



TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU



TEXAS

The University of Texas at Austin

## Tutorial on MPI programming

Victor Eijkhout `eijkhout@tacc.utexas.edu`

TACC/XSEDE MPI training 2020

# 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.



# Basics

# Part I

## The SPMD model

# Overview

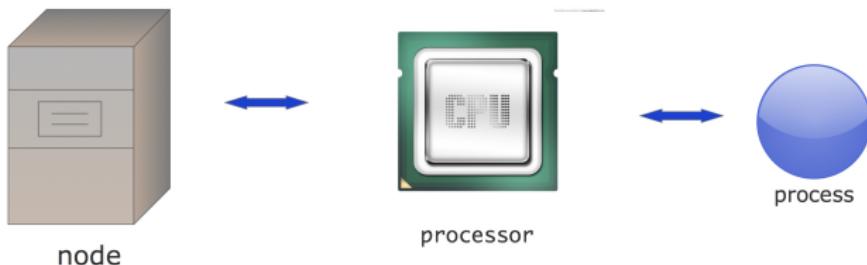
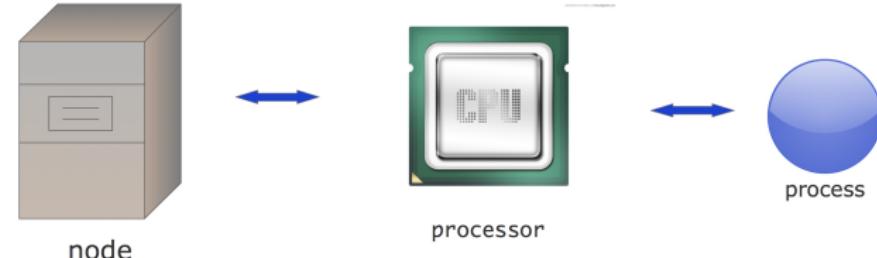
In this section you will learn how to think about parallelism in MPI.

Commands learned:

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

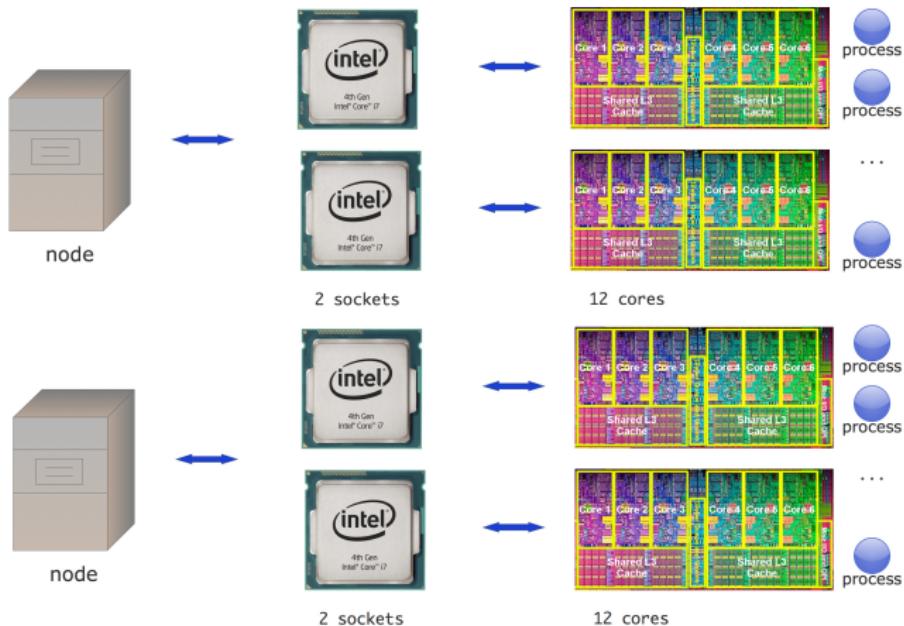
# The MPI worldview: SPMD

# Computers when MPI was designed



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

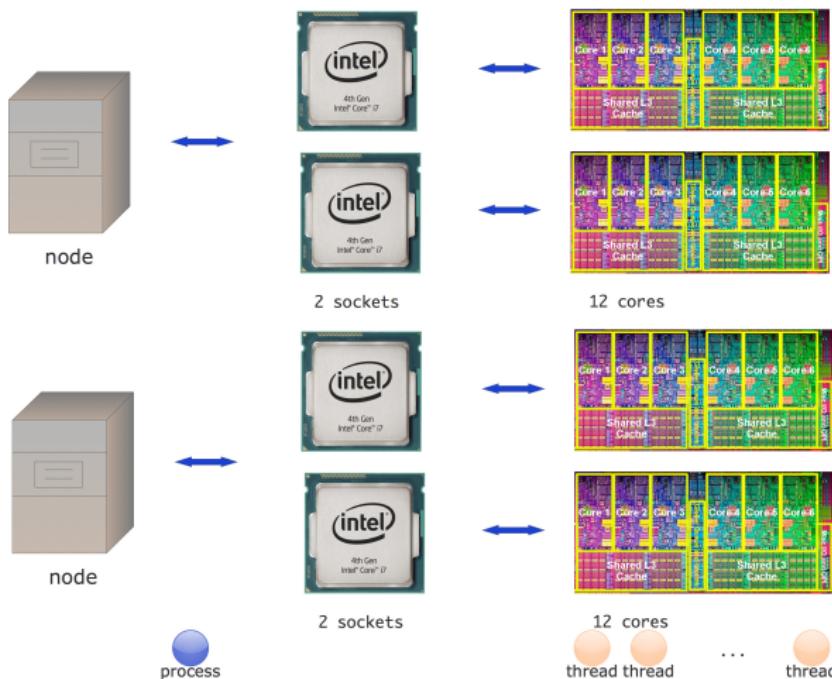
# Pure MPI



A node has multiple sockets, each with multiple cores.

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

# Hybrid programming



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

# 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.

# 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 neighbours.

# Practicalities

# 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.

Run your program with something like

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

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

At TACC:

```
ibrun yourprog
```

the number of processes is determined by SLURM.

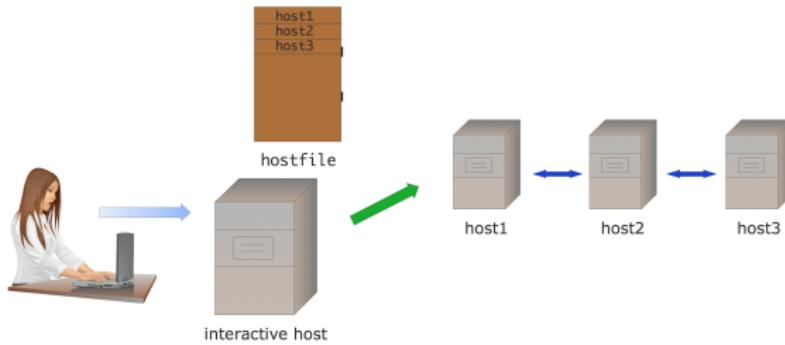
# Do I need a supercomputer?

- 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...

# 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*.

# How to make exercises

- Directory: exercises-mpi-c or cxx or f or f08 or p
- 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 python2 # or python3
```

No compilation needed. Run:

```
ibrun python2 yourprogram # or python3
```

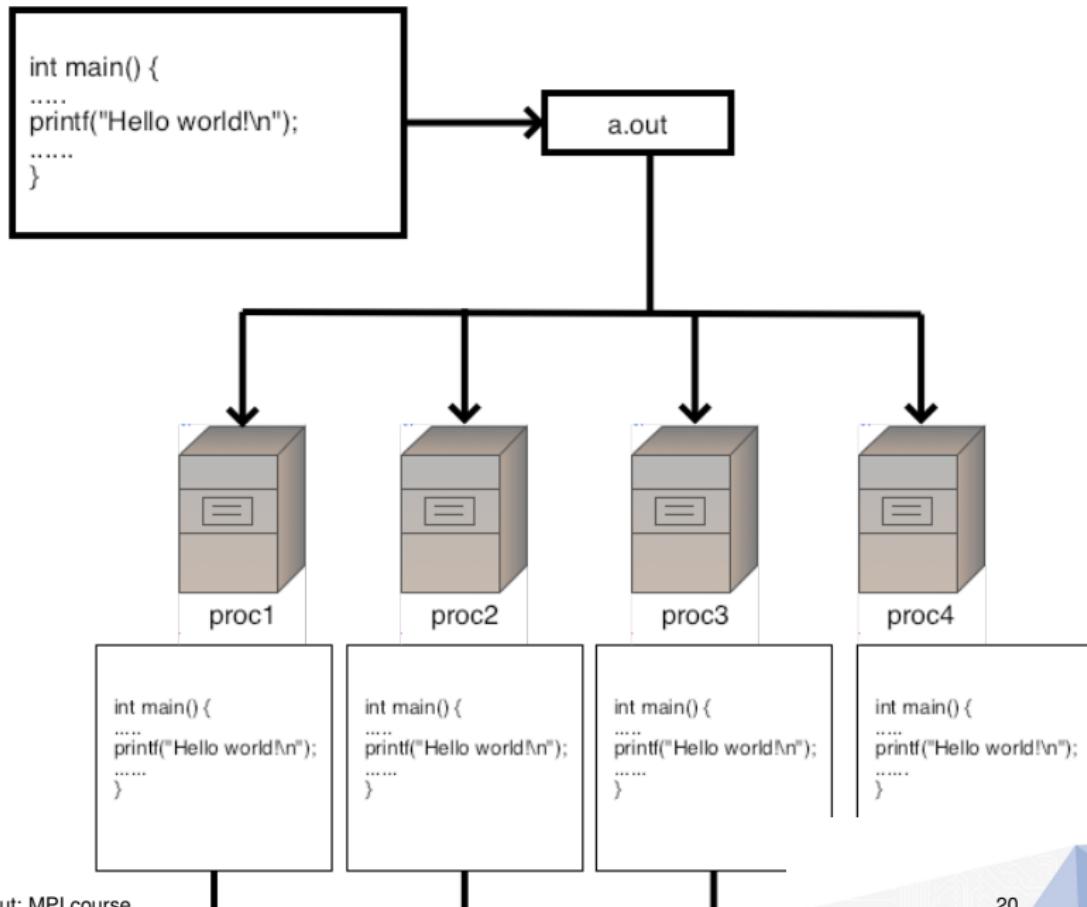
- Add an exercise of your own to the makefile: add the name to the EXERCISES

## Exercise 1 (hello)

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

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

# In a picture



# We start learning MPI!

# MPI definitions

You need an include file:

```
#include "mpi.h" // for C
use mpi          ! for Fortran90
use mpi_f08      ! for Fortran2008
```

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

```
module load intel/18.0.2
or newer. Not in gcc7.
```

# MPI Init / Finalize

Then put these calls around your code:

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

and for Fortran:

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

# About error codes

MPI routines return an integer error code

- In C: function result. Can be ignored.
- 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.  
Just ignore them ...

# Python bindings

```
module load python2 # also python3
```

```
from mpi4py import MPI
```

Run:

```
ibrun python2 yourprogram.py # or python3
```

No initialization needed.

## 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.

# Process identification

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

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

For now, the communicator will be `MPI_COMM_WORLD`.

Note: mapping of ranks to actual processes and cores is not predictable!

# 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)

# About routine prototypes: Fortran

## 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, ierr ) ! F90 style
|| CALL MPI_Comm_size( comm, size ) ! F2008 style
```

- Fortran90: final parameter always error parameter. Do not forget!
- Fortran2008: final parameter optional.
- Fortran90: MPI\_... types are INTEGER.
- Fortran2008: MPI\_... types are Type.

# 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)
```

## MPI\_Comm\_size

Semantics:

`MPI_COMM_SIZE(comm, size)`

IN `comm`: communicator (handle)

OUT `size`: number of processes in the group of `comm` (integer)

C:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran:

```
MPI_Comm_size(comm, size, ierror)
```

TYPE(`MPI_Comm`), INTENT(IN) :: `comm`

INTEGER, INTENT(OUT) :: `size`

INTEGER, OPTIONAL, INTENT(OUT) :: `ierror`

Python:

```
MPI.Comm.Get_size(self)
```

## MPI\_Comm\_rank

Semantics:

`MPI_COMM_RANK(comm, rank)`

IN `comm`: communicator (handle)

OUT `rank`: rank of the calling process in group of `comm` (integer)

C:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

```
MPI_Comm_rank(comm, rank, ierror)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
MPI.Comm.Get_rank(self)
```

## Exercise 3 (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 4 (commrank)

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

## Exercise (optional) 5

Now use the command `MPI_Get_processor_name` in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run a program that uses two different nodes.

The character buffer needs to be allocated by you, it is not created by MPI, with size at least `MPI_MAX_PROCESSOR_NAME`.

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!

## MPI\_Get\_processor\_name

C:

```
int MPI_Get_processor_name(char *name, int *resultlen)
name : buffer char[MPI_MAX_PROCESSOR_NAME]
```

Fortran:

```
MPI_Get_processor_name(name, resultlen, ierror)
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
INTEGER, INTENT(OUT) :: resultlen
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

```
MPI.Get_processor_name()
```

# 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

# A practical example

# 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 range 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.)

## Part II

### Collectives

# 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**

# Concepts

# Collectives

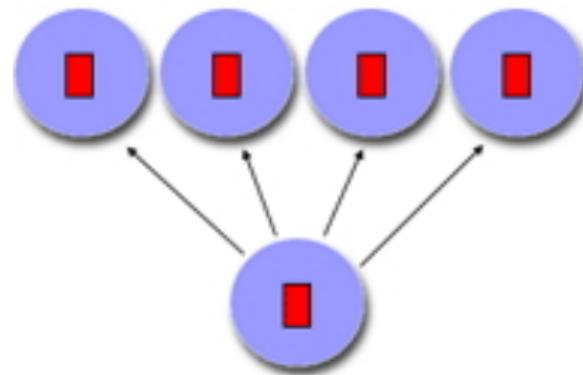
Gathering and spreading information:

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

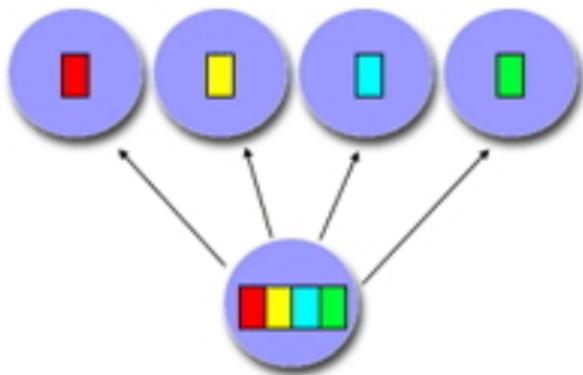
Root process: the one doing the collecting or disseminating.

Basic cases:

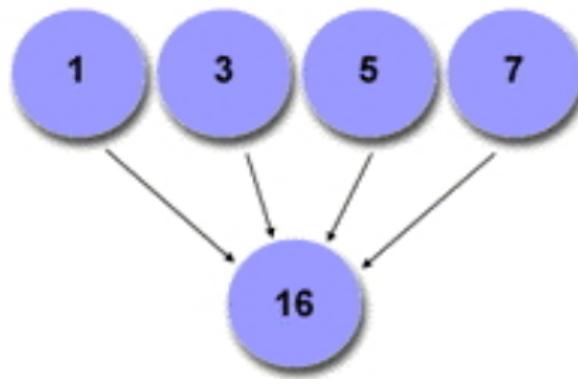
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



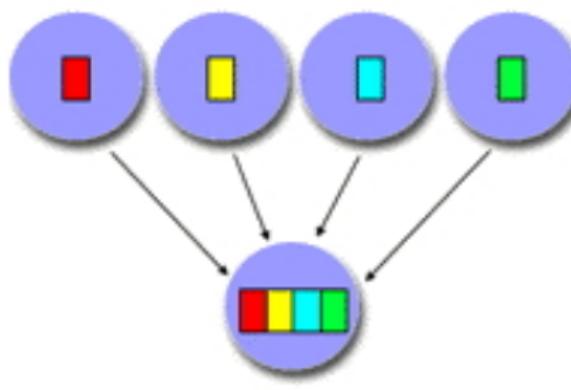
broadcast



scatter



reduction



gather

## Exercise 7

How would you realize the following scenarios with MPI collectives?

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

# 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

# Motivation for 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?

# Motivation for 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?

- ➊ The calculation of the average  $\mu$  is a reduction.

# Motivation for 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?

- ① The calculation of the average  $\mu$  is a reduction.
- ② Every process needs to compute  $x_i - \mu$  for its value  $x_i$ , so use allreduce operation, which does the reduction and leaves the result on all processors.

# Motivation for 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?

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

## Another 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.

## Reduction to single process

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

# 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.
- MPI\_Datatype is MPI\_INT, MPI\_REAL8 et cetera.
- MPI\_Op is MPI\_SUM, MPI\_MAX et cetera.

# 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.

# MPI operators

<i>MPI operator</i>	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.

# 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

# 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

# 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 8 (randommax)

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

(The operator is `MPI_SUM` for C/Fortran, or `MPI.SUM` for Python.)

Each process then scales its value by this sum. Compute the sum of the scaled numbers and check that it is 1.

# Random numbers in C

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

# 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)
```

# Random numbers in Python

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

## Exercise (optional) 9

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?

# 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.

# 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

# Allreduce vs Reduce/Bcast

Allreduce is equivalent to reduce and broadcast.

- One line less code.
- Gives the implementation more possibilities for optimization.
- Is actually twice as fast: allreduce same time as reduce.

# Gauss-Jordan elimination

<https://youtu.be/aQYuwatlWME>

## Exercise 10 (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  $c = 1, \dots, n$

$$A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{rc}$$

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) 11

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

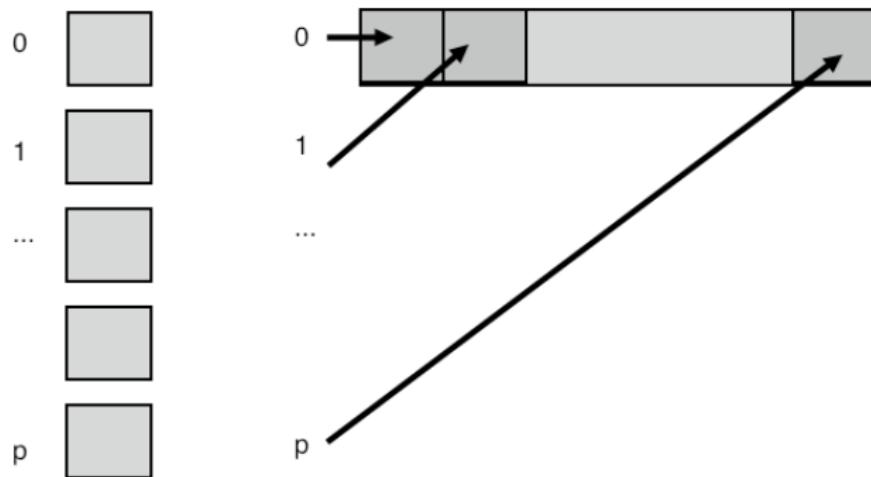
## **Gather/Scatter, Scan, Barrier, and others**

# Gather/Scatter

```
int MPI_Gather(  
    void *sendbuf, int sendcnt, MPI_Datatype sendtype,  
    void *recvbuf, int recvcnt, MPI_Datatype recvtype,  
    int root, MPI_Comm comm  
);  
int MPI_Scatter(  
    void* sendbuf, int sendcount, MPI_Datatype sendtype,  
    void* recvbuf, int recvcount, MPI_Datatype recvtype,  
    int root, MPI_Comm comm)
```

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

# Gather pictured

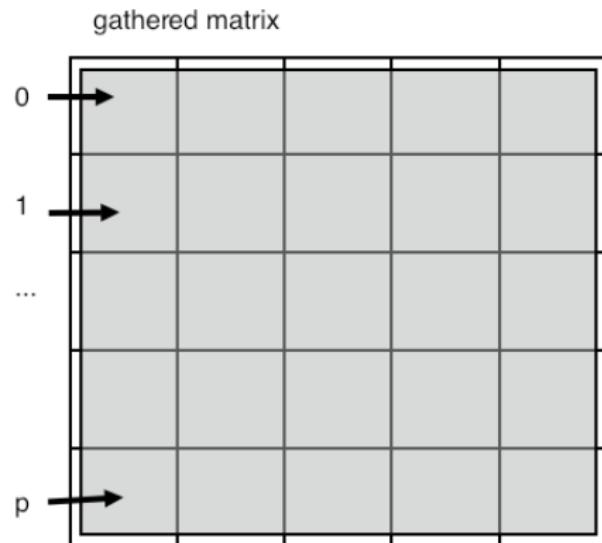
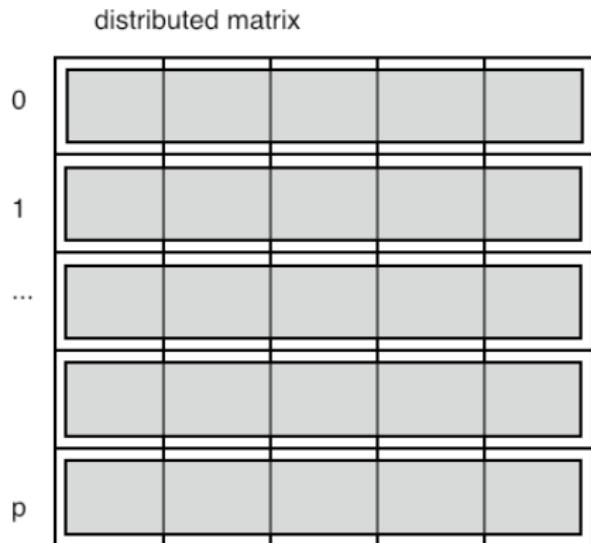


## Exercise 12

Let each process compute a random number. You want to print the maximum value and on what processor it is computed. What collective(s) do you use?  
Write a short program.

# Popular application of gather

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



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

# MPI\_Allgather

C:

```
int MPI_Allgather(const void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Iallgather(const void *sendbuf, int sendcount,
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

Fortran:

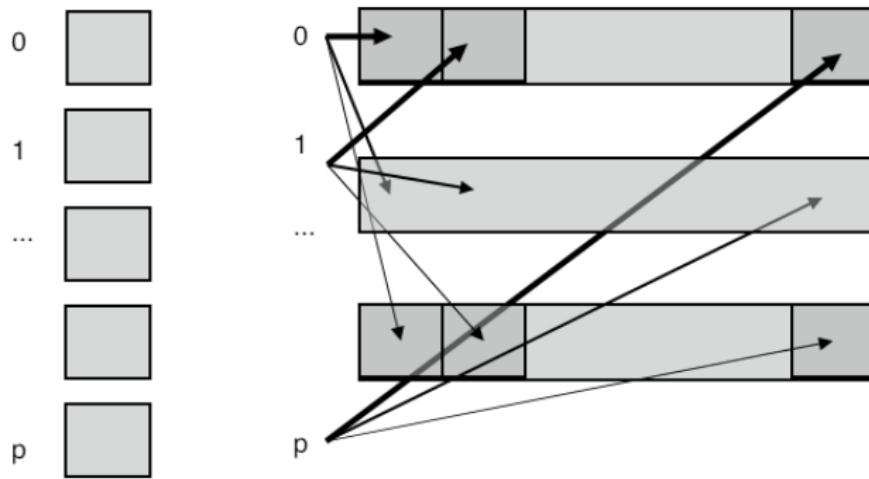
```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT,
               RECVTYPE, COMM, IERROR)
<type>    SENDBUF (*), RECVBUF (*)
INTEGER     SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM,
INTEGER     IERROR
MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT,
                RECVTYPE, COMM, REQUEST, IERROR)
<type>    SENDBUF (*), RECVBUF (*)
INTEGER     SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM
INTEGER     REQUEST, IERROR
```

C++ Syntax

Parameters:

sendbuf : Starting address of send buffer (choice).  
sendcount: Number of elements in send buffer (integer)  
sendtype: Datatype of send buffer elements (handle).

# Allgather pictured



# 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.

# MPI\_Scan

C:

```
int MPI_Scan(const void* sendbuf, void* recvbuf,
             int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
IN sendbuf: starting address of send buffer (choice)
OUT recvbuf: starting address of receive buffer (choice)
IN count: number of elements in input buffer (non-negative integer)
IN datatype: data type of elements of input buffer (handle)
IN op: operation (handle)
IN comm: communicator (handle)
```

Fortran:

```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Python:

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

## V-type collectives

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

# MPI\_Gatherv

C:

```
int MPI_Gatherv(
    const void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, const int recvcounts[], const int displs[],
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Semantics:

IN sendbuf: starting address of send buffer (choice)  
IN sendcount: number of elements in send buffer (non-negative integer)  
IN sendtype: data type of send buffer elements (handle)  
OUT recvbuf: address of receive buffer (choice, significant only at root)  
IN recvcounts: non-negative integer array (of length group size) containing the number  
IN displs: integer array (of length group size). Entry i specifies the displacement rel  
IN recvtype: data type of recv buffer elements (significant only at root) (handle)  
IN root: rank of receiving process (integer)  
IN comm: communicator (handle)

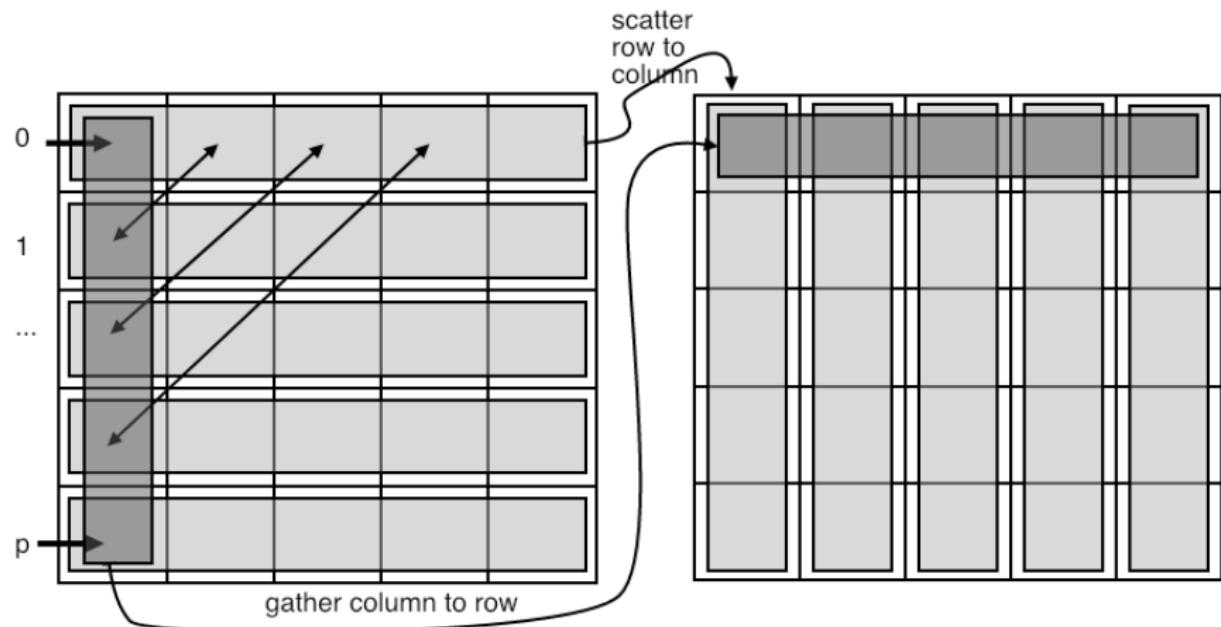
Fortran:

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root,
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
EIJNODENMPIcouGEPTIONAL, INTENT(OUT) :: ierror
```

## All-to-all

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

# Data transposition



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

## All-to-allv

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

# Radix sort

Sort 4 numbers on two processes:

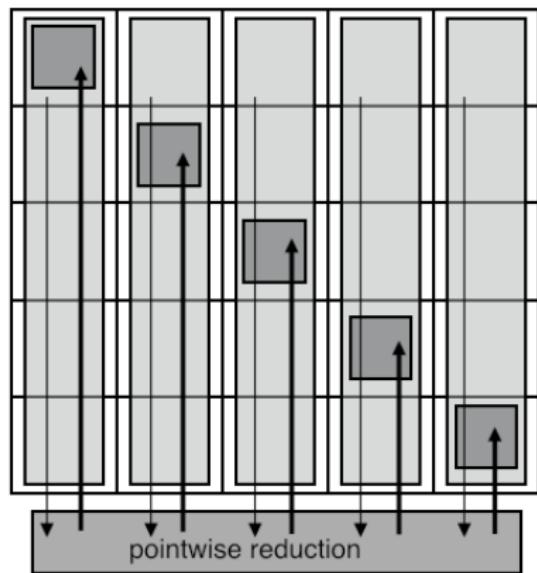
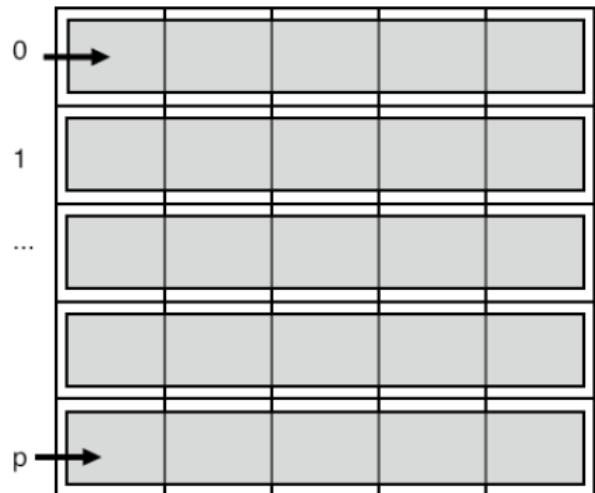
	proc0		proc1	
array	2	5	7	1
binary	010	101	111	001
stage 1				
last digit	0	1	1	1
	(this serves as bin number)			
sorted	010		101	111 001
stage 2				
next digit	1	0	1	0
	(this serves as bin number)			
sorted	101	001	010	111
stage 3				
next digit	1	0	0	1
	(this serves as bin number)			
sorted	001	010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

## Reduce-scatter

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

## Example: sparse matrix setup

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



# Barrier

```
|| int MPI_BARRIER( MPI_Comm comm )
```

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

# User-defined operators

# MPI Operators

Define your own reduction operator

- Define operator between partial result and new operand

```
|| 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

Semantics:

`MPI_OP_CREATE( function, commute, op)`

[ IN `function`] user defined function (function)

[ IN `commute`] true if commutative; false otherwise.

[ OUT `op`] operation (handle)

C:

```
int MPI_Op_create(MPI_User_function *function, int commute,  
                  MPI_Op *op)
```

Fortran:

`MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)`

EXTERNAL FUNCTION

LOGICAL COMMUTE

INTEGER OP, IERROR

Python:

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

# Example

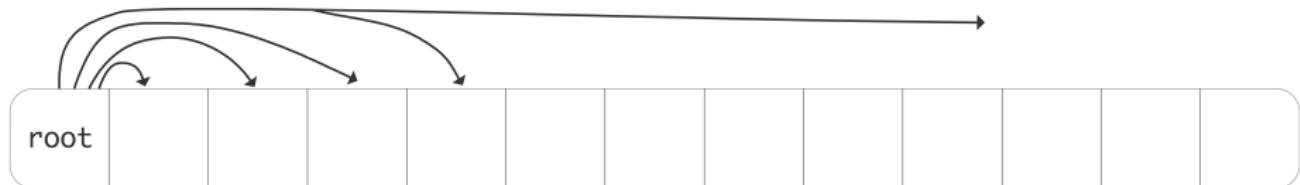
Smallest nonzero:

```
// 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;
}
```

# Performance of collectives

# Naive realization of collectives

Broadcast:



Single message:

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

# Naive realization of collectives

Broadcast:



Single message:

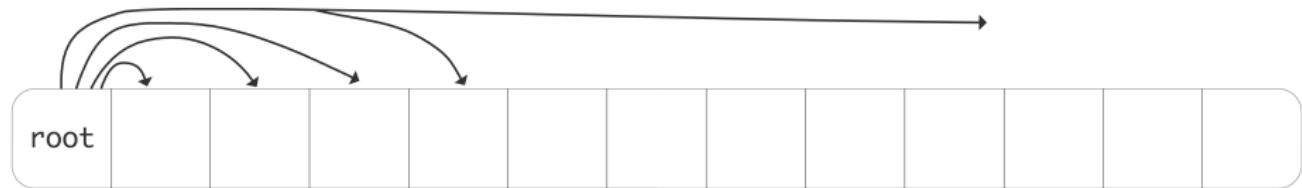
$$\alpha = \text{message startup} \approx 10^{-6} \text{ s}, \quad \beta = \text{time per word} \approx 10^{-9} \text{ s}$$

- Time for message of  $n$  words:

$$\alpha + \beta n$$

# Naive realization of collectives

Broadcast:



Single message:

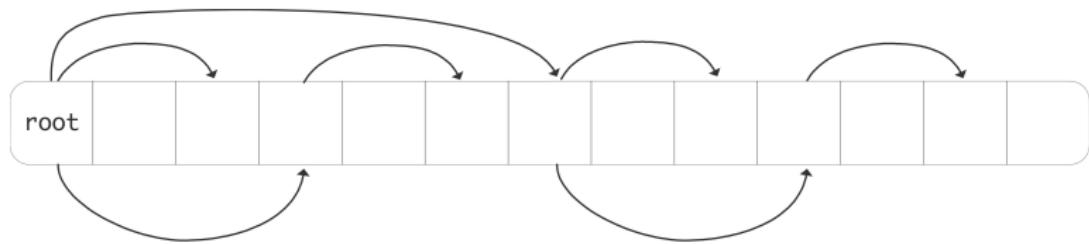
$$\alpha = \text{message startup} \approx 10^{-6} \text{ s}, \quad \beta = \text{time per word} \approx 10^{-9} \text{ s}$$

- Time for message of  $n$  words:

$$\alpha + \beta n$$

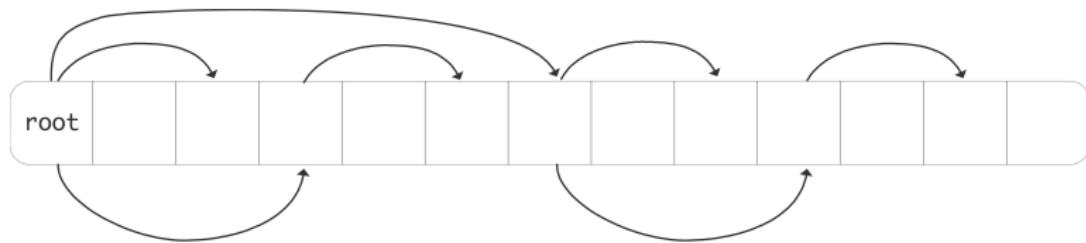
- Time for collective? Can you improve on that?

## Better implementation of collective



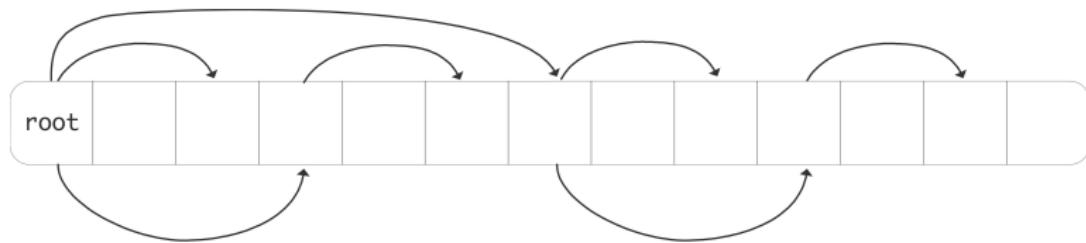
- What is the running time now?

## Better implementation of collective



- What is the running time now?
- Can you come up with lower bounds on the  $\alpha, \beta$  terms? Are these achieved here?

## Better implementation of collective



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

# Part III

Point-to-point communication

# Overview

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

Commands learned:

- `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Isend`, `MPI_Irecv`
- `MPI_Wait...`
- Mention of `MPI_Test`, `MPI_Bsend`/Ssend/Rsend.

# **Point-to-point communication**

# MPI point-to-point mechanism

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

# Ping-pong

A sends to B, B sends back to A

What is the code for A? For B?

# Ping-pong

A sends to B, B sends back to A

Process A executes the code

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

Process B executes

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

# Ping-pong in MPI

Remember SPMD:

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

# MPI\_Send

C:

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

Semantics:

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

Fortran:

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

Python native:

```
MPI.Comm.send(self, obj, int dest, int tag=0)
```

Python numpy:

```
MPI.MPIArraySend(self, buf, int dest, int tag=0)
```

## MPI\_Recv

C:

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

Semantics:

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

Fortran:

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

Python native:

```
Eijkhout: MPI course 109
```

## Status object

- 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
- Use `MPI_STATUS_IGNORE` if the above does not apply

## Exercise 13 (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

C:

```
double MPI_Wtime(void);
```

Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

Python:

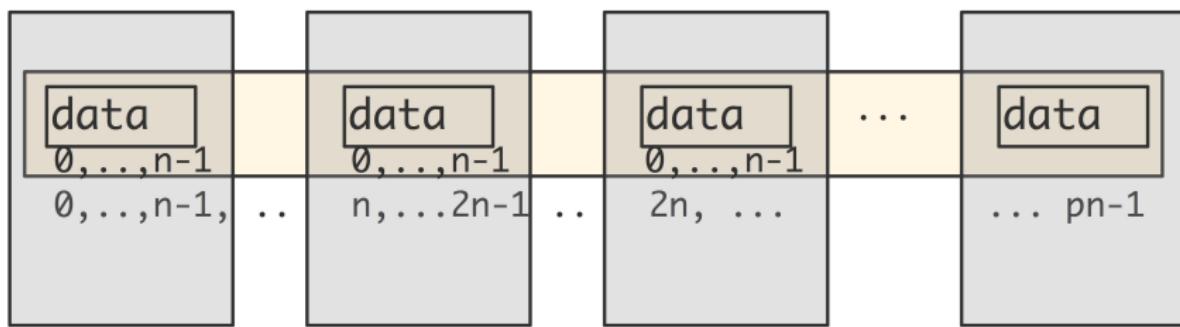
```
MPI.Wtime()
```

# Distributed data

# 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’.

# 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);  
|| }
```

# Load balancing

If the distributed array is not perfectly divisible:

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

This gives a load balancing problem. Better solution?

(for future reference)

Let

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

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

Result:

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

# Inner product calculation

Given vectors  $x, y$ :

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

Start out with a distributed vector.

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

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

## Exercise (optional) 14 (inprod)

Implement an inner product routine: let  $x$  be a distributed vector of size  $N$  with elements  $x[i] = i$ , and compute  $x^t x$ . As before, the right value is  $(2N^3 + 3N^2 + N)/6$ .

Use the inner product value to scale to vector so that it has norm 1. Check that your computation is correct.

## Exercise (optional) 15

Implement a (very simple-minded) Fourier transform: if  $f$  is a function on the interval  $[0, 1]$ , then the  $n$ -th Fourier coefficient is

$$f_n \hat{=} \int_0^1 f(t) e^{-t/\pi} dt$$

which we approximate by

$$f_n \hat{=} \sum_{i=0}^{N-1} f(ih) e^{-in/\pi}$$

- Make one distributed array for the  $e^{-inh}$  coefficients,
- make one distributed array for the  $f(ih)$  values
- calculate a couple of coefficients

# **Local information exchange**

# Motivation

Partial differential equations:

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

Simple case:

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

Finite difference approximation:

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

## 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.

# 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

# Operating on distributed data

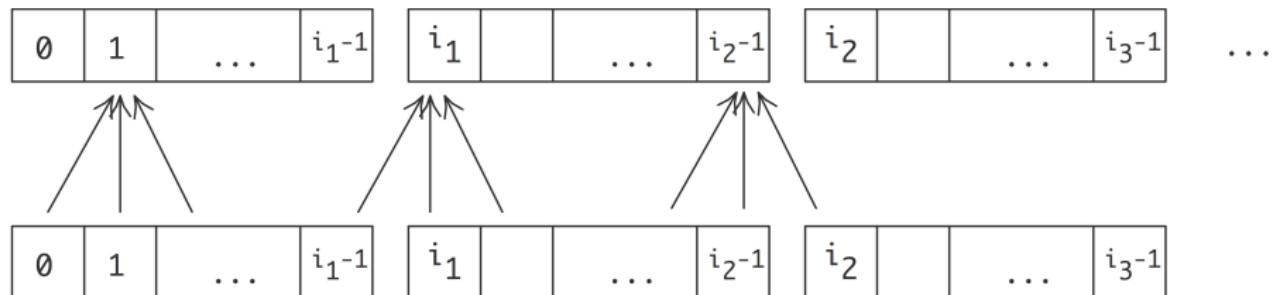
Array of numbers  $x_i : i = 0, \dots, N$

compute

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

'owner computes'

This leads to communication:



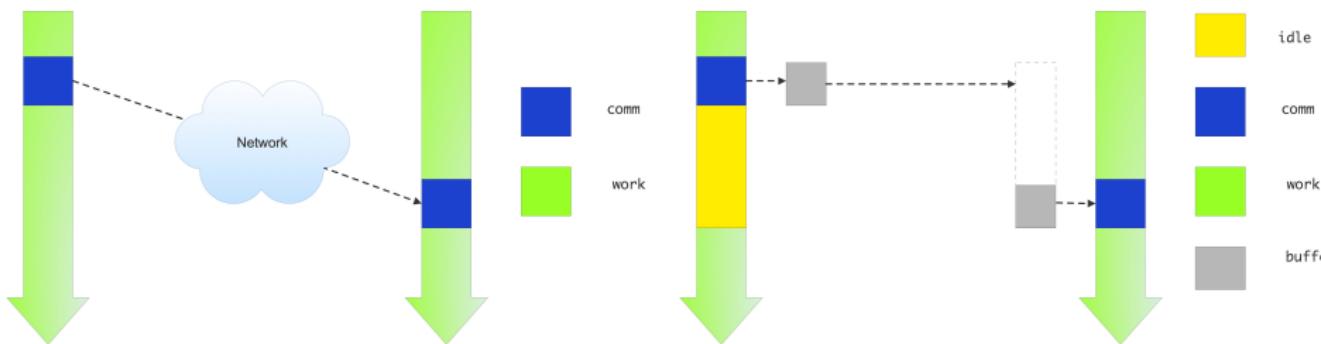
so we need a point-to-point mechanism.

# Blocking communication

# Blocking send/recv

**MPI\_Send** and **MPI\_Recv** are *blocking* operations:

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



Ideal vs actual send/recv behaviour.

# Deadlock

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

A subtlety.

This code may actually work:

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

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

# Protocol

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

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

Small messages bypass this: ‘eager’ send.

Definition of ‘small message’ controlled by environment variables.

## Exercise 16 (serialsend)

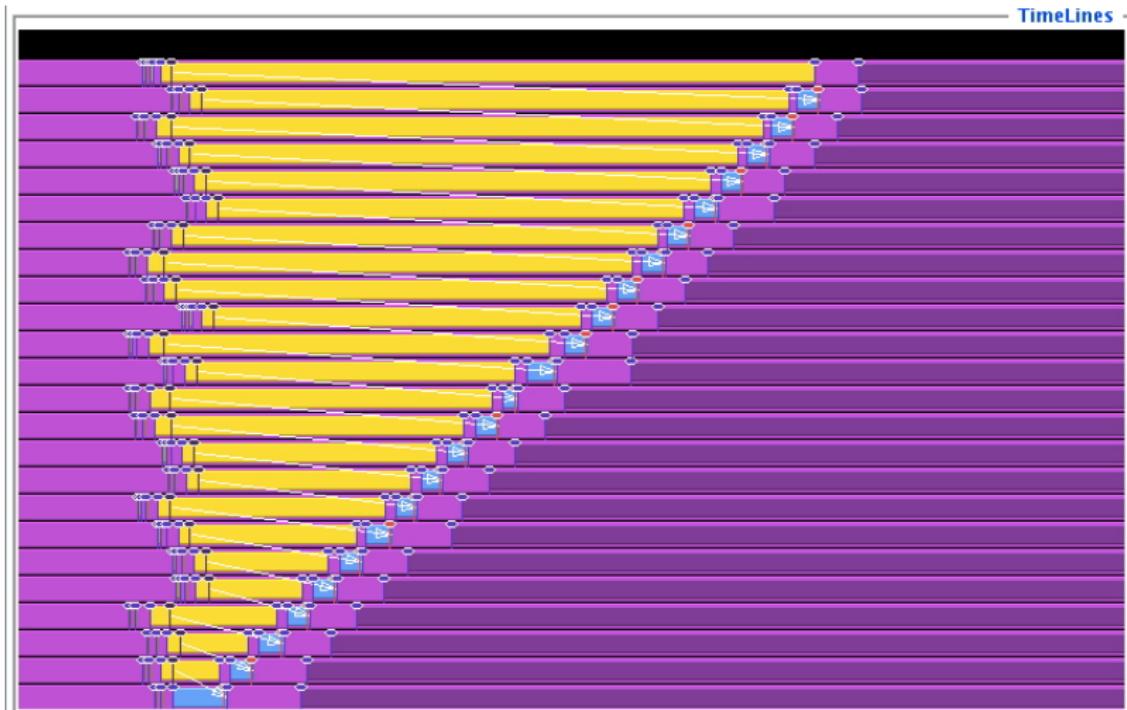
(Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

- ① Give the paper to your right neighbour;
- ② Accept the paper from your left neighbour.

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 neighbour.
- ② If you are not the leftmost student, turn to your left and accept the paper from your left neighbour.

# TAU trace: serialization



## The problem here...

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

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

## Exercise (optional) 17

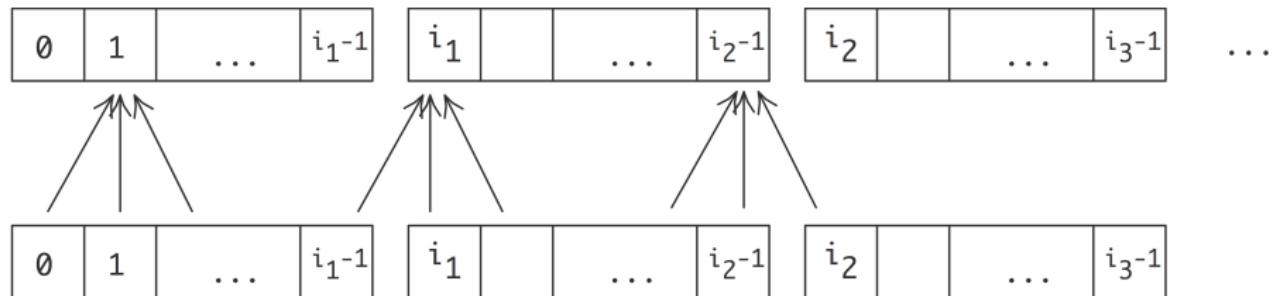
Implement the above algorithm using **MPI\_Send** and **MPI\_Recv** calls. Run the code, and use TAU to reproduce the trace output of figure 121. If you don't have TAU, can you show this serialization behaviour using timings?

# **Pairwise exchange**

# Operating on distributed data

Take another look:

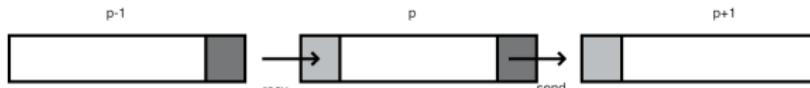
$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \dots, N-1$$



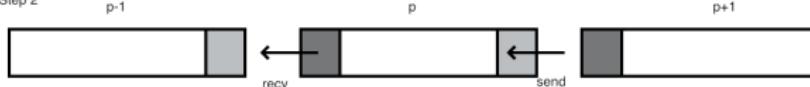
- One-dimensional data and linear process numbering;
- Operation between neighbouring indices: communication between neighbouring processes.

# 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!

# 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

Semantics:

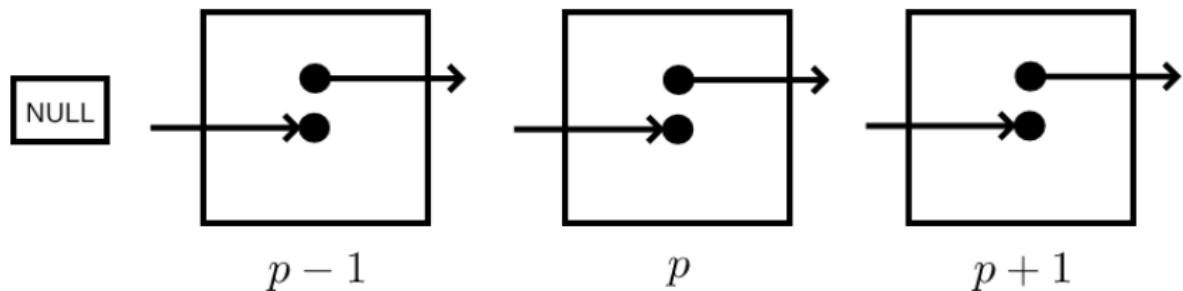
```
MPI_SENDRECV(
    sendbuf, sendcount, sendtype, dest, sendtag,
    recvbuf, recvcount, recvtype, source, recvtag,
    comm, status)
IN sendbuf: initial address of send buffer (choice)
IN sendcount: number of elements in send buffer (non-negative integer)
IN sendtype: type of elements in send buffer (handle)
IN dest: rank of destination (integer)
IN sendtag: send tag (integer)
OUT recvbuf: initial address of receive buffer (choice)
IN recvcount: number of elements in receive buffer (non-negative integer)
IN recvtype: type of elements in receive buffer (handle)
IN source: rank of source or MPI_ANY_SOURCE (integer)
IN recvtag: receive tag or MPI_ANY_TAG (integer)
IN comm: communicator (handle)
OUT status: status object (Status)
```

C:

```
int MPI_Sendrecv(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf, int recvcount, MPI_Datatype recvtyp
Eijkhout: MPI course, int recvtag,
```

# SPMD picture

What does process  $p$  do?



## Sendrecv with incomplete pairs

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

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

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

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

# 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) 18 (right send)

Revisit exercise 16 and solve it using **MPI\_Sendrecv**.

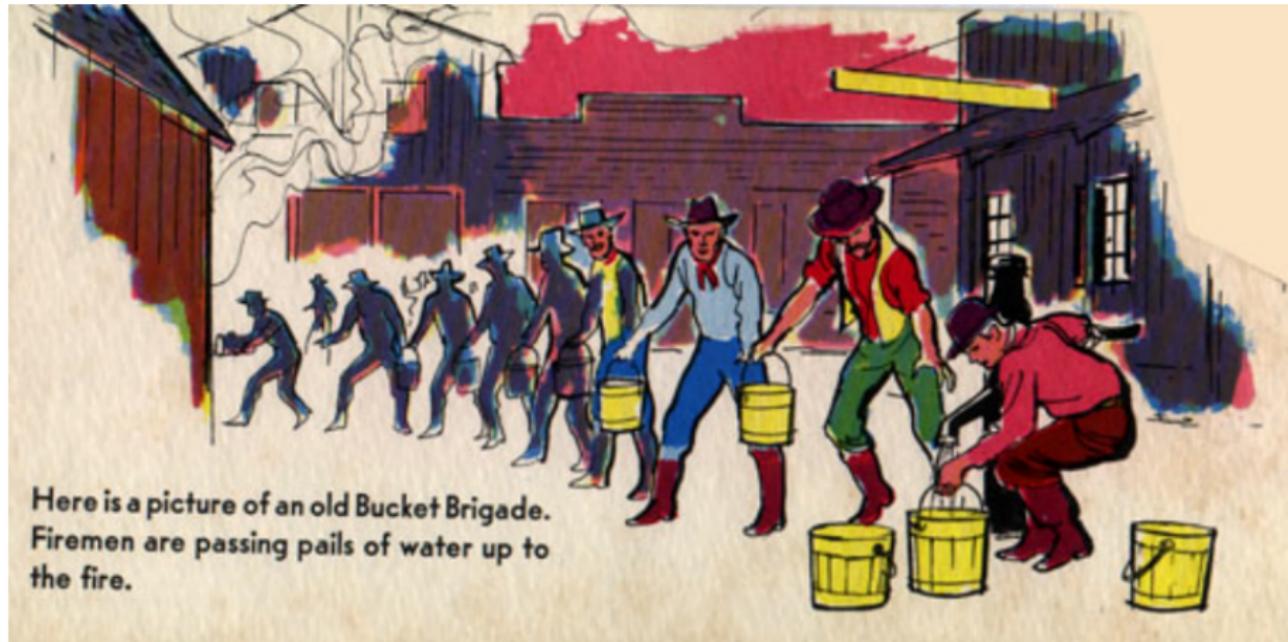
If you have TAU installed, make a trace. Does it look different from the serialized send/recv code? If you don't have TAU, run your code with different numbers of processes and show that the runtime is essentially constant.

## Exercise 19 (sendrecv)

Implement the above three-point combination scheme using **MPI\_Sendrecv**; every processor only has a single number to send to its neighbour.

# 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.

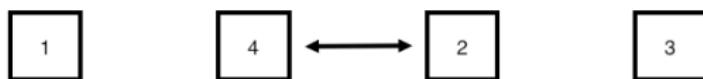
## Exercise 20 (bucketblock)

Take the code of exercise 17 and modify it so that the data from process zero gets propagated to every process. Specifically: compute

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

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

# Odd-even transposition sort



↔ transpose performed  
↔ no transpose needed

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.

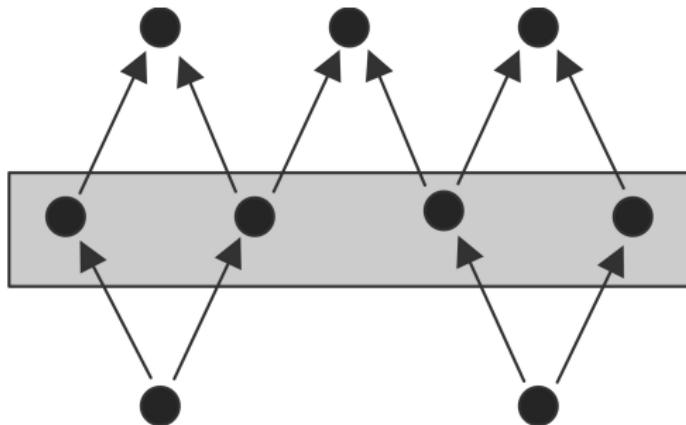
The exchange sort algorithm is split in even and odd stages, where in the even stage, processors  $2i$  and  $2i + 1$  compare and swap data, and in the odd stage, processors  $2i + 1$  and  $2i + 2$  compare and swap. You need to repeat this  $P/2$  times, where  $P$  is the number of processors; see figure 136.

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.

# **Irregular exchanges: non-blocking communication**

# Sending with irregular connections

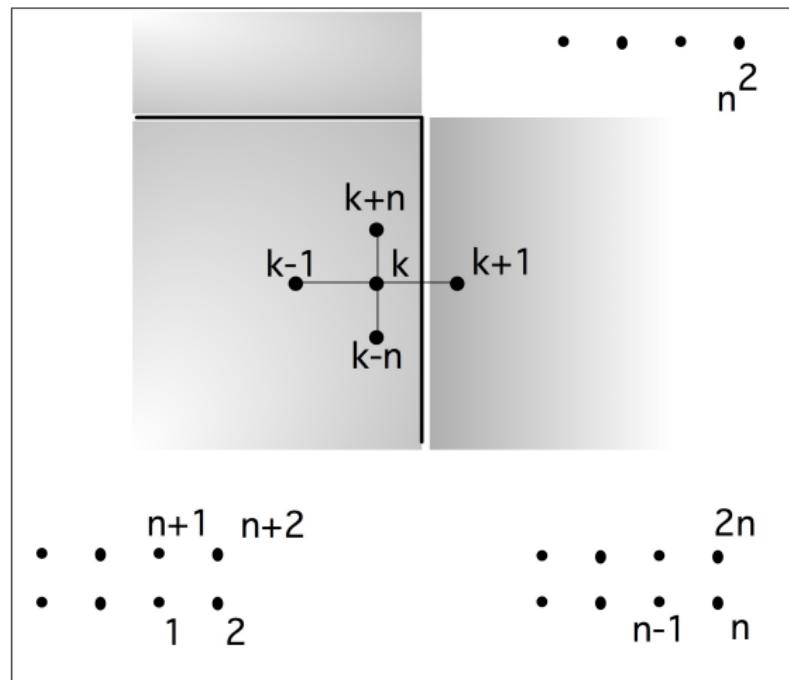
Graph operations:



# **Communicating other than in pairs**

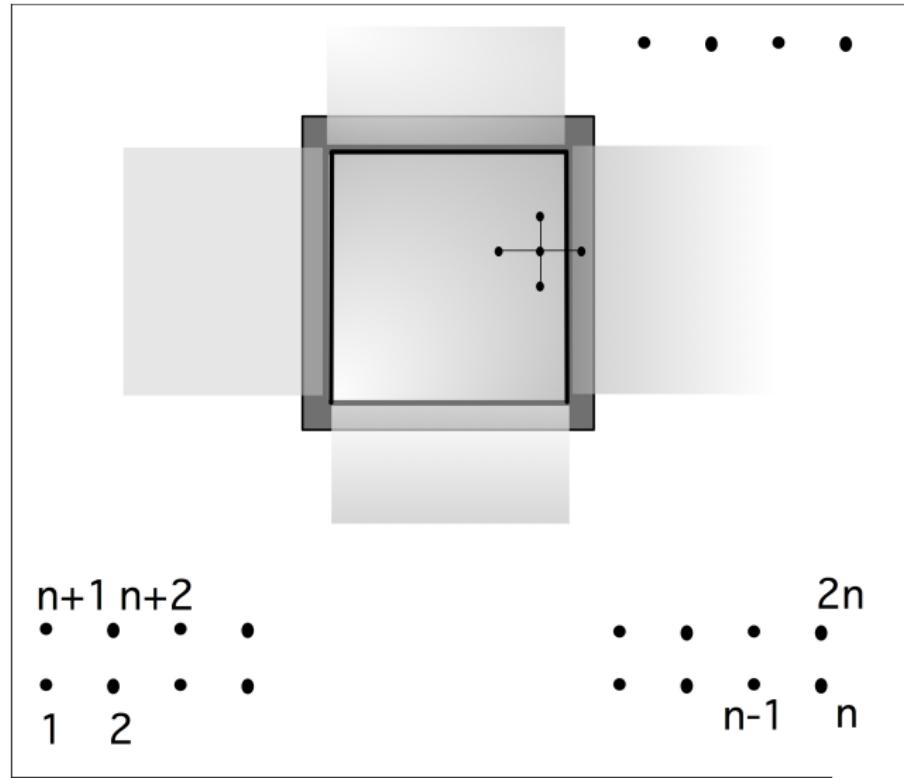
## PDE, 2D case

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



# Halo region

The halo region of a process, induced by a stencil



# PDE matrix

$$A = \left( \begin{array}{cccc|ccc|c} 4 & -1 & & & 0 & -1 & -1 & & 0 \\ -1 & 4 & -1 & & & & & & \\ \ddots & \ddots & \ddots & & & & \ddots & & \\ & \ddots & \ddots & -1 & & & \ddots & & \\ \hline 0 & & -1 & 4 & 0 & 0 & & -1 & \\ -1 & & 0 & & 4 & -1 & & -1 & -1 \\ & -1 & & & -1 & 4 & -1 & & -1 \\ & \uparrow & \ddots & & \uparrow & \uparrow & \uparrow & & \uparrow \\ k-n & & & & k-1 & k & k+1 & & k+n \\ \hline & & -1 & & & & -1 & 4 & \\ & & & & & \ddots & & & \ddots \end{array} \right)$$

## How do you approach this?

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

Instead:

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

# Non-blocking send/recv

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

# Syntax

Very much like blocking **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)
```

The **MPI\_Request**s can be tested:

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

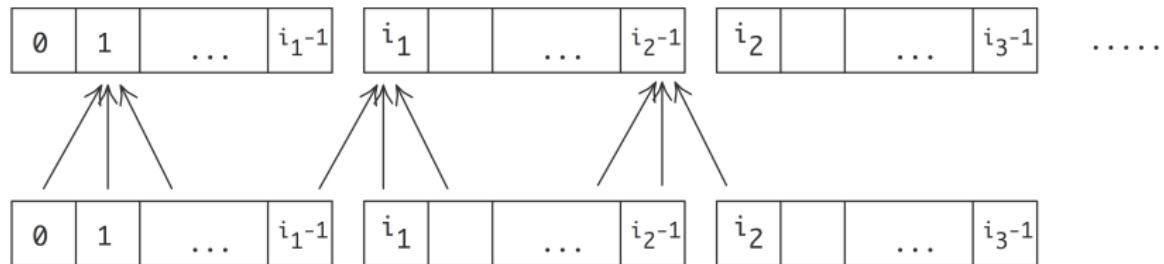
- ignore status: **MPI\_STATUSES\_IGNORE**
- also **MPI\_Wait**, **MPI\_Waitany**, **MPI\_Waitsome**

## Exercise 22 (isendirecv)

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

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

on a distributed array. (Hint: use `MPI_PROC_NULL` at the ends.)



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

# Comparison

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

# Buffer use in blocking/non-blocking case

Blocking:

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

Non-blocking:

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

## Latency hiding

Other motivation for non-blocking calls:

overlap of computation and communication, provided hardware support.

Also known as 'latency hiding'.

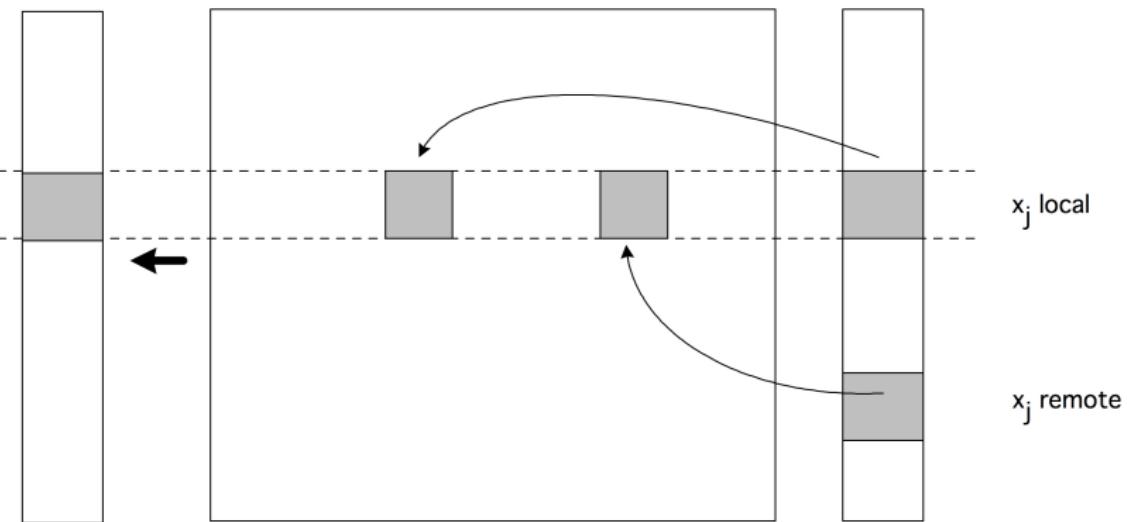
Example: three-point combination operation (see above):

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

# Matrices in parallel

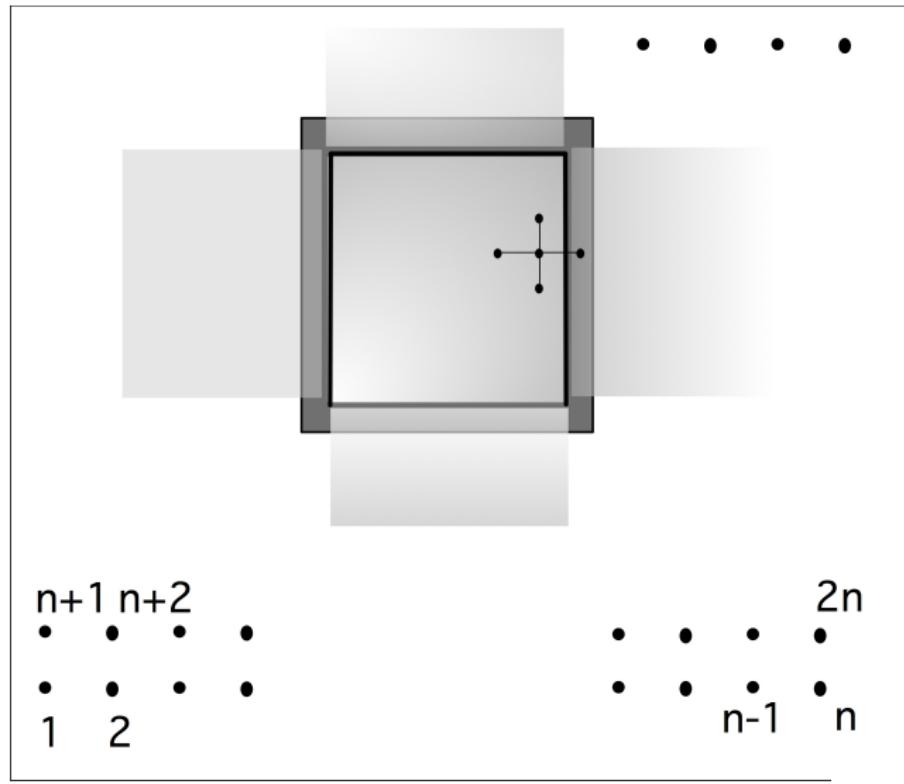
$$y \leftarrow Ax$$

and  $A, x, y$  all distributed:



# Hiding the halo

Interior of a process domain can overlap with halo transfer:



## Exercise 23 (isendirecvarray)

Take your code of exercise 22 and modify it to use latency hiding. Operations that can be performed without needing data from neighbours 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.

# 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
}
```

# 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) 24 (bucketpipenonblock)

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

## 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.

too obscure to go into.

# Advanced topics

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



# 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

# 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**/Unpack
- F90 types

# Discussion

# 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.

# Datatype topics

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

# Datatypes

# 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

# 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.

# Contiguous type

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



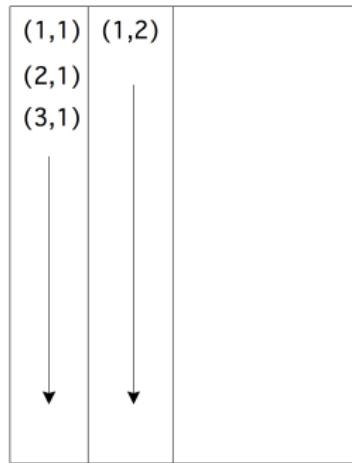
This one is indistinguishable from just sending `count` instances of the `old_type`.

# Example: non-contiguous data

Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

Logical:

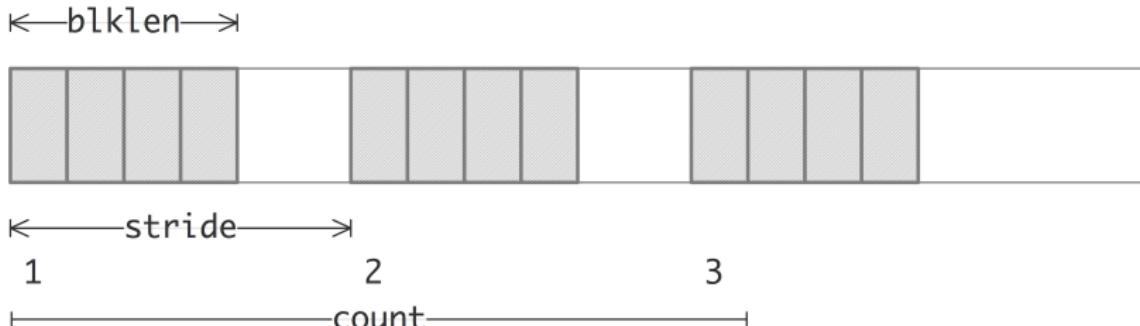


Physical:

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

# 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.

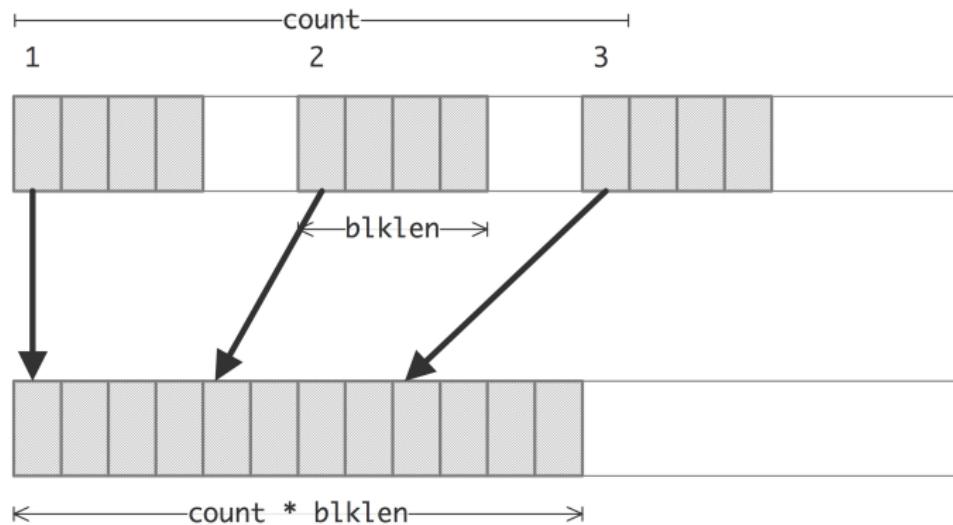
```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    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);
}
```

# Different send and receive types

Send and receive type can differ. Example:

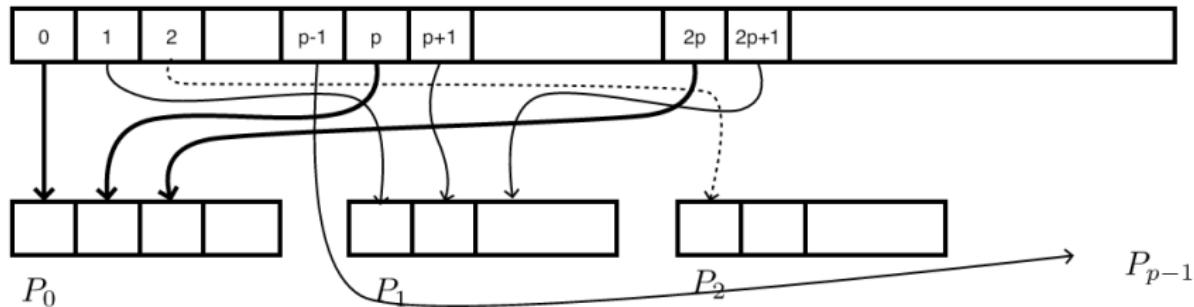
Sender type: vector

receiver type: contiguous or elementary



Receiver has no knowledge of the stride of the sender.

## Illustration of the next exercise



Sending strided data from process zero to all others

## Exercise 25 (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.

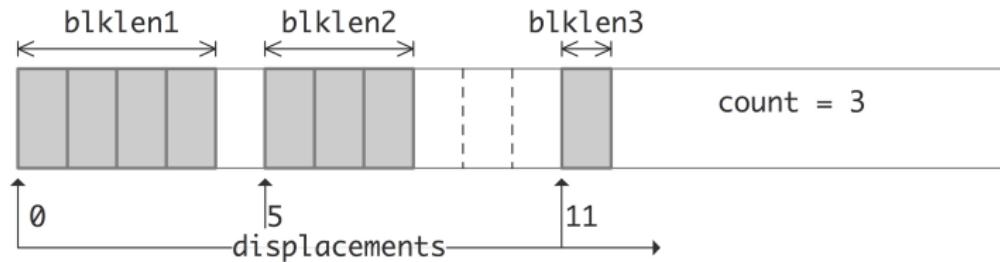
For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?

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

## Exercise 26

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

# Indexed type



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

## Hindexed type

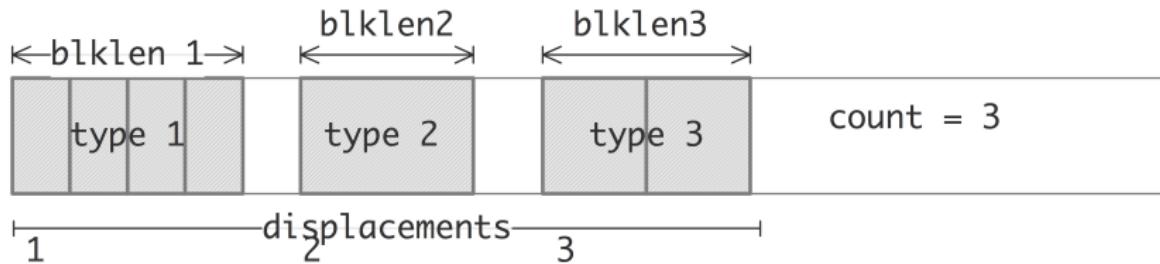
Similar to indexed but using byte offsets:  
explicit memory address.

Example usage scenario: send linked list.

Use **MPI\_Get\_address**

# Heterogeneous: Structure type

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

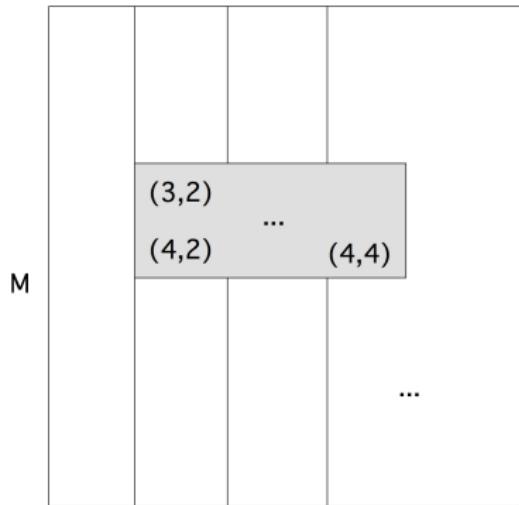


This gets very tedious...

# **Subarray type**

# Submatrix storage

Logical:



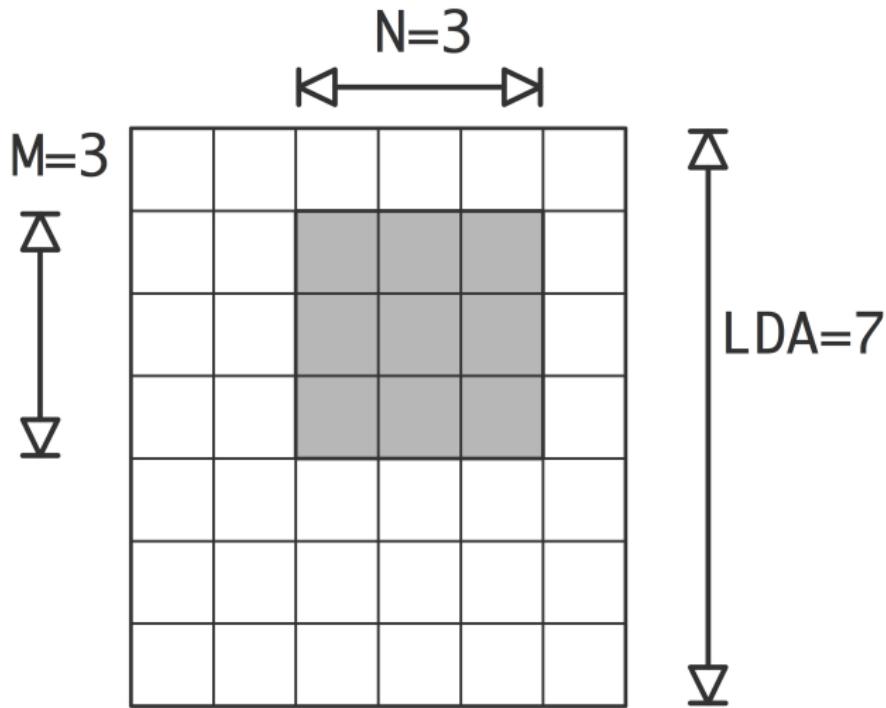
Physical:



- Location of first element
- Stride, blocksize

# BLAS/Lapack storage

Three parameter description:



How about as a 'block within a block'?

# 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 27 (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.

# 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 )
```

# Packed data

# 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);
```

# Example

```
// pack.c
if (procno==sender) {
    MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
    for (int i=0; i<nsends; i++) {
        double value = rand()/(double)RAND_MAX;
        MPI_Pack(&value,1,MPI_DOUBLE,buffer,buflen,&position,comm);
    }
    MPI_Pack(&nsends,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,&nsends,1,MPI_INT,comm);
    for (int i=0; i<nsends; 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==nsends);
```

# Part V

## Communicator manipulations

# 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.

# 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`.

# 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(...);
```

# Use of a library

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

# Use of a library

```
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;
}
```

# Wrong way

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

# Right way

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

# Disjoint splitting

Split a communicator in multiple disjoint others.

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

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

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

# Row/column example

Simulate a processor grid

create subcommunicator per column (or row):

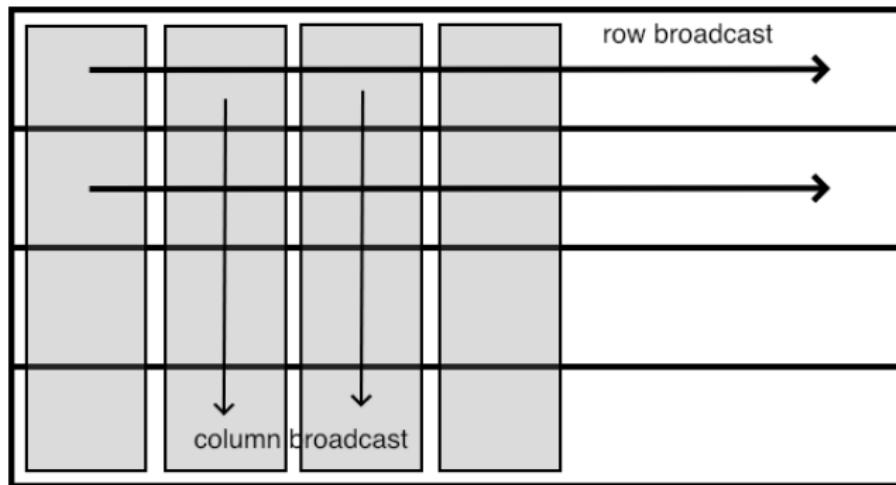
```
MPI_Comm_rank( MPI_COMM_WORLD, &procno );
proc_i = procno % proc_column_length;
proc_j = procno / proc_column_length;

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?

# Row and column communicators



Row and column broadcasts in subcommunicators

## Exercise 28 (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:

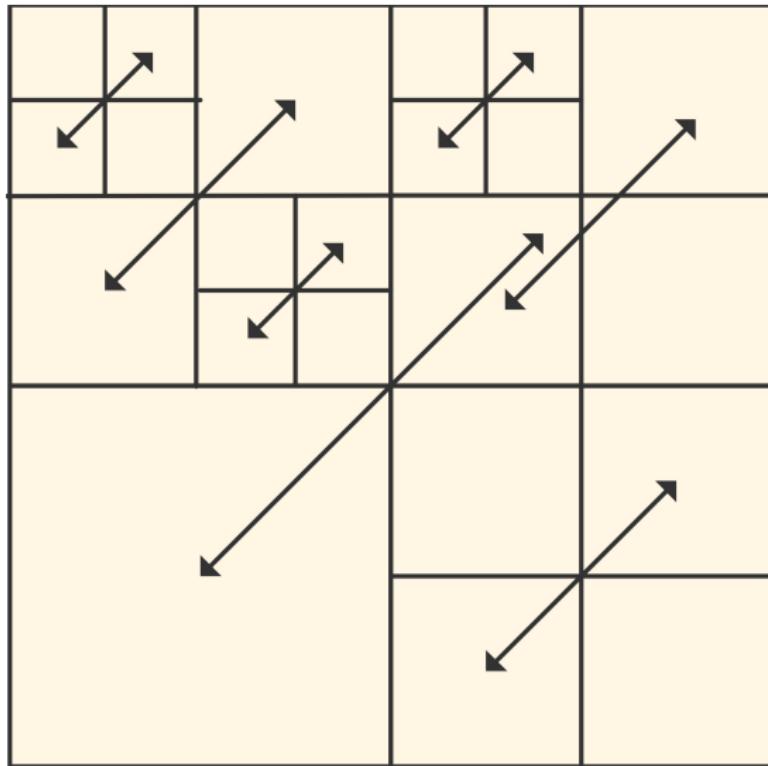
Global ranks:	Ranks in row:	Ranks in column:
0 1 2	0 1 2	0 0 0
3 4 5	0 1 2	1 1 1

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

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

## Exercise 29

Implement a recursive algorithm for matrix transposition:



# 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.

```
// commssplittype.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);

ASSERT(new_procno<CORES_PER_NODE);
```

## More

- Non-disjoint subcommunicators through process groups.
- Intra-communicators and inter-communicators.  
These will come up later in the section about process management.
- Process topologies: cartesian and graph.  
There will also be a section about this, later.

# Part VI

## MPI File I/O

# Overview

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

Commands learned:

- **MPI\_File\_open**/write/close
- parallel file pointer routines: **MPI\_File\_set\_view**/write\_at

# 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.

# 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 ...

# 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
```

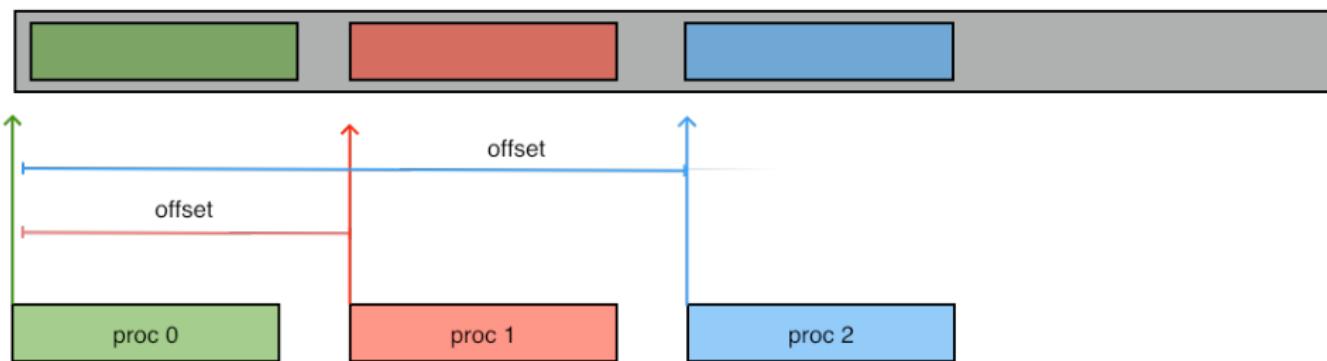
# How do you make it unique for a process?

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

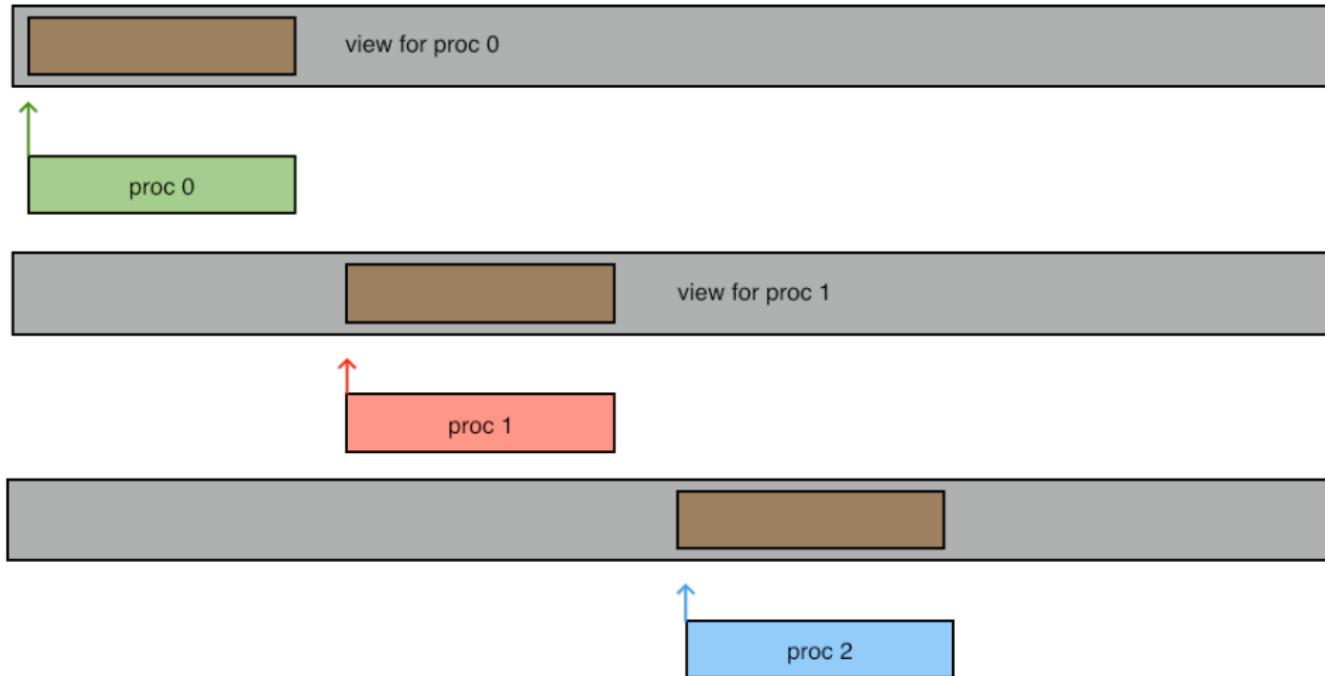
or

```
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);
```

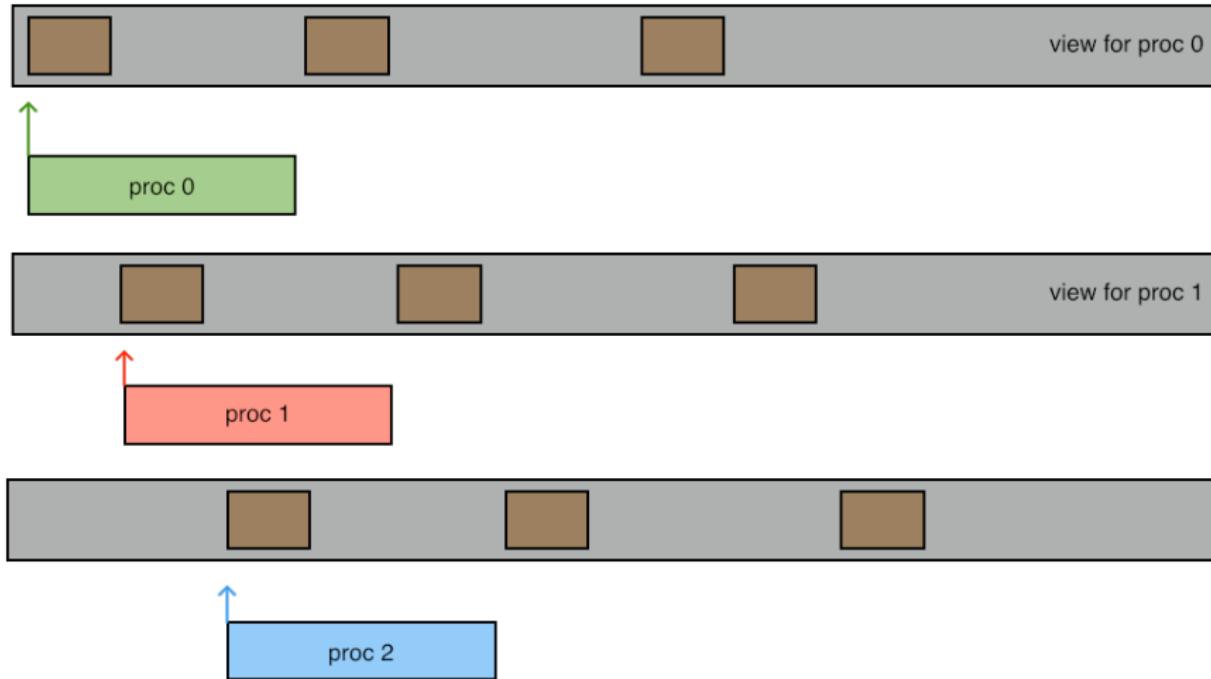
# Write at an offset



# Write to a view



# Write to a view



Made with Vex

## Exercise 30 (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 31 (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 32 (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

# 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`

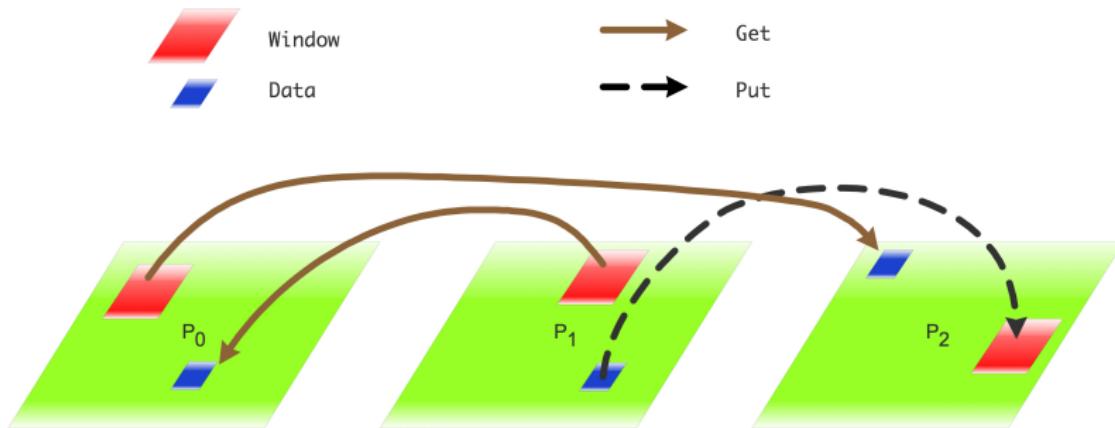
# **Basic mechanisms**

# Motivation

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

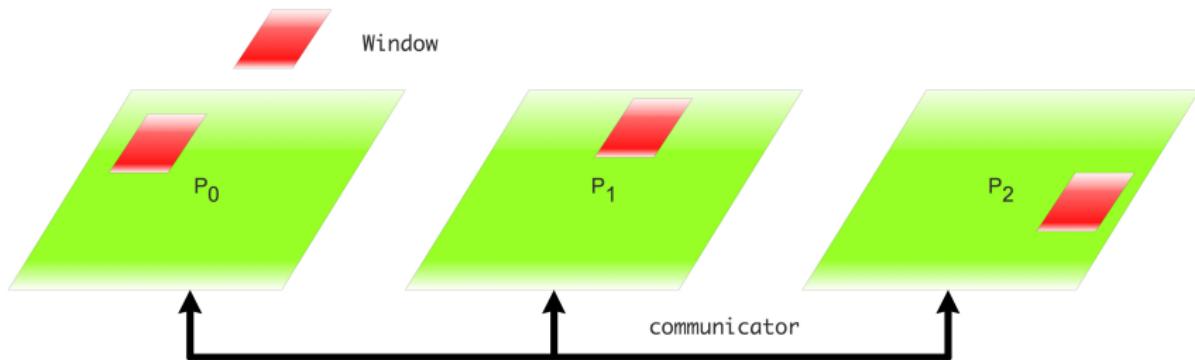
- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbours. Uncertain when this happens.
- Other irregular data structures: distributed hash tables.

# 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.

# 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: ***MPI\_Win\_allocate***, can use dedicated fast memory.

Also call ***MPI\_Win\_free*** when done. This is important!

# Window allocation

Instead of passing buffer, let MPI allocate:

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

# 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

C:

```
int MPI_Put(
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank,
    MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win)
```

Semantics:

IN origin\_addr: initial address of origin buffer (choice)  
IN origin\_count: number of entries in origin buffer (non-negative integer)  
IN origin\_datatype: datatype of each entry in origin buffer (handle)  
IN target\_rank: rank of target (non-negative integer)  
IN target\_disp: displacement from start of window to target buffer (non-negative integer)  
IN target\_count: number of entries in target buffer (non-negative integer)  
IN target\_datatype: datatype of each entry in target buffer (handle)  
IN win: window object used for communication (handle)

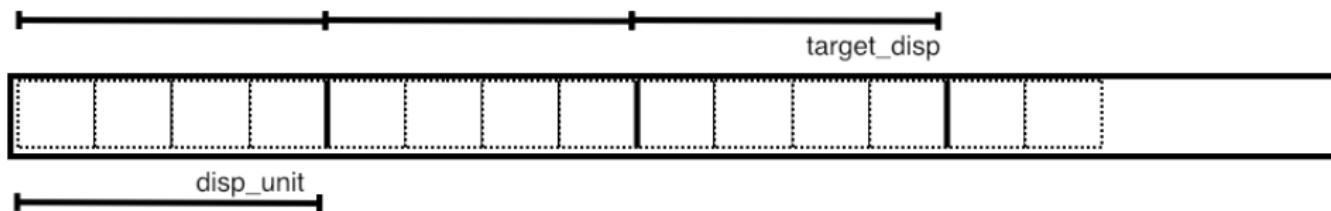
Fortran:

```
MPI_Put(origin_addr, origin_count, origin_datatype,
         target_rank, target_disp, target_count, target_datatype, win, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, tar
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_c
Eijkhede MPP course, INTENT(IN) :: win
```

# Location in the window

Location to write:

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



## Exercise 33 (rightput)

Revisit exercise 16 and solve it using **MPI\_Put**.

## Exercise 34 (randomput)

Write code where process 0 randomly writes in the window on 1 or 2.

## Exercise (optional) 35 (randomput )

Replace **MPI\_Win\_create** by **MPI\_Win\_allocate**.

## 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

C:

```
int MPI_Get(
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank,
    MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win)
```

Semantics:

IN origin\_addr: initial address of origin buffer (choice)  
IN origin\_count: number of entries in origin buffer (non-negative integer)  
IN origin\_datatype: datatype of each entry in origin buffer (handle)  
IN target\_rank: rank of target (non-negative integer)  
IN target\_disp: displacement from start of window to target buffer (non-negative integer)  
IN target\_count: number of entries in target buffer (non-negative integer)  
IN target\_datatype: datatype of each entry in target buffer (handle)  
IN win: window object used for communication (handle)

Fortran:

```
MPI_Get(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count, target_datatype, win, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, tar
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_c
Eijkhede MPP course, INTENT(IN) :: win
```

# MPI\_Accumulate

C:

```
int MPI_Accumulate
  (const void *origin_addr, int origin_count,MPI_Datatype origin_datatype,
   int target_rank,MPI_Aint target_disp, int target_count,MPI_Datatype target_datatype,
   MPI_Op op, MPI_Win win)
int MPI_Raccumulate
  (const void *origin_addr, int origin_count,MPI_Datatype origin_datatype,
   int target_rank,MPI_Aint target_disp, int target_count,MPI_Datatype target_datatype,
   MPI_Op op, MPI_Win win,MPI_Request *request)
```

## Input Parameters

origin\_addr : Initial address of buffer (choice).

origin\_count : Number of entries in buffer (nonnegative integer).

origin\_datatype : Data type of each buffer entry (handle).

target\_rank : Rank of target (nonnegative integer).

target\_disp : Displacement from start of window to beginning of target buffer (nonnegative integer).

target\_count : Number of entries in target buffer (nonnegative integer).

target\_datatype : Data type of each entry in target buffer (handle).

op : Reduce operation (handle).

win : Window object (handle).

## Output Parameter

# Ordering and synchronization

# 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.

# 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) 36 (countdown)

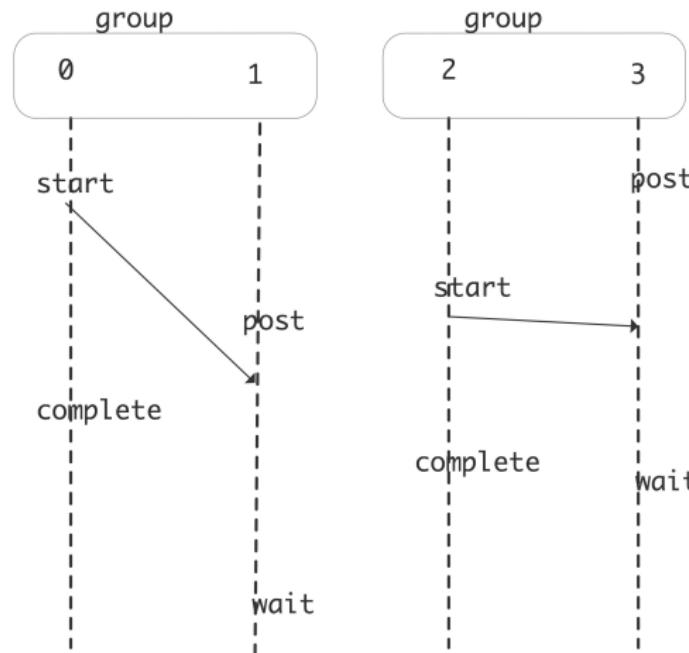
Implement a shared counter:

- 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?

# A second active synchronization

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



More fine-grained than fences.

# **Passive target synchronization**

# 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

## Exercise 37 (onesidedbuild)

- Let each process have an empty array of sufficient length and a stack pointer that maintains the first free location.
- Now let each process randomly put data in a free location of another process' array.
- Use window locking. (Why is active target synchronization not possible?)

# Atomic operations

## Emulating shared memory with one-sided communication

- One process stores a table of work descriptors, and a pointer to the first unprocessed descriptor;
- Each process reads the pointer, reads the corresponding descriptor, and increments the pointer; and
- A process that has read a descriptor then executes the corresponding task.

# Race condition example

process 1:  $I = I + 2$

process 2:  $I = I + 3$

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

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

# Atomic operations

Race condition problem in read/write:

- Result of `MPI_Get` is only known after the fence.
- Another process may have done an `MPI_Put` in that epoch.
- No guarantee on the correctness of the result of the `MPI_Get`

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

```
int MPI_Fetch_and_op
  (const void *origin_addr, void *result_addr,
   MPI_Datatype datatype,
   int target_rank, MPI_Aint target_disp,
   MPI_Op op, MPI_Win win)
```

```
// passive.cxx
if (procno==repository) {
    // Repository processor creates a table of inputs
    // and associates that with the window
}
if (procno!=repository) {
    float contribution=(float)procno,table_element;
    int loc=0;
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
    // read the table element by getting the result from adding zero
    err = MPI_Fetch_and_op
        (&contribution,&table_element,MPI_FLOAT,
         repository,loc,MPI_SUM,the_window); CHK(err);
    MPI_Win_unlock(repository,the_window);
}
```

## Exercise (optional) 38

Redo exercise 36 using **`MPI_Fetch_and_op`**. The problem is again to make sure all processes have the same view of the shared counter.

Does it work to make the fetch-and-op conditional? Is there a way to do it unconditionally? What should the ‘break’ test be, seeing that multiple processes can update the counter at the same time?



# Advanced (MPI-3) topics

## Justification

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

## Part VIII

More about collectives

# User-defined operators

## Exercise 39 (onenorm)

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

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

# Non-blocking collectives

# 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

# 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
- No guaranteed progress.

## MPI\_Ibcast

C:

```
int MPI_Ibcast(
    void* buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm,
    MPI_Request *request
)
```

Fortran:

```
MPI_Ibcast(buffer, count, datatype, root, comm, ierror)
TYPE(*), DIMENSION(..) :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
TYPE(MPI_Request), intent(out) :: request
```

# 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
```

## Exercise 40 (procgridnonblock)

▶ Earlier procgrid exercise

Revisit exercise 28. 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 non-blocking broadcasts, and time the difference between a blocking and a non-blocking solution.

# Non-blocking barrier

## Just what is a barrier?

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

## Use case: adaptive refinement

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

## 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

C:

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

Input Parameters

comm : communicator (handle)

Output Parameters

request : communication request (handle)

Fortran2008:

```
MPI_Ibarrier(comm, request, ierror)
Type(MPI_Comm),intent(int) :: comm
TYPE(MPI_Request),intent(out) :: request
INTEGER,intent(out),optional :: ierror
```

```
// ibarriertest.c
for ( ; ; step++) {
    int barrier_done_flag=0;
    MPI_Test(&barrier_request,&barrier_done_flag,
                MPI_STATUS_IGNORE);
    if (barrier_done_flag) {
        break;
    } else {
        int flag; MPI_Status status;
        MPI_Iprobe
            ( MPI_ANY_SOURCE,MPI_ANY_TAG,
              comm, &flag, &status );
    }
}
```

## Exercise 41 (ibarrierupdate)

- Let each process send to a random number of randomly chosen neighbours. 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?

# Problem with ‘progress’

- Problem: `MPI_Test` is local
- Something needs to force the barrier information to propagate
- Solution: force progress with `MPI_Iprobe`
- Frowny face: barrier completion takes much longer than you'd expect.

# Part IX

## Shared memory

# 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.

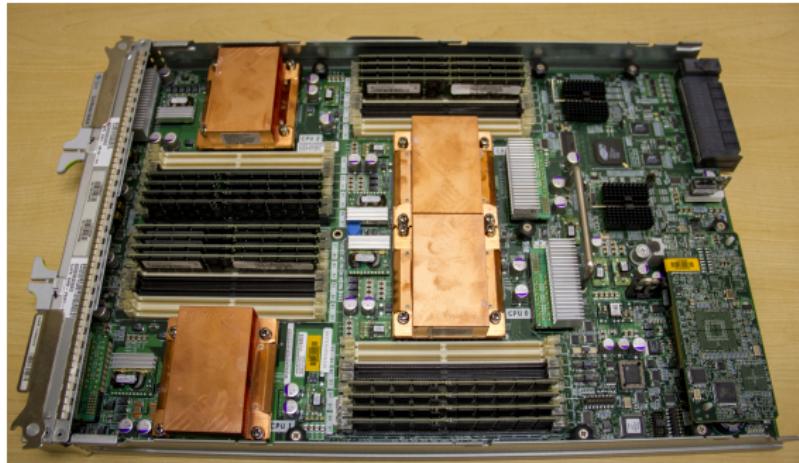
# MPI shared memory

- 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.

# 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.*

## Shared memory per cluster node



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

# 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`.

# Discover shared memory

- **MPI\_Comm\_split\_type** splits into communicators of same type.
- Only supported type: **MPI\_COMM\_TYPE\_SHARED** splitting by shared memory.

```
// commssplittype.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);

ASSERT(new_procno<CORES_PER_NODE);
```

# 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_Aint window_size; double *window_data; MPI_Win
    node_window;
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 );
```

# 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 )  
|| ;
```

## Boring example: heat equation

```
// sharedshared.c
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&
    shared_baseptr,&shared_window);
```

## 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 42 (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 X

## Process management

# Overview

This section discusses processes management; intra communicators.

Commands learned:

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

# 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’)

# Space for processes

Probably a machine dependent component.

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);
```

# Manager program

```
MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
              (void*)&universe_sizep, &flag);
MPI_Comm everyone; /* intercommunicator */
int nworkers = universe_size-world_size;
MPI_Comm_spawn(worker_program, /* executable */
               MPI_ARGV_NULL, nworkers,
               MPI_INFO_NULL, 0, MPI_COMM_WORLD, &everyone,
               errorcodes);
```

# Worker program

```
|| MPI_Comm_size(MPI_COMM_WORLD,&nworkers);  
|| MPI_Comm parent;  
|| MPI_Comm_get_parent(&parent);  
|| MPI_Comm_remote_size(parent, &remotesize);
```

# Part XI

## Process topologies

# Overview

This section discusses graph topologies.

Commands learned:

- **`MPI_Dist_graph_create`**, `MPI_DIST_GRAPH`
- `MPI_Neighbor_...`

# Process topologies

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

# 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

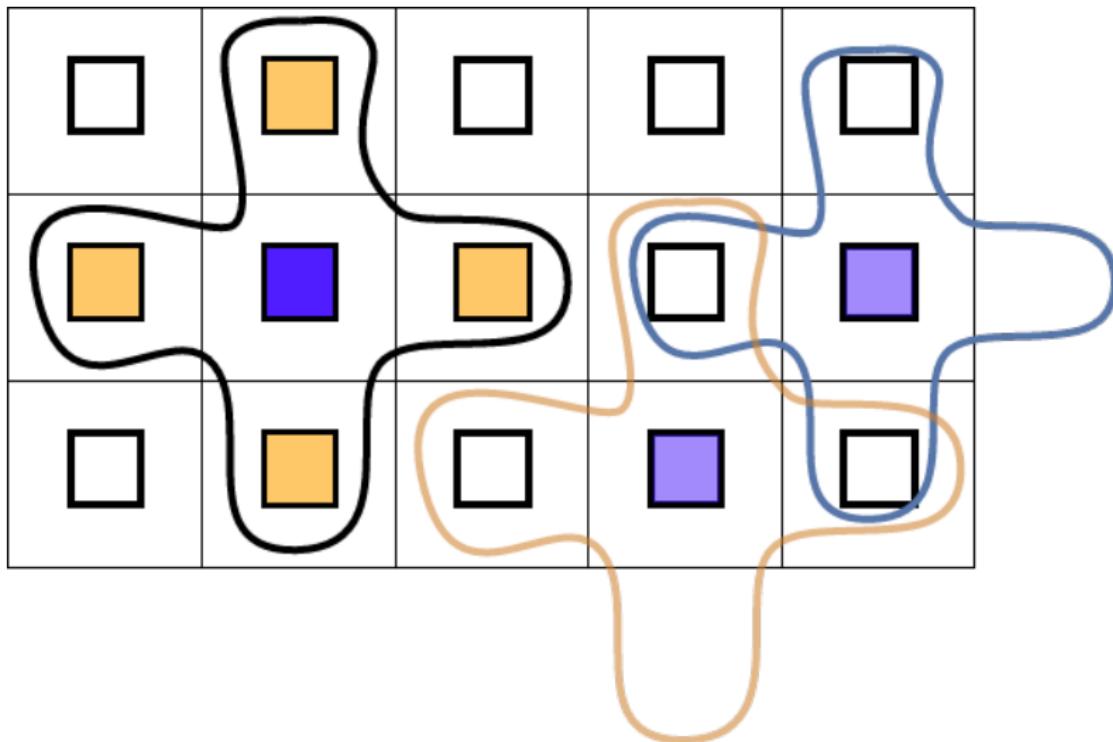
# MPI-1 topology

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

## MPI-2 and 3 topology

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

## Neighbourhood collective



Distributed graph topology where each node has four neighbours

# Why neighbourhood 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.

# Create graph topology

```
int MPI_Dist_graph_create
  (MPI_Comm comm_old, int n, 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 processes 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

# Neighbourhood collectives

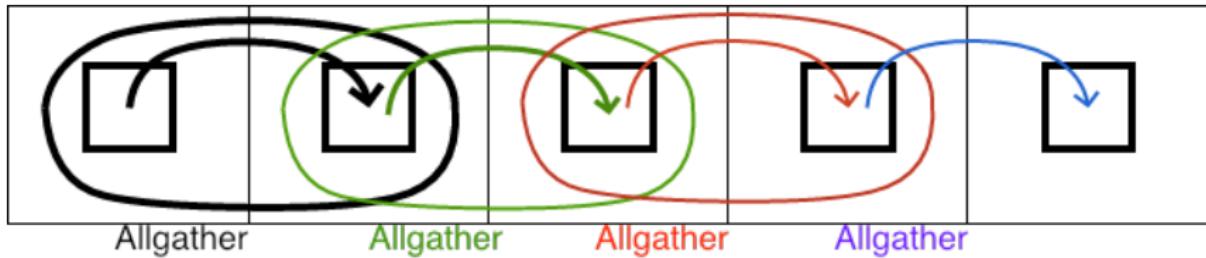
```
|| int MPI_Neighbor_allgather
||   (const void *sendbuf, int sendcount, MPI_Datatype sendtype,
||    void *recvbuf, int recvcount, MPI_Datatype recvtype,
||    MPI_Comm comm)
```

Like an ordinary **MPI\_Allgather**, but  
the receive buffer has a length degree.

## Exercise 43 (rightgraph)

Revisit exercise 16 and solve it using `MPI_Dist_graph_create`. Use figure 292 for inspiration.

## Inspiring picture for the previous exercise



Solving the right-send exercise with neighbourhood collectives



# Other

# Part XII

## Performance

# Overview

We briefly touch on issues that influence MPI performance.

# Performance measurement

# 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.

# Example

```
// pingpong.c
int src = 0, tgt = nprocs/2;
double t, send=1.1, recv;
if (procno==src) {
    t = MPI_Wtime();
    for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Send(&send, 1, MPI_DOUBLE, tgt, 0, comm);
        MPI_Recv(&recv, 1, MPI_DOUBLE, tgt, 0, comm, MPI_STATUS_IGNORE);
    }
    t = MPI_Wtime() - t; t /= NEXPERIMENTS;
    printf("Time for pingpong: %e\n", t);
} else if (procno==tgt) {
    for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Recv(&recv, 1, MPI_DOUBLE, src, 0, comm, MPI_STATUS_IGNORE);
        MPI_Send(&recv, 1, MPI_DOUBLE, src, 0, comm);
    }
}
```

# Global timing

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

Alternative:

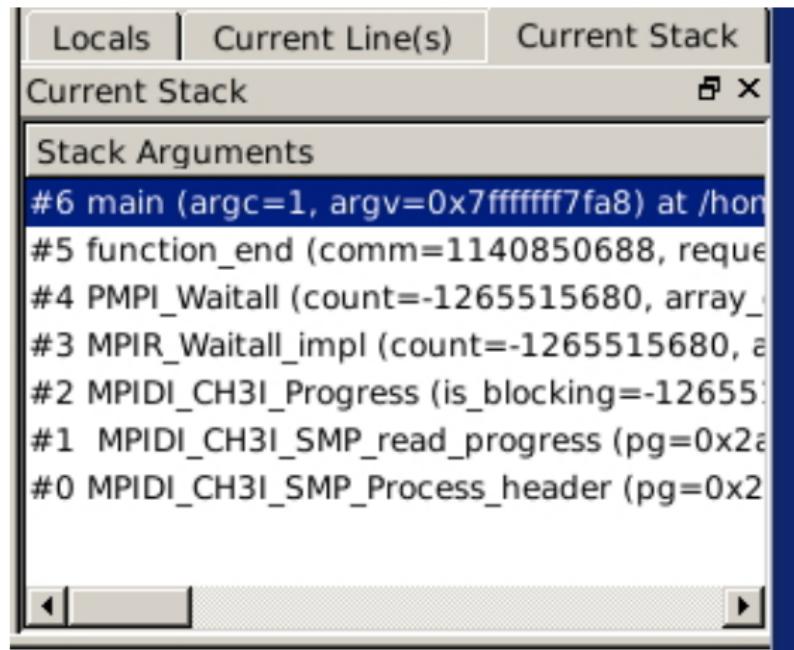
```
||   MPI_Barrier(comm)
||   t = MPI_Wtime();
||   // something happens here
||   MPI_Barrier(comm)
||   t = MPI_Wtime()-t;
```

# Profiling

See other lecture: MPIP, TAU, et cetera.

## 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.



The screenshot shows a window titled "Current Stack" with three tabs at the top: "Locals", "Current Line(s)", and "Current Stack". The "Current Stack" tab is selected. Below the tabs, there is a "Stack Arguments" section. The stack trace is listed as follows:

```
#6 main (argc=1, argv=0x7fffffff7fa8) at /home/.../main.c:10
#5 function_end (comm=1140850688, request=1)
#4 PMPI_Waitall (count=-1265515680, array_size=1, array=0x2a00000000000000)
#3 MPIR_Waitall_impl (count=-1265515680, array_size=1, array=0x2a00000000000000)
#2 MPIDI_CH3I_Progress (is_blocking=-1265515680)
#1 MPIDI_CH3I_SMP_read_progress (pg=0x2a00000000000000)
#0 MPIDI_CH3I_SMP_Process_header (pg=0x2a00000000000000)
```

At the bottom of the window are navigation buttons: a left arrow, a right arrow, and a double-right arrow.

# Programming for performance

# Eager limit

- Optimization for small messages: bypass rendez-vous protocol (slide 119)
- 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.

# 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`.

# Blocking versus non-blocking

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

# 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`**.

# 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

# 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.

# Graph topology and neighborhood collectives

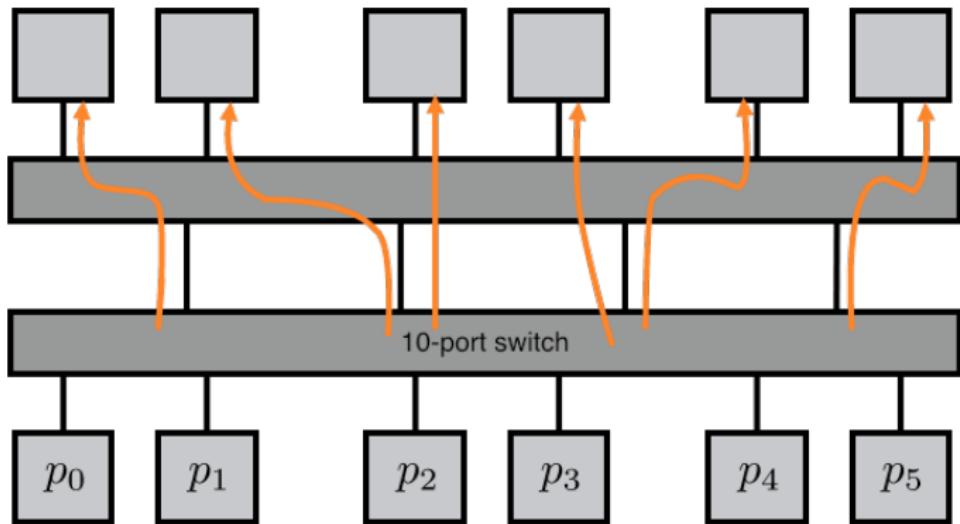
- Mapping problem to architecture sometimes not trivial
- Load balancers: *ParMetis, Zoltan*
- Graph topologies: `MPI_Dist_graph_adjacent`:  
allowed to reorder ranks for proximity
- Neighborhood collectives allow MPI to schedule optimally.
  - `MPI_Neighbor_allgather` (**and** `allgather_v`)
  - `MPI_Neighbor_alltoall`

# Network issues

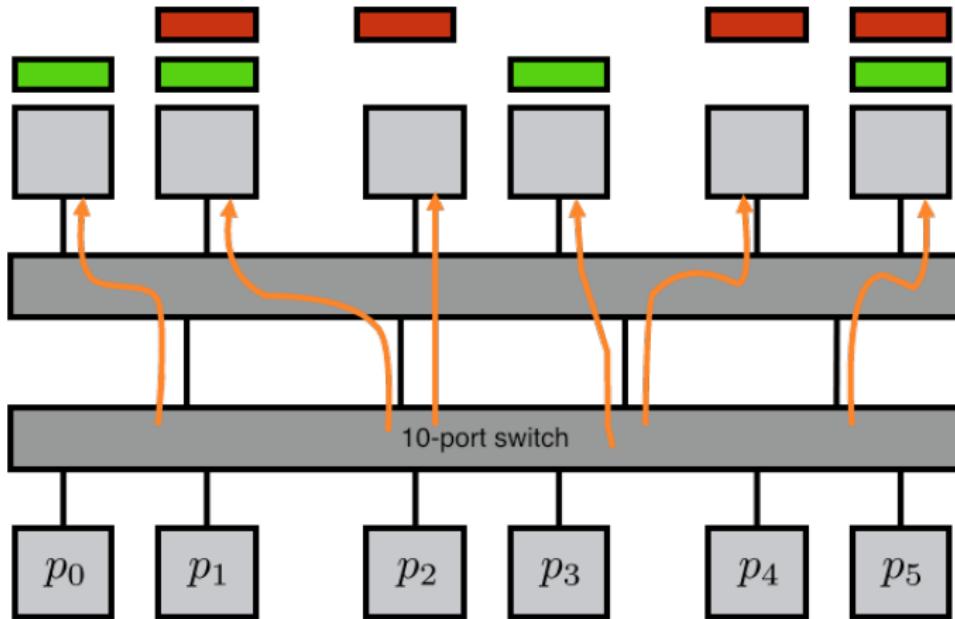
Network contention means that

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

# Output routing



# Contention



# 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.