TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Tutorial on MPI programming, Advanced Topics

Victor Eijkhout eijkhout@tacc.utexas.edu

TACC training, 2018

TACC

One-sided communication

Overview

This section concernes one-sided operations, which allows ‘shared memory’ type programming.

Commands learned:

- `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
- Active target synchronization `MPI_Win_create`, `MPI_Win_fence`
- `MPI_Post/Wait/Start/Complete`
- Passive target synchronization `MPI_Win_lock/unlock`
- Atomic operations: `MPI_Fetch_and_op`

Table of Contents

1 Active target synchronization

2 Passive target synchronization

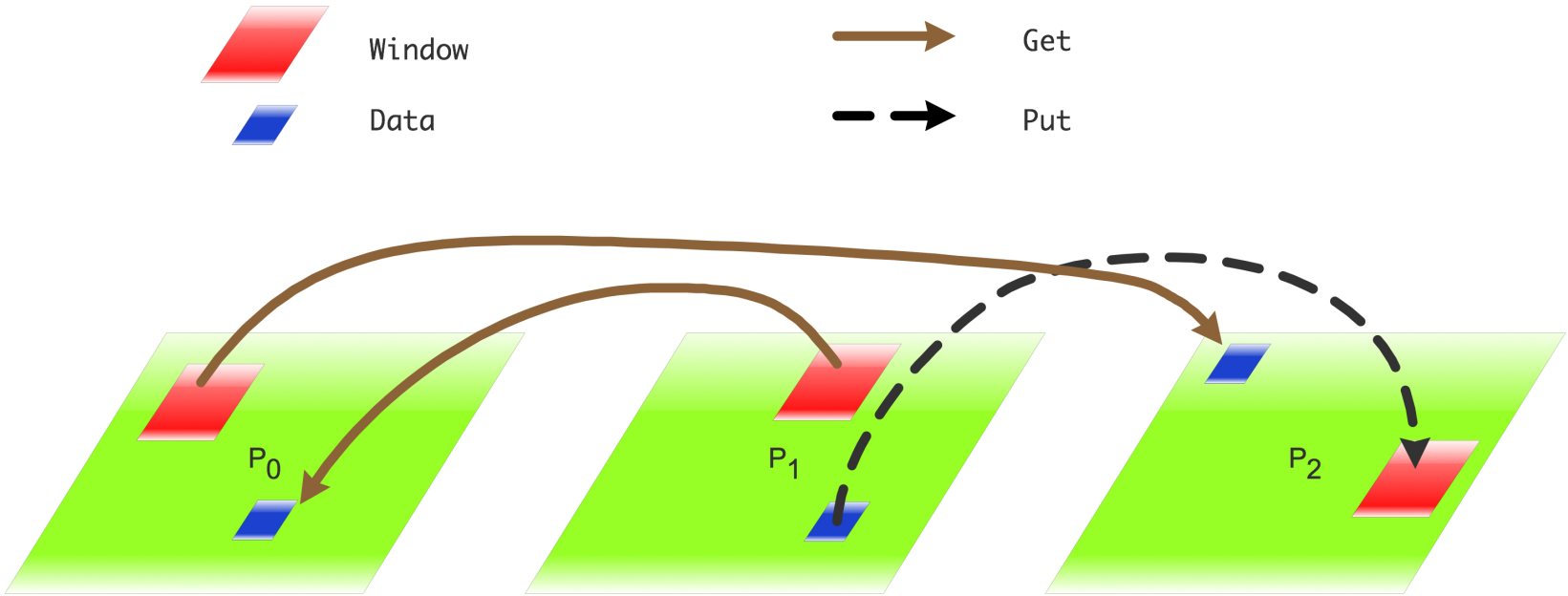
3 More

Motivation

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

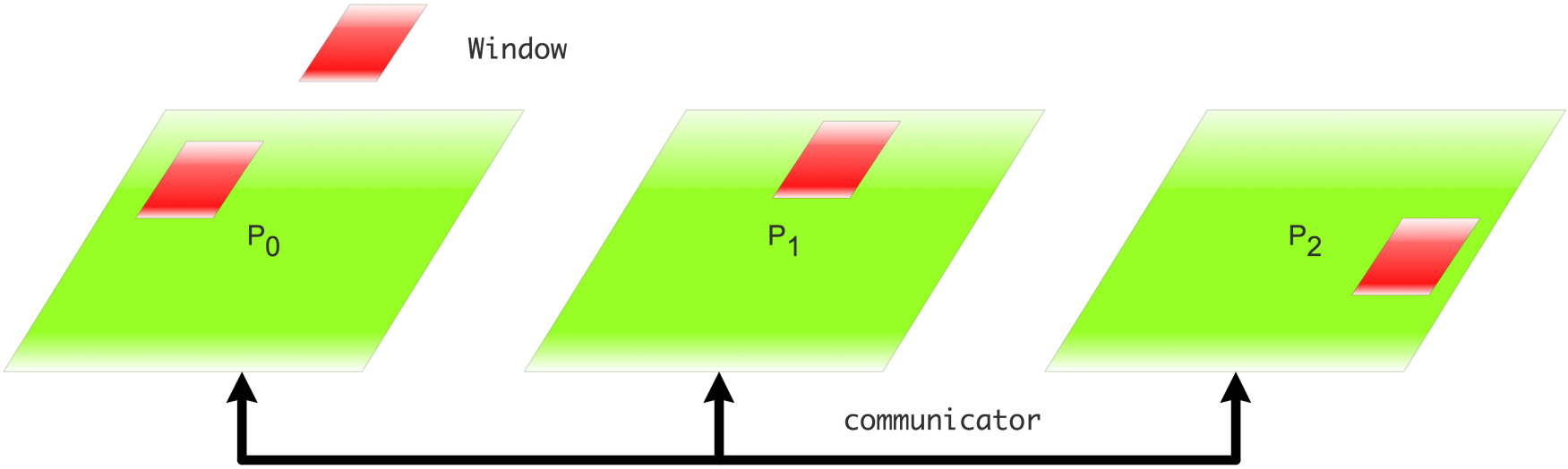
- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbours. Uncertain when this happens.
- Other irregular data structures: distributed hash tables.

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.

```
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 targe
  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-negat
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, target_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: ta
TYPE(MPI_Win), INTENT(IN) :: win
```

Exercise 1 (`rightput`)

Revisit exercise 15 and solve it using `MPI_Put`

Exercise 2 (`randomput`)

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

Exercise (optional) 3 (`randomput`)

Replace `MPI_Win_create` by `MPI_Win_allocate`.

```
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 targe
    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 targe
    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
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
```

Exercise (optional) 4 (`countdown`)

Implement a shared counter:

- One process maintains a counter;
- Iterate: all others at random moments update this counter.
- When the counter is zero, everyone stops iterating.

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

Race conditions in one-sided communication

- One process stores a table of work descriptors, and a pointer to the firt 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.

Atomic operations

Race condition problem in read/write:
result of `MPI_Get` is only known after the fence.

```
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);
}
```

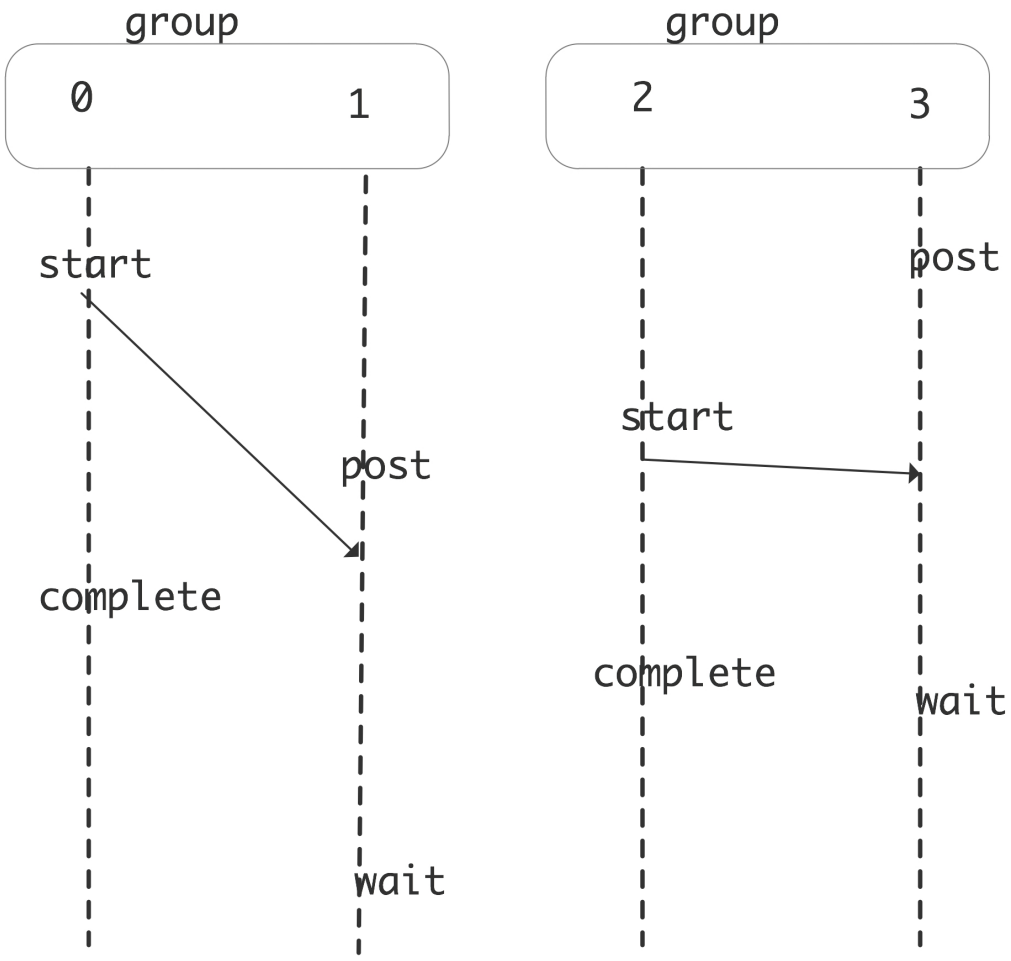
Exercise (optional) 5

Redo exercise 4 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?

A second active synchronization

Use `Post,Wait,Start,Complete` calls



More fine-grained than fences.

Table of Contents

1 Active target synchronization

2 Passive target synchronization

3 More

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)
```

Exercise 6 (`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?)

Table of Contents

1 Active target synchronization

2 Passive target synchronization

3 More

Shared memory interface

- 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`.

Justification

MPI basic concepts suffice for many applications, especially on medium scale. Various advanced concepts let your code deal with unusual scenarios or very large scale runs.

Complicated data

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

Table of Contents

4 Discussion

5 Datatypes

6 Subarray type

7 Packed data

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.

Table of Contents

4 Discussion

5 Datatypes

6 Subarray type

7 Packed data

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 );
```

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)
```

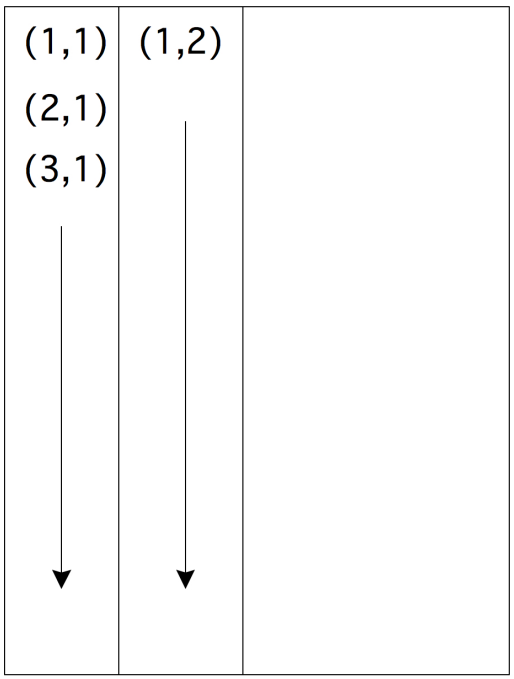
This one is indistinguishable from just sending `count` instances of the `old_type`.

Example: non-contiguous data

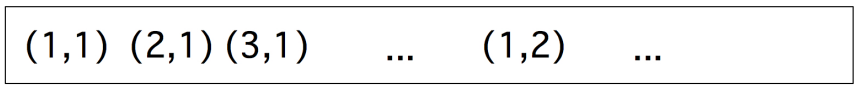
Matrix in column storage:

- Columns are contiguous
- Rows are not contiguous

Logical:

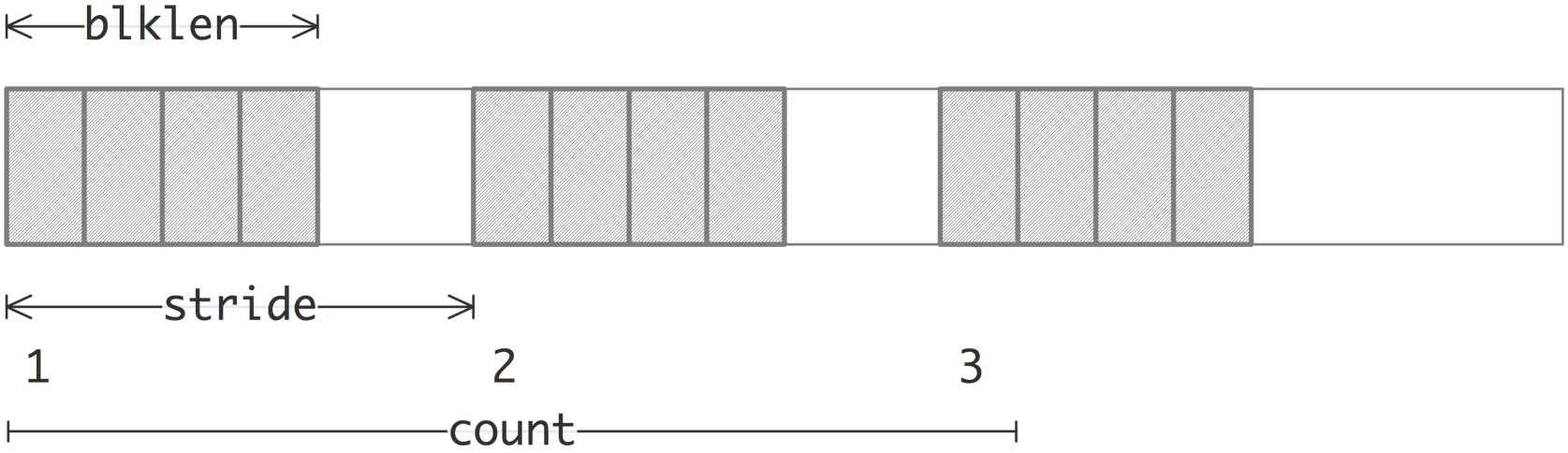


Physical:



Vector type

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



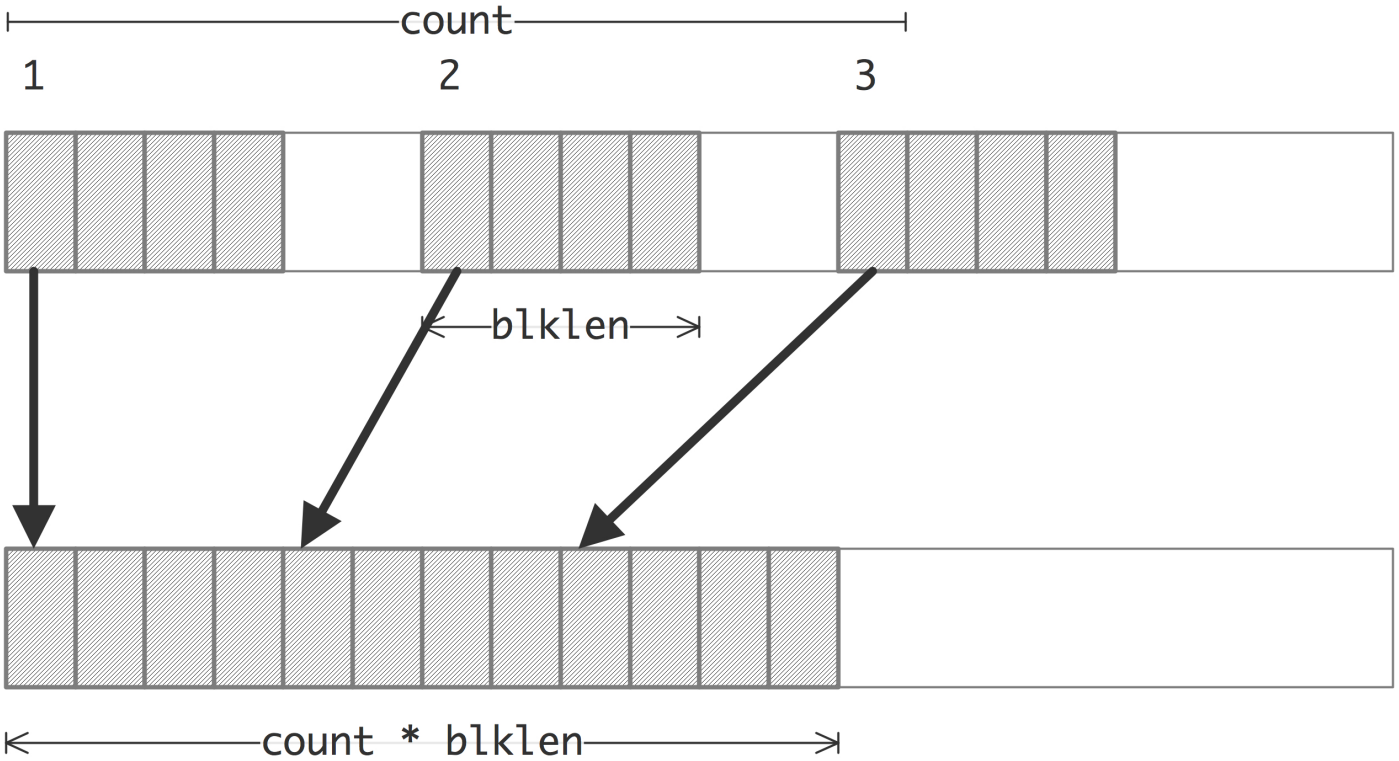
Used to pick a regular subset of elements from an array.

```
// 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

Sender type: vector

receiver type: contiguous or elementary



Receiver has no knowledge of the stride of the sender.

Exercise 7 (`stridesend`)

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

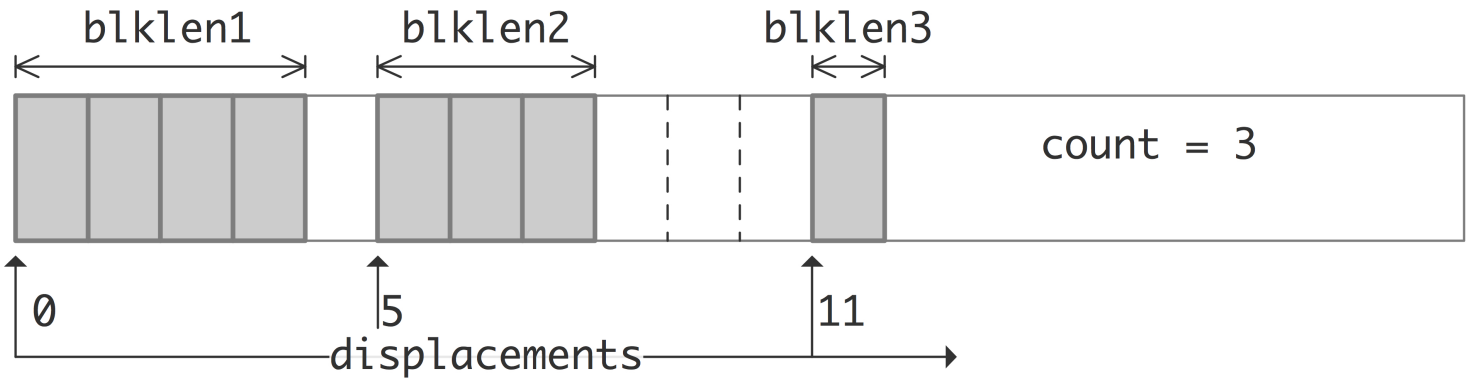
For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?

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

Exercise 8

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

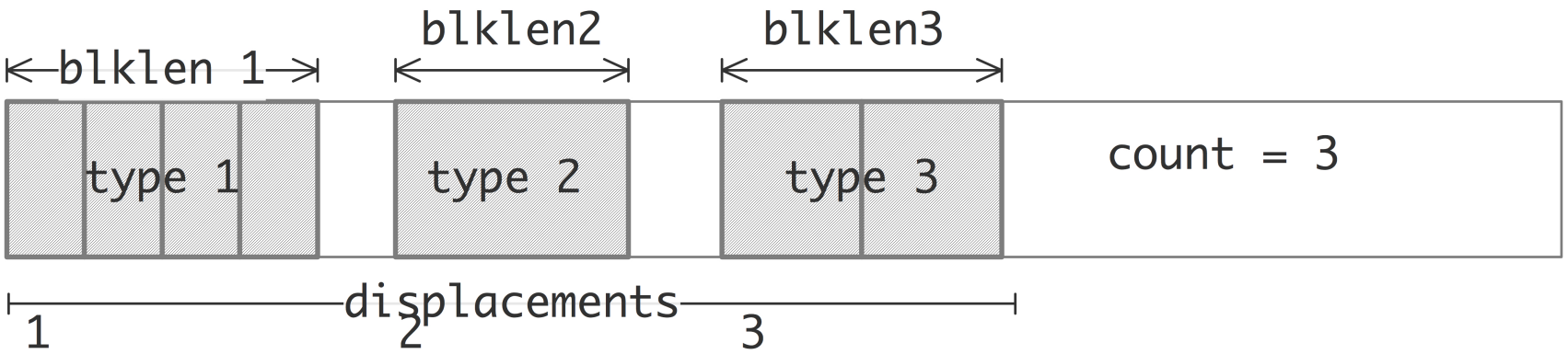



```
int MPI_Type_indexed(
  int count, int blocklens[], int displacements[],
  MPI_Datatype old_type, MPI_Datatype *newtype);
```

Also `hindexed` with byte offsets.

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. . .

Table of Contents

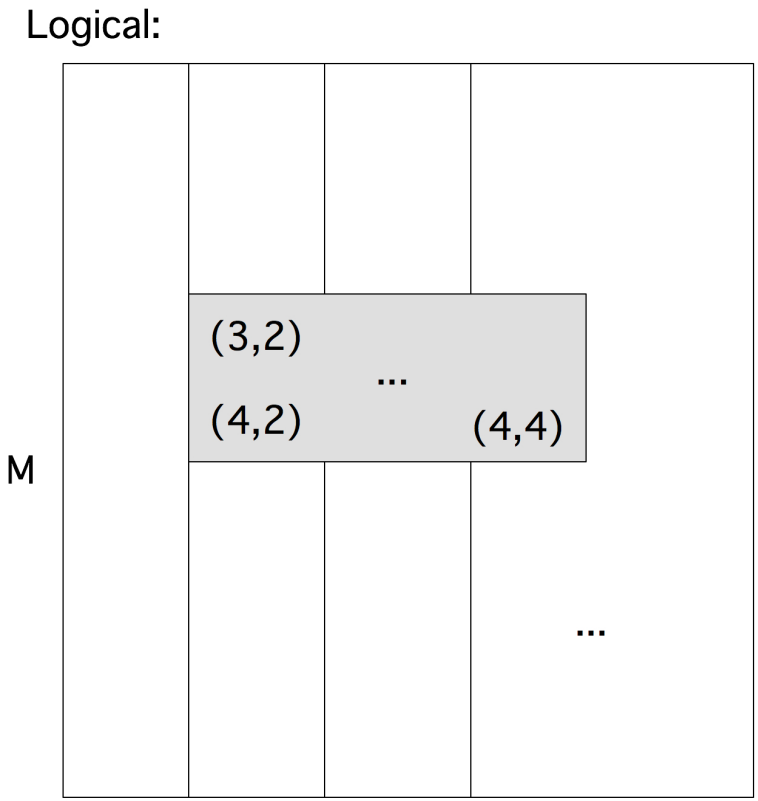
4 Discussion

5 Datatypes

6 Subarray type

7 Packed data

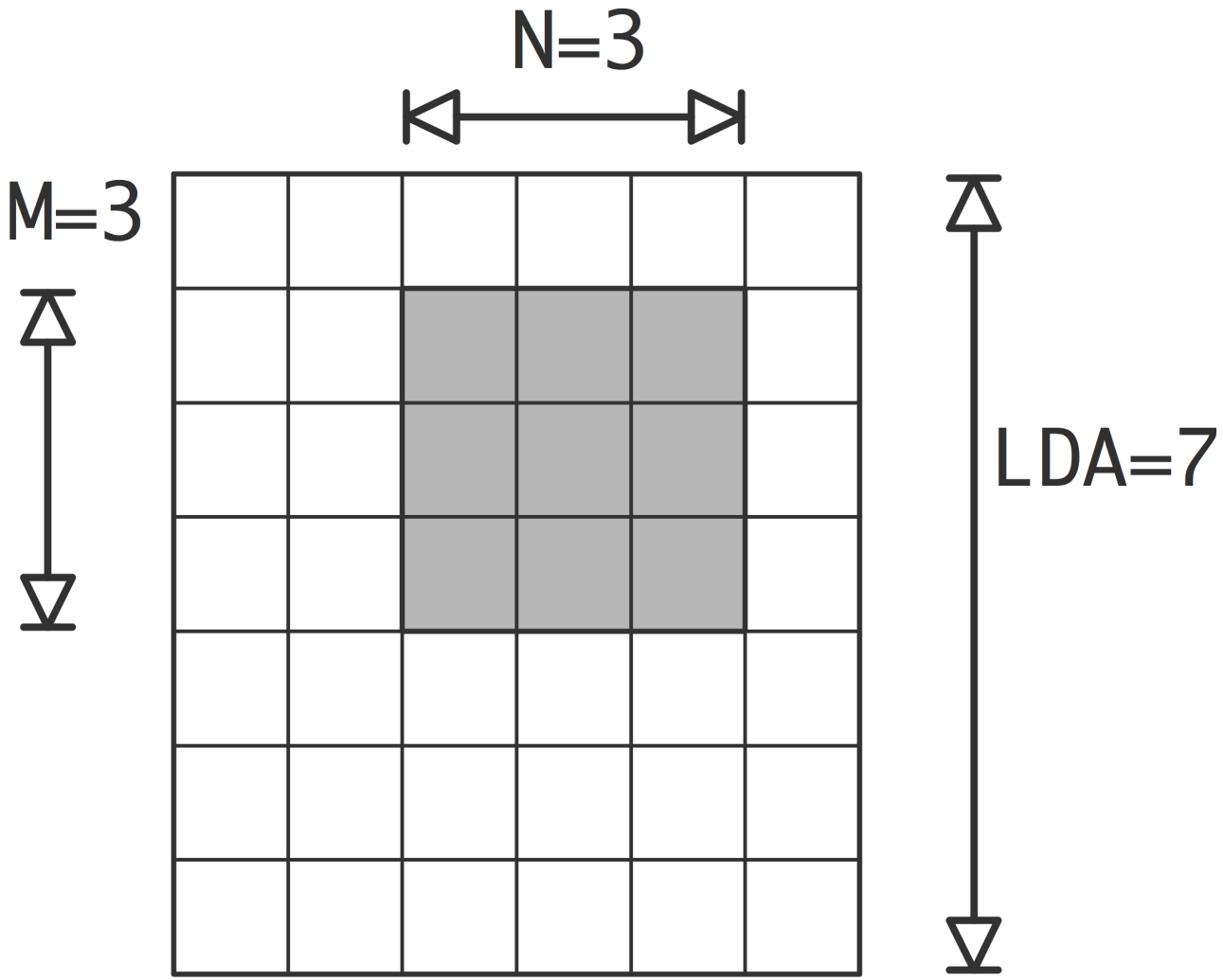
Submatrix storage



- Location of first element
- Stride, blocksize

BLAS/Lapack storage

Three parameter description:



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 9 (`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.

This is one of the rare cases where you should use `ibrun -np 27` and such: 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
CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )
```

Table of Contents

4 Discussion

5 Datatypes

6 Subarray type

7 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
  ASSERT(irecv_value==nsends);
```

Sub-computations

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

```
// commdup_wrong.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

```
// commdup_right.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

```
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 );
```

Exercise 10 (`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 2×3 processor grid you should find:

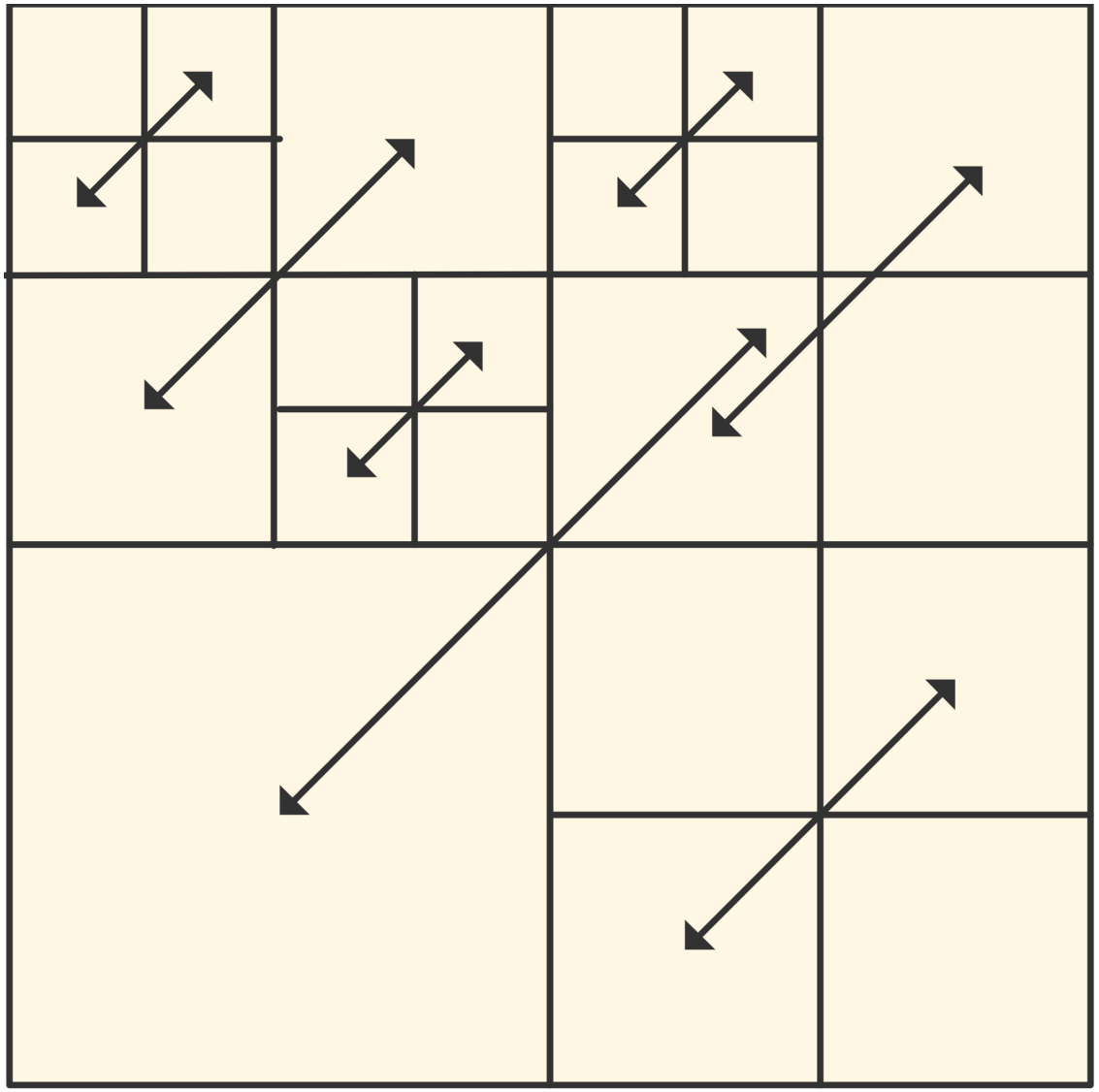
```
Global ranks:  Ranks in row:  Ranks in colum:
  0  1  2       0  1  2        0  0
  3  4  5       0  1  2        1  1
```

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 11

Implement a recursive algorithm for matrix transposition:



More

- Non-disjoint subcommunicators through process groups.
- Intra-communicators and inter-communicators.
- Process topologies: cartesian and graph.

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);
```

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);
```

Exercise 12 (`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 13 (`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 14 (`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.

Process management

Overview

This section discusses processes management; intra communicators.

Commands learned:

- `MPI_Comm_spawn`, `MPI_COMM_UNIVERSE`
- `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.
- `MPI_COMM_UNIVERSE`: 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 time 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);
```

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
- MPI can optimize for locality
- Express operations in terms of topology.

Illustration

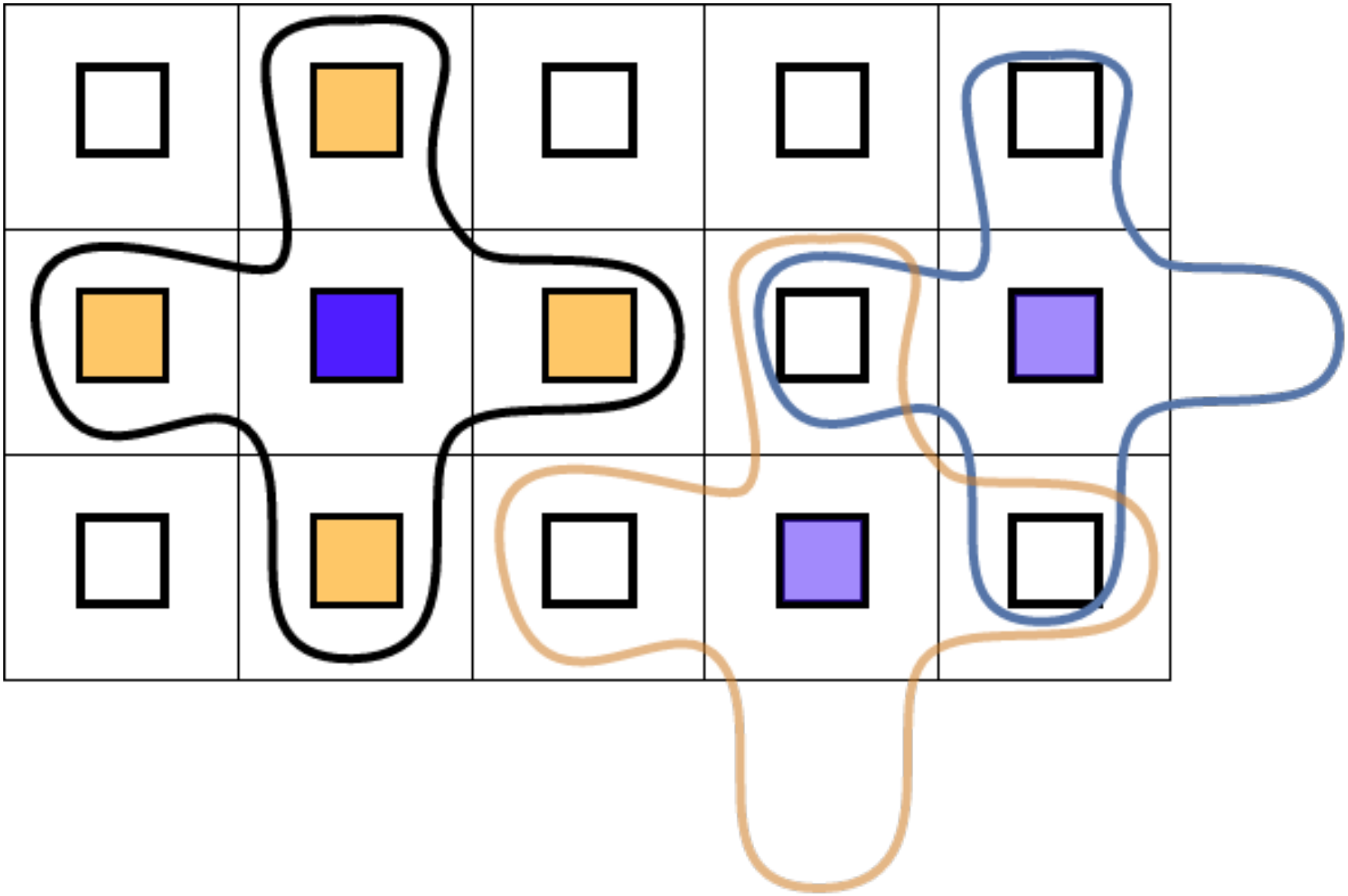


Illustration of a distributed graph topology where each node has four neighbours

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 we
     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`: just set to $\equiv 1$
- `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 `Allgather`, but
the receive buffer has a length `degree`.

Exercise 15 (`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.

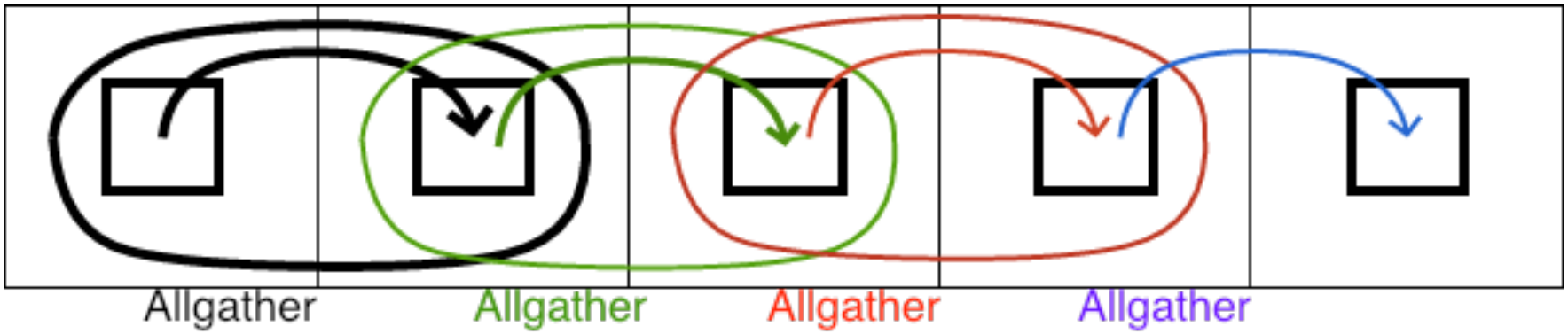
Including boundary conditions for first and last process, that becomes the following program:

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.

Exercise 16 (`rightgraph`)

Revisit exercise 15 and solve it using `MPI_Dist_graph_create`. Use figure 89 for inspiration.

Inspiring picture for the previous exercise



Solving the right-send exercise with neighbourhood collectives