High Performance Computing

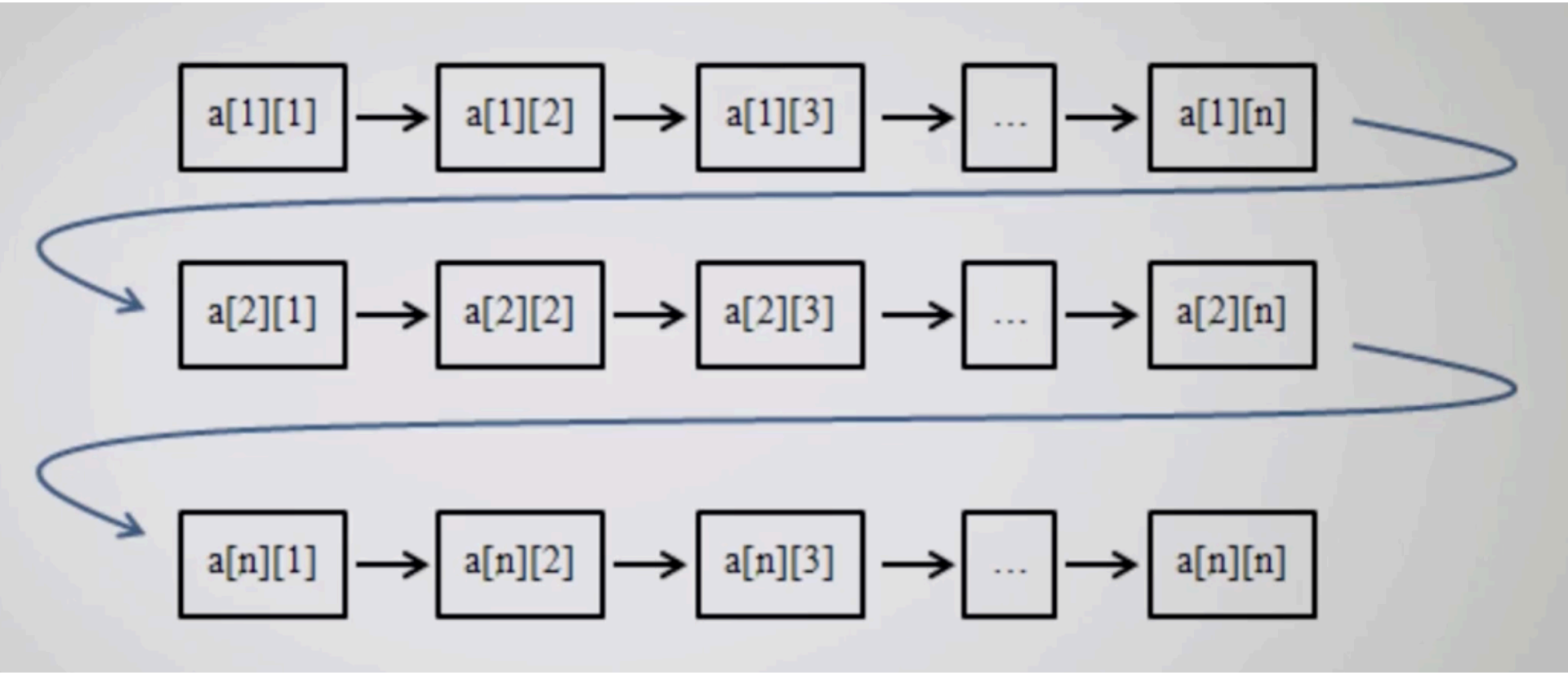
Term 4 2018/2019

Lecture 6

Additional features of MPI

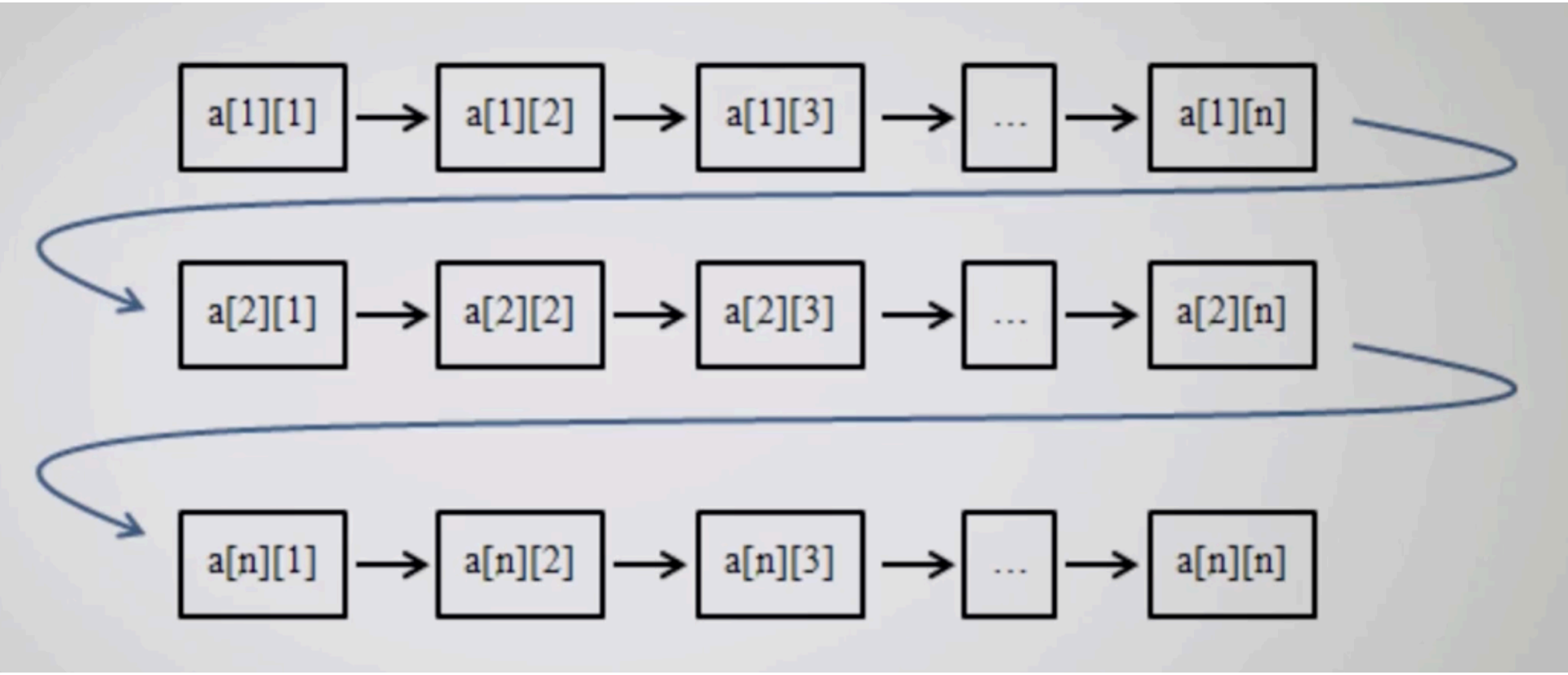
Derived datatypes

We have a C-type 2D array. We want to send rows and columns.



Derived datatypes

We have a C-type 2D array. We want to send rows and columns.

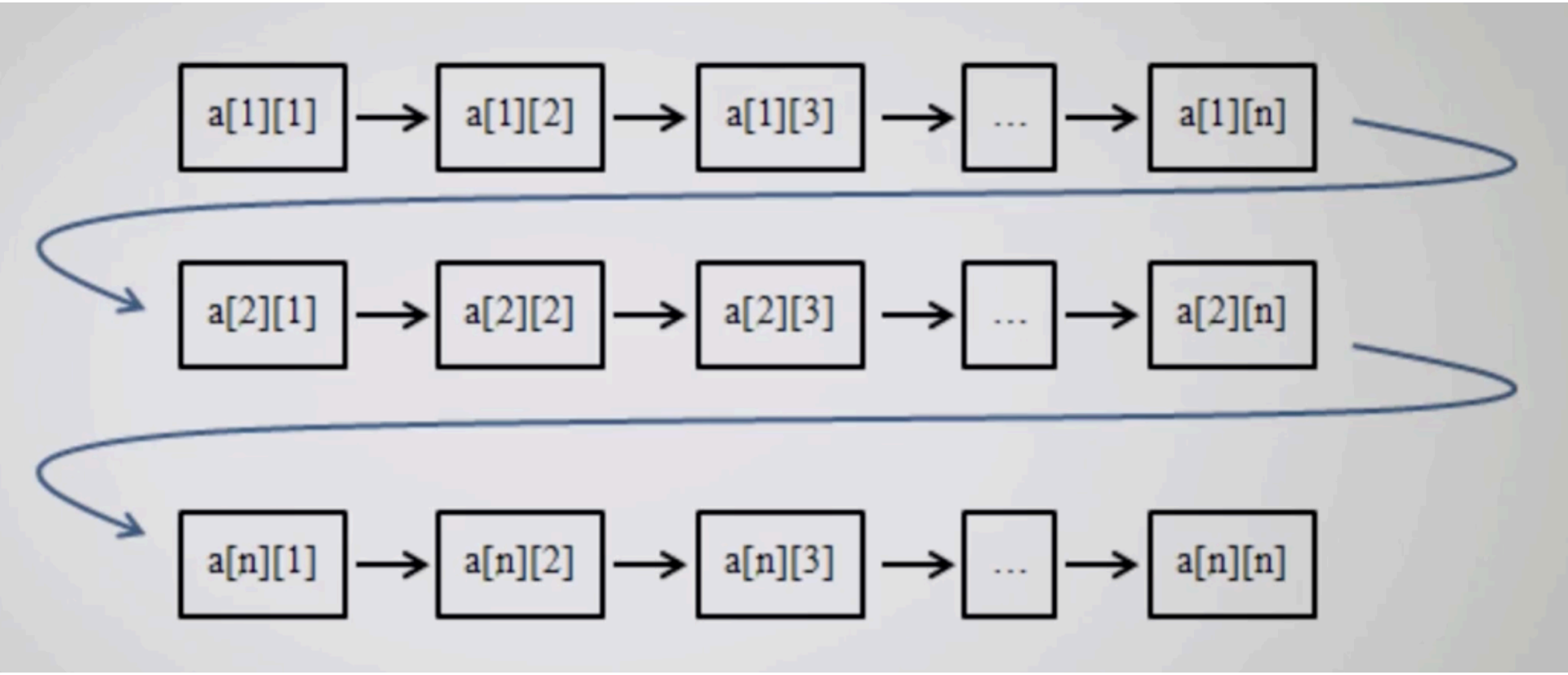


Sending the row is easy:

```
MPI_Send (&a[1][1], n , MPI_INT,
          1, 1, MPI_COMM_WORLD);
```

Derived datatypes

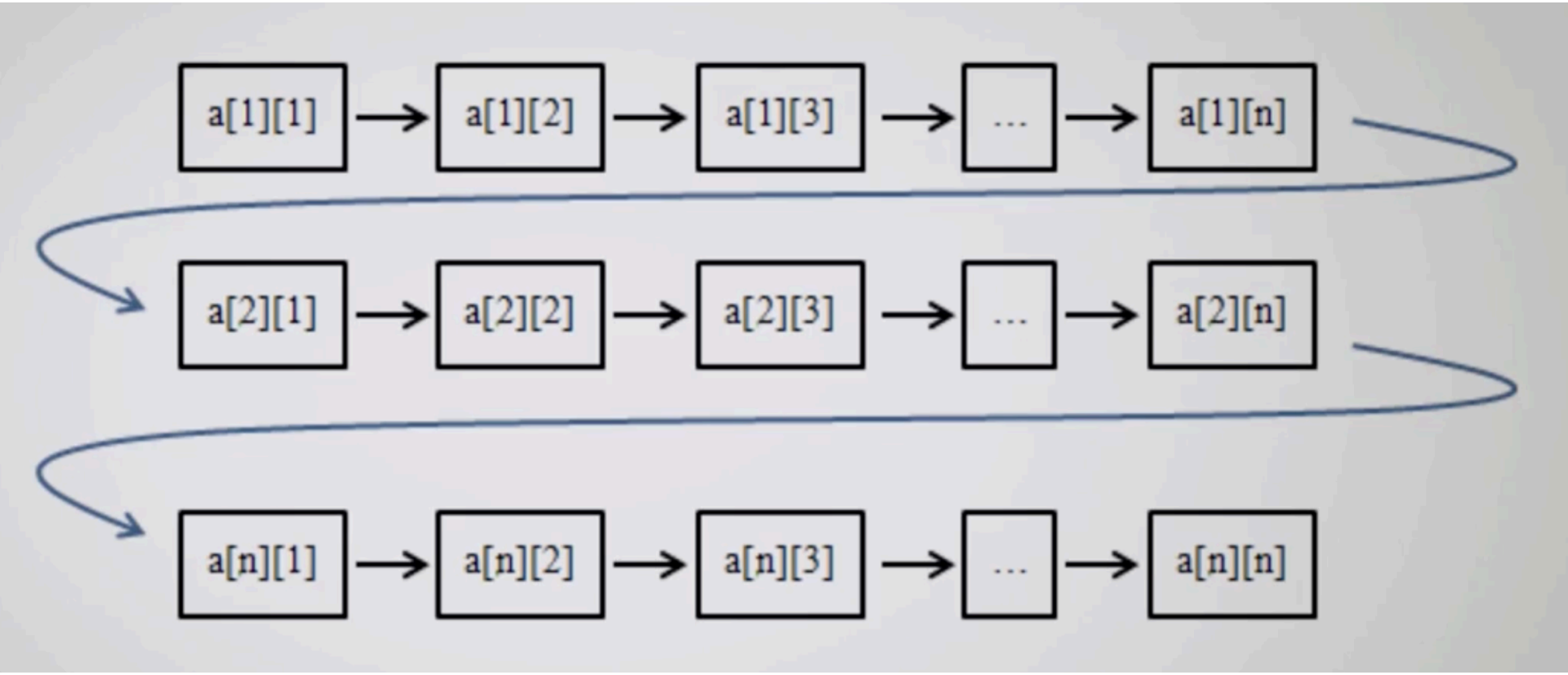
We have a C-type 2D array. We want to send rows and columns.



What about the column?

Derived datatypes

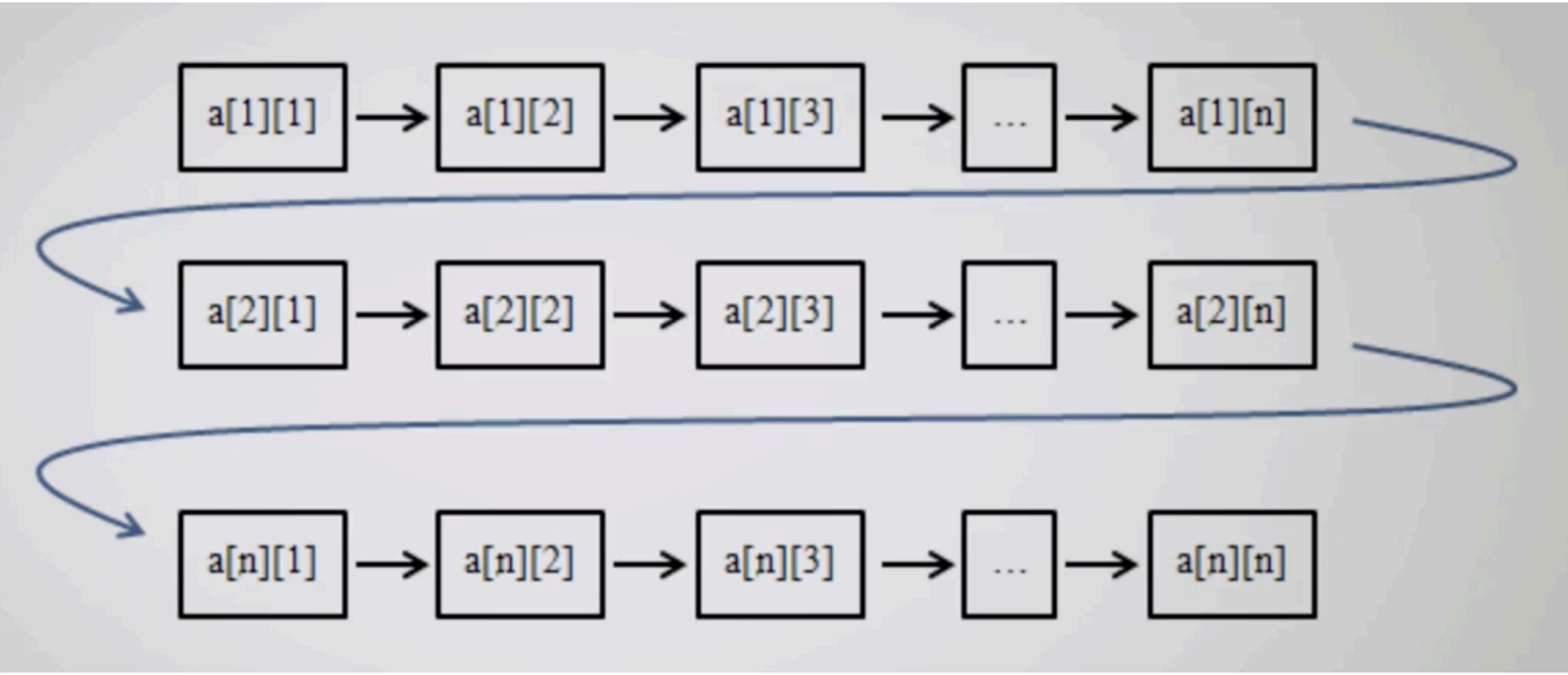
We have a C-type 2D array. We want to send rows and columns.



What about the column?

Derived datatypes

We have a C-type 2D array. We want to send rows and columns.



What about the column?

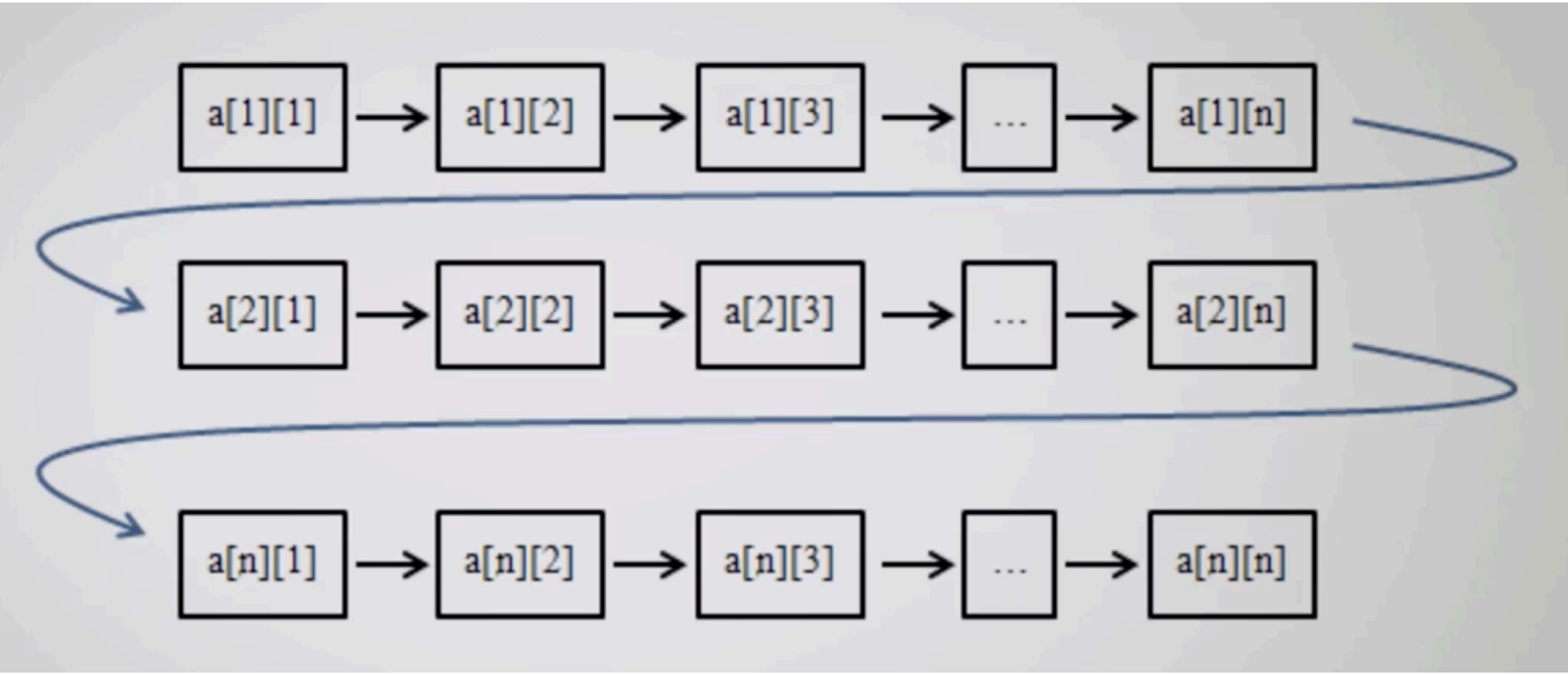
Data send = Establishing The Communication (Latency) + Actual Sending (Packet-wise)

We want to send as much data as possible in one go.

Solution 1: Create a temporary memory buffer with intermediate data

Derived datatypes

We have a C-type 2D array. We want to send rows and columns.



What about the column?

Instead we can specify which data from the memory will be taken (create a stencil).

i.e. „n blocks, each size 1, spaced by n“

Derived datatypes

Standard way of creating the derived datatypes:

1. Creating of the new MPI data type using the constructors:

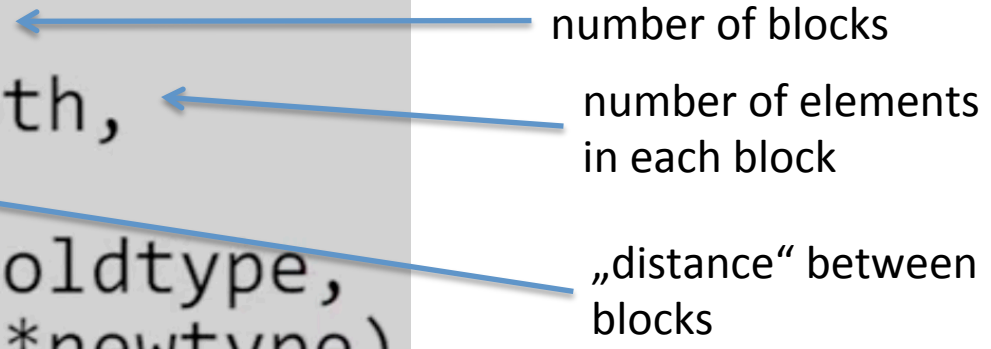
MPI_Type_vector
MPI_Type_indexed
MPI_Type_struct

2. New MPI Type is registered using „MPI_Type_commit“ and can be further used

3. Delete the MPI Type using MPI_Type_free

MPI_Type_vector

```
int MPI_Type_vector(
        int count,
        int blocklength,
        int stride,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```



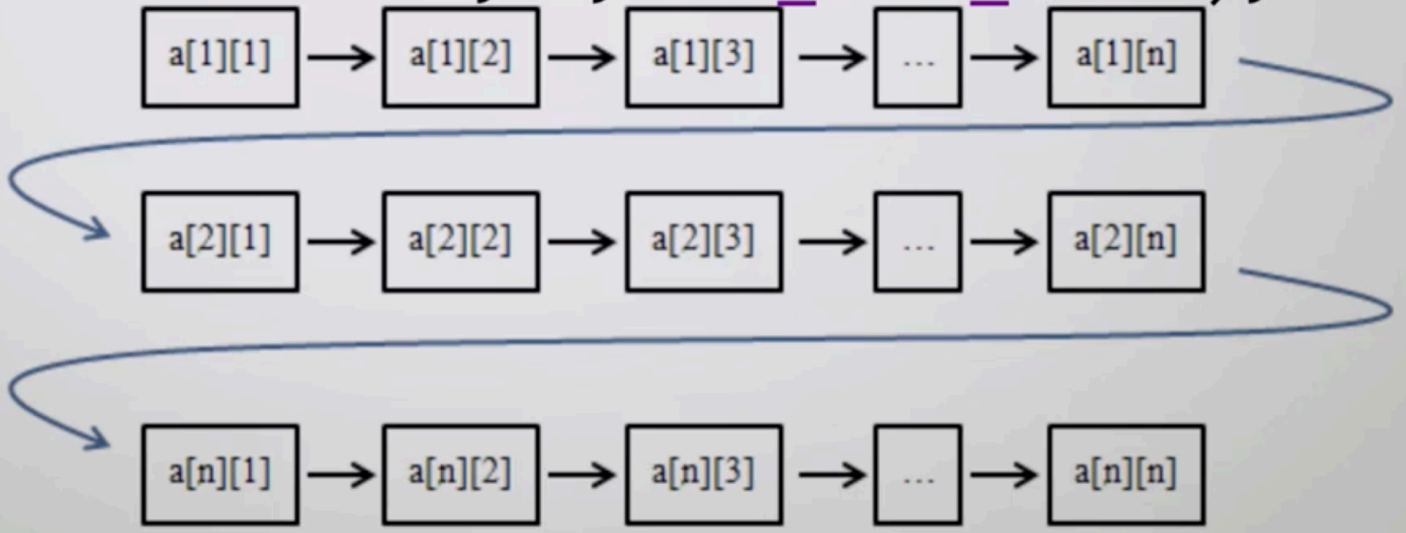
MPI_Type_indexed
MPI_Type_struct

are ideologically the same

MPI_Type_vector example

```
MPI_Datatype ColumnType;
MPI_Type_vector (n, 1, n,
                 &ColumnType,
                 MPI_INT);
MPI_Type_commit (&ColumnType);

MPI_Send (&a[1][1], 1 , ColumnType,
          1, 1, MPI_COMM_WORLD);
```



Coding tasks

1. Compare the time of sending of 1 column of 2D array a) element by element; b) using an intermediate 1D array and c) using MPI_Type_vector

2. Send even rows.

3. Send even columns.

40 minutes.

MPI Groups and Communicators

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

to which new communicator the process will belong
(same color = same communicator)

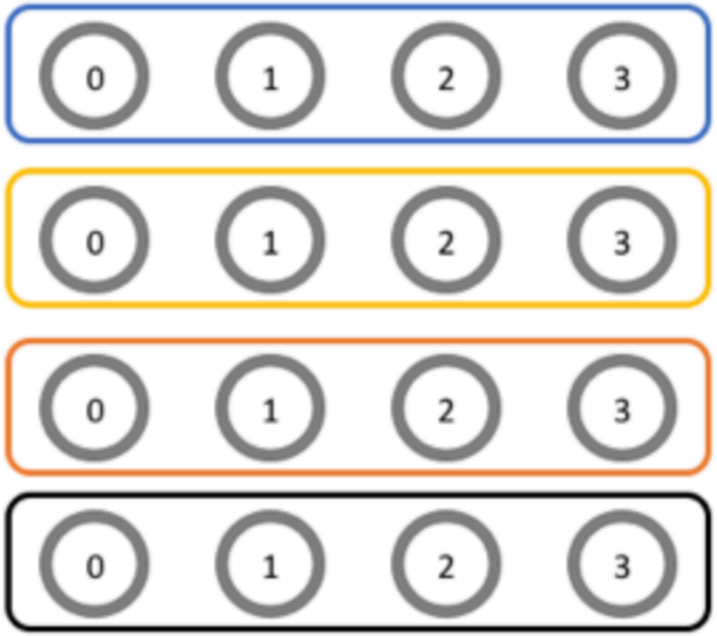
to find out the rank (smallest number becomes root)

MPI Groups and Communicators

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

to which new communicator the process will belong (same color = same communicator)

to find out the rank (smallest number becomes root)



MPI Groups and Communicators

```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
        world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

MPI Groups and Communicators

Other functions for communicators (for basic knowledge, we will not go through them):

`MPI_Comm_dup` creates a duplicate of the original communicator

`MPI_Comm_create` creates a new communicator from a group of processes

MPI Groups and Communicators

Other functions for communicators (for basic knowledge, we will not go through them):

`MPI_Comm_dup` creates a duplicate of the original communicator

`MPI_Comm_create` creates a new communicator from a group of processes

A **group** is an ordered set of process identifiers. The ordering is given by associating with each process identifier a unique **rank** from 0 to group.size − 1

A **communicator** encapsulates all communication between the set of processes.

MPI Groups and Communicators

```
int MPI_Comm_group(
    MPI_Comm comm,
    MPI_Group *group);
```

```
int MPI_Group_incl( MPI_Group group,
                    int n,
                    int *ranks,
                    MPI_Group *newgroup)

int MPI_Group_excl( MPI_Group group,
                    int n,
                    int *ranks,
                    MPI_Group *newgroup)
```

MPI Groups and Communicators

Example:

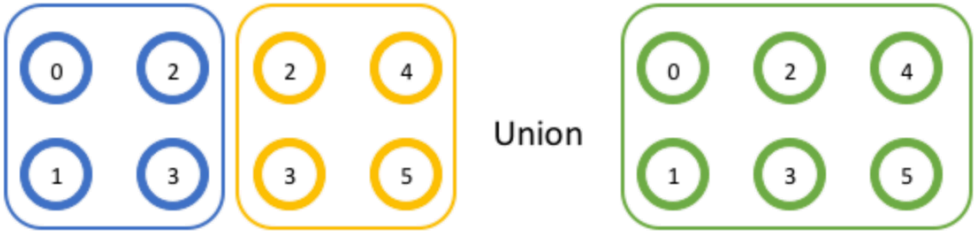
```
MPI_Group ngr;
int ranks[2];
ranks[0]=0;
ranks[1]=2;
ranks[2]=4;
MPI_Group_incl(gr,3,&ranks,&ngr);
```

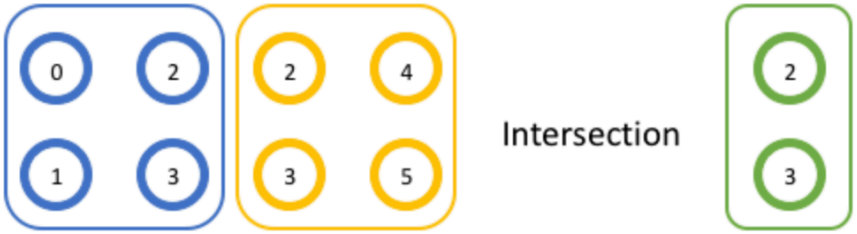
MPI Groups and Communicators

Working with groups:

```
int MPI_Group_union(MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *newgroup)
int MPI_Group_intersection(
                    MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *newgroup)
int MPI_Group_difference(
                    MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *newgroup)
int MPI_Group_free(MPI_Group *group)
```

MPI_GROUP_EMPTY





MPI Groups and Communicators

Creating communicators from groups:

```
MPI_Comm newcomm;

int MPI_Comm_dub(
        MPI_Comm oldcomm,
        MPI_Comm *newcomm);

int MPI_Comm_create(
        MPI_Comm comm,
        MPI_Group group,
        MPI_Comm *newcomm)
int MPI_Comm_free(MPI_Comm comm)
```

One group can belong to several communicators, but all communications only within a given communicator

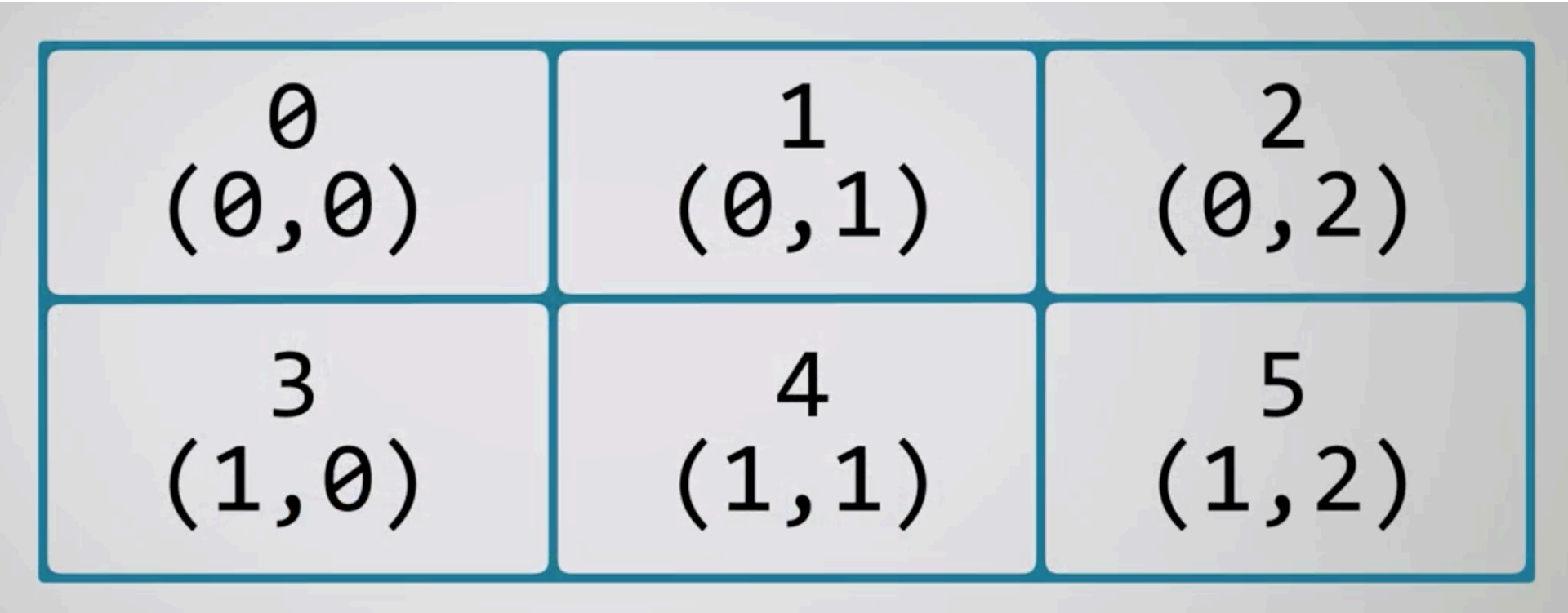
MPI Virtual Topology

Way to address the processes:

So far 1D array (ranks from 0 to world.size)

But this is sometimes inconvenient (imagine you have 1000000 processes to split a 3D matrix)

Cartesian topology

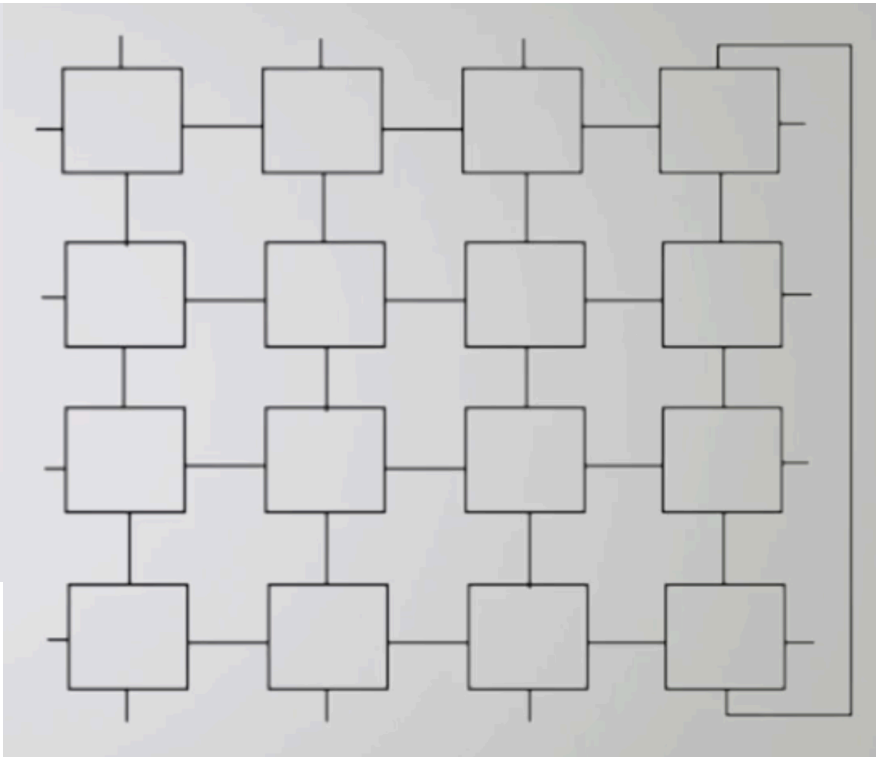


MPI Virtual Topology

```
int MPI_Cart_create(
        MPI_Comm comm_old,
        int ndims,
        int *dims,
        int *periods,
        int reorder,
        MPI_Comm *comm_2D)
```

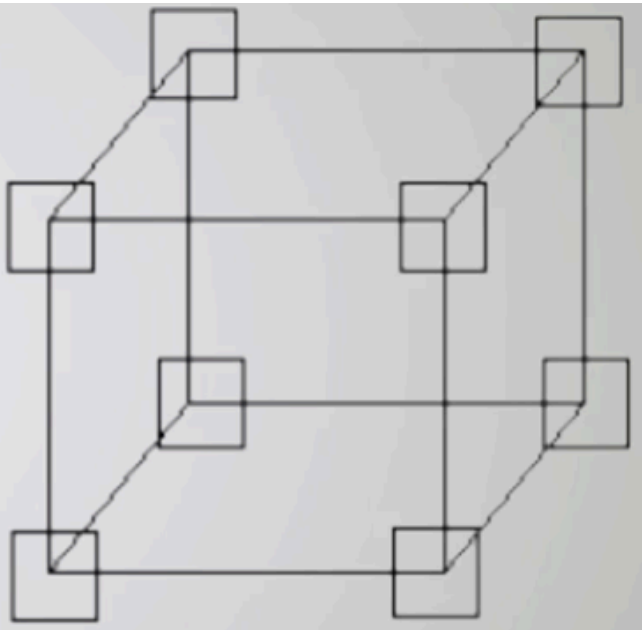
MPI Virtual Topology

```
MPI_Comm comm_2D;
int ndim = 2;
int dims[2], periods[2];
dims[0] = 4;
dims[1] = 4;
periods[0] = 1;
periods[1] = 1;
MPI_Cart_create( MPI_COMM_WORLD,
     ndim, &dims, &periods,
     0,&comm_2D);
```



MPI Virtual Topology

```
MPI_Comm comm_3D;
int ndim = 3;
int dims[3], periods[3];
dims[0] = 2;
dims[1] = 2;
dims[2] = 2;
periods[0] = 0;
periods[1] = 0;
periods[2] = 0;
MPI_Cart_create( MPI_COMM_WORLD,
     ndim, &dims, &periods,
     0, &comm_3D);
```



Ways of using I/O in parallel programs

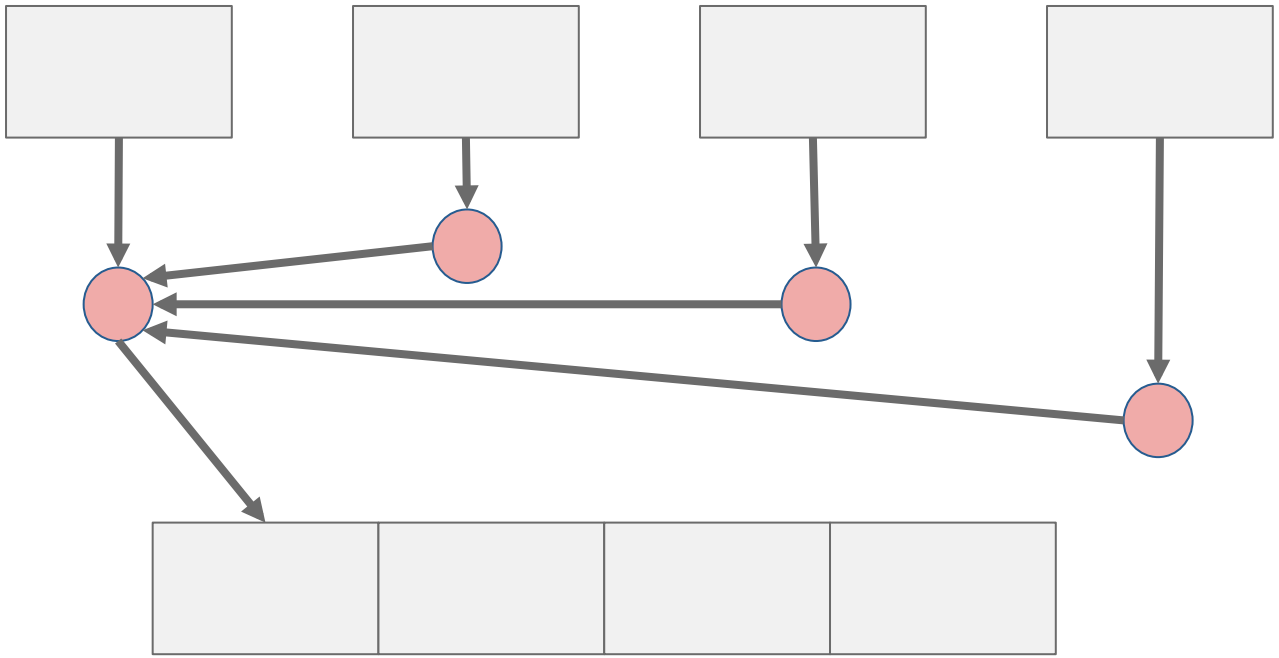
Sequential I/O: All processes send data to rank 0, and 0 writes it to the file

Pros:

- Easy to write

Cons:

- Poor performance and scalability



Ways of using I/O in parallel programs

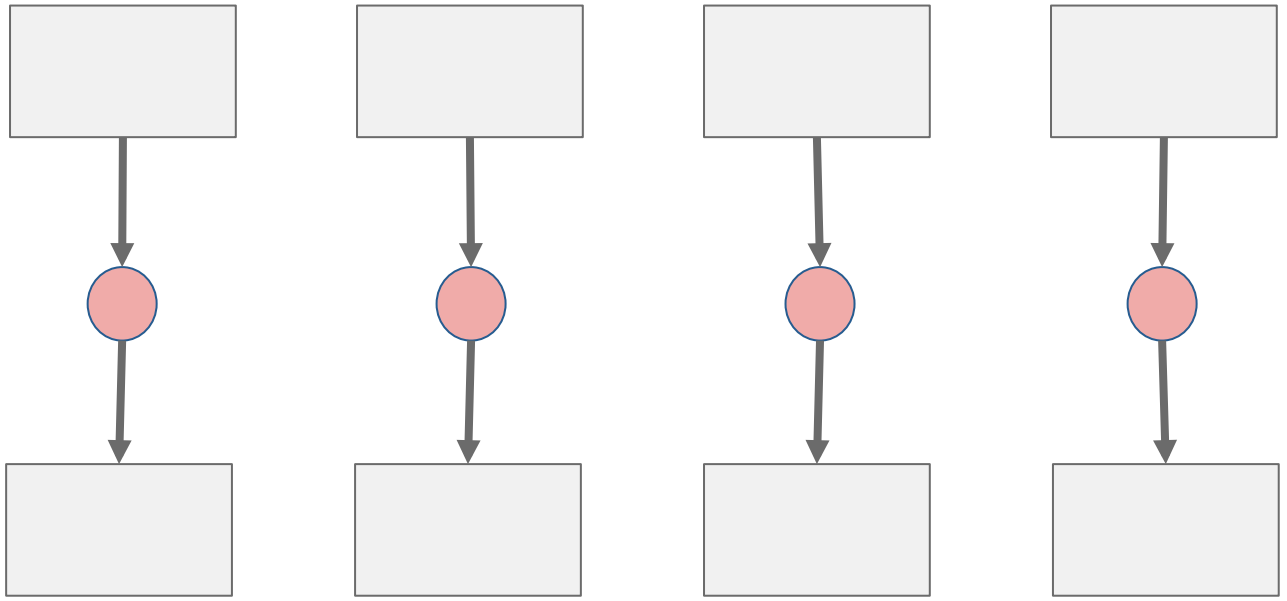
Independent I/O: All processes write to different files

Pros:

- High performance

Cons:

- Lots of small files to manage
- Difficult to read back data from different number of processes

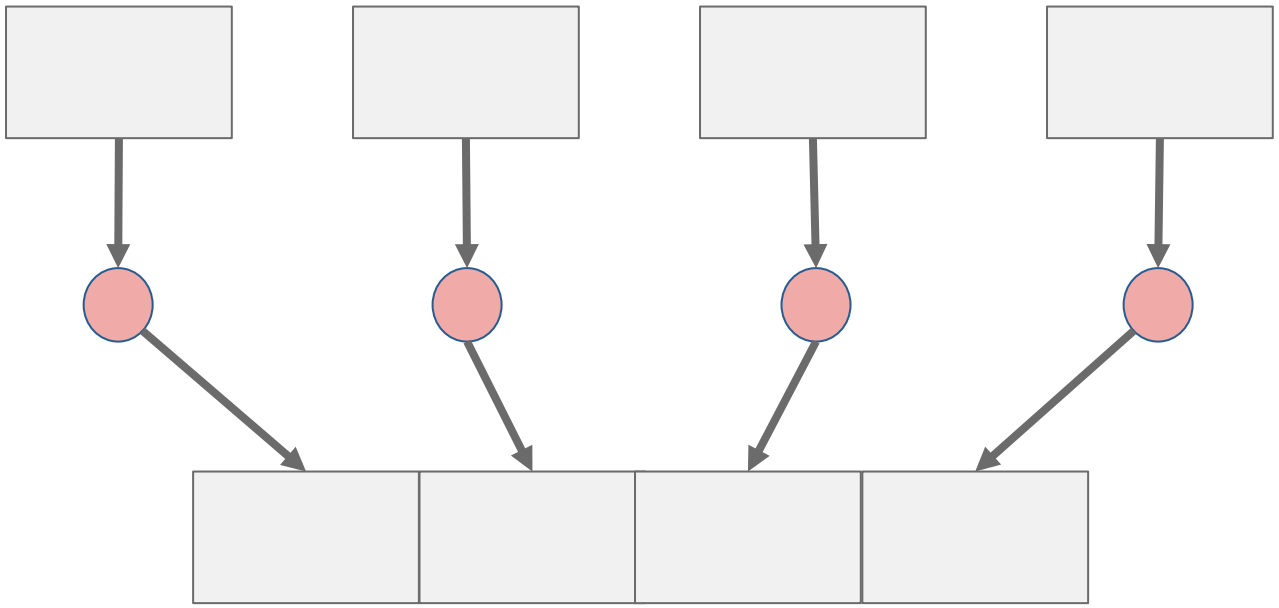


Ways of using I/O in parallel programs

MPI-Parallel I/O: All processes write to the same file

Cons:

- Probably the best way to organise high-performance I/O even for noncontiguous data



MPI-Parallel I/O using individual file pointers

```
MPI_File fh;
MPI_Status status;
int rank;
int buffer[SIZE];
char filename[10];
…

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_File_open(MPI_COMM_WORLD, filename,
        MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

MPI_File_seek(fh, rank * SIZE * sizeof(int), MPI_SEEK_SET);

MPI_File_read(fh, buffer, SIZE, MPI_INT, &status);

MPI_File_close(&fh);
```

MPI-Parallel I/O using single file pointer

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype filetype, ROMIO_CONST char *datarep, MPI_Info info)
```

```
#include <mpi.h>
...
   MPI_File fh;
   MPI_Status status;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
   MPI_File_open(MPI_COMM_WORLD, filename,
                        MPI_MODE_CREATE | MPI_MODE_WRONLY,
                        MPI_INFO_NULL, &fh);

   MPI_File_set_view(fh, rank * SIZE * sizeof(int),
                              MPI_INT, MPI_INT, "native", MPI_INFO_NULL);

   MPI_File_write(fh, buf, SIZE, MPI_INT, &status);

   MPI_File_close(&fh);
...
```

Task

Each processor generates an array of 10 elements with its rank, i.e.

rank0: [0,0,0,0,0,0,0,0,0,0]
rank1: [1,1,1,1,1,1,1,1,1,1]

write a code that will write those arrays to the file using parallel MPI I/O. The arrays should appear in the file in order.