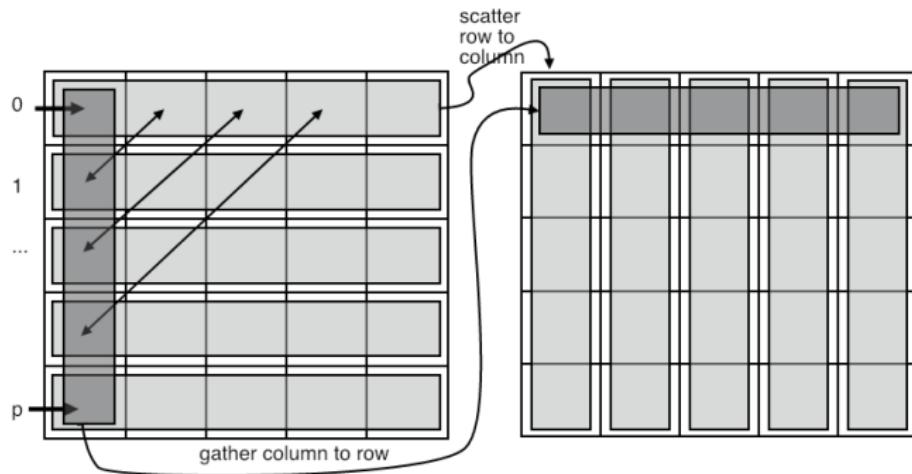
## 209. Matching collectives

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

```
|| if (rank==root)
    MPI_Reduce( &x /* ... */ root,comm );
else
    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`.

## 210. Transpose as gather/scatter



Every process needs to do a scatter or gather.

## 211. Simultaneous collectives

Transpose matrix by scattering all rows simultaneously.

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

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

## 212. Persistent collectives (MPI-4)

Similar to persistent send/recv:

```
|| MPI_Allreduce_init( ...., &request );
|| for ( ... ) {
||   MPI_Start( request );
||   MPI_Wait( request );
|| }
|| MPI_Request_free( &request );
```

Available for all collectives and neighborhood collectives.

## 213. Example

```
// powerpersist.c
double localnorm,globalnorm=1.;
MPI_Request reduce_request;
MPI_Allreduce_init
( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
  comm,MPI_INFO_NULL,&reduce_request);
for (int it=0; it<10; it++) {
    matmult(indata,outdata,buffersize);
    localnorm = localsum(outdata,buffersize);
    MPI_Start( &reduce_request );
    MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
    scale(outdata,indata,buffersize,1./sqrt(globalnorm));
}
MPI_Request_free( &reduce_request );
```

Note also the MPI\_Info parameter.

## 214. Persistent vs non-blocking

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.

# Non-blocking barrier

## 215. Just what is a barrier?

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

## 216. Use case: adaptive refinement

- 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.

## 217. Use case: distributed termination detection

- 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

# MPI\_Ibarrier

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

## 218. Step 1

Do sends, post barrier.

```
// ibarrierprobe.c
if (i_do_send) {
/*
 * Pick a random process to send to,
 * not yourself.
 */
int receiver = rand()%nprocs;
MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
 * Everyone posts the non-block barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);
```

## 219. Step 2

Poll for barrier and messages

```
for ( ; ; step++) {
    int barrier_done_flag=0;
    MPI_Test(&barrier_request,&barrier_done_flag,
             MPI_STATUS_IGNORE);
    //stop if you're done!
    if (barrier_done_flag) {
        break;
    } else {
        // if you're not done with the barrier:
        int flag; MPI_Status status;
        MPI_Iprobe
            ( MPI_ANY_SOURCE,MPI_ANY_TAG,
              comm, &flag, &status );
        if (flag) {
            // absorb message!
```

## Exercise 44 (ibarrierupdate)

- Let each process send to a random number of randomly chosen neighbors. Use `MPI_Isend`.
- Write the main loop with the `MPI_Test` call.
- Insert an `MPI_Iprobe` call and process incoming messages.
- Can you make sure that all sends are indeed processed?

# Part X

## Shared memory

## 220. Shared memory myths

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.

## 221. MPI shared memory

- 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.

## 222. Shared memory treatments in MPI

- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers ‘fake shared memory’: yes, 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 shared memory.

## 223. Shared memory per cluster node



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

## 224. Shared memory interface

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
- Use `MPI_Win_shared_query` to get pointer to another process' window data.
- You can now use `memcpy` instead of `MPI_Put`.

## 225. Discover shared memory

- `MPI_Comm_split_type` splits into communicators of same type.
- Only supported type: `MPI_COMM_TYPE_SHARED` splitting by shared memory.  
(MPI-4: split by other hardware features through  
`MPI_COMM_TYPE_HW_GUIDED` and `MPI_Get_hw_resource_types`)

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type(MPI_COMM_WORLD,
    →MPI_COMM_TYPE_SHARED,procno,info,&sharedcomm);
MPI_Comm_size(sharedcomm,&new_nprocs);
MPI_Comm_rank(sharedcomm,&new_procno);
```

## 226. Allocate shared window

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

- Has to be on a communicator on shared memory
- memory will be allocated contiguously  
convenient for address arithmetic,  
not for NUMA: set `alloc_shared_noncontig` true in  
`MPI_Info` object

```
// sharedbulk.c
MPI_Win node_window;
MPI_Aint window_size; double *window_data;
if (onnode_procid==0)
    window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
( window_size,sizeof(double),MPI_INFO_NULL,
  nodecomm,
  &window_data,&node_window);
```

## 227. Get pointer to other windows

Use `MPI_Win_shared_query`:

```
|| MPI_Aint window_size0; int window_unit; double *win0_addr;  
|| MPI_Win_shared_query  
|| ( node_window,0,  
||   &window_size0,&window_unit, &win0_addr );
```

# MPI\_Win\_shared\_query

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

## 228. Exciting example: bulk data

- 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
- every else points to this object.

## Exercise 45 (shareddata)

Let the ‘shared’ data originate on process zero in `MPI_COMM_WORLD`.  
Then:

- create a communicator per shared memory domain;
- create a communicator for all the processes with number zero on their node;
- broadcast the shared data to the processes zero on each node.

# Part XI

## Process management

## 229. Overview

This section discusses processes management; intra communicators.

Commands learned:

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

## 230. Process management

- 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’)

## 231. Space for processes

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:

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

## 232. Manager program

```
int universe_size, *universe_size_attr,uflag;
MPI_Comm_get_attr
(comm_world,MPI_UNIVERSE_SIZE,
&universe_size_attr,&uflag);
universe_size = *universe_size_attr;
if (!uflag) universe_size = world_n;
int work_n = universe_size - world_n;
if (world_p==0) {
    printf("A universe of size %d leaves room for %d workers\n",
           universe_size,work_n);
    printf(.. spawning from %s\n",procname);
}
```

## 233. Manager program (cont'd)

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

## 234. Worker program

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

## 235. Were you spawned?

```
// spawnapp.c
MPI_Comm comm_parent;
MPI_Comm_get_parent(&comm_parent);
int is_child = (comm_parent!=MPI_COMM_NULL);
if (is_child) {
    int nworkers,workerno;
    MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
    MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
    printf("I detect I am worker %d/%d running on %s\n",
        workerno,nworkers,procname);
```

# Session model

## 236. World and session model

Process management so far: ‘world model’

New: ‘session model’; world is just one session.

```
MPI_Session_init(info,errhandler,&session);  
  
const char pset_name[] = "mpi://WORLD";  
MPI_Group_from_session_pset  
(lib_shandle,pset_name,&wgroup);  
MPI_Comm_create_from_group  
(wgroup,"parcompbook-example",  
 MPI_INFO_NULL,MPI_ERRORS_RETURN,&world_comm);
```

## Part XII

### Process topologies

## 237. Overview

This section discusses graph topologies.

Commands learned:

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

## 238. Process topologies

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

## 239. Process reordering

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

## 240. MPI-1 topology

- Cartesian topology
- Graph topology, globally specified.  
Not exactly scalable!

## 241. MPI-3 topology

- Graph topologies locally specified: scalable!
- Neighborhood collectives:  
expression close to the algorithm.

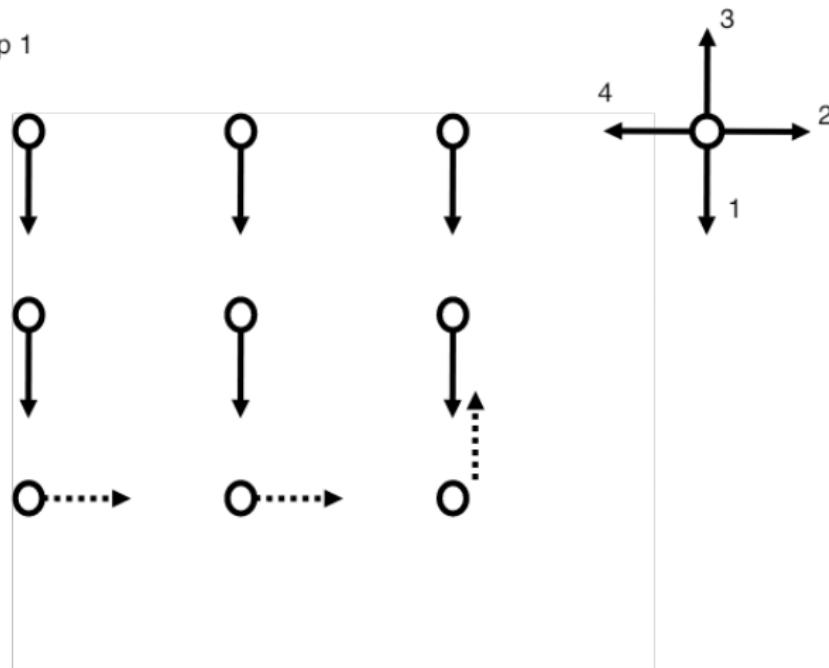
## 242. Example: 5-point stencil

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

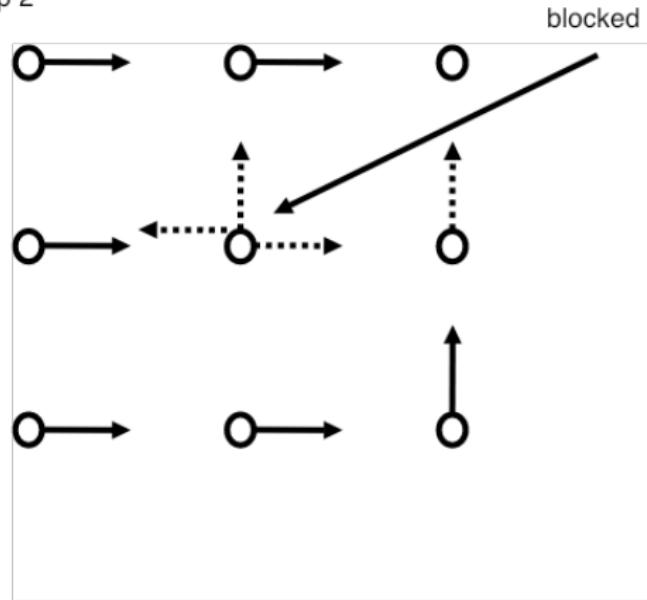
## 243. Step 1

Step 1



## 244. Step 2

Step 2

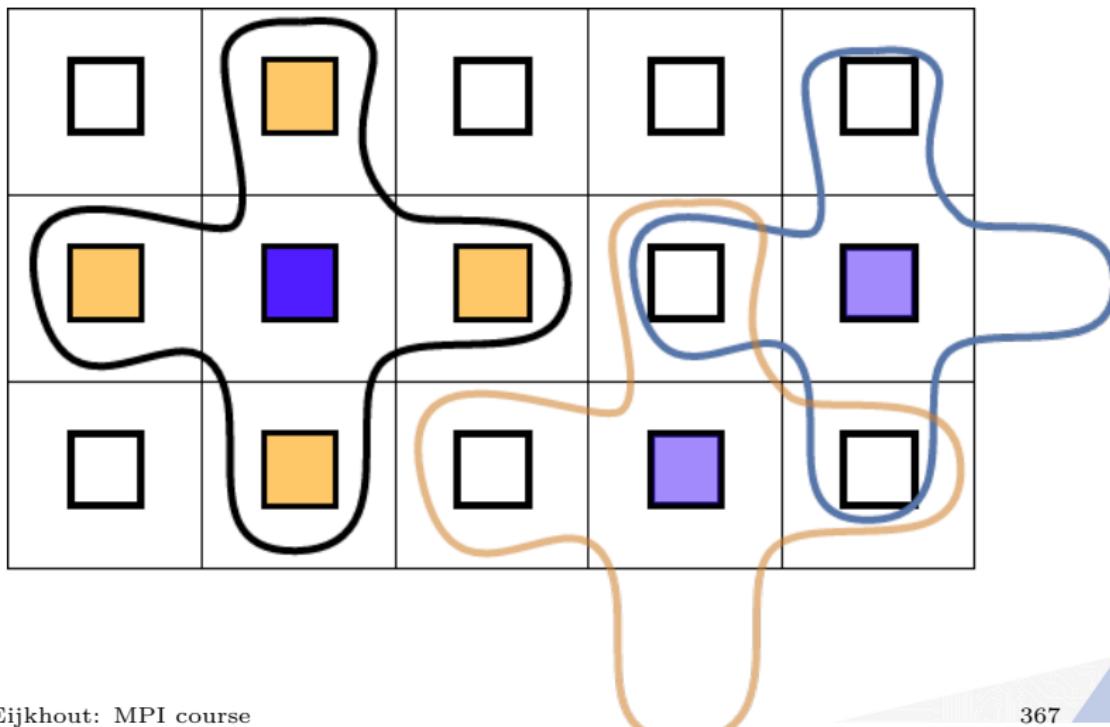


The middle node is blocked because all its targets are already receiving or a channel is occupied:  
one missed turn

## 245. Neighborhood collective

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

`MPI_Neighbor_allgather`



## 246. Why neighborhood collectives?

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

## 247. Create graph topology

```
int MPI_Dist_graph_create  
  (MPI_Comm comm_old, int nsources, const int sources[],  
   const int degrees[], const int destinations[],  
   const int weights[], MPI_Info info, int reorder,  
   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

## 248. Neighborhood collectives

```
|| int MPI_Neighbor_allgather  
|| (const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
||   void *recvbuf, int recvcount, MPI_Datatype recvtype,  
||   MPI_Comm comm)
```

Like an ordinary `MPI_Allgather`, but  
the receive buffer has a length degree.

## 249. Neighbor querying

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.

# MPI\_Dist\_graph\_neighbors

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

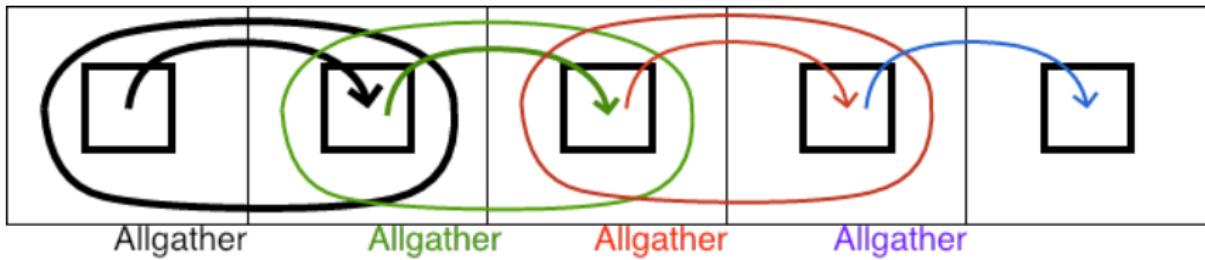
## Exercise 46 (rightgraph)

▶ Earlier rightsend exercise

Revisit exercise 17 and solve it using `MPI_Dist_graph_create`. Use figure ?? for inspiration.

Use a degree value of 1.

## 250. Inspiring picture for the previous exercise



Solving the right-send exercise with neighborhood collectives

## 251. Hints for the previous exercise

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 ??.

# Other

## Part XIII

Tracing, performance, and such

## 252. Overview

We briefly touch on peripheral issues issues to MPI.

# Errors

## 253. Built-in handlers

Default: global termination.

```
|| 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`:

## 254. Handlers on specific classes

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`.

## 255. Handling errors

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

## 256. Define new errors

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

# Performance measurement

## 257. Timers

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

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

but probably aren't.

## 258. Example

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

## 259. Global timing

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

Alternative:

```
|| MPI_Barrier(comm)
  t = MPI_Wtime();
  // something happens here
  MPI_Barrier(comm)
  t = MPI_Wtime()-t;
```

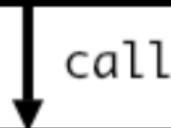
## 260. Profiling

See other lecture: MPIP, TAU, et cetera.

## 261. Your own profiling interface

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, ... );  
}
```



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

# Programming for performance

## 262. Eager limit

- Optimization for small messages: bypass rendez-vous protocol (slide 91)
- 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.

## 263. Eager limit setting

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

## 264. Blocking versus non-blocking

- 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.

## 265. Progress

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`.

## 266. Persistent sends

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`

## 267. Buffering

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

## 268. Graph topology and neighborhood collectives

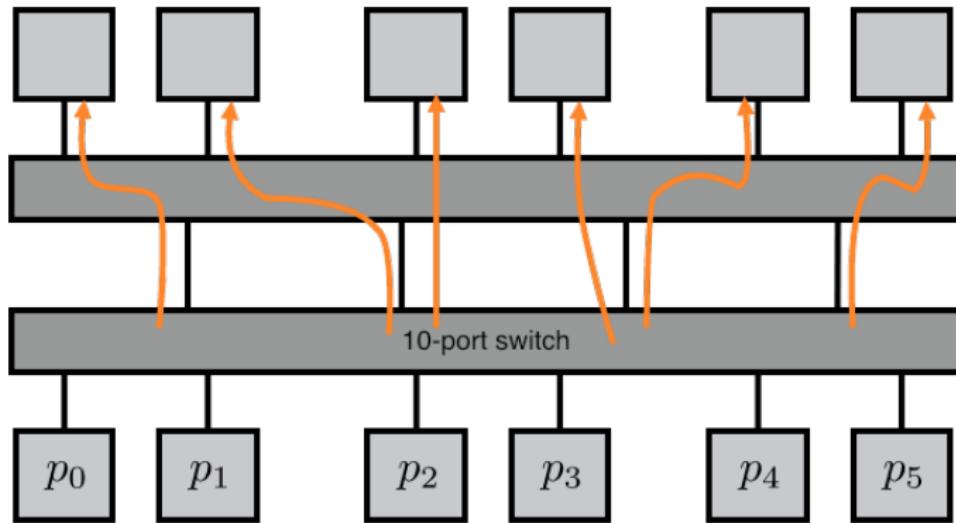
- Mapping problem to architecture sometimes not trivial
- Load balancers: ParMetis, Zoltan
- Graph topologies: MPI\_Dist\_graph\_adjacent:  
allowed to reorder ranks for proximity
- Neighborhood collectives allow MPI to schedule optimally.
  - **MPI\_Neighbor\_allgather** (and **MPI\_Neighbor\_allgather\_v**)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

## 269. Network issues

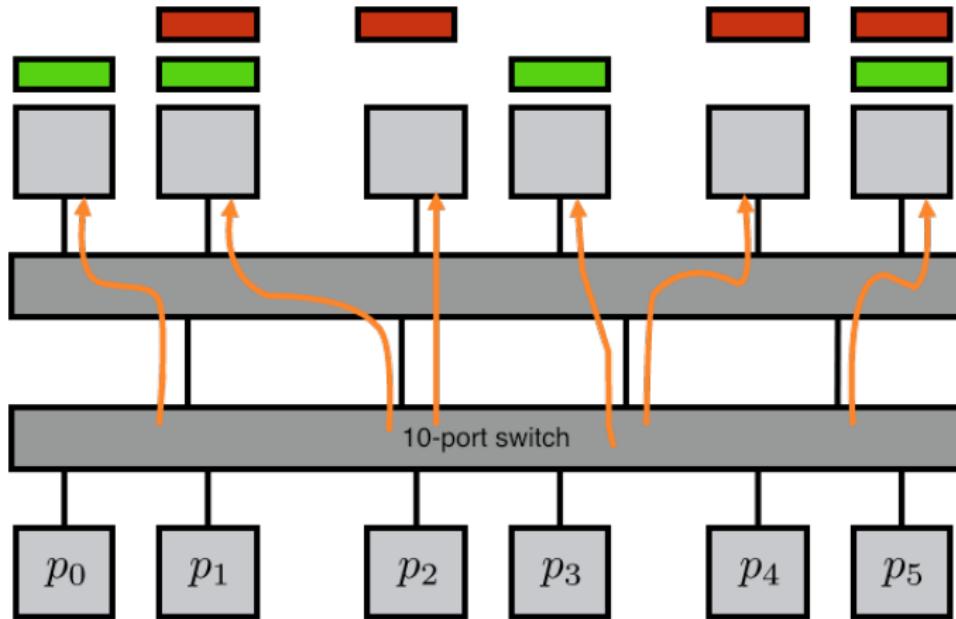
Network contention means that

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

## 270. Output routing



## 271. Contention



## 272. Offloading and onloading

- 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.