TRACK II

DISTRIBUTED-MEMORY PARALLEL PROGRAMMING

2025 IHPCSS

LISBON, PORTUGAL

Andrew Kirby

Instructor

University of Wyoming, USA

John Cazes

Hands-On Helper

Texas Advanced Computing Center, USA

Erik Draeger

Hands-On Helper

LLNL, USA

Sebastian Kuckuk

Hands-On Helper

NHR@FAU, Germany

Erwin Laure

Session Chair, Day 2

Max Planck Computing & Data Center, German

Ramses van Zon

Session Chair, Day 2

SciNet, University of Toronto, Canada

Matthais Brust

Evaluator

University of Luxembourg

Julie Wernert

Evaluator

Indiana University, USA

LECTURE CONTENT INSPIRATION

Ludovic Capelli - Pawsey
David Henty - EPCC

EPCC Archer Tutorials
SC17 MPI Tutorial

https://web.cels.anl.gov/~thakur/sc17-mpi-tutorial/

TRACK OVERVIEW

- **Section I** – Intro MPI Review
- **Section II** - Derived Data Types
- **Section III** – Virtual Topologies
- **Section IV** – Neighborhood Collectives
- **Section V** – Hybrid Programming MPI+X

TRACK AIMS

- A *Practical* Course
  - Understand the message-passing model for parallel programming
  - Write parallel programs in C, Fortran, or Python using the MPI library
- Though *Interactive Hands-On* Examples

TIMETABLE

Designed as 30-minute blocks

- 20 minutes of teaching
- 10 minutes of practice

DISTRIBUTED MEMORY = ?

Exit

Go to **wooclap.com** and use the code **NPCDTK**

How to participate?

Copy participation link

1. Go to wooclap.com
2. Enter the event code in the top banner

Event code
NPCDTK

wooclap

Questions - / 1 + Messages 100%

0

BRIDGES SETUP (I)

Connecting

```
ssh <username>@bridges2.psc.edu
```

Clone Repo (exercises + slides)

```
git clone https://github.com/ackirby88/IHPCSS2025-mpi
```

Interactive Reservations

- `./interactive8-day1.sh`
- `./interactive8-day2.sh`

BRIDGES SETUP (II)

```
# (1) First time you log into bridges2
[user@bridges2-login01 ~]$ module load python anaconda3 cuda intel-mpi
[user@bridges2-login01 ~]$ module list
[user@bridges2-login01 ~]$ module save mpi-track
```

```
# (2) Next time you log into bridges2
[~]$ module restore mpi-track
```

```
[~]$ module list
    Currently Loaded Modules:
      1) allocations/1.0
      2) psc.allocations.user/1.0
      3) intel-mpi/2021.10.0
      4) python/3.8.6
      5) anaconda3/2024.10-1
      6) cuda/12.6.1
```

BRIDGES - RUNNING ON THE LOGIN NODE (C/F90)

- Go to the repository
  - `cd <repo>/exercises/Exercise.0-Preamble/<lang>/`
- Compile the code (c/f90)
  - `make`
- Run the executable
  - `./mpi_hello_world`

Repo: https://github.com/ackirby88/IHPCSS2025-mpi

BRIDGES - RUNNING ON THE LOGIN NODE (PYTHON)

- Go to the repository
  - `cd <repo>/exercises/Exercise.0-Preamble/python/`
- Python Environment Setup
  - `source 0_source_me_first.sh`
- Run the executable
  - `python 1_hello_mpi.py`

Repo: https://github.com/ackirby88/IHPCSS2025-mpi

DISTRIBUTED-MEMORY PARALLEL COMPUTING

MOTIVATION

CLOCK FREQUENCY

- 1998 First 0.1GHz CPU, Pentium II Xeon 400.
- 1999 First 1GHz CPU, AMD Athlon.
- 2001 First 2GHz CPU, Intel Pentium 4.
- 2002 First 3GHz CPU, Intel Pentium 4.
- 2012 First 4GHz CPU, AMD FX-4170.
- 2013 First 5GHz CPU, AMD FX-9590.
- 2023 First 6GHz CPU, Intel Core i9-13900KS.

NEED FOR PARALLEL COMPUTING

- Can only increase clock frequency so much → physics!
  - Power increases approximately $P \propto C \cdot V^2 \cdot f$
    - C = capacitance
    - V = voltage
    - f = frequency
      - Increasing Voltage increases heat!
  - Signal Propagation Delay (speed of light)
  - Quantum Effects and Material Limits

✓ Instead of trying to increase clock frequency, why not use multiple cores?

DAWN OF PARALLEL COMPUTING

- Early Parallel computing traces its roots back to the 1940s and 1950s
  - **1950s-60s**: Concepts of vector and array processing emerged
  - **1970s**: Parallel architectures like
    SIMD (Single Instruction, Multiple Data) -- ILLIAC IV (1966)
    MIMD (Multiple Instruction, Multiple Data) -- Gamma 60 (1957)
- 1980s: Rise of Distributed-Memory Systems
  - Early Message Passing Libraries (vendor-specific)
    - e.g., PVM, Express, and NX
- 1990s: The Emergence of the Message-Passing Interface (MPI)
  - Portable, standardized interface for message passing

MESSAGE PASSING INTERFACE (MPI) HISTORY

- 1992 — MPI Forum formed
- 1994 — MPI-1
  - pt2pt, collective operations, communicators
- 1997-2003 — MPI-2
  - One-sided communication (RMA), parallel I/O
- 2012 — MPI-3
  - Improved RMA, non-blocking collectives, enhanced MPI+X
- 2021 — MPI-4
  - Better support for GPU-aware MPI
  - Partitioned communication for better overlap of comm & computation
- 2025 — MPI-5
  - Standard Application Binary Interface (ABI)
  - And more!

MPI

- The MPI Library is the most important piece of software in parallel programming.
- All of the world's largest supercomputers are programmed using MPI.
- Writing parallel programs using MPI is fun!

SECTION I

INTRO MPI REVIEW

PROGRAMMING MODELS

Serial Programming

Concepts

Arrays
Subroutines
Control Flow
Variables
Human-Readable
Object-Oriented

Languages

Python
Julia
C/C++
Fortran
struct
if/then/else

Implementations

gcc
icc
gfortran
javac
crayftn
nvcc

Message-Passing Parallel Programming

Concepts

Processes
Send/Recv
SPMD
Collectives
Groups
Communicators

Libraries

MPI
MPI_Init()

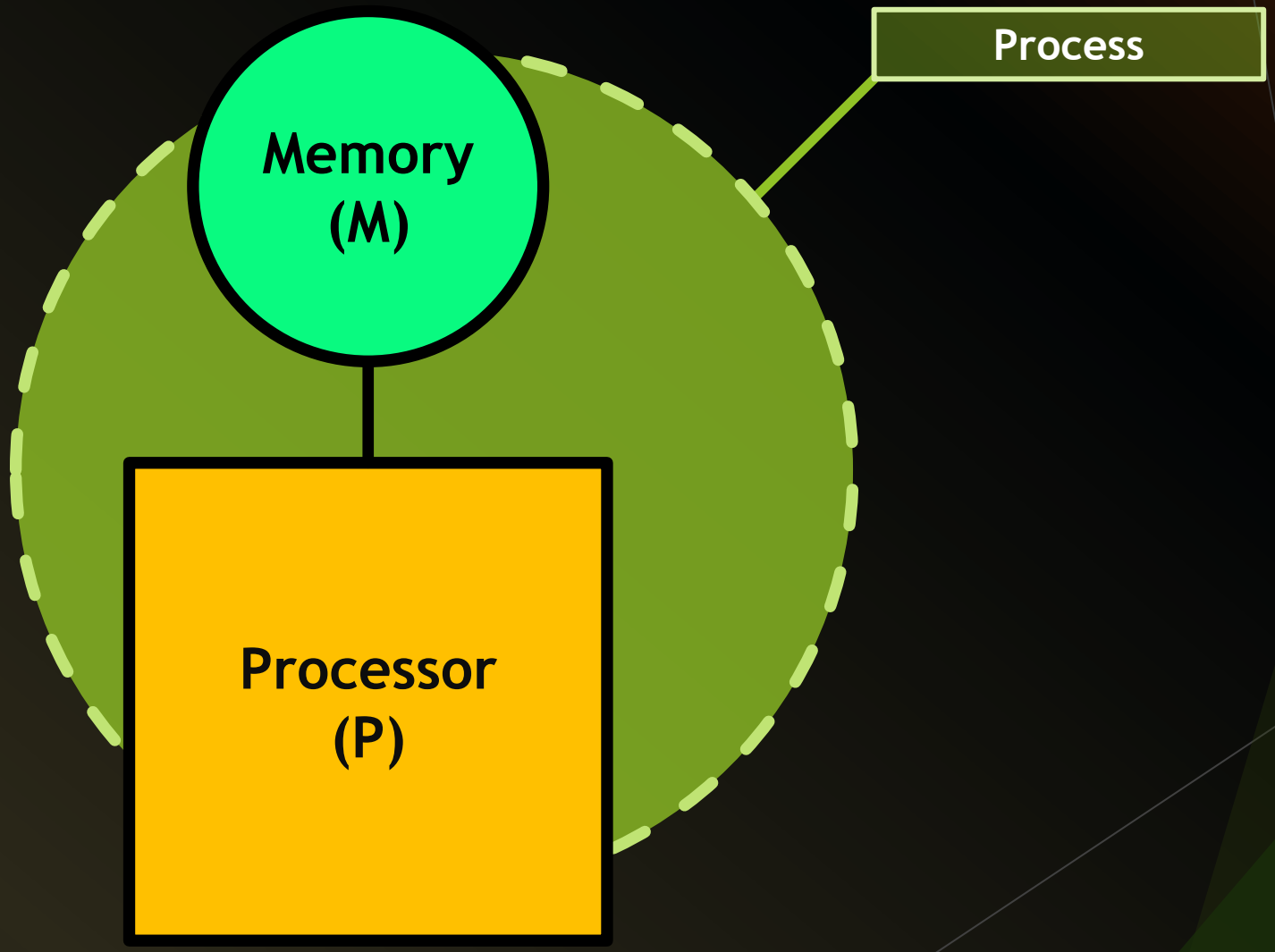
Implementations

Intel MPI
MPICH2
OpenMPI
Cray MPI
IBM MPI

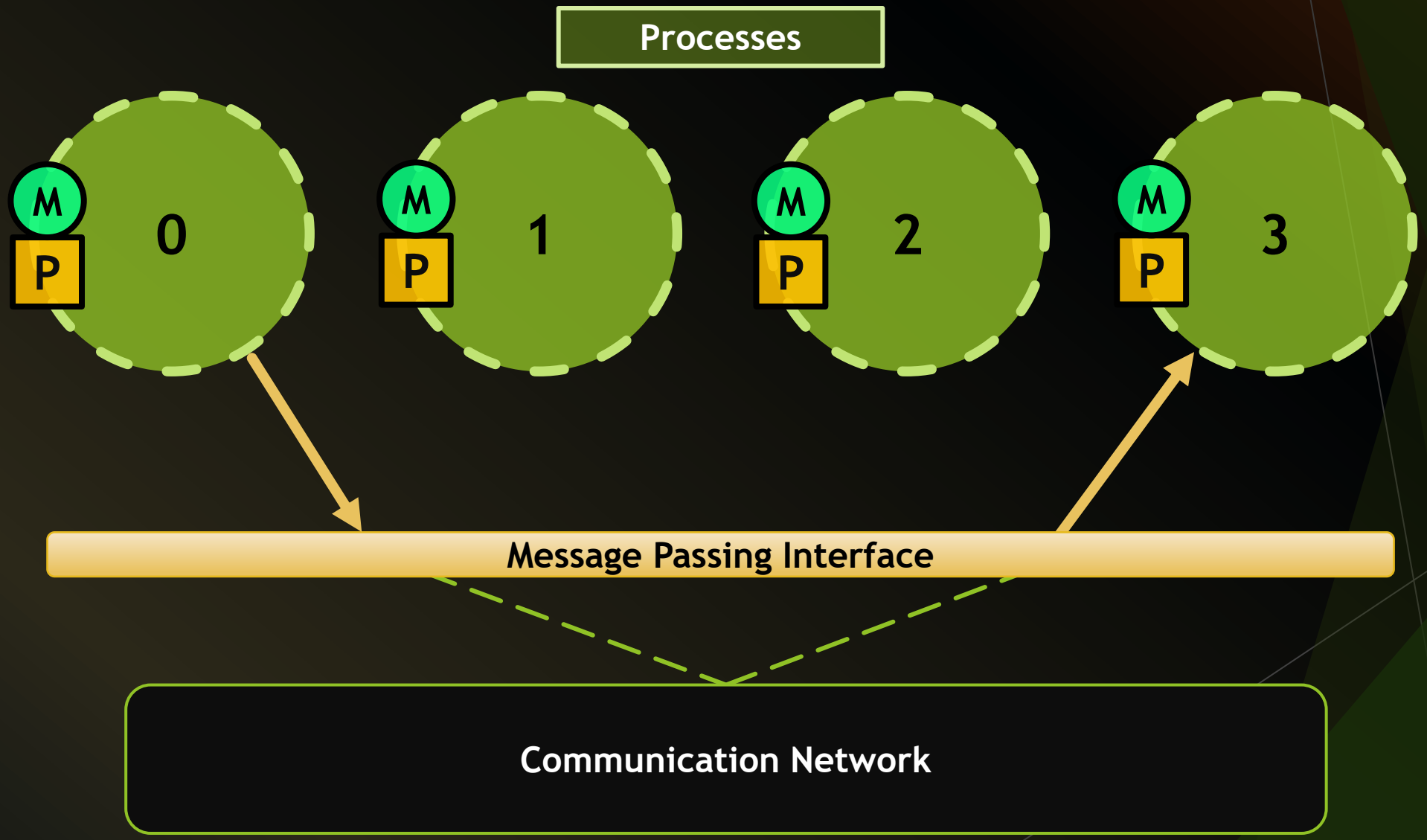
MESSAGE PASSING MODEL

- **Processes** — Message Passing based on notion of processes.
  i.e., an instance of the program,
  together with the program's data
- **Parallelism** — Many processes co-operating on same task.
- **Private Data** — Each process has access only to its **OWN** data.
- **Communication** — Processes sending & receiving messages.

SEQUENTIAL PARADIGM



PARALLEL PARADIGM

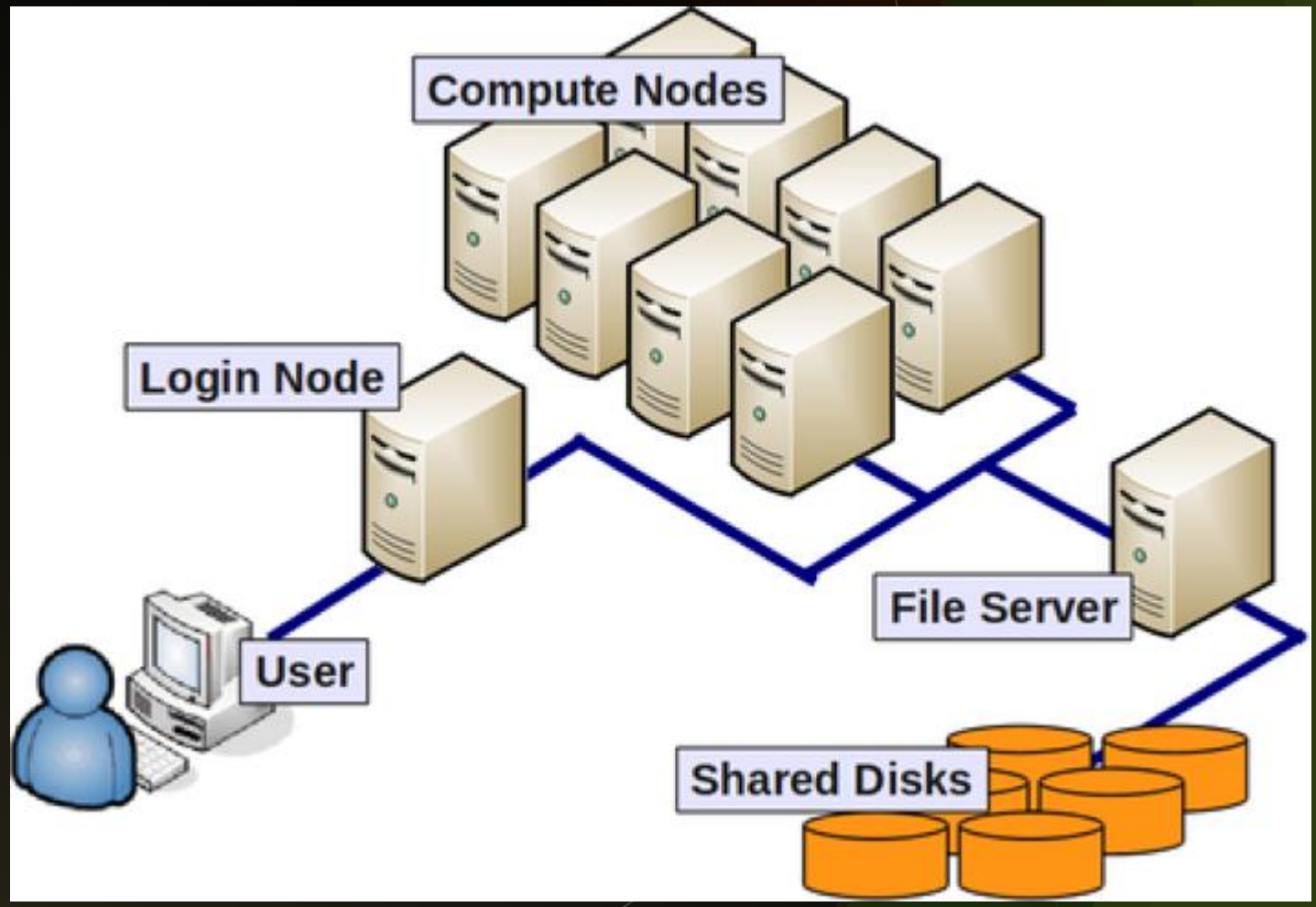


CLUSTER OF WORKSTATIONS

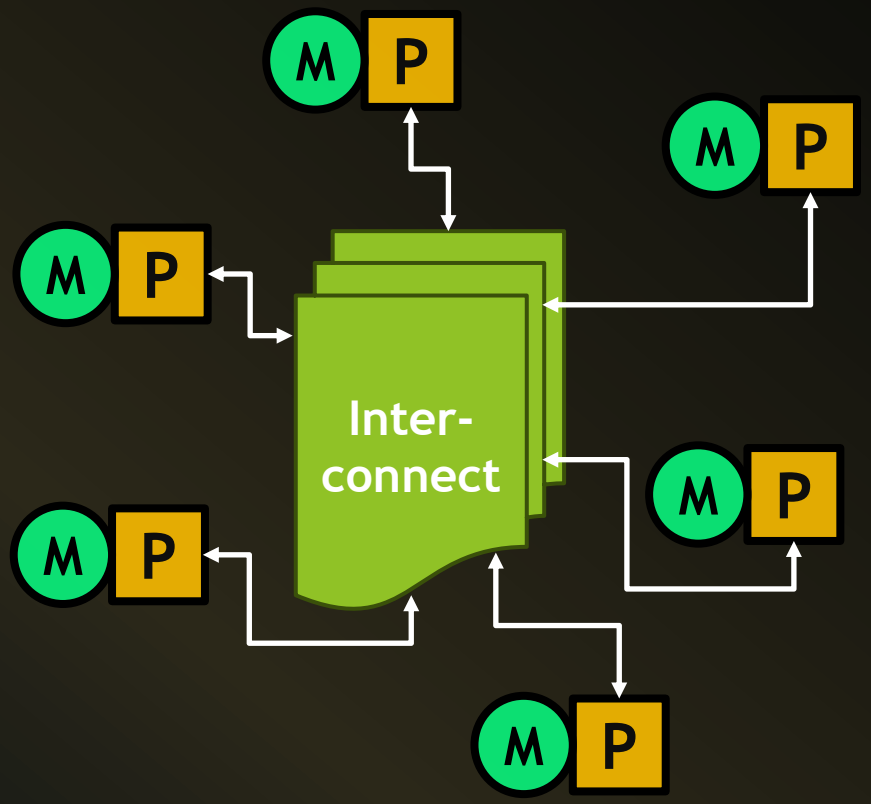
- Aim to run a *single* computation *across all* workstations.

“SUPERCOMPUTER” ARCHITECTURE

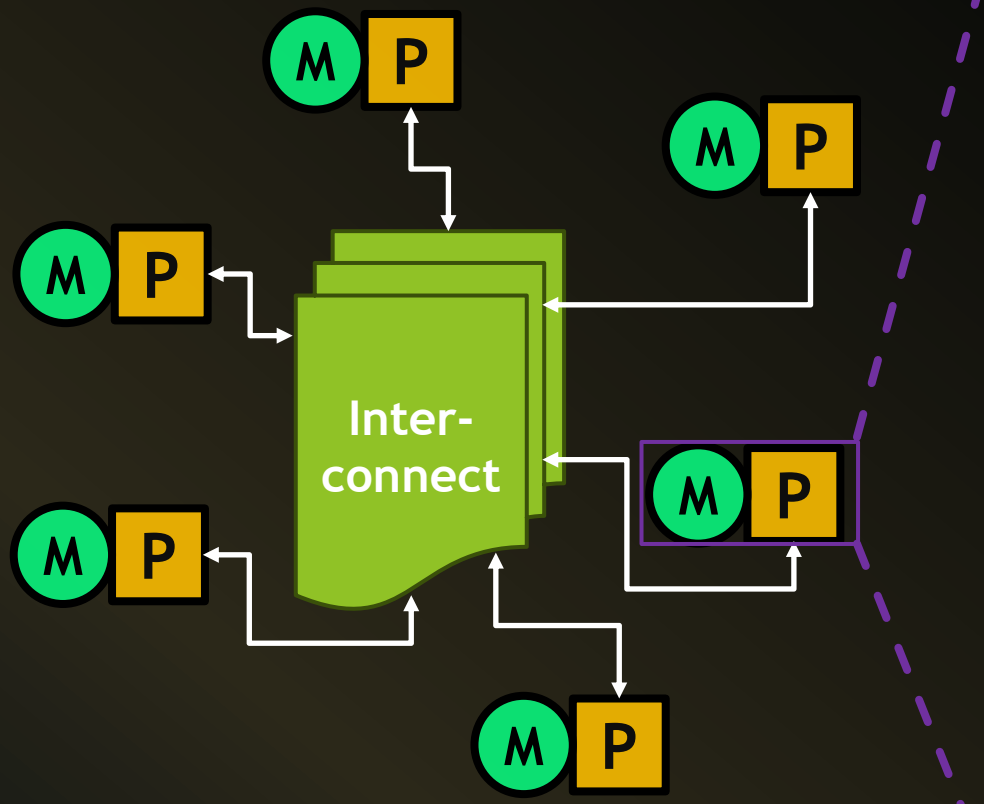
- Log Nodes
- Compute Nodes
  Each contains:
  - CPU(s) w/ DRAM
  - GPU(s) w/ VRAM
  - Interconnect switch

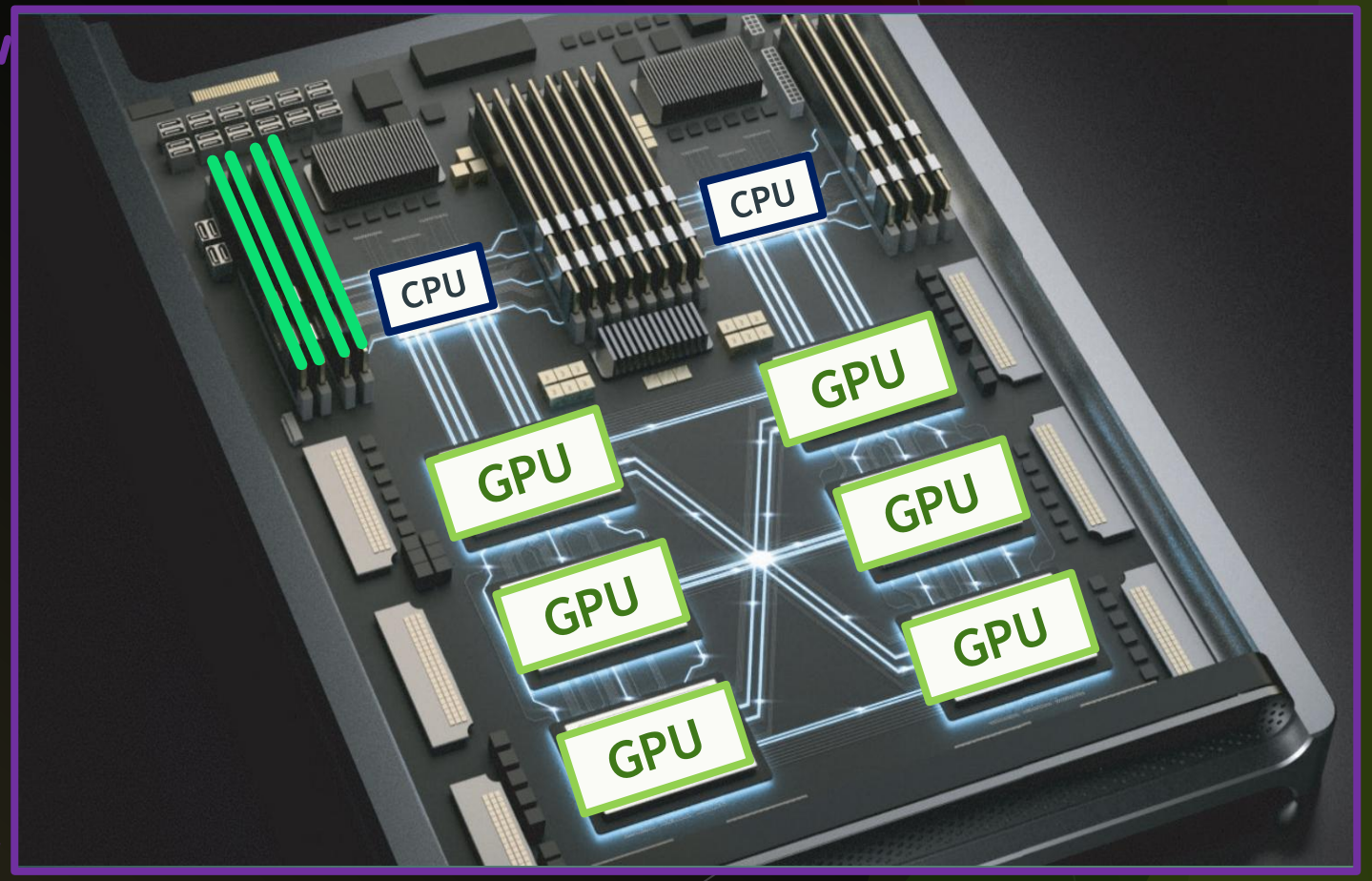


DISTRIBUTED-MEMORY ARCHITECTURES



DISTRIBUTED-MEMORY ARCHITECTURES





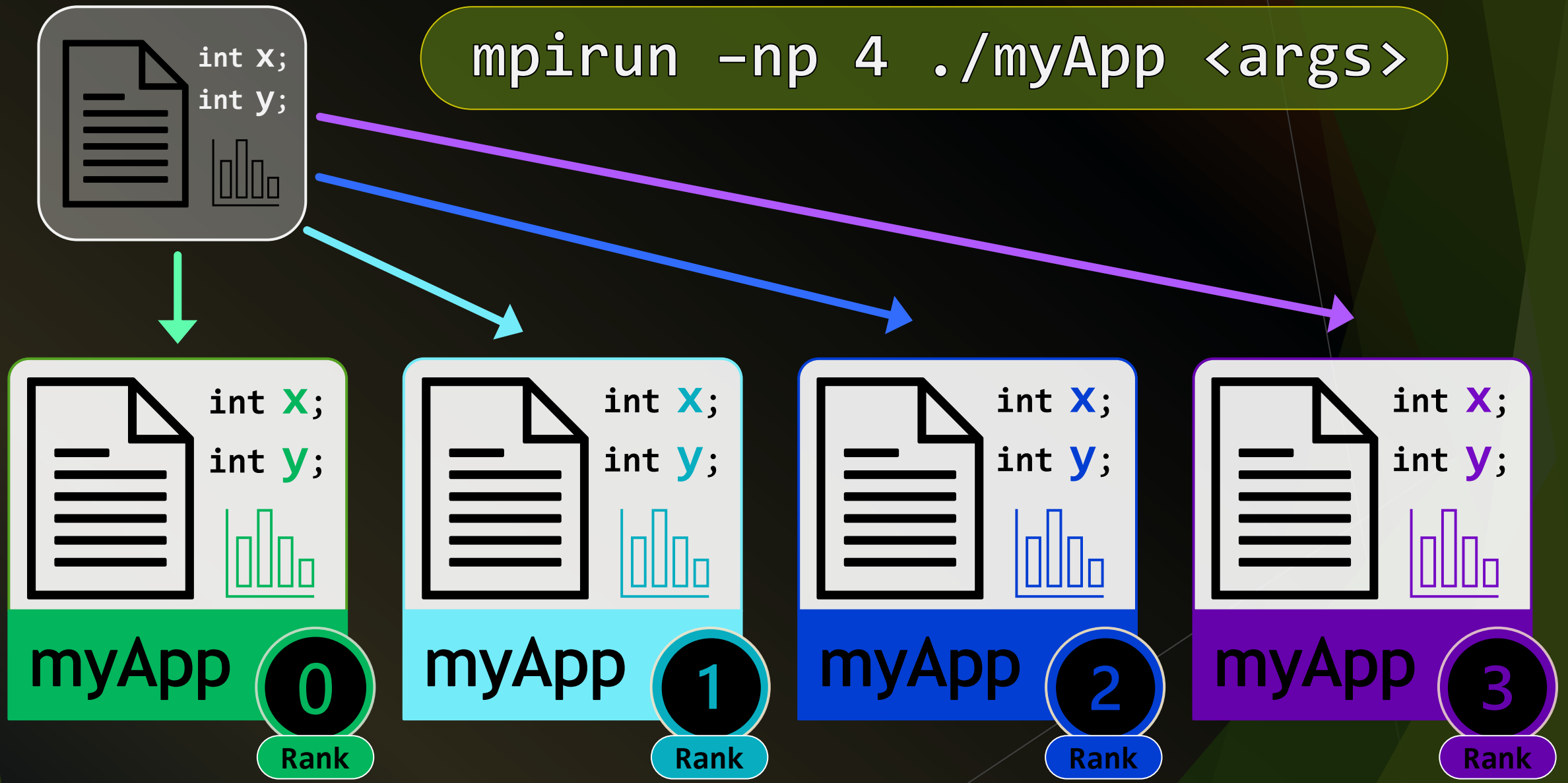
SPMD

- SPMD — Most message passing programs use the Single-Program-Multiple-Data (SPMD) model.
- Program — All processes run (their own copy of) the same program.
- Data — Each process has a separate copy of the data.
- Process ID — To be useful, each process has a unique identifier.
- Path — Processes can follow different control paths, based on their process ID.
- Partitioning — Usually run one process per processor/core/GPU.

DISTRIBUTED-MEMORY PROCESSING

```
mpirun –np 4 ./myApp <args>
```

DISTRIBUTED-MEMORY PROCESSING



DISTRIBUTED-MEMORY PROCESSING

```
X = 0;
```

```
>> x = 0;
```

```
if(rank==0) X = 1;
```

```
>> x = 1;
```


```
X = 0;
```

```
>> x = 0;
```

```
if(rank==0) X = 1;
```

```
>> x = 0;
```

MPI IS SIMPLE

Many parallel programs can be written using six functions:

- Setup — `MPI_Init`
- Teardown — `MPI_Finalize`
- Who am I? — `MPI_Comm_rank`
- How many of us? — `MPI_Comm_size`
- Message Passing — `MPI_Send`
- `MPI_Recv`

MINIMAL MPI PROGRAM - HELLO WORLD!

```c
#include <mpi.h>
#include <stdio.h>

int main( int argc, char *argv[]){
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    return 0;
}
```

```
mpirun -np 4 ./hello_world.mpi

>> Hello, world!
   Hello, world!
   Hello, world!
   Hello, world!
```

MINIMAL MPI PROGRAM – BETTER HELLO WORLD!

```c
#include <mpi.h>
#include <stdio.h>

int main( int argc, char *argv[]){
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

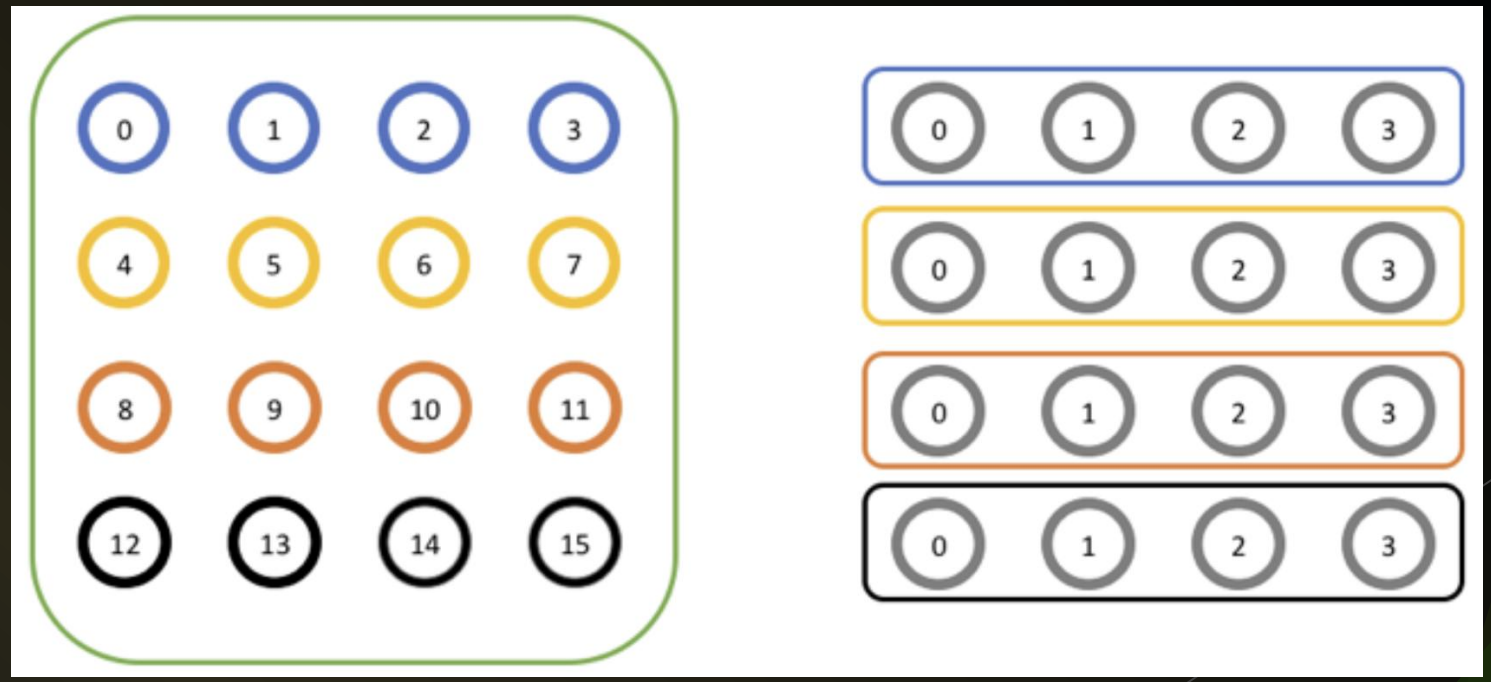
```
mpirun -np 4 ./hello_world.mpi

>> I am 1 of 4
   I am 3 of 4
   I am 0 of 4
   I am 2 of 4
```

COMMUNICATORS

- All MPI communications take place within a *communicator*
  - A communicator is fundamentally a group of processes
  - There is a pre-defined comm: MPI_COMM_WORLD
    - Contains ALL processes
  - A message can ONLY be received within the same communicator from which it was sent

USE OF COMMUNICATORS

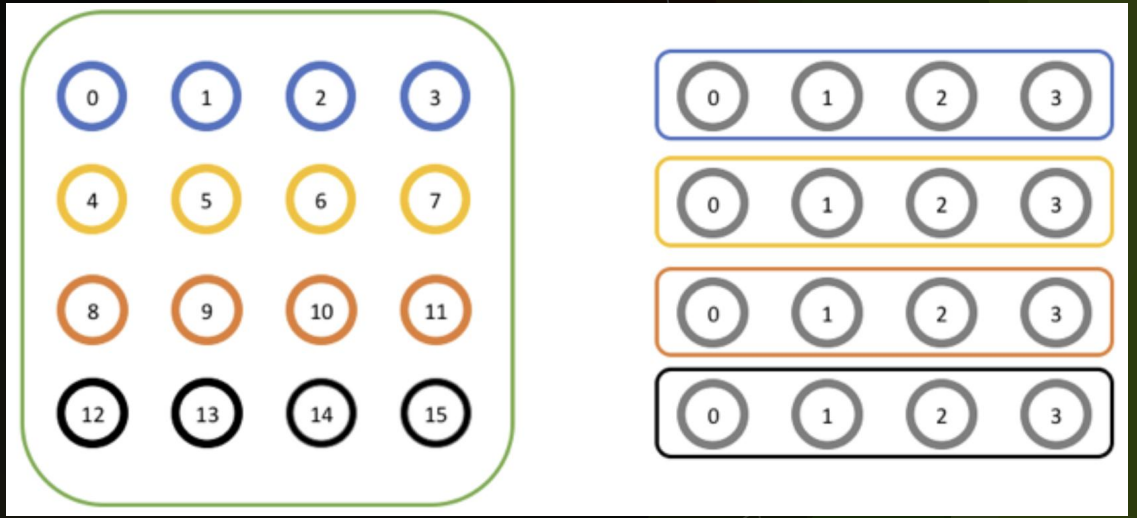
- Split MPI_COMM_WORLD into pieces
  - Each process has a new rank within each sub-communicator
  - Guarantees messages from the pieces do not interact

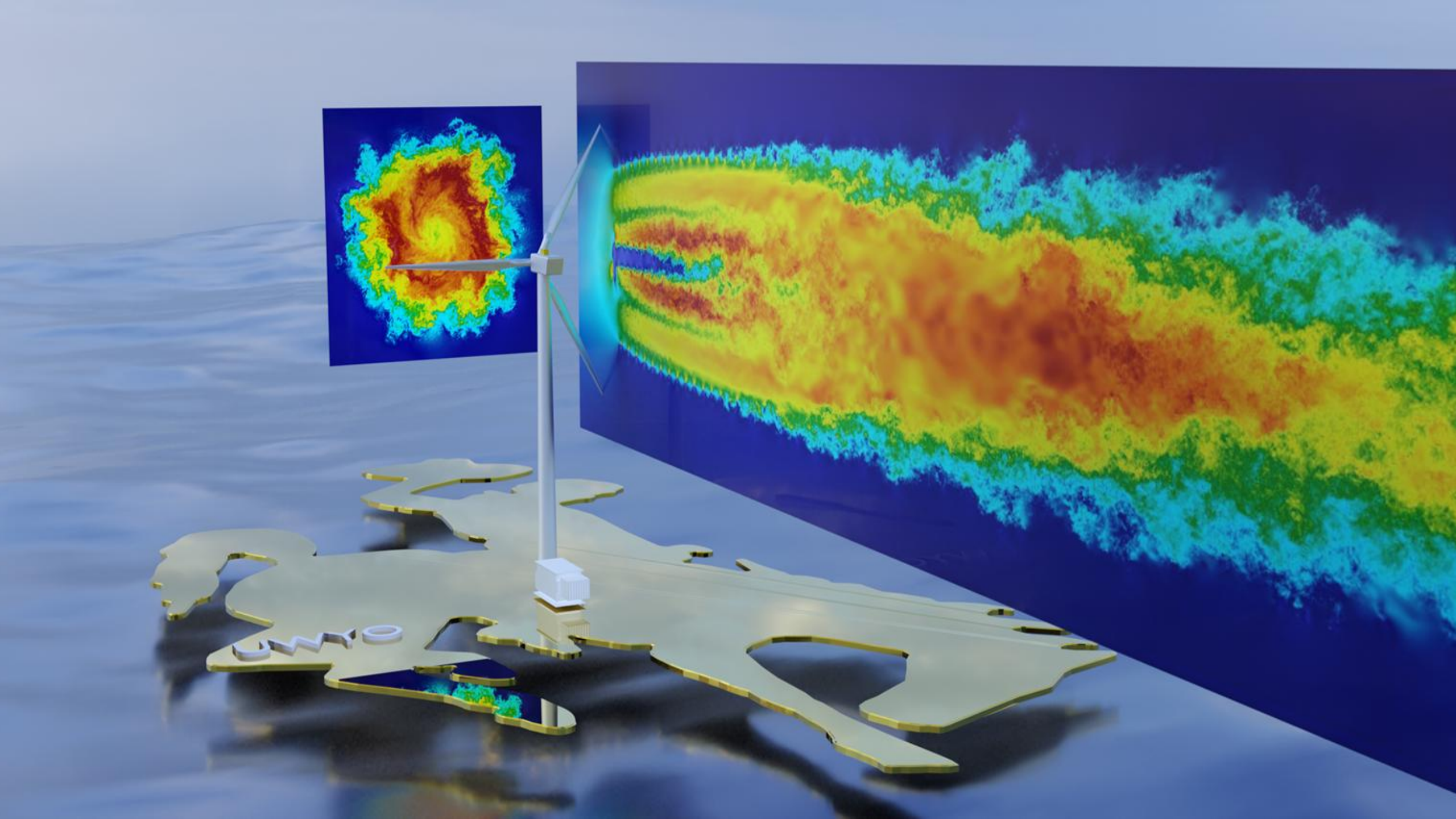


USE OF COMMUNICATORS

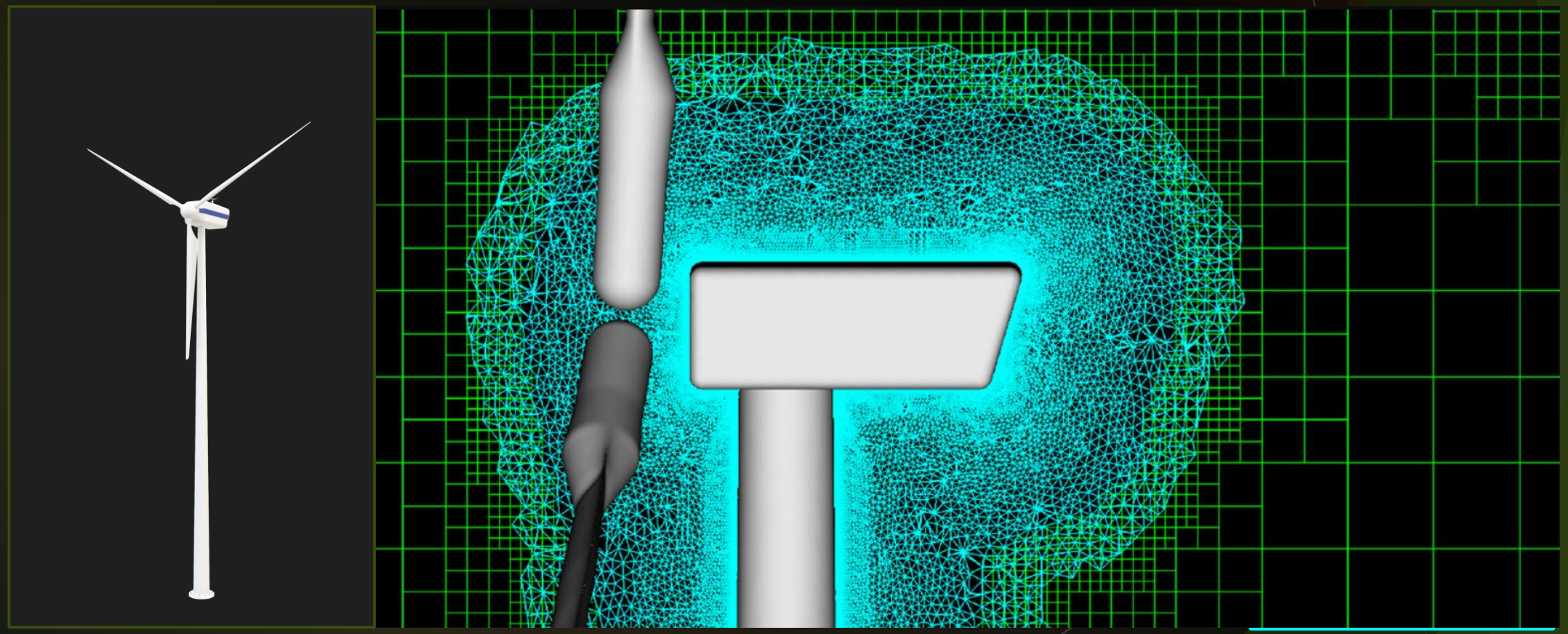
```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

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);
```

```
int MPI_Comm_split
```



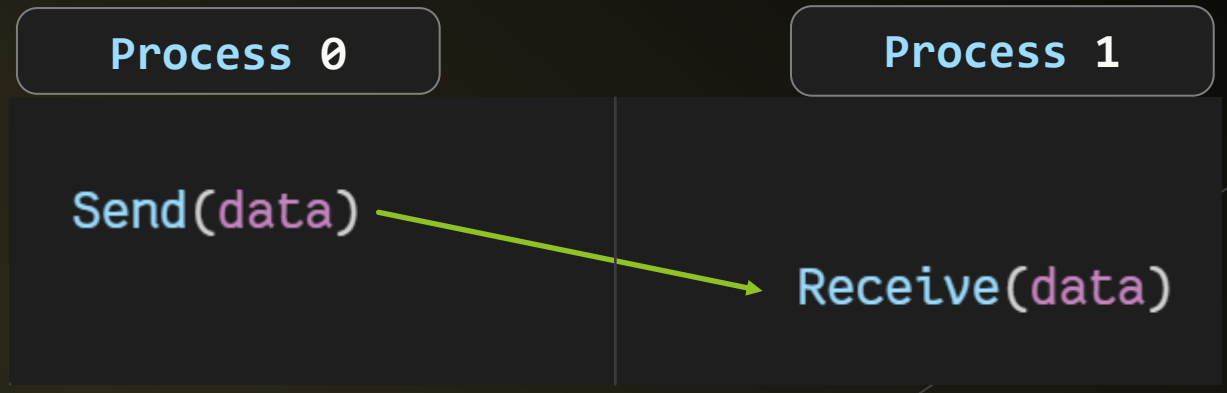


NONUNIFORM MESHES / MPI SUBGROUPS



POINT-TO-POINT COMMUNICATIONS

- MPI requires explicit data movement
  - YOU, the programmer, must say exactly what data goes where and when.
  - Cooperative data exchange
    - Data is explicitly *sent* by one process and explicitly *received* by another process.



MESSAGES

- **Transfer** A message transfers a number of data items of a certain type from the memory of one process to the memory of another process.
- **Contents** A message usually contains…
  - ID of the sending process (in the communicator below)
  - ID of the receiving process (in the communicator below)
  - Type of the data items
  - Number of data items
  - Data itself (i.e., beginning memory address)
  - Message identifier (i.e., tag)
  - Communicator (i.e., comm. context for group of processes)

COMMUNICATION MODES

- Synchronous Send is not complete until the message has started to be received.

```
// Blocking synchronous send
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
```

- Asynchronous Send completes as soon as the message has been copied to buffer.

```
//  Basic send with user-provided buffering
MPI_Buffer_attach(buffer, n*sizeof(double) + MPI_BSEND_OVERHEAD);
int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
```

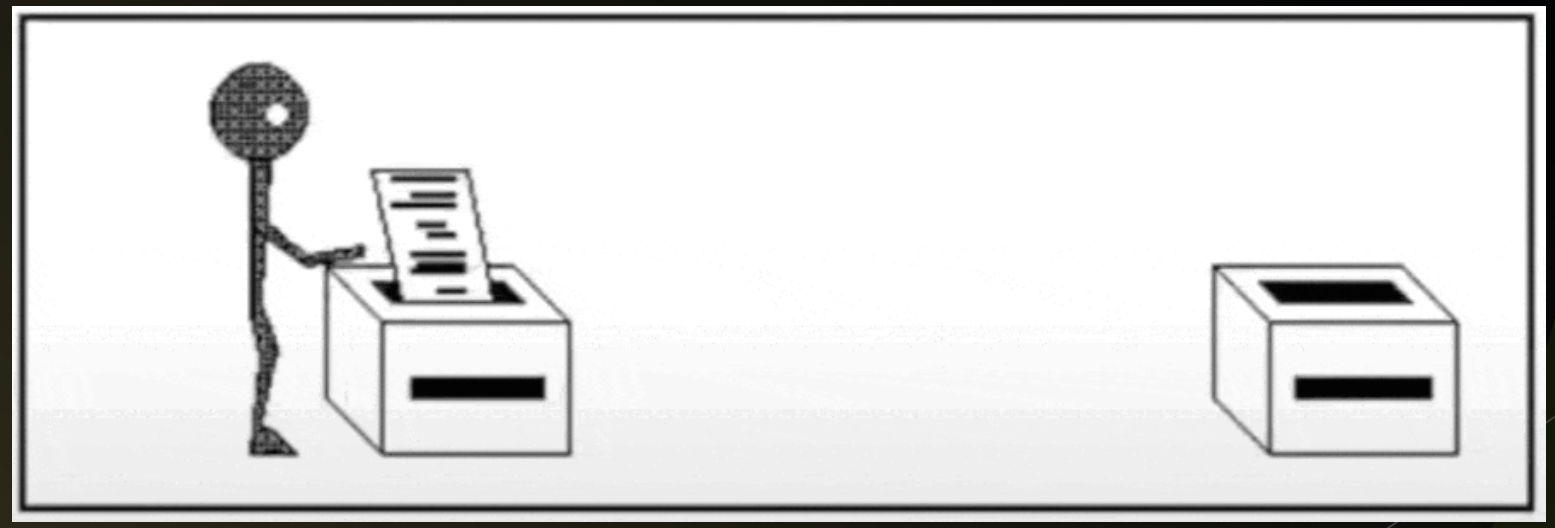
- Receives are usually synchronous – the receiving process must wait until the message arrives.

```
// Blocking receive for a message
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
```

SYNCHRONOUS SEND

```
int MPI_Ssend
```

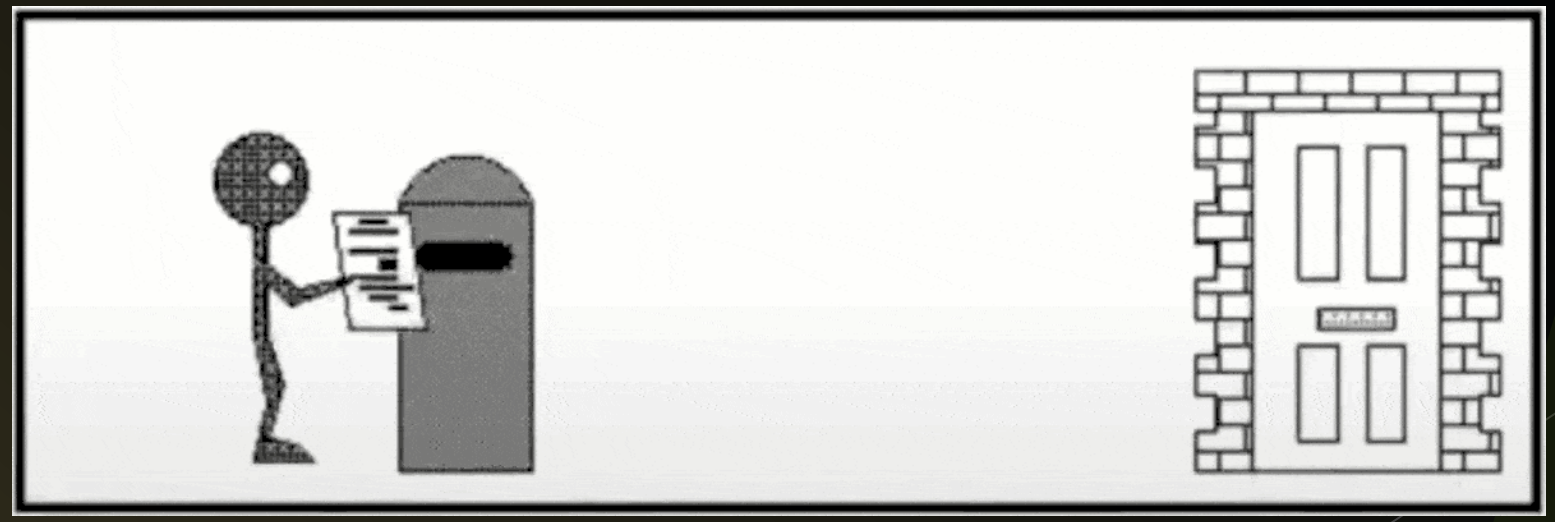
- Analogous to faxing a letter or making a phone call.
- "Beep" indicates when the letter started to be received.



ASYNCHRONOUS SEND

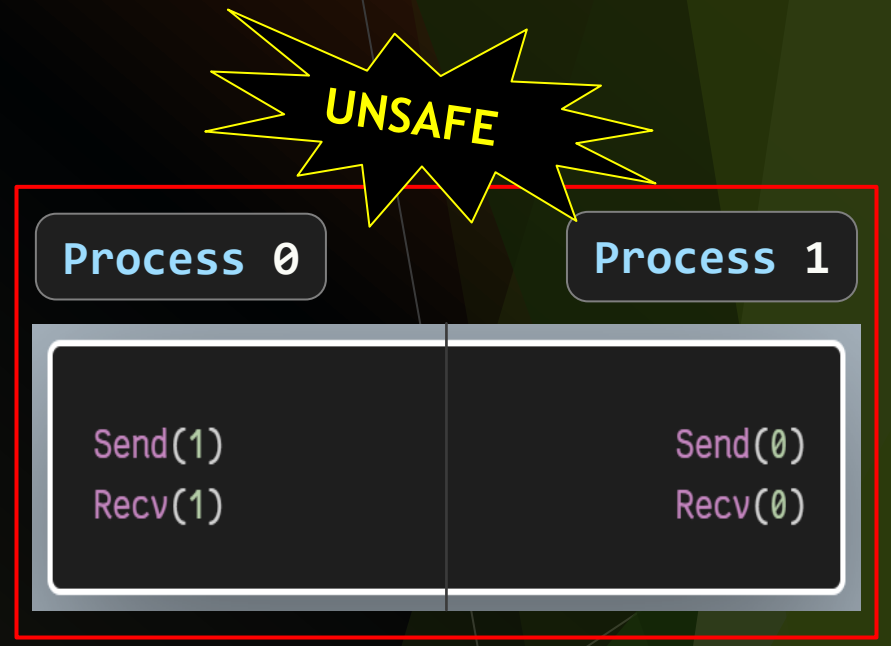
```
int MPI_Bsend
```

- Analogous to mailing a letter.
- Only know when the letter has been posted, not when it has been received.



SEND

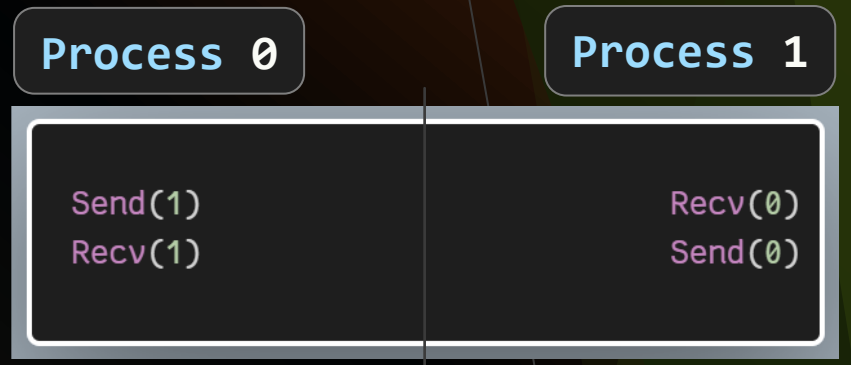
- MPI_Ssend runs the risk of deadlock.
- MPI_Bsend less likely to deadlock, but
  (a) user must supply buffer,
  (b) routine may fail if buffer is exhausted.



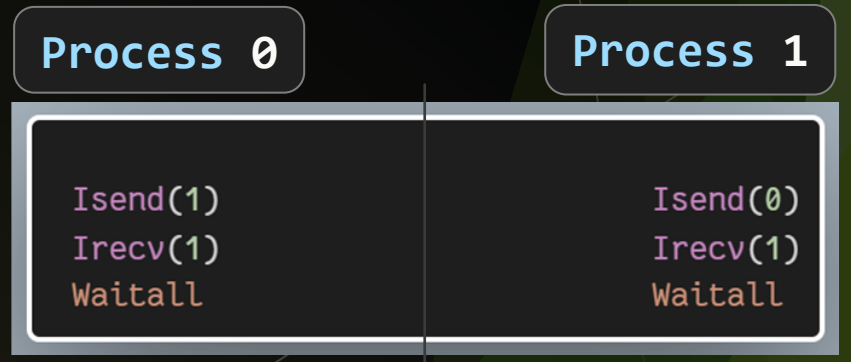
- MPI_Send tries to solve these problems
  - Buffer space provided by system
  - Send will normally be asynchronous (like Bsend)
  - If buffer is full, Send becomes synchronous (like Ssend)
  - Unlikely to fail (but could still deadlock)

POSSIBLE SOLUTIONS

- **Pingpong** communications
  - Match communications deliberately to avoid deadlock.
  - Process 0 sends, then receives
    Process 1 receives, then sends

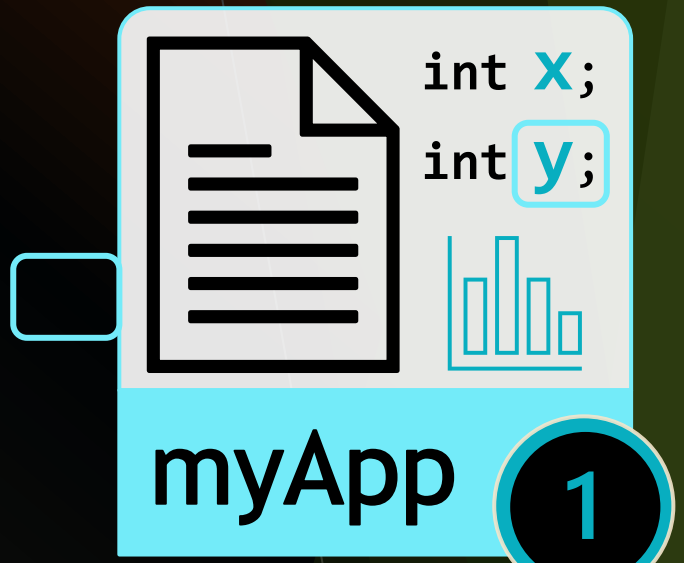


- **Non-blocking** communications (e.g., MPI_**I**send, MPI_**I**recv)
  - MPI_**I**<*function*>: *Immediate* version of function.
  - **Return**: Functions that return immediately, without waiting for the operation to complete.
  - **Purpose**: Allows a program to overlap communication with computation.
  - **Completion**: You must explicitly check or wait for completion using MPI_Wait, MPI_Test, etc.



PT2PT COMMUNICATION - TEST YOUR KNOWLEDGE


```
X = 23
if(rank==0)
 MPI_Send(X,1)

if(rank==1)
 MPI_Recv(Y,0)
```

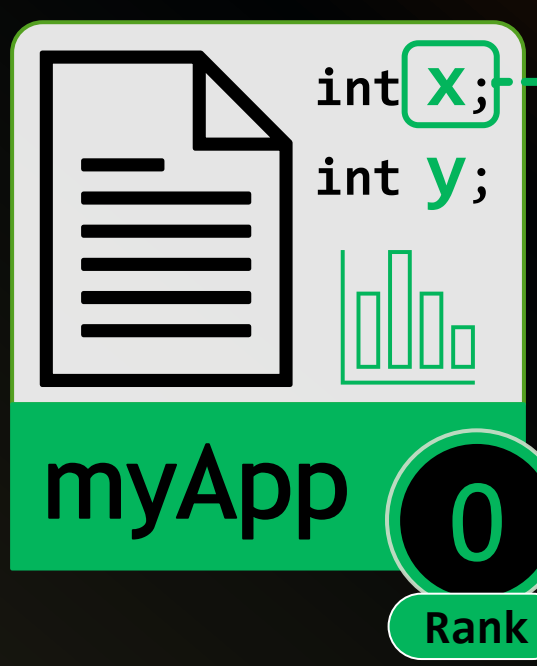



```
>> x = 23
   y = ?
```

```
>> x = 23
   y = ?
```

PT2PT COMMUNICATION - TEST YOUR KNOWLEDGE


```
X = 23
if(rank==0)
 MPI_Send(X,1)

if(rank==1)
 MPI_Recv(Y,0)
```



Send

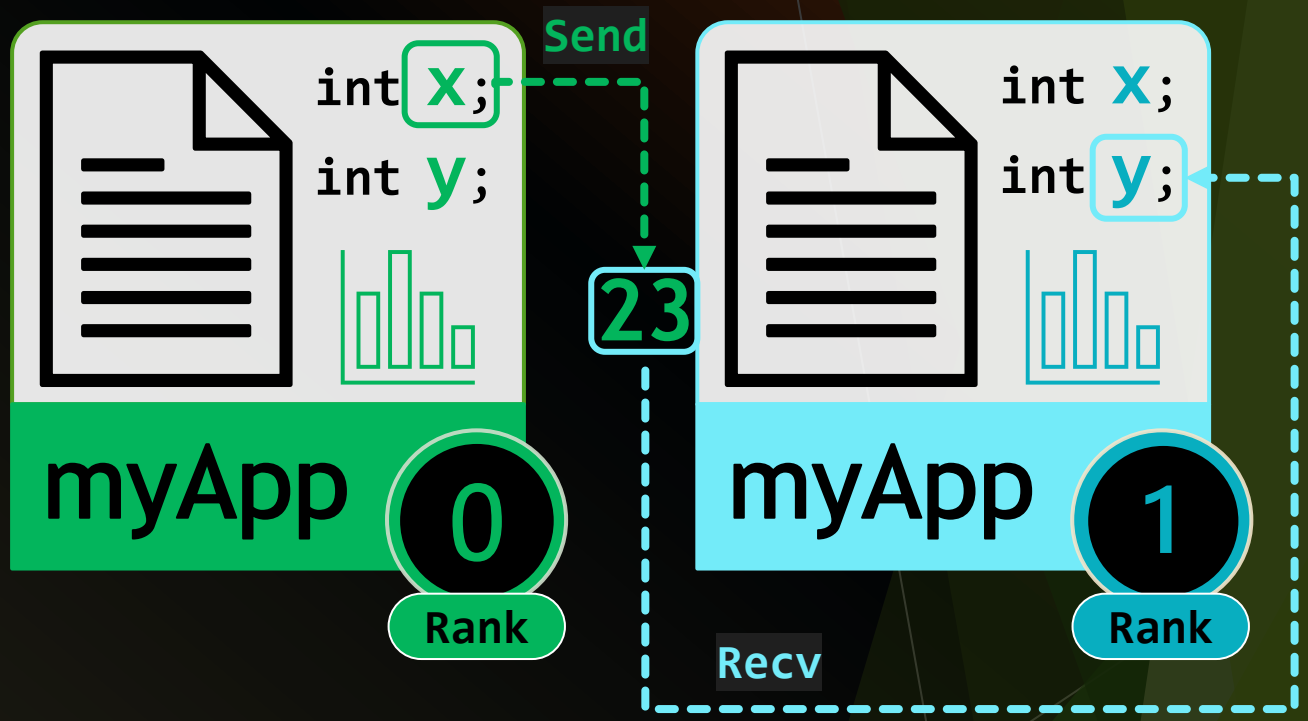
23


```
>> x = 23
   y = ?
```

```
>> x = 23
   y = ?
```

PT2PT COMMUNICATION - TEST YOUR KNOWLEDGE


```
X = 23
if(rank==0)
 MPI_Send(X,1)

if(rank==1)
 MPI_Recv(Y,0)
```



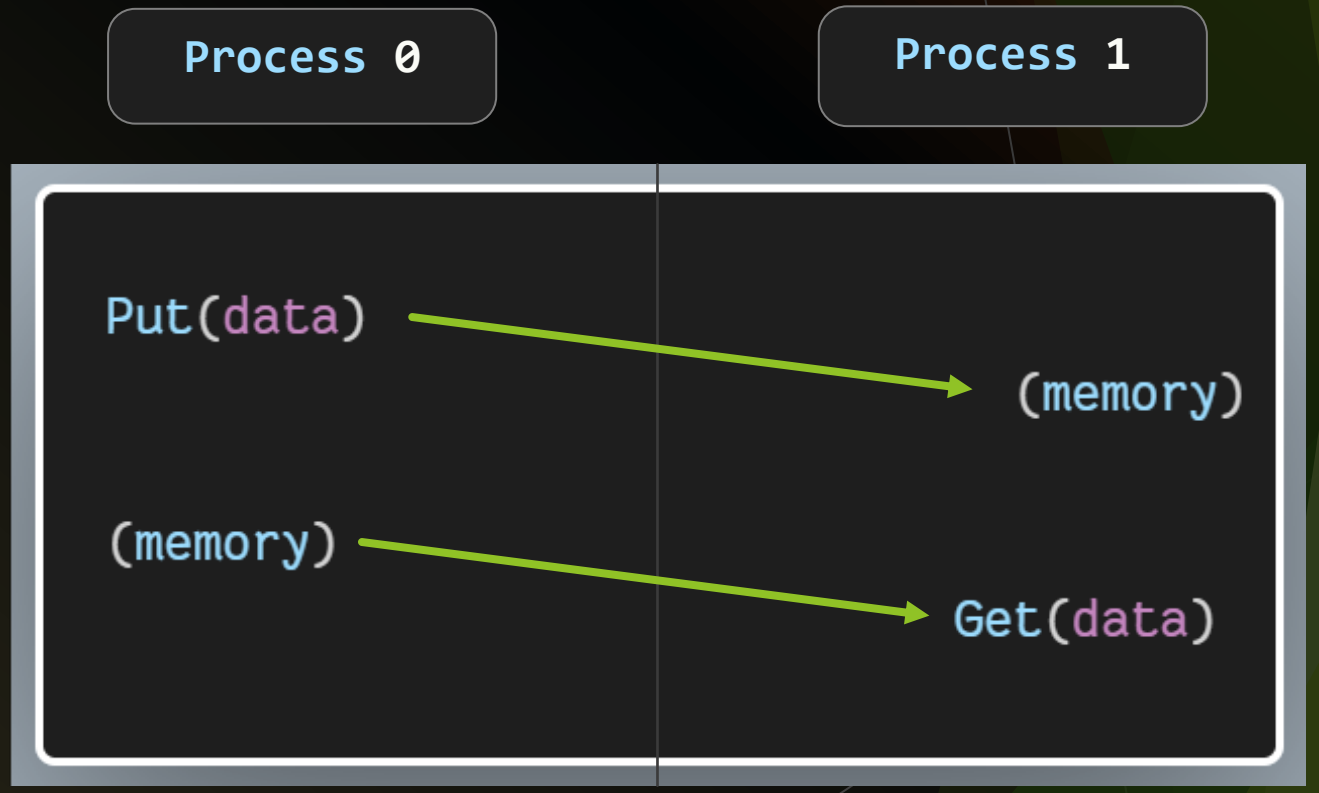
>> x = 23
y = ?

>> x = 23
y = 23

ONE-SIDED OPERATIONS FOR COMMUNICATION (RMA)

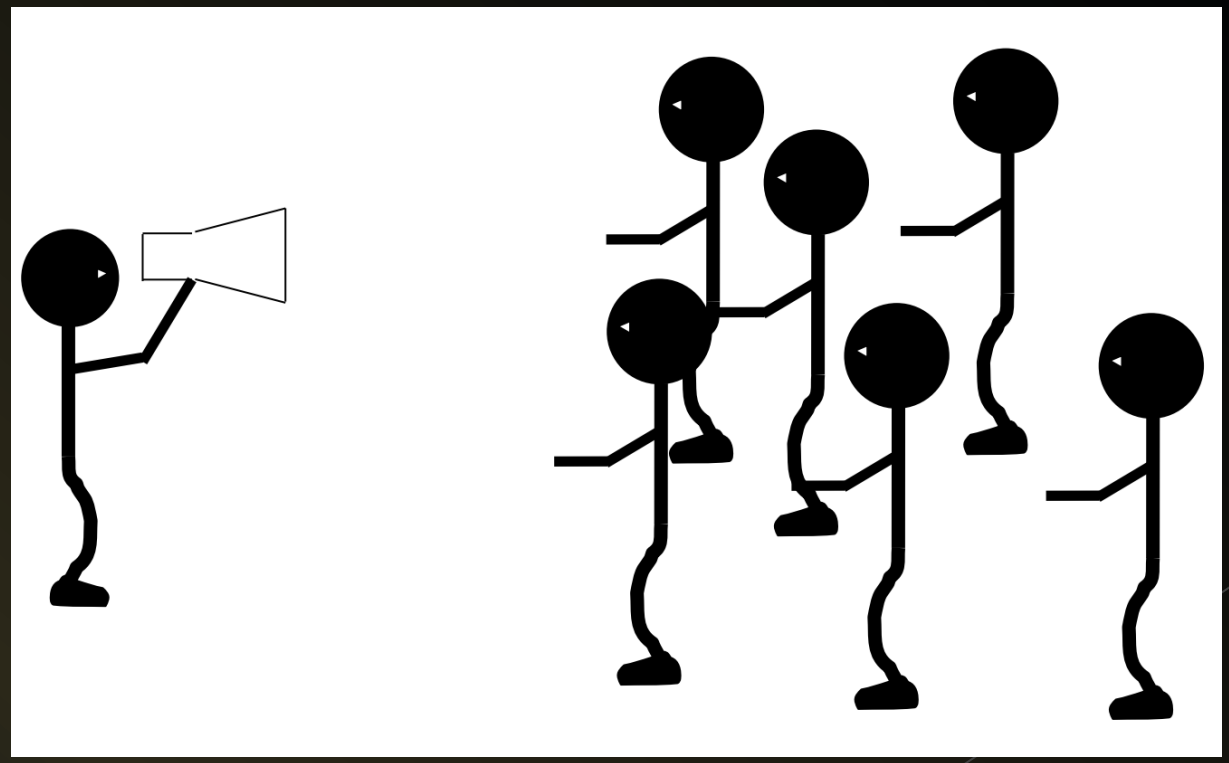
```
// Put data into a memory
window on a remote process

int MPI_Put(…);
```

```
// Get data from a memory
window on a remote process

int MPI_Get(…);
```

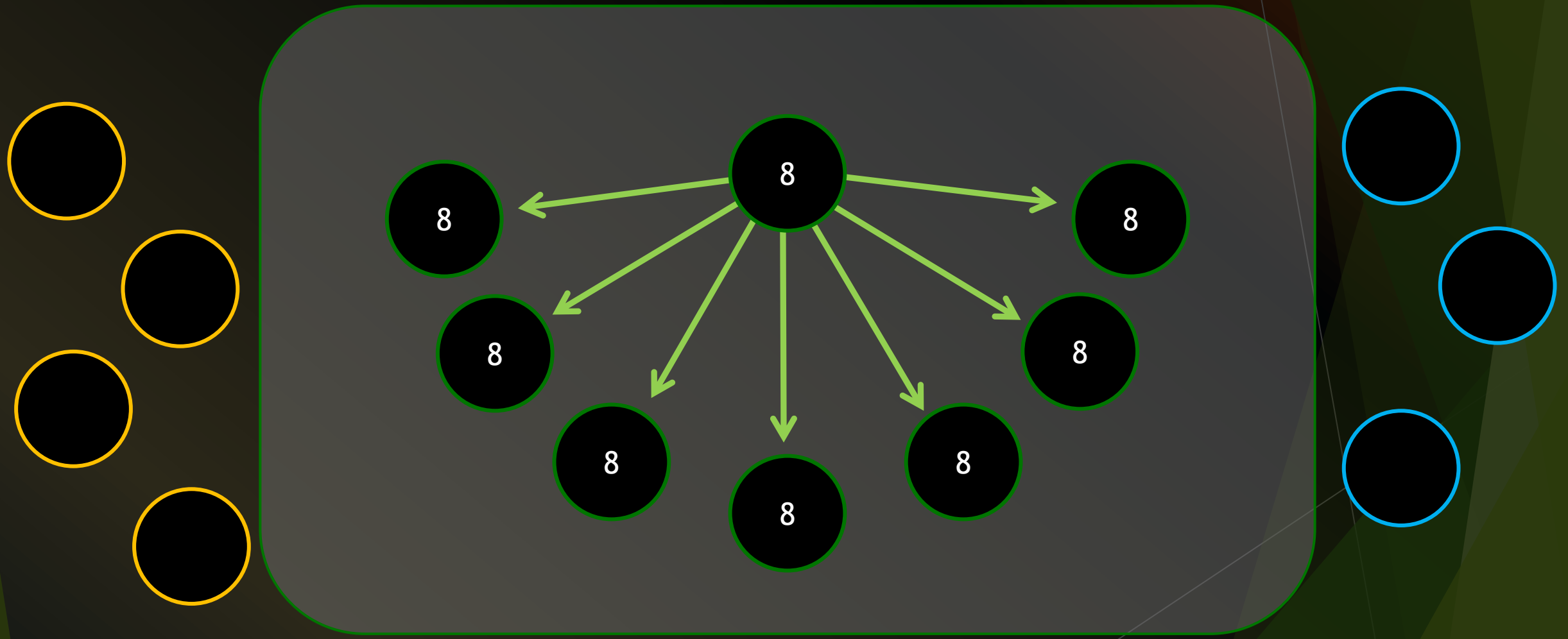


COLLECTIVE COMMUNICATIONS

- Group communication between *sets* of processes
- Can be done using simple (pt2pt) messages, but often implemented separately for efficiency
- Called by all processes in a communicator.

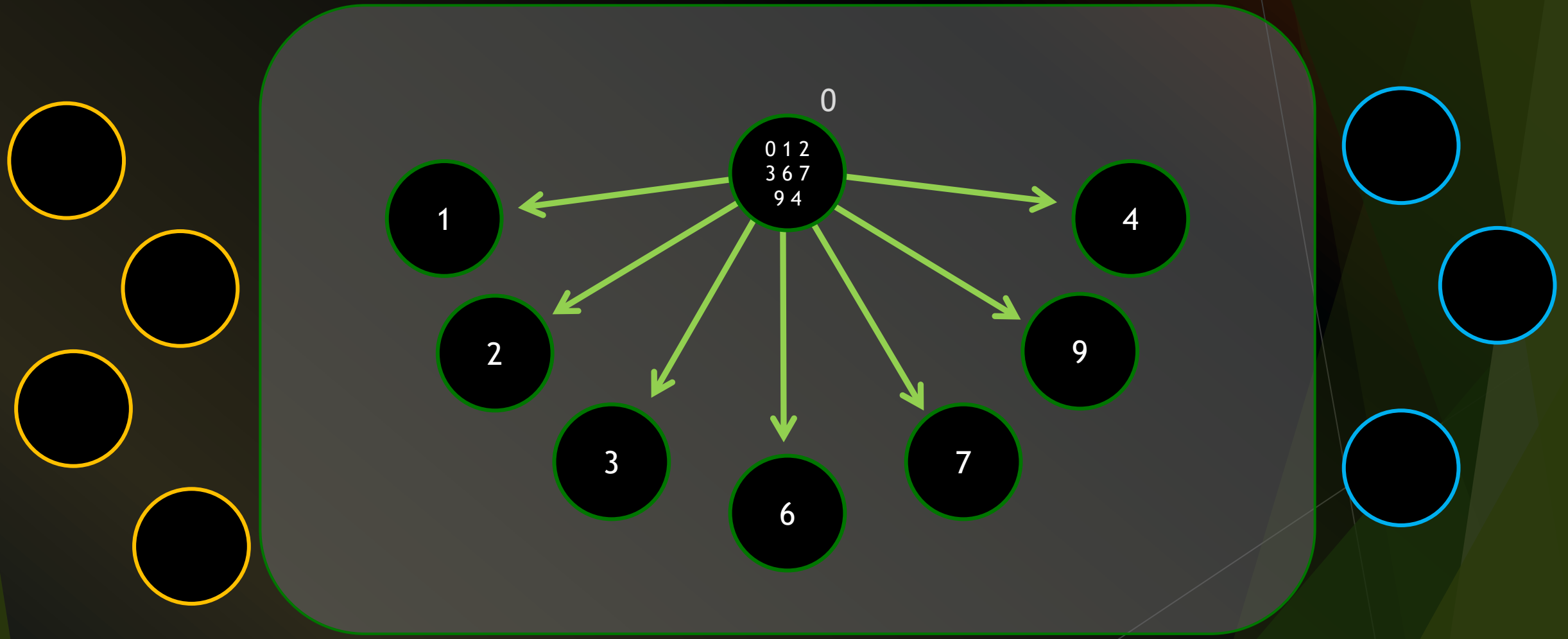


BROADCAST



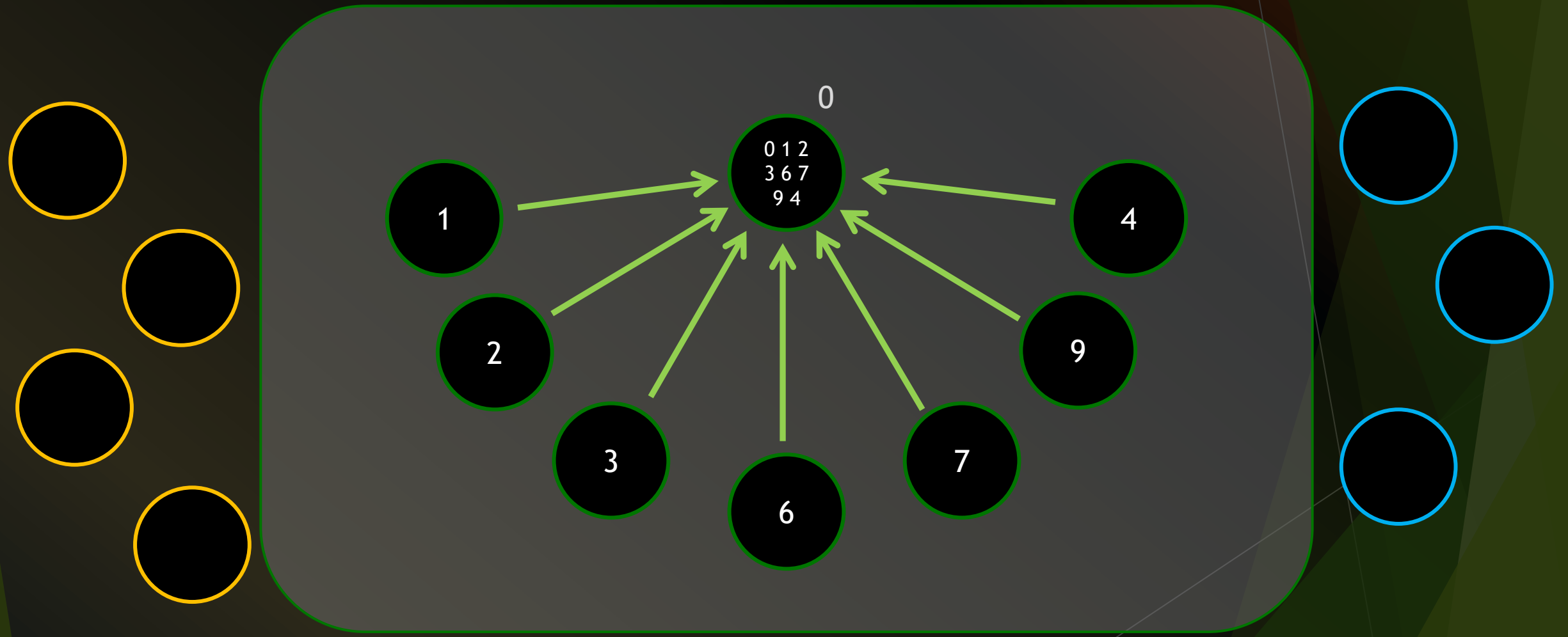
- From one process to all processes in comm

SCATTER



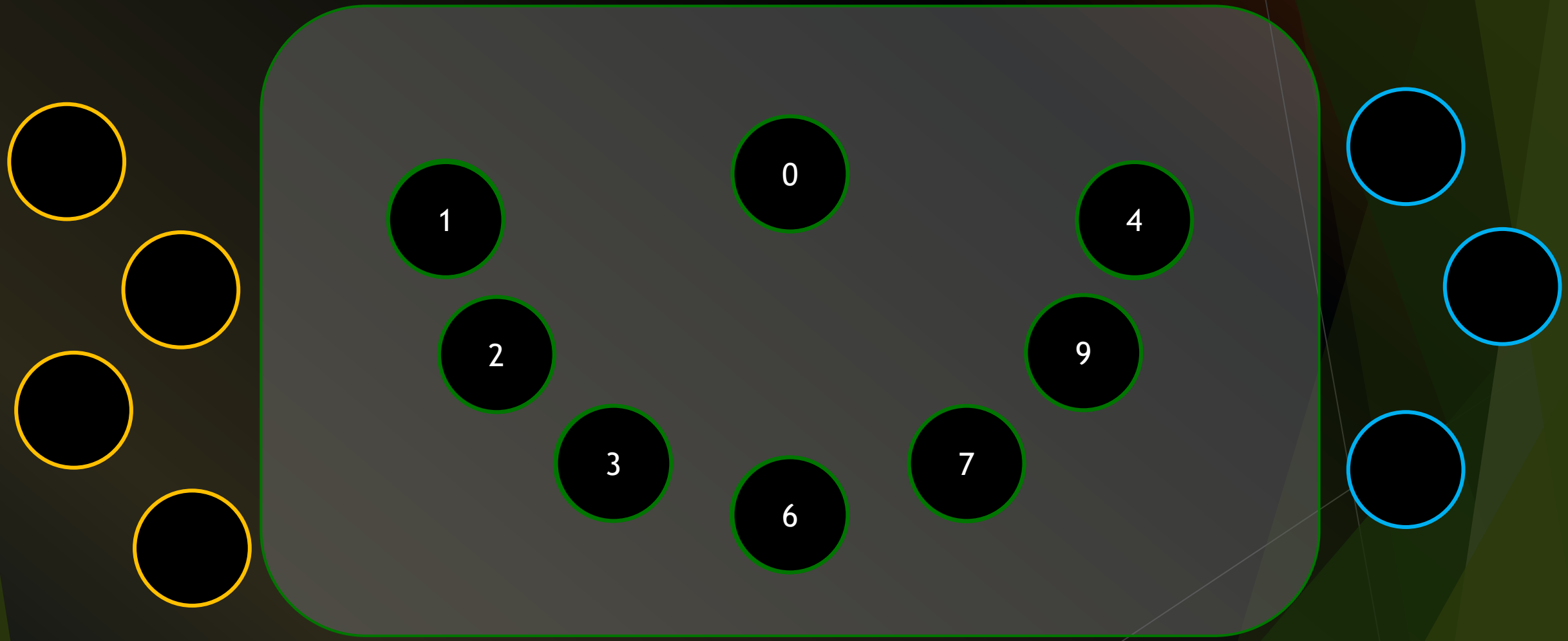
- Information scatter to all processes in comm

GATHER



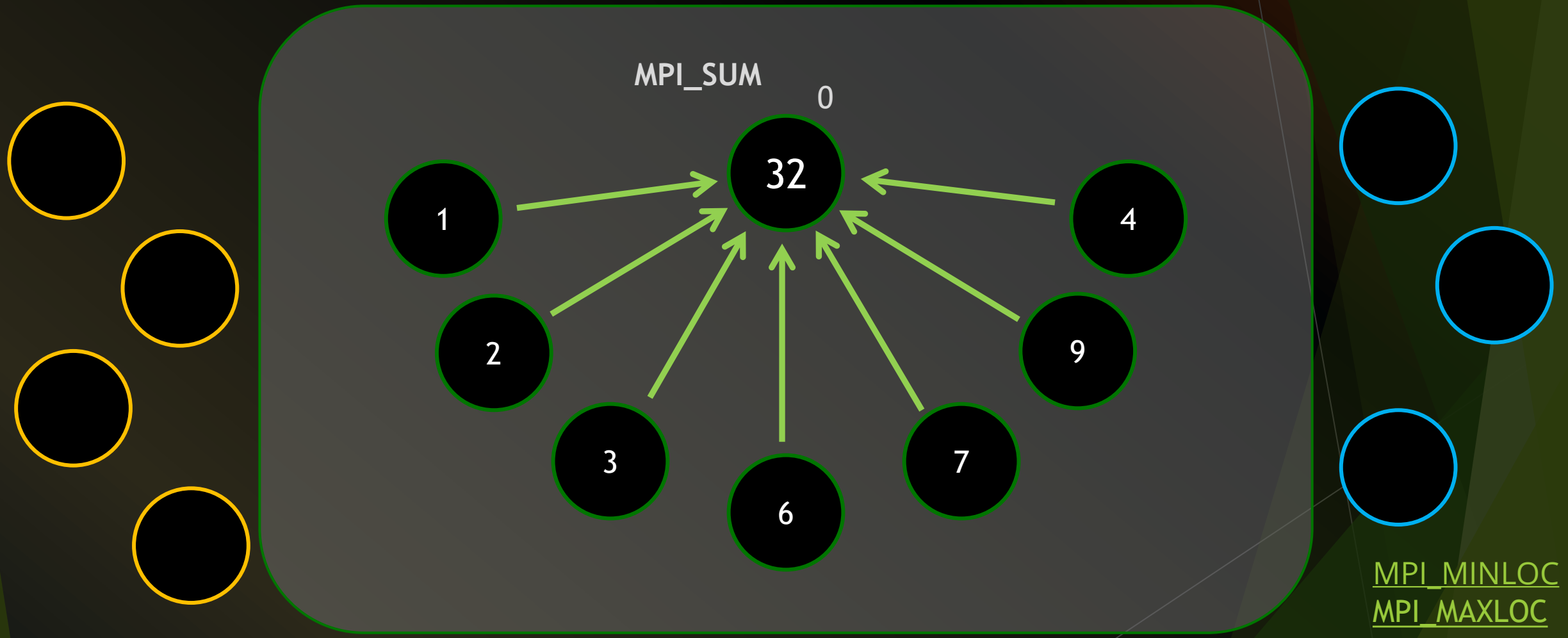
- Information gathered to one process in comm

ALLGATHER



- Information gathered to all processes in comm

REDUCE



MPI_MINLOC
MPI_MAXLOC

- Form a global sum, product, max, min, etc. to one process in comm

SUMMARY

- Message-Passing is a programming model
  - Implemented by MPI
  - The Message-Passing Interface (MPI) is a library of functions
- Essential to understand the basic concepts
  - Private variables
  - Explicit communications
  - SPMD
- Difficulty comes from the Message-Passing model
  - Very different from sequential programming

LANGUAGE INTERFACES

- Official MPI Bindings (defined in the MPI Standard)
  - C/C++, Fortran
- Third-Party Bindings (Community Libraries)
  - Python, Julia, Java, Rust, Go, R, MATLAB
- Python: I added some Python examples to this course
  - mpi4py package: wrappers for standard library
  - https://mpi4py.readthedocs.io/
  - Two versions of comms routines: upper and lower case
    - comm.Send() - for NumPy arrays
    - comm.send() - for arbitrary datatypes

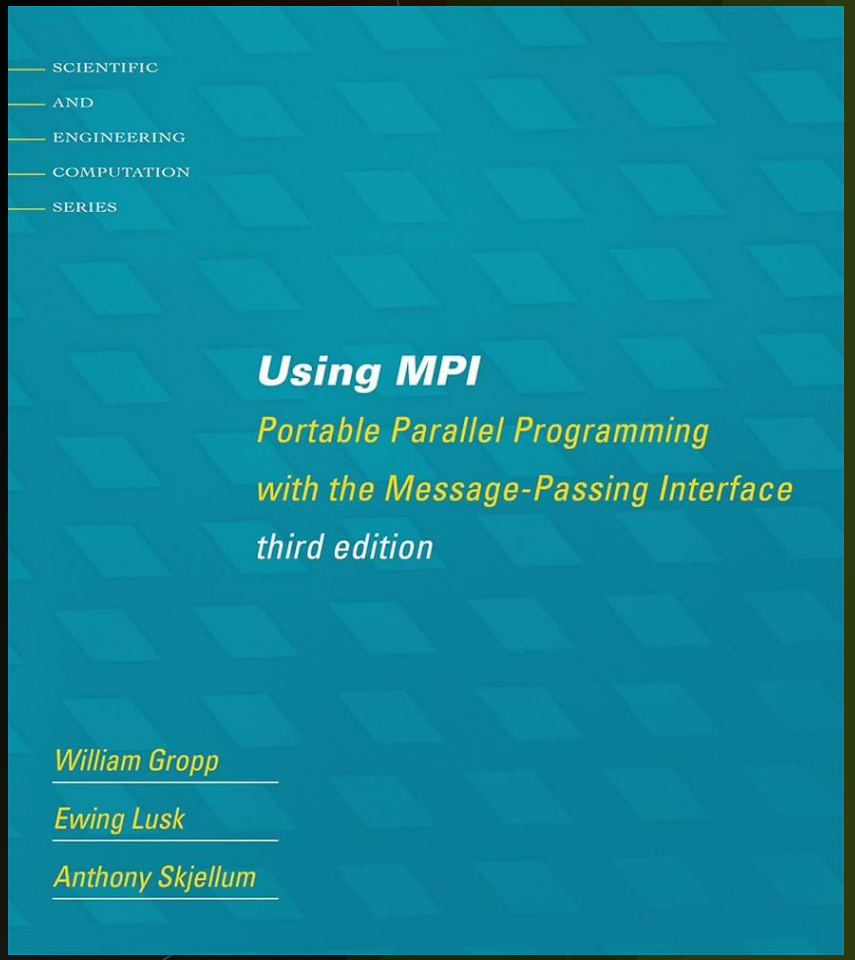
HPC PYTHON SESSION (WEDNESDAY - 9:00 AM)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
processor_name = MPI.Get_processor_name()

print(f"Hello world from rank {rank} out of {size} on
processor {processor_name}")
```

Ramses van Zon

DOCUMENTATION

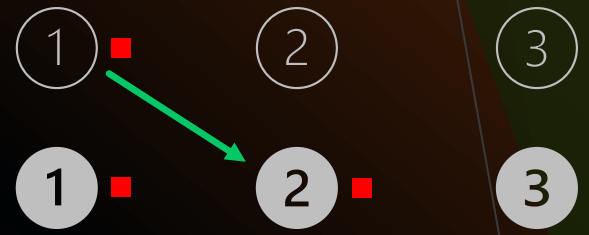
- MPI Standard available online
  - See: http://www.mpi-forum.org/docs/
  - Currently on version 5.0
- Manual (man) pages on Bridges2
  - Must load MPI module
  - e.g., user@bridges2$ man MPI_Init
- Excellent Resource
  - https://rookiehpc.org/
- Textbook: Using MPI



SECTION II

DERIVED DATA TYPES

MPI_SEND/MPI_RECV



1
RANK

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

```
// [in] Initial address of send buffer
// [in] Maximum number of elements to send
// [in] Datatype of each send buffer entry
// [in] Rank of destination (integer)
// [in] Message tag (integer)
// [in] Communicator (handle)
```

2
RANK

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

```
// [out] Initial address of receive buffer
// [in] Maximum number of elements to recv
// [in] Datatype of each recv buffer entry
// [in] Rank of source (integer)
// [in] Message tag (integer)
// [in] Communicator (handle)
// [out] Status object (status)
```

MPI INTRINSIC DATA TYPES

C/C++		FORTRAN
MPI_CHAR	MPI_C_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_FLOAT_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_DOUBLE_COMPLEX	MPI_INTEGER1
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER2
MPI_LONG	MPI_C_BOOL	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_LOGICAL	MPI_REAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT8_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT16_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_INT32_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_INT64_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT8_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT16_T	MPI_LOGICAL
MPI_DOUBLE	MPI_UINT32_T	MPI_BYTE
MPI_LONG_DOUBLE	MPI_UINT64_T	MPI_PACKED
	MPI_BYTE	
	MPI_PACKED	

MPI DATA TYPES

Recall : MPI just gets the starting address of the memory you want to send/receive.

i.e.,

```
MPI_Send (const void *sbuf, … )
MPI_Recv ( void *rbuf, … )
```

What happens if we want to send custom data without making a copy?

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

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

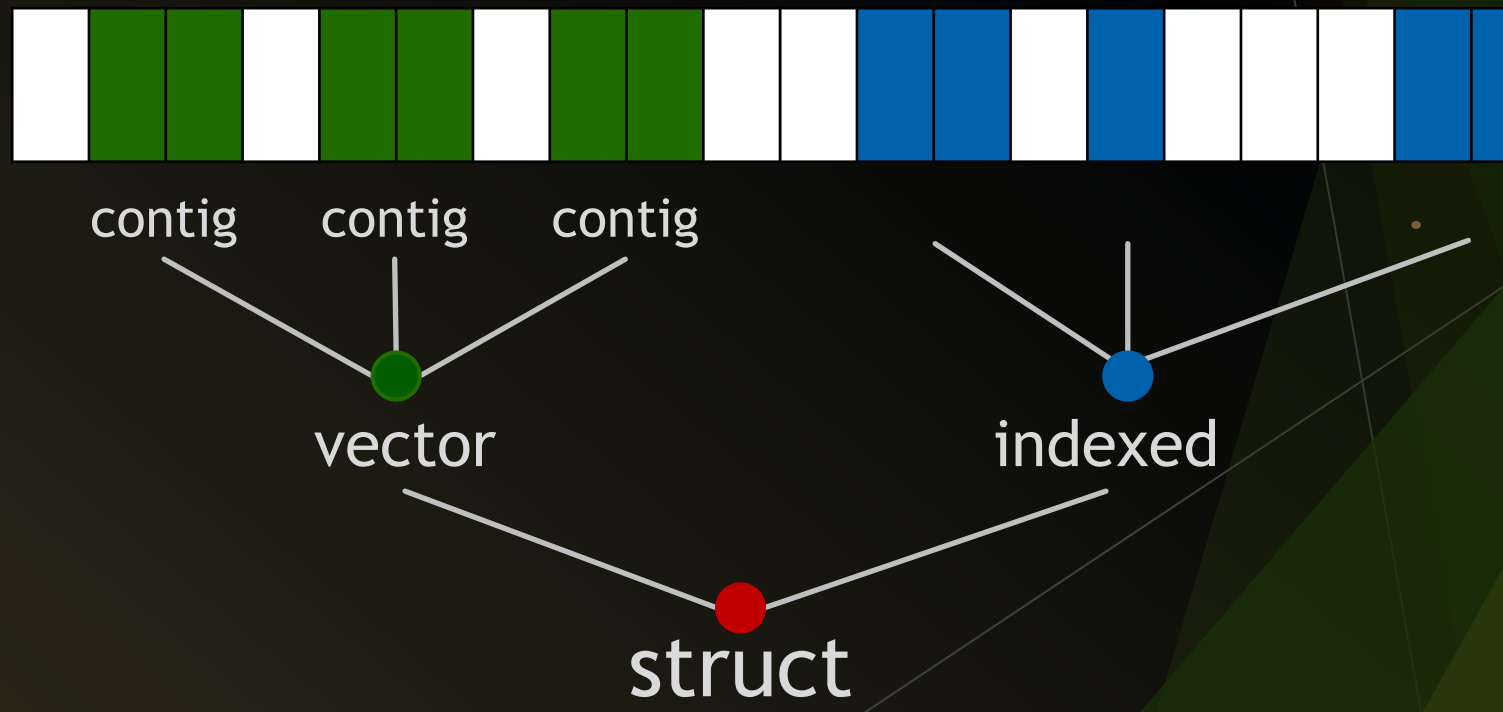
MPI DATA TYPES

We can serialize custom data into a message stream

>> Declarative specification of data layout <<

Derived Data Types

- Contiguous
- Vector
- Indexed
- Struct



MPI DERIVED DATA TYPES

`MPI_Type_contiguous`	> contiguous datatype (no gaps)
`MPI_Type_vector`	> regularly spaced datatype (gaps)
`MPI_Type_create_hvector`	> like vector, but stride is specified in bytes
`MPI_Type_indexed`	> variably-spaced datatype
`MPI_Type_create_indexed_block`	> special case of indexed
`MPI_Type_create_subarray`	> subarray within a multidimensional array
`MPI_Type_create_darray`	> distribution of a ndim-array into a grid of ndim-logical processes
`MPI_Type_create_struct`	> fully general datatype

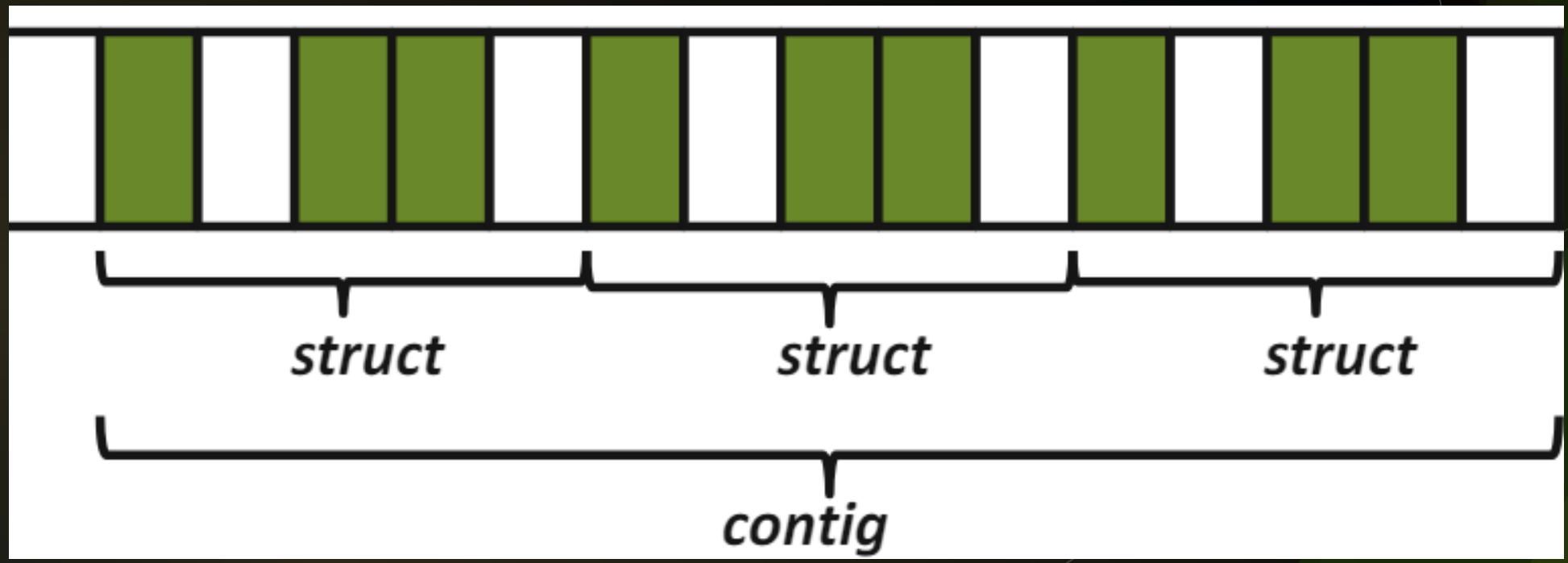
Likely discussed in Parallel I/O Session (Wednesday)

MPI_TYPE_CONTIGUOUS

Creates a contiguous datatype.

```
int MPI_Type_contiguous(
    int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

```
// count:     Number of entries.
// oldtype:   Old datatype (handle).
// newtype:   New datatype (handle).
```



MPI_TYPE_CONTIGUOUS

EXAMPLE

Creates a contiguous datatype.

```
int MPI_Type_contiguous(
    int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

```
MPI_Datatype row_type;
int MPI_Type_contiguous(
            4,   // count
    MPI_FLOAT,   // old type
    &row_type); // new type
MPI_Type_commit(&row_type);

MPI_Send(&A[2][0],
         1, row_type,
         dest, tag, comm);
```

`float` A[4][4]

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

MPI_TYPE_COMMIT / FREE / DUP

Types must be committed before use!

- Only the DDT's that are used
- MPI_Type_commit may perform extensive optimizations

```
MPI_Datatype ParticleType;
// Build DDT ...
MPI_Type_commit(&ParticleType);
```

Types should be deleted when finished

- Free up MPI resources
- Does not affect types constructed from it

```
MPI_Type_free(&ParticleType);
```

Types may be duplicated

- May be useful for holding state of type before modification

```
MPI_Datatype ParticleTypeDuplicate;
MPI_Type_dup(ParticleType,
             &ParticleTypeDuplicate);
```

HANDS ON ACTIVITY #1

MPI_Type_contiguous

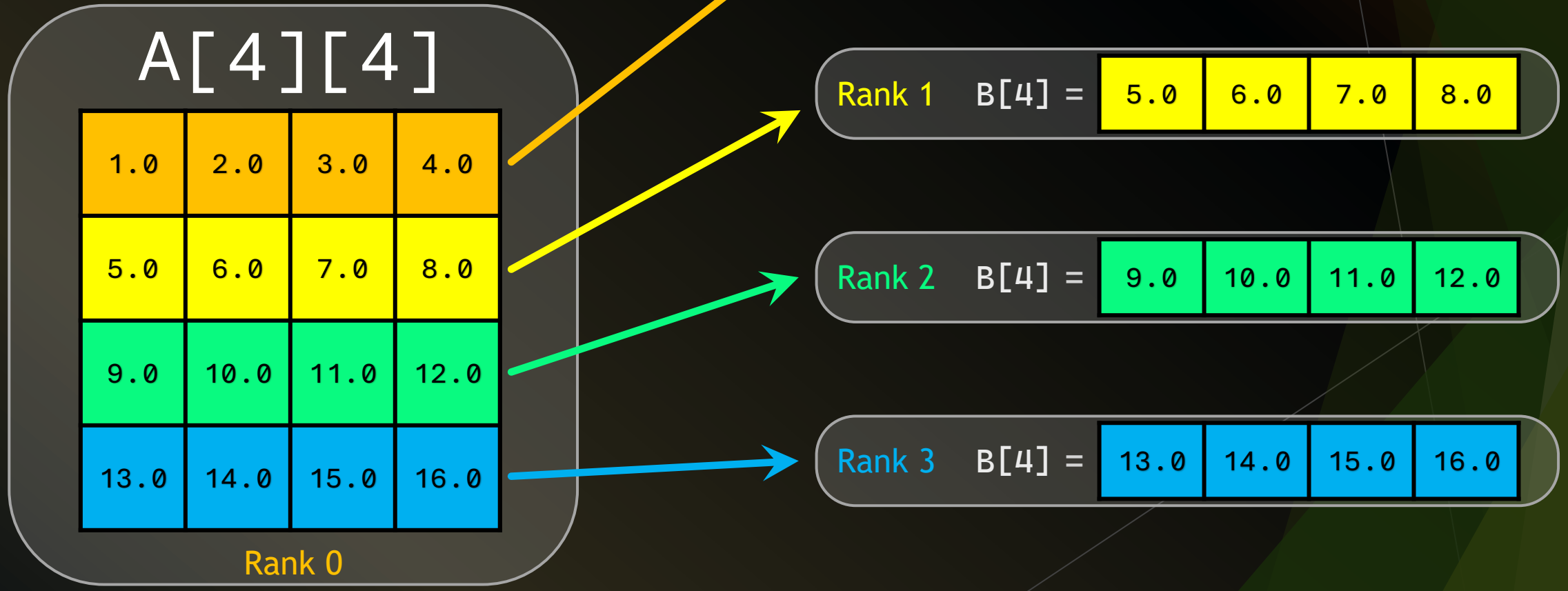
MPI_Type_vector

MPI_Type_indexed

MPI_Type_create_struct

EXERCISE.1 ddt.1.contiguous DERIVED DATA TYPES

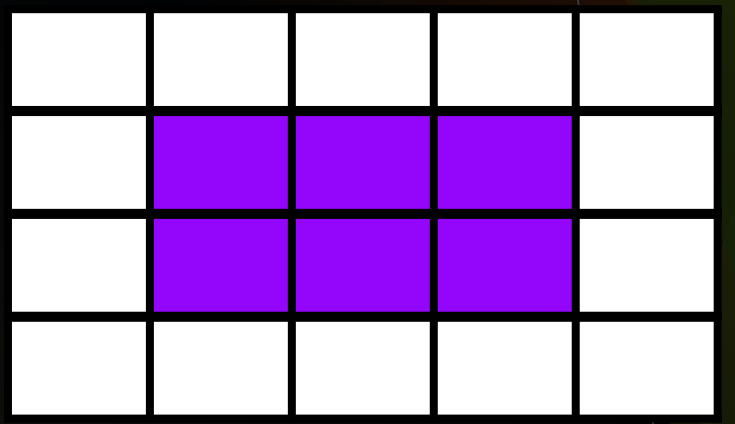
```
int MPI_Type_contiguous(
    int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```



MPI_TYPE_VECTOR

Creates a vector (strided) datatype.

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

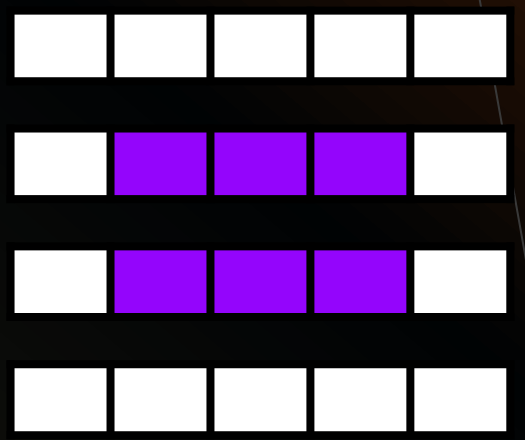


How can we send the subarray?

MPI_TYPE_VECTOR

Creates a vector (strided) datatype.

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



How can we send the subarray?

MPI_TYPE_VECTOR

EXAMPLE #1

Creates a vector (strided) datatype.

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

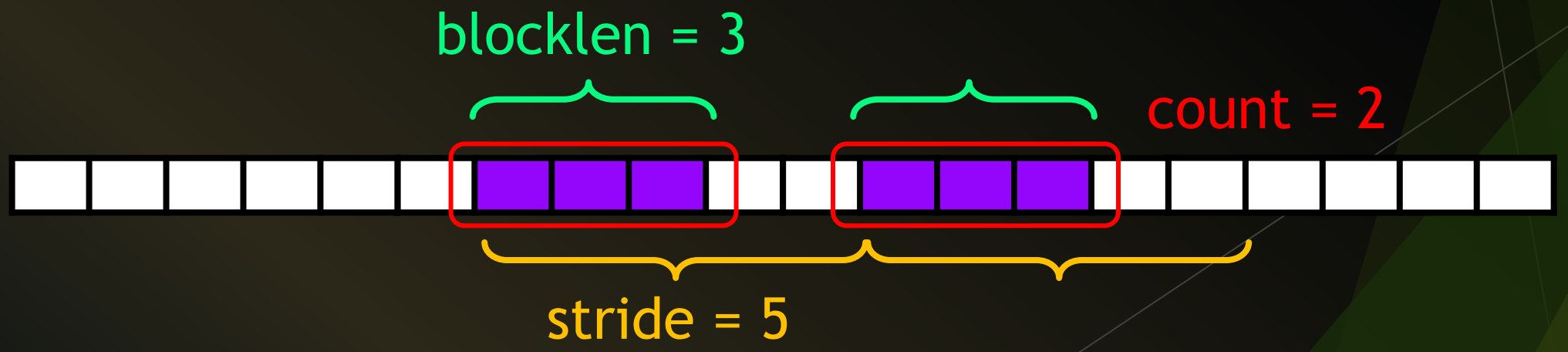
// count: Number of blocks.
> "How many chunks of data?"

// blocklens: Number of elements in each block.
> "How many things in a chunk?"

// stride: Number of elements between start of each block.
> "How many units (of things) between chunks?

// oldtype: Old datatype (handle).

// newtype: New datatype (handle).



MPI_TYPE_VECTOR

EXAMPLE #2

Creates a vector (strided) datatype.

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

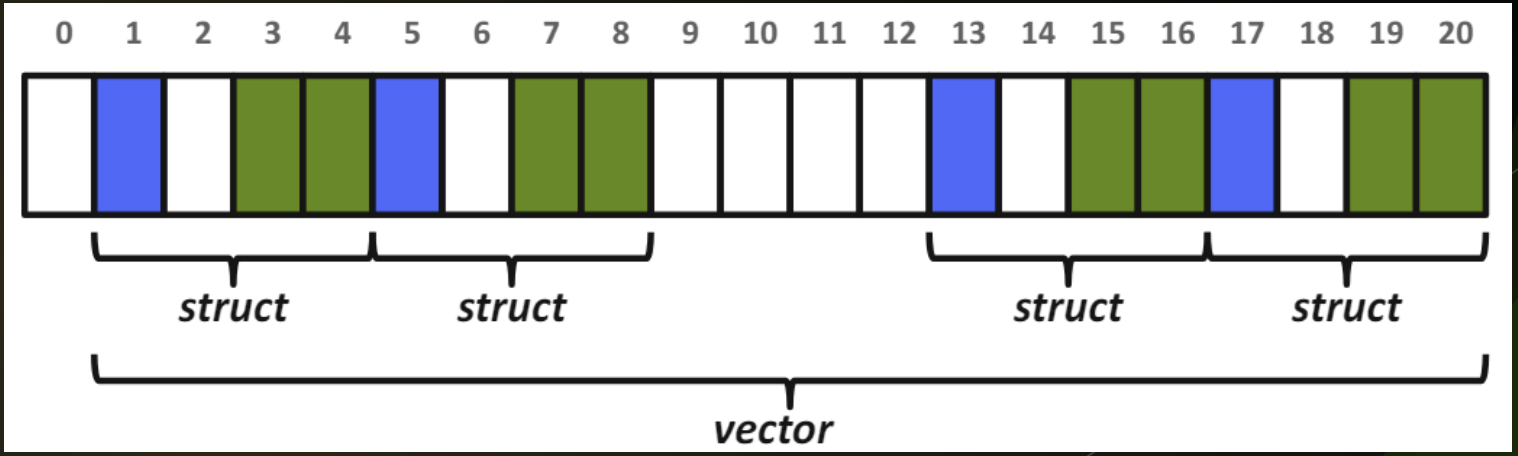
// count: Number of blocks.
> "How many chunks of data?"

// blocklens: Number of elements in each block.
> "How many things in a chunk?"

// stride: Number of elements between start of each block.
> "How many units (of things) between chunks?

// oldtype: Old datatype (handle).

// newtype: New datatype (handle).



MPI_TYPE_VECTOR

EXAMPLE #3

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

```
MPI_Datatype HALO_Y_DIR_TYPE;
int MPI_Type_vector(
                  by, // count
                   1, // blocklen
                bx+2, // stride
          MPI_DOUBLE, // old type
  &HALO_Y_DIR_TYPE);// new type
```

HANDS ON ACTIVITY #1

MPI_Type_contiguous

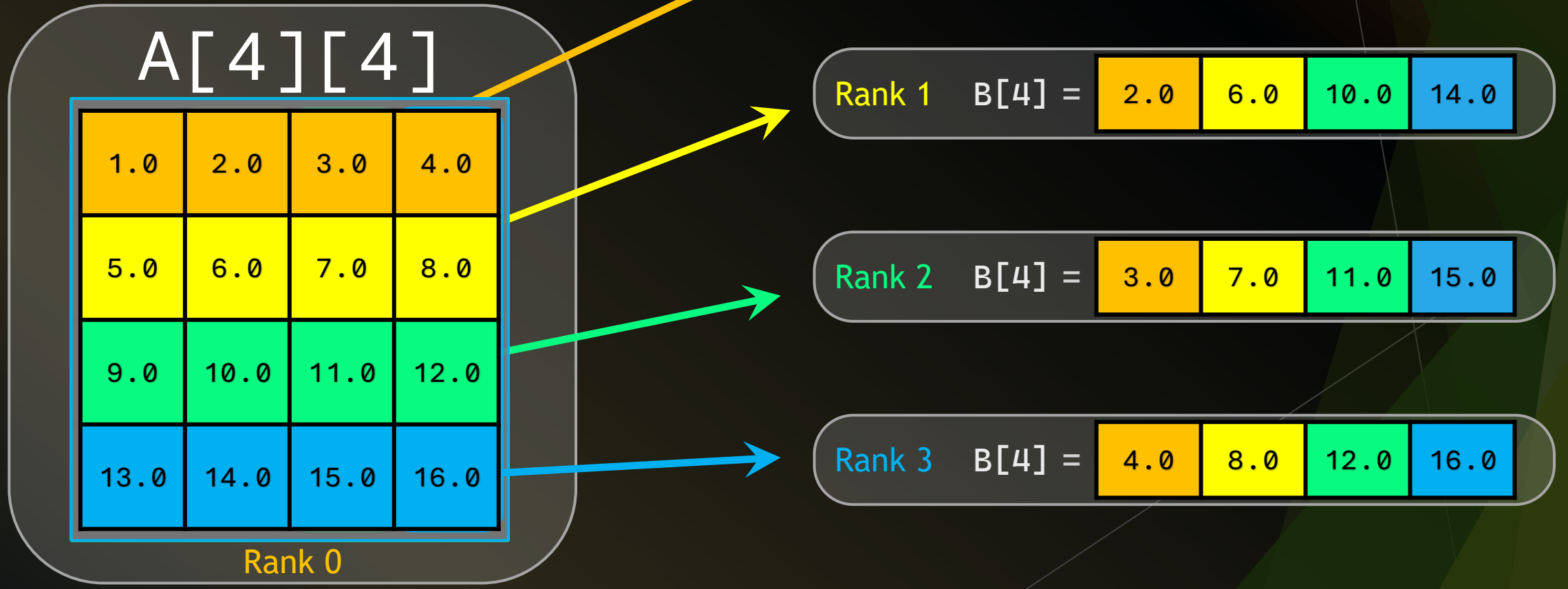
MPI_Type_vector

MPI_Type_indexed

MPI_Type_create_struct

EXERCISE.1 ddt.2.vector


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

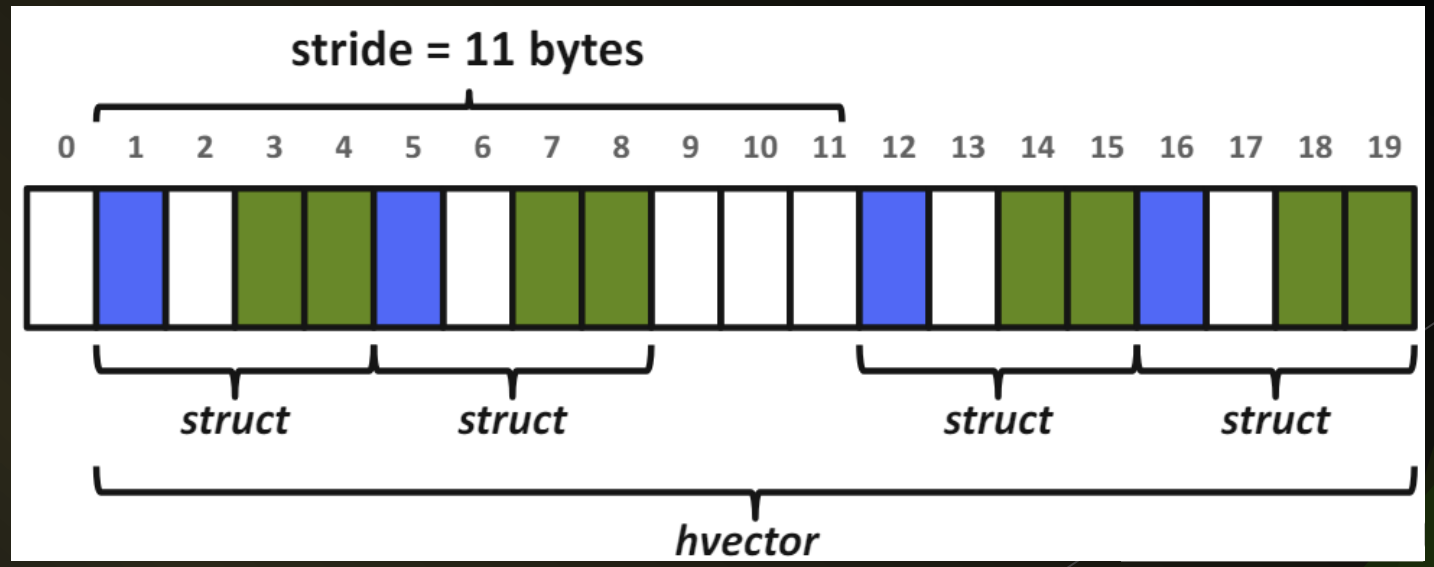


MPI_TYPE_CREATE_HVECTOR (MPI_TYPE_HVECTOR DEPRECATED)

Creates a vector (strided) data type with offset in bytes.

```
int MPI_Type_create_hvector(
    int count,
    int blocklen,
    MPI_Aint stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

```
// count :      Number of blocks.
// blocklens :  Number of elements in each block.
// stride :     Number of bytes between start of
                each block.
// oldtype :    Old datatype (handle).
// newtype :    New datatype (handle).
```

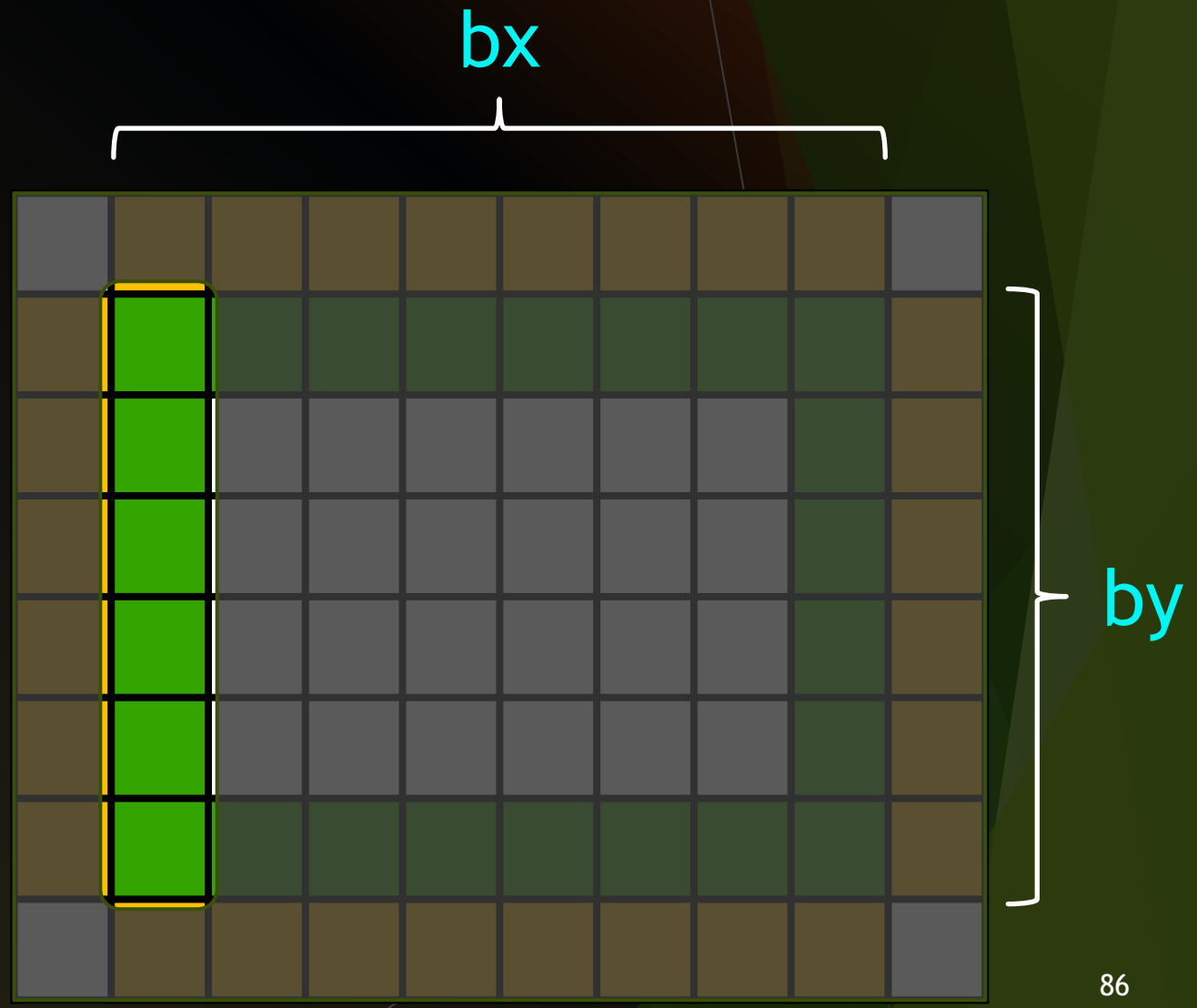


MPI_TYPE_CREATE_HVECTOR

EXAMPLE

```
int MPI_Type_create_hvector(
    int count,
    int blocklen,
    MPI_Aint stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

```
MPI_Datatype HALO_Y_DIR_HTYPE;
int MPI_Type_create_hvector(
                  by, // count
                   1, // blocklen
 (bx+2)*sizeof(double), // byte stride
          MPI_DOUBLE, // old type
 &HALO_Y_DIR_HTYPE);// new type
```



MPI_TYPE_INDEXED

Creates an indexed datatype.

```
int MPI_Type_indexed(
    int count,
    int *array_of_blocklens,
    int *array_of_displacements,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

```
// count:                Number of data blocks.
// array_of_blocklens:   Size of each block (ints).
// array_of_displacements:
                         Array of displacements (ints).
                         In units of extent of oldtype.
// oldtype:              Old datatype (handle).
// newtype:              New datatype (handle).
```

MPI_TYPE_INDEXED EXAMPLE

```
int blocklens[6] = {1,1,2,1,2,1};
int displace[6] = {0,3,5,9,13,17};
int MPI_Type_indexed(6,
            blocklens,
             displace,
           MPI_DOUBLE,
    &new_index_type);
```

```
// count:                Number of data blocks.
// array_of_blocklens: Size of each block (ints).
// array_of_displacements:
                         Array of displacements (ints).
                         In units of extent of oldtype.
// oldtype:              Old datatype (handle).
// newtype:              New datatype (handle).
```

HANDS ON ACTIVITY #1

MPI_Type_contiguous

MPI_Type_vector

MPI_Type_indexed

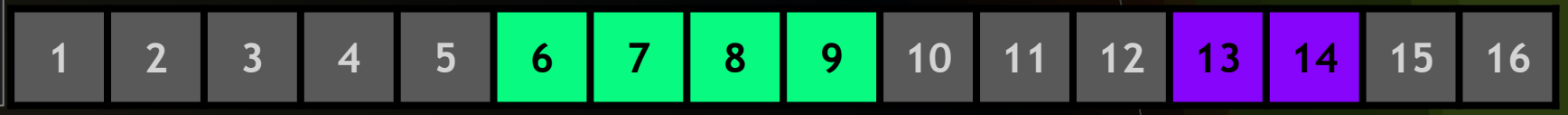
MPI_Type_create_struct

EXERCISE.1 ddt.3.indexed DERIVED DATA TYPES

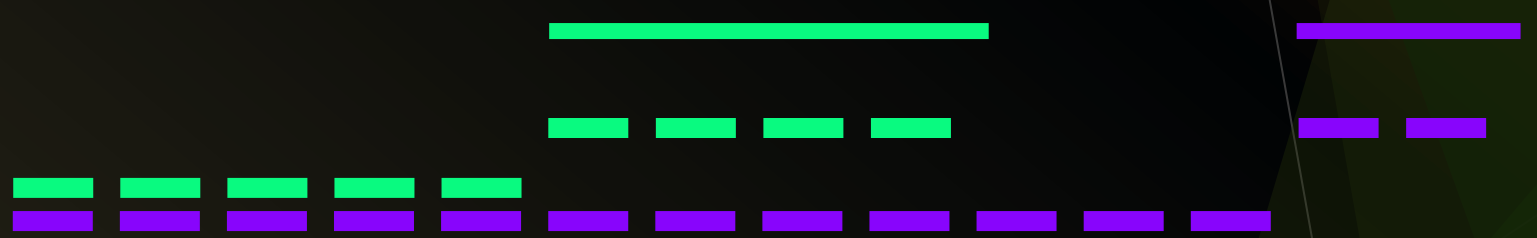
```
int MPI_Type_indexed(
 int count,
 int *array_of_blocklens,
 int *array_of_displacements,
 MPI_Datatype oldtype,
 MPI_Datatype *newtype);
```

Rank 0

A[16]



- Block Count
- Block Lengths
- Displacements



Rank 0 B[6] = 6 7 8 9 13 14

Rank 2 B[6] = 6 7 8 9 13 14

Rank 1 B[6] = 6 7 8 9 13 14

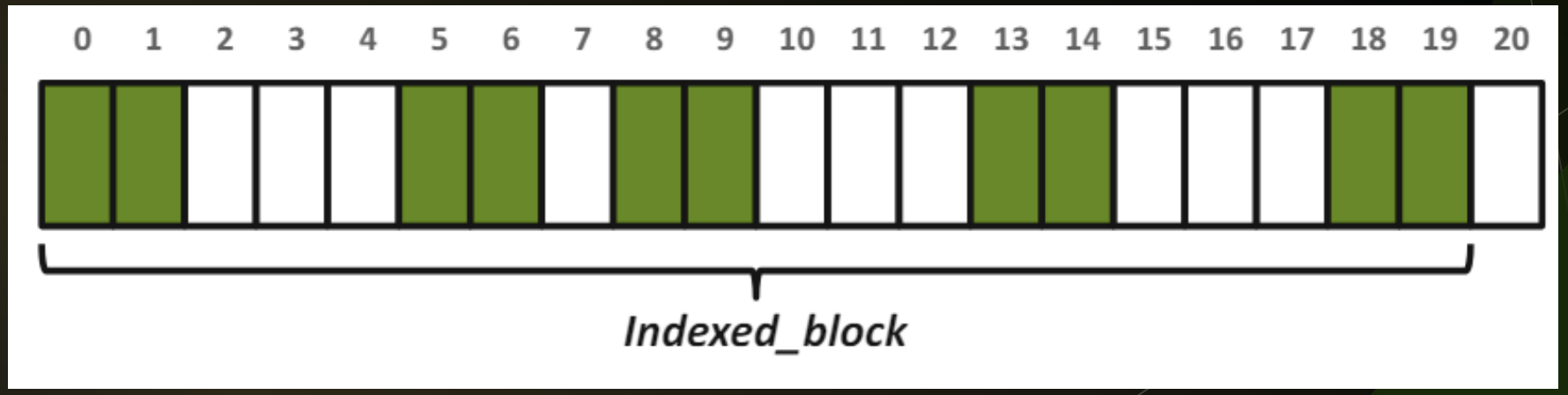
Rank 3 B[6] = 6 7 8 9 13 14

MPI_TYPE_CREATE_INDEXED_BLOCK

Creates an indexed data type with the same block length for all blocks.

```
int MPI_Type_create_indexed_block(
    int count,
    int blocklen,
    int *array_of_displacements,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

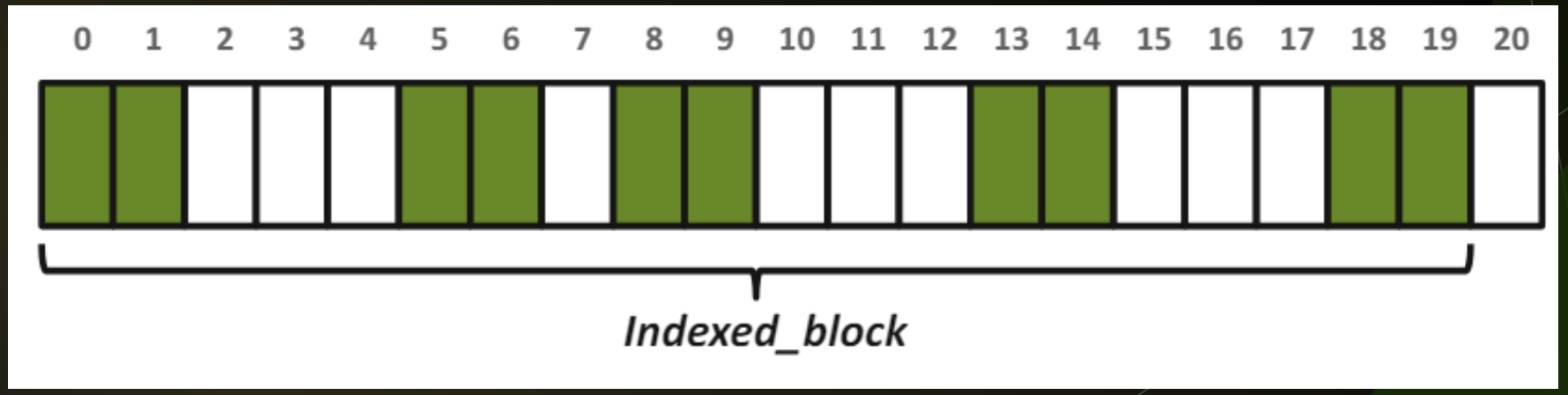
```
// count:                Length of array of displacements.
// blocklen:             Size of block (int).
// array_of_displacements:
                         Array of displacements (ints).
                         In units of extent of oldtype.
// oldtype:              Old datatype (handle).
// newtype:              New datatype (handle).
```



MPI_TYPE_CREATE_INDEXED_BLOCK EXAMPLE

```
int displacement[5] = {0,5,8,13,18};
int MPI_Type_create_indexed_block(
                                5,
                                2,
                    displacement,
                      MPI_DOUBLE,
     &new_indexed_block_type);
```

```
// count:          Length of array of displacements.
// blocklen:       Size of block (int).
// array_of_displacements:
                   Array of displacements (ints).
                   In units of extent of oldtype.
// oldtype:        Old datatype (handle).
// newtype:        New datatype (handle).
```



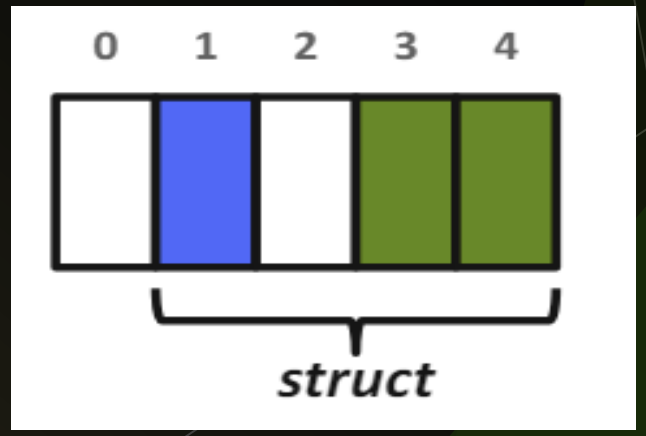
MPI_TYPE_CREATE_STRUCT

Creates a structured data type.

```
int MPI_Type_create_struct(
      int count,
      int *array_of_blocklens,
MPI_Aint *array_of_displacements,
    MPI_Datatype *array_of_types,
    MPI_Datatype *newtype);
```

```
// count:              Number of blocks (type entries).
// array_of_blocklens:  Array of block lengths.
// array_of_displacements:
                        Array of byte displacements.
// array_of_types: Array of old datatypes.
// newtype: New datatype (handle).
```

```
typedef struct {
    int id;
    char flag;
    float x, y;
} data_struct;
```



MPI_TYPE_EXTENT

For creating struct datatypes, we need

- need byte displacements/offsets

- MPI_Type_extent (not safe!)

```
struct {
    float x, y, z, velocity;
    int n, type;
} Particle;
Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);
count = 2;
blockcounts[0] = 4;          blockcount[1] = 2;
oldtypes[0]    = MPI_FLOAT;  oldtypes[1] = MPI_INT;
displ[0]       = 0;                  displ[1] = 4*extent;
```

```
int MPI_Type_create_struct(
     int count,
     int *array_of_blocklens,
MPI_Aint *array_of_displacements,
    MPI_Datatype *array_of_types,
    MPI_Datatype *newtype);
```

MIND THE ALIGNMENTS!

C struct may be automatically padded by the compiler!

```
struct {
    char a;
    int  b;
    char c;
} x;
```

```
struct {
    char a;
    char gap0[3];
    int  b;
    char c;
    char gap1[3];
} x;
```

MPI_GET_ADDRESS

- Good for handling inner padding issue…
  - But does not account for trailing struct padding
  - Important when sending more than one struct

```
struct {
    float x, y, z, velocity;
    int n, type;
} Particle;
Particle particles[NELEM];
```

```
MPI_Get_address(&Particle, &base);
MPI_Get_address(&Particle.x, &F_offset);
MPI_Get_address(&Particle.n, &I_offset);
count = 2;
blockcounts[0] = 4;          blockcount[1] = 2;
oldtypes[0]    = MPI_FLOAT;    oldtypes[1] = MPI_INT;
displ[0]       = F_offset - base;  displ[1] = I_offset - base;
```

```
int MPI_Type_create_struct(
     int count,
     int *array_of_blocklens,
MPI_Aint *array_of_displacements,
    MPI_Datatype *array_of_types,
    MPI_Datatype *newtype);
```

MPI_TYPE_CREATE_RESIZED

Returns a new data type with new extent and upper and lower bounds.

```
int MPI_Type_create_resized(
    MPI_Datatype oldtype,
    MPI_Aint lwrbnd,
    MPI_Aint extent,
    MPI_Datatype *newtype);
```

```
// oldtype:   Input data type (handle).
// lwrbnd:    New lower bound of
              data type (# of bytes).
// extent:    New extent of data type
              (# of bytes).
// newtype:   New datatype (handle).
```

- Returns new datatype that is identical to oldtype,
  - except the lower bound of this new datatype is set to be “lb”,
  - and its upper bound is set to be “lb + extent”.

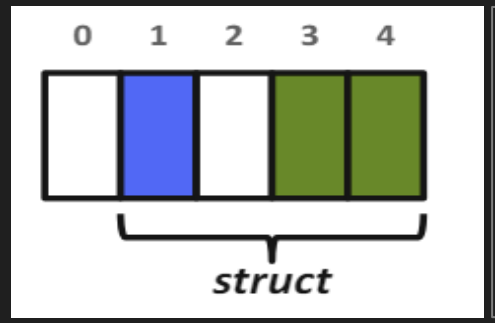
MPI_TYPE_CREATE_RESIZED EXAMPLE

```
/* Sending an array of structs portably */
struct PartStruct particle[100];
MPI_Datatype ParticleType;

/* check if extent is correct */
MPI_Type_get_extent(ParticleType, &lb, &extent);
if (extent != sizeof(particle[0])) {
    MPI_Datatype old = ParticleType;
    MPI_Type_create_resized(old, // [in]  old datatype
                            0, // [in]  new lower bound byte offset
            sizeof(particle[0]), // [in]  new byte extent of data type
                 &ParticleType); // [out] new datatype
    MPI_Type_free(&old);
}
MPI_Type_commit(&ParticleType);
```

MPI_TYPE_CREATE_STRUCT, REVISITED

EXAMPLE

```
data_struct Data;
MPI_Aint base, member_offset;
MPI_Aint offsets[3];
int blockcounts[3];
MPI_Datatype oldtypes[3];

// compute base address in struct
MPI_Get_address(&Data, &base);

// description of the 1 MPI_INT field: id
MPI_Get_address(&Data.id, &member_offset);
offsets[0] = member_offset - base;
blockcounts[0] = 1;
oldtypes[0] = MPI_INT;

// description of the 1 MPI_CHAR field: flag
MPI_Get_address(&Data.flag, &member_offset);
offsets[1] = member_offset - base;
blockcounts[1] = 1;
oldtypes[1] = MPI_CHAR;

// description of the 2 MPI_FLOAT fields: x,y
MPI_Get_address(&Data.x, &member_offset);
offsets[2] = member_offset - base;
blockcounts[2] = 2;
oldtypes[2] = MPI_FLOAT;

MPI_Datatype new_struct_type;
int MPI_Type_create_struct(
                3, // # of blocks
    blockcounts, // # of elements in each block
        offsets, // Byte displacements of each block
       oldtypes, // Type of elements in each block
&new_struct_type); // New data type

// check if extent is correct of new_struct_type
MPI_Type_get_extent(new_struct_type, &lb, &extent);
...
```




```
typedef struct {
    int id;
    char flag;
    float x, y;
} data_struct;
```

HANDS ON ACTIVITY #1

MPI_Type_contiguous

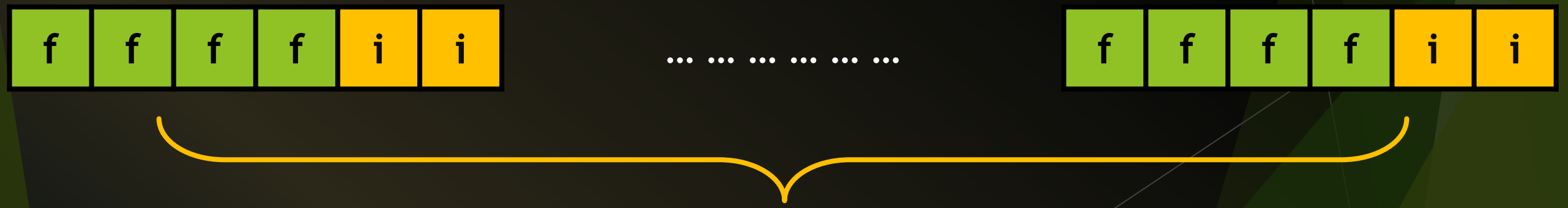
MPI_Type_vector

MPI_Type_indexed

MPI_Type_create_struct

```
int MPI_Type_create_struct(
      int count,
      int *array_of_blocklens,
MPI_Aint *array_of_displacements,
MPI_Datatype *array_of_types,
MPI_Datatype *newtype);
```

```
typedef struct {
    float x, y, z, velocity;
    int n, type;
} Particle;

Particle particles[NELEM];
```



particles[NELEM];

MPI_OP_CREATE

Creates a user-defined combination function handle.

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

// function: User-defined function.
// commute: True if commutative; false otherwise.
// op : Custom operation (handle).

```
MPI_Op           op;
MPI_Datatype     ctype;
mpiret = MPI_Type_contiguous(7, MPI_DOUBLE, &ctype);
mpiret = MPI_Type_commit(&ctype);
```

```
mpiret = MPI_Op_create((MPI_User_function *) stats_mpifunc, 1, &op);
mpiret = MPI_Allreduce(flatin, flatout, nvars, ctype, op, mpicomm);
```

MPI_OP_CREATE

Creates a user-defined combination function handle.

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

```
// function:    User-defined function.
// commute:     True if commutative;
                false otherwise.
// op :         Custom operation (handle).
```

```
mpiret = MPI_Op_create((MPI_User_function *) stats_mpifunc, 1, &op);
mpiret = MPI_Allreduce(flatin, flatout, nvars, ctype, op, mpicomm);
```

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

MPI_OP_CREATE

EXAMPLE

```c
// Define Custom Collective Function
void accumulate_values_func(void *invec,
                            void *inoutvec,
                             int *len,
                    MPI_Datatype *datatype){
  const int IX=0, IY=1, IZ=2, NINDEX=3;
  double *in = (double *) invec;
  double *inout = (double *) inoutvec;

  /* sum values */
  for (int i = 0; i < *len; ++i) {
    inout[IX] += in[IX];
    inout[IY] += in[IY];
    inout[IZ] += in[IZ];

    /* advance to next data set */
    in += NINDEX;
    inout += NINDEX;
  }
}
```

```c
// Construct MPI Type & Op
MPI_Op op;
MPI_Datatype ctype;
double values[3*npoints];     // TODO: init
double value_sums[3*npoints]; // TODO: zero

/* create contiguous type & commit */
MPI_Type_contiguous(3, // stack 3 doubles
                    MPI_DOUBLE,
                    &ctype);
MPI_Type_commit(&ctype);

/* create MPI Op & perform Allreduce */
MPI_Op_create(
  (MPI_User_function *) accumulate_values_func,
               1, // commutative operation
           &op); // [out] MPI Op (handle)

MPI_Allreduce(values, value_sums, npoints,
              ctype, op, comm);
```

DDT REVIEW: SELECTION ORDER

Simple and effective performance model

- More parameters == slower performance

predefined < contig < vector < indexed_block < index < struct

Advice to developers

- Construct datatypes hierarchically bottom-up
- Awesome Tool made by Ludovic Capelli!
  https://rookiehpc.org/mpi/tools/datatype_creator/index.html

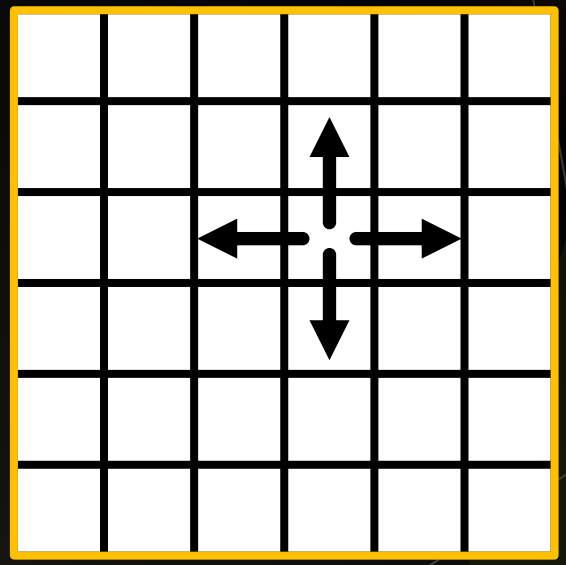
SECTION III

VIRTUAL TOPOLOGIES

STRUCTURED MESHES

A *structured mesh* is
a mesh where one can find a connectivity pattern that applies to *every* element in the mesh.

Typical Communication Pattern

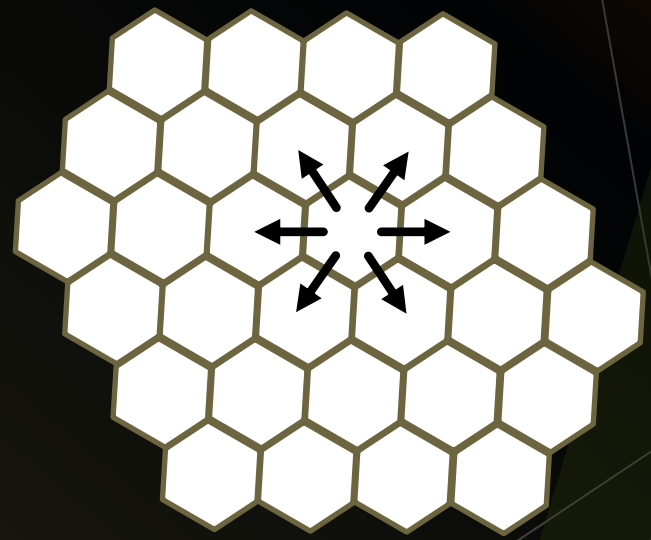


4 Neighbors

STRUCTURED MESHES

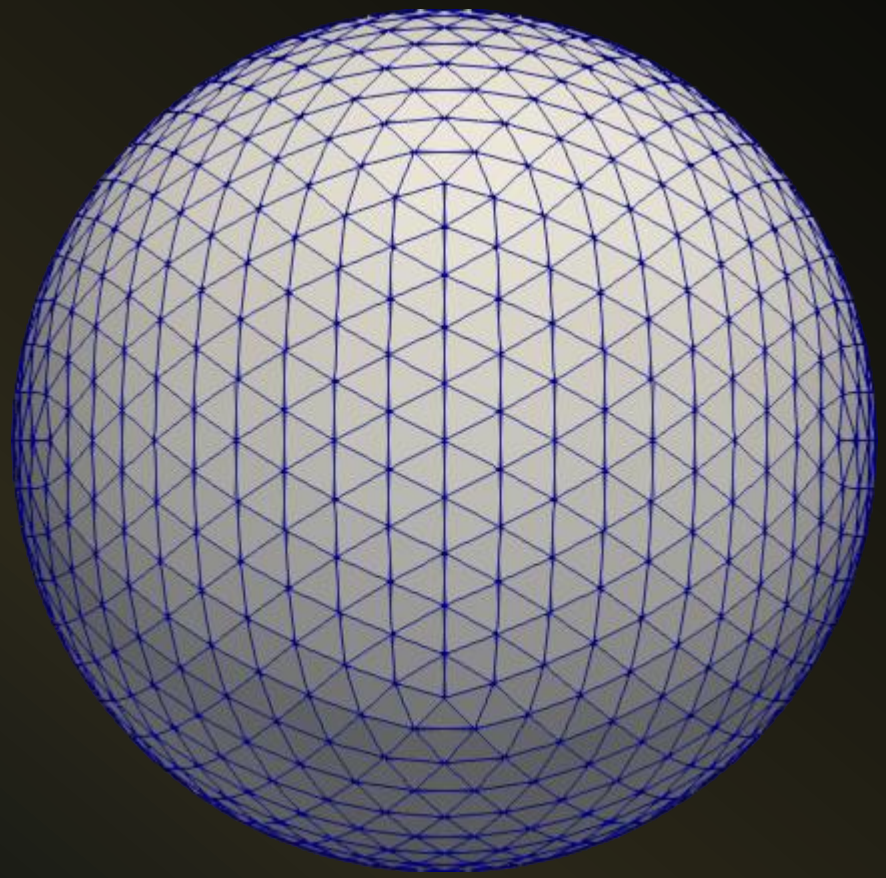
A *structured mesh* is
a mesh where one can find a connectivity pattern that applies to *every* element in the mesh.

Typical Communication Pattern



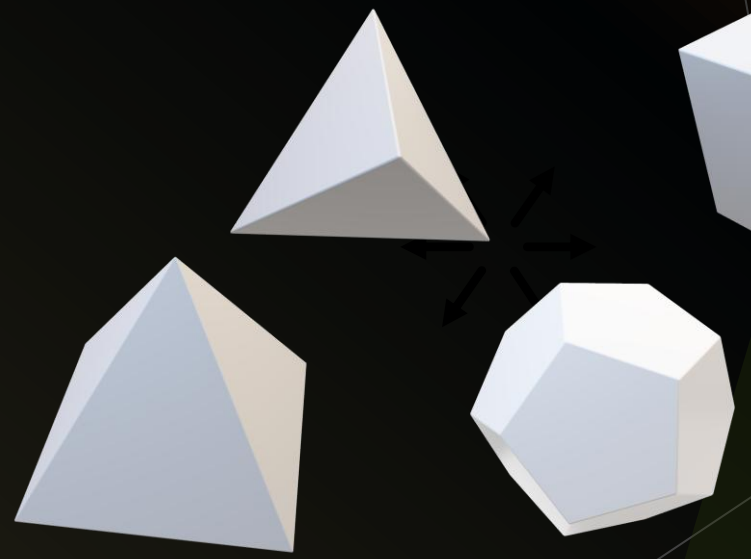
6 Neighbors

STRUCTURED MESHES



Tetrahedrons

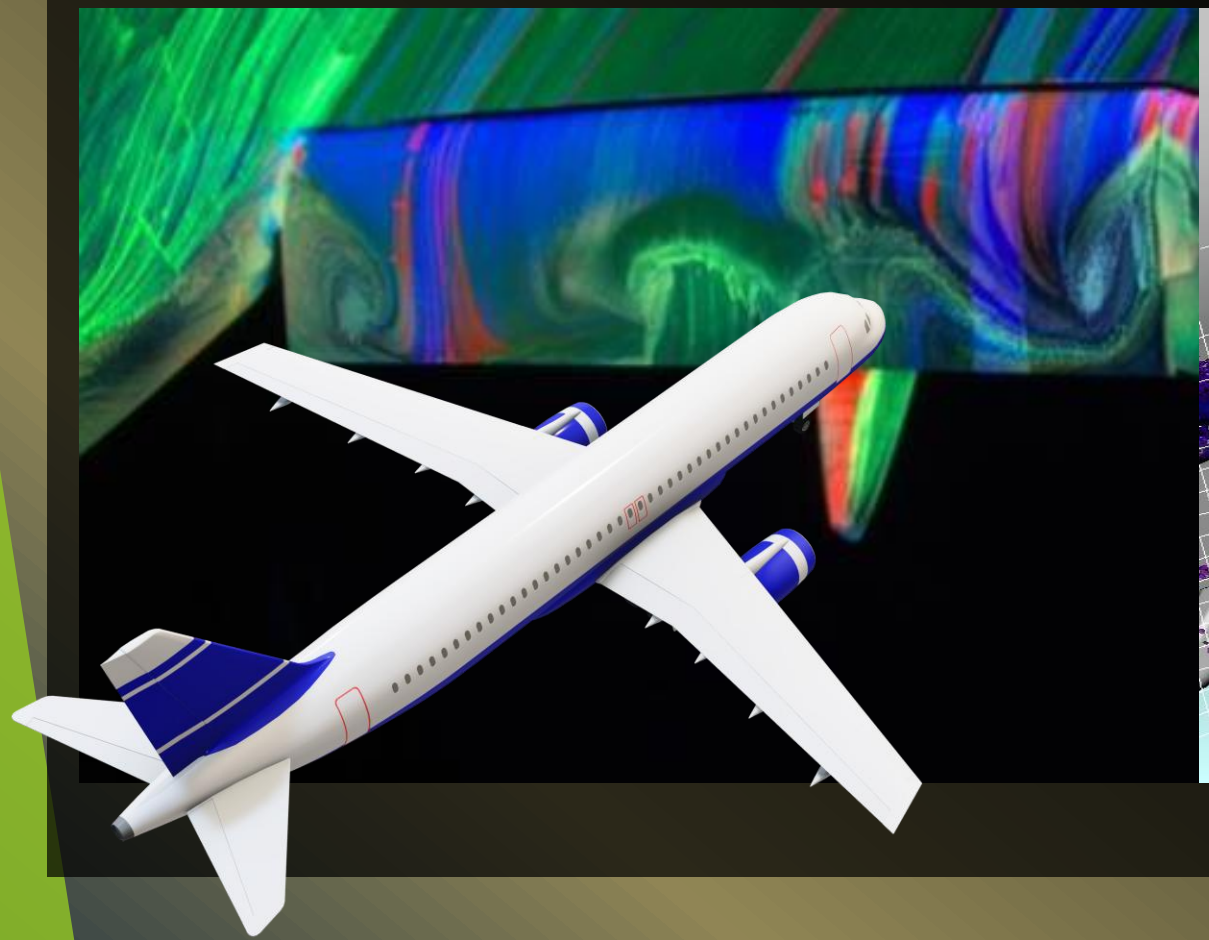
Hexahedrons

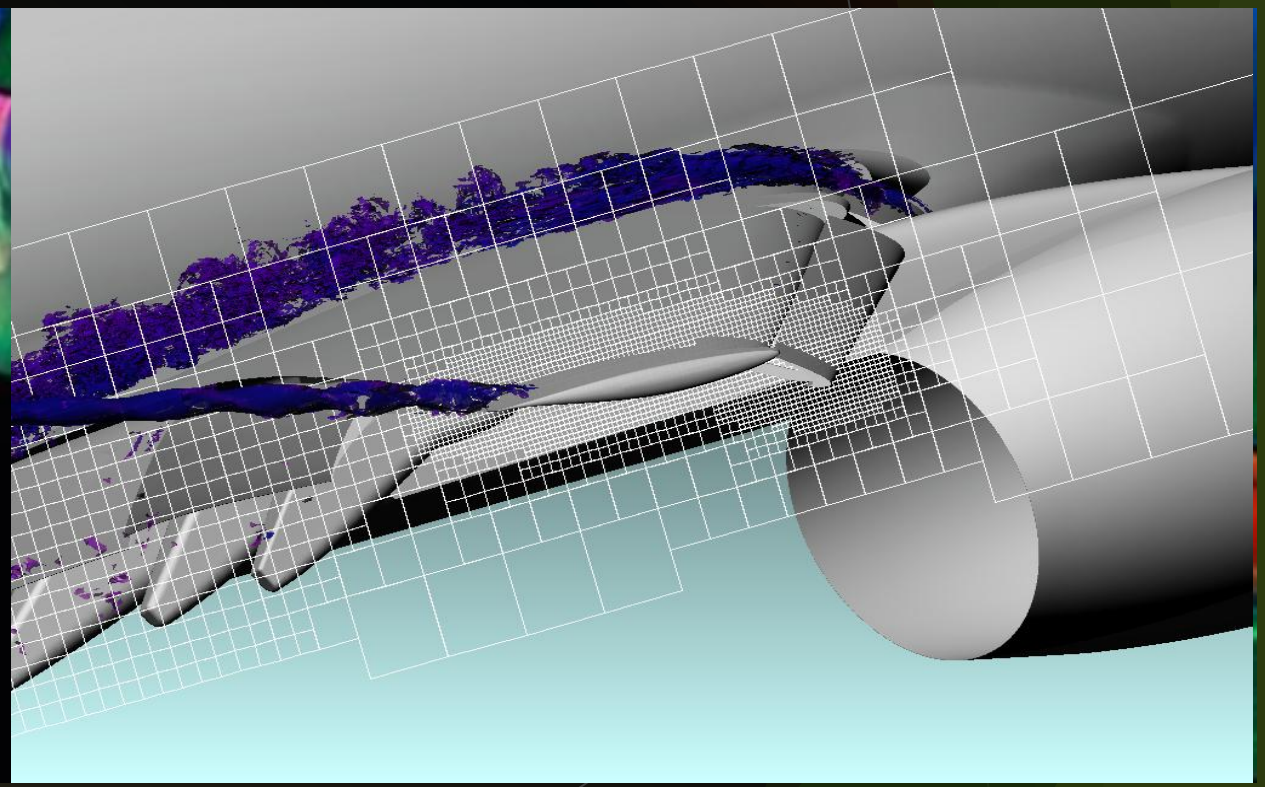


Pyramids

Dodecahedrons

EXAMPLE: COMPUTATIONAL FLUID DYNAMICS

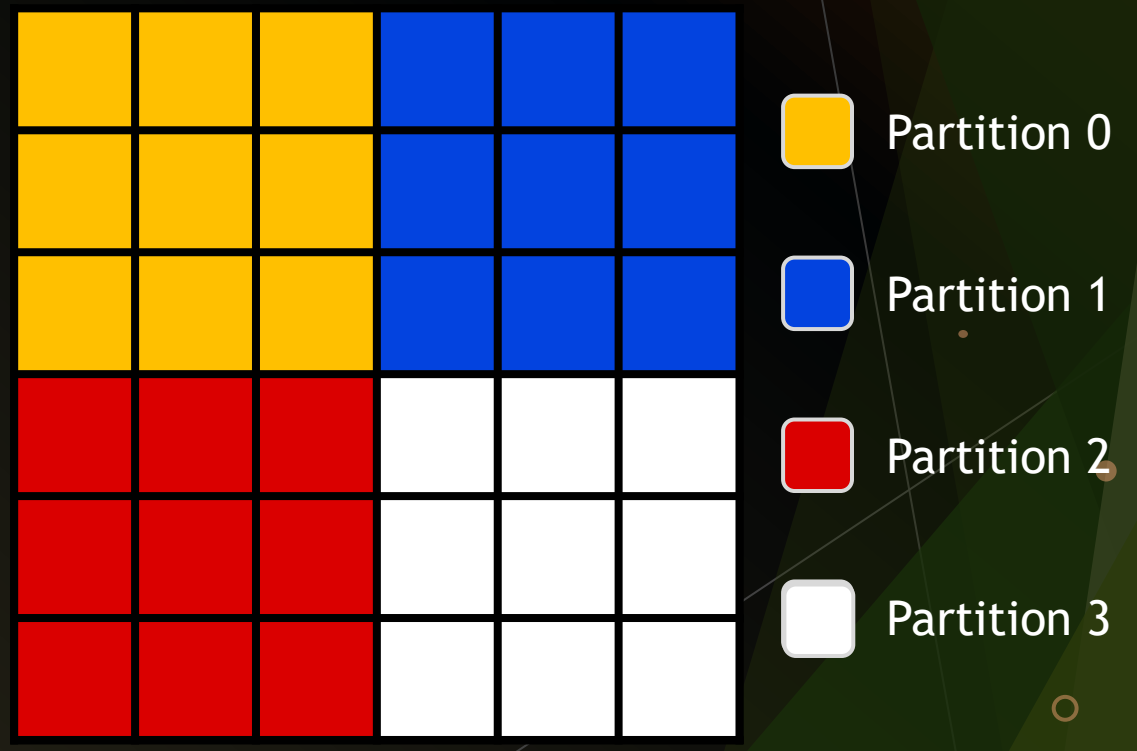




LOAD BALANCING

Basic Partitioning

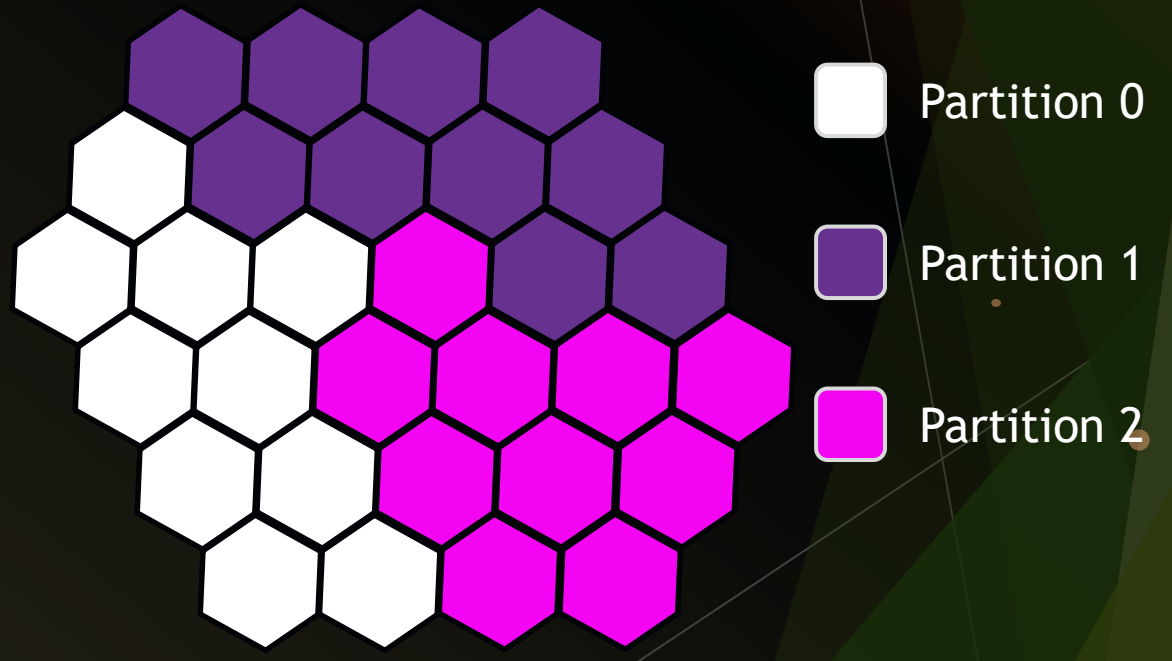
Load balancing a structured mesh typically comes down to partitioning the mesh to have an equal number of elements per Processing Element (PE).



LOAD BALANCING

Basic Partitioning

Load balancing a structured mesh typically comes down to partitioning the mesh to have an equal number of elements per Processing Element (PE).



LOAD BALANCING: A LITTLE LESS TRIVIAL

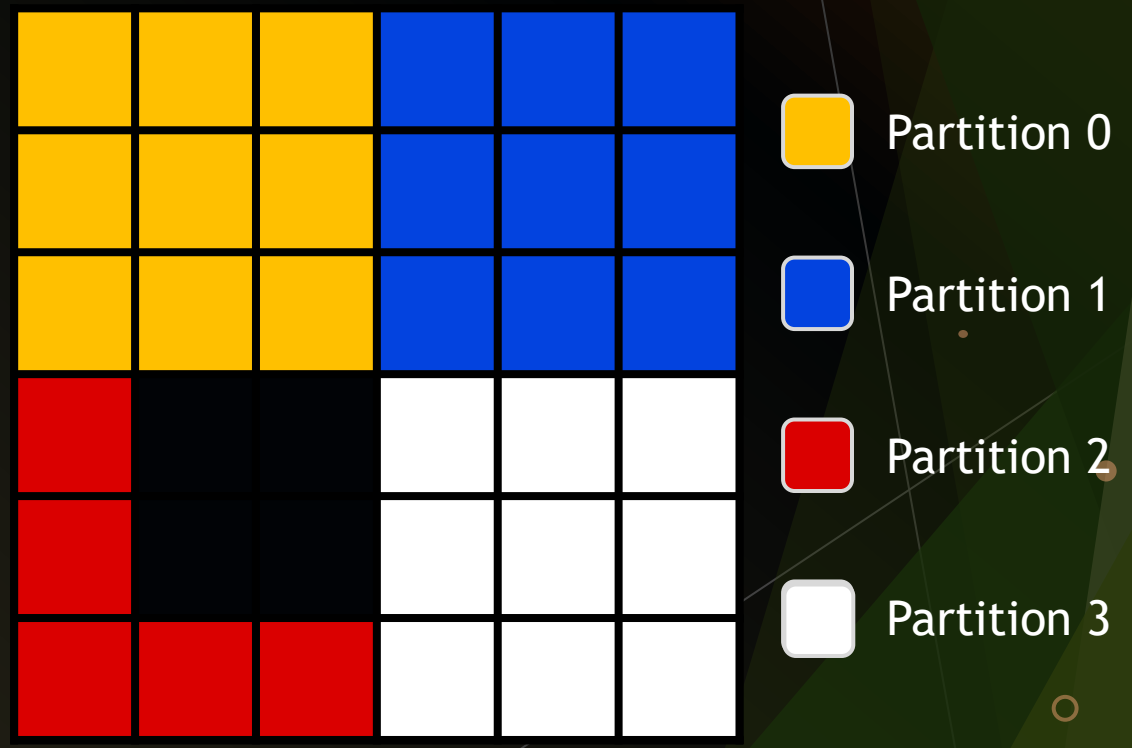
Basic Partitioning

Pros

- ✓ No need to alter the underlying decomposition.

Cons

- ➢ Load-imbalance present: 3 MPI processes have 9 elements, whereas 1 MPI process has 5 elements.



LOAD BALANCING: A LITTLE LESS TRIVIAL

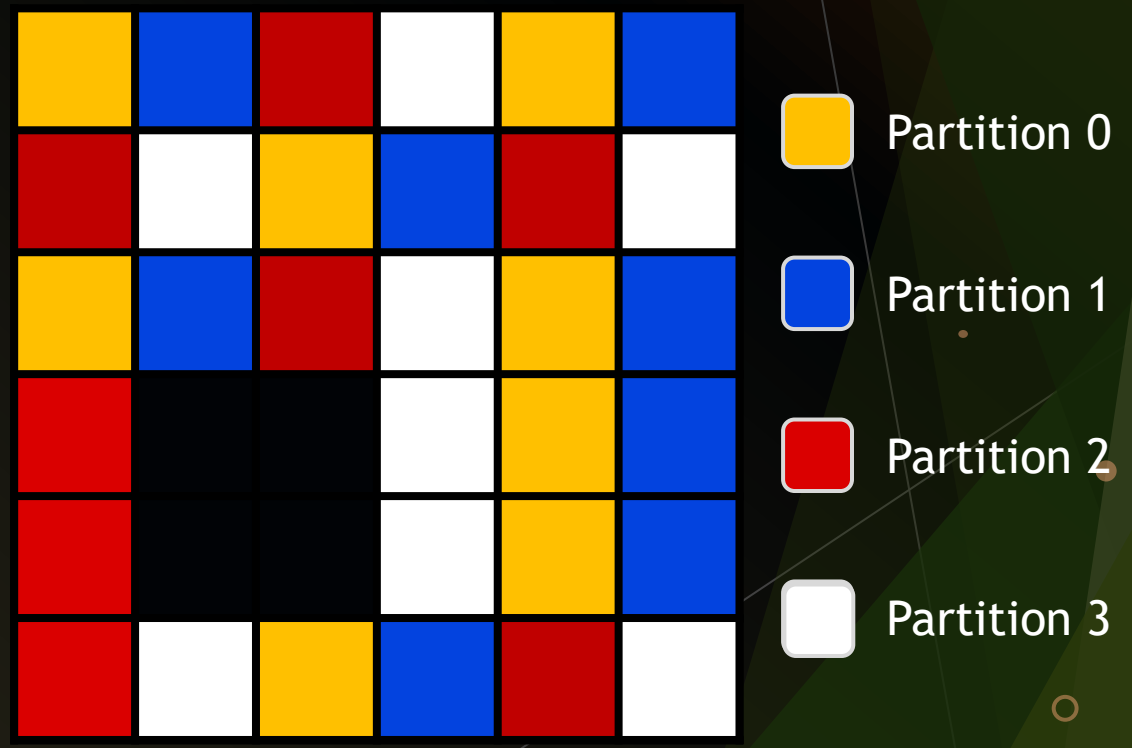
Cyclic Partitioning

Pros

- ✓ Every MPI process is assigned 8 elements.

Cons

- ➢ Hindered data locality as elements are rarely consecutive.



LOAD BALANCING: A LITTLE LESS TRIVIAL

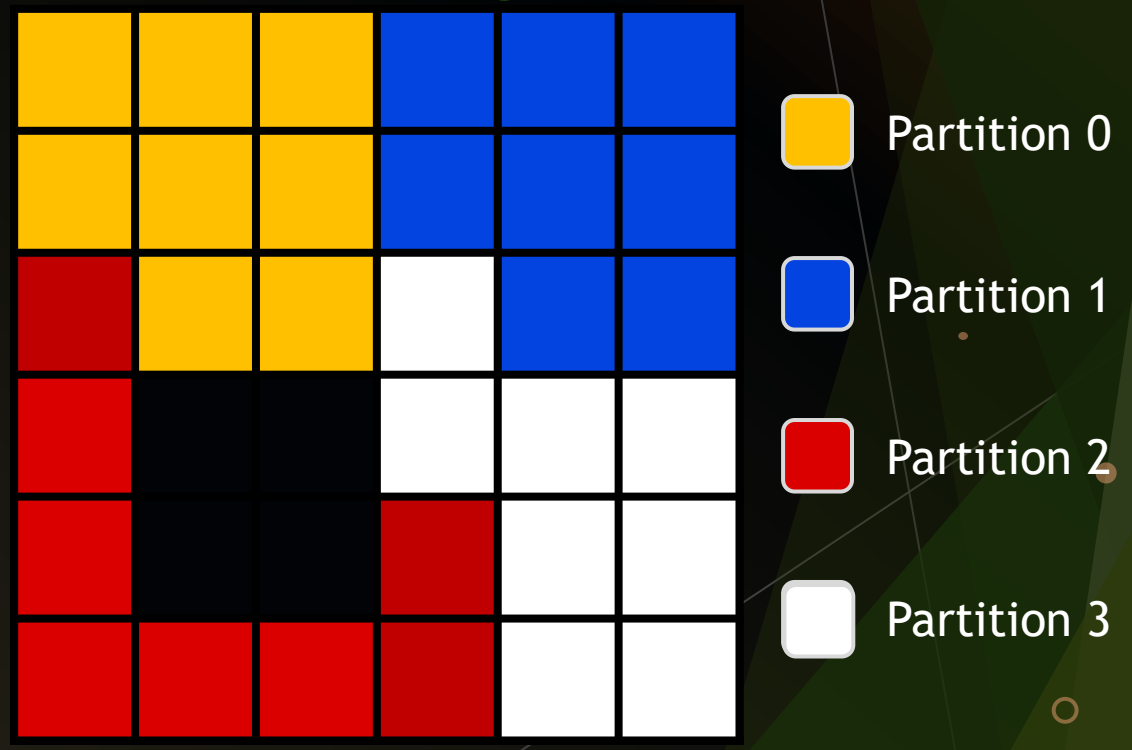
Advanced Partitioning

Pros

- ✓ Every PE has an identical number of elements (8).
- ✓ Most elements are consecutive

Cons

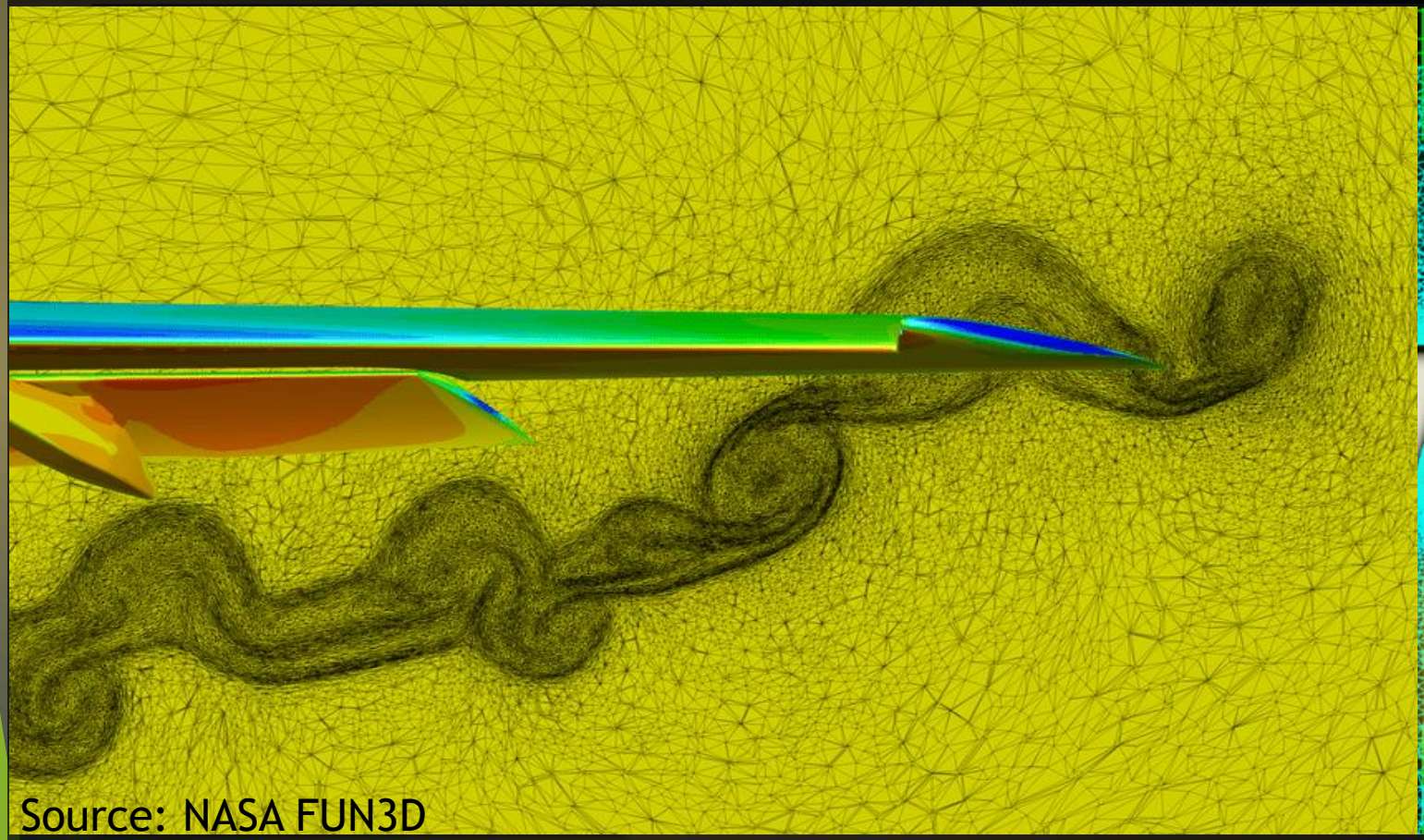
- ➢ More complex.
- ➢ Takes more time to complete.



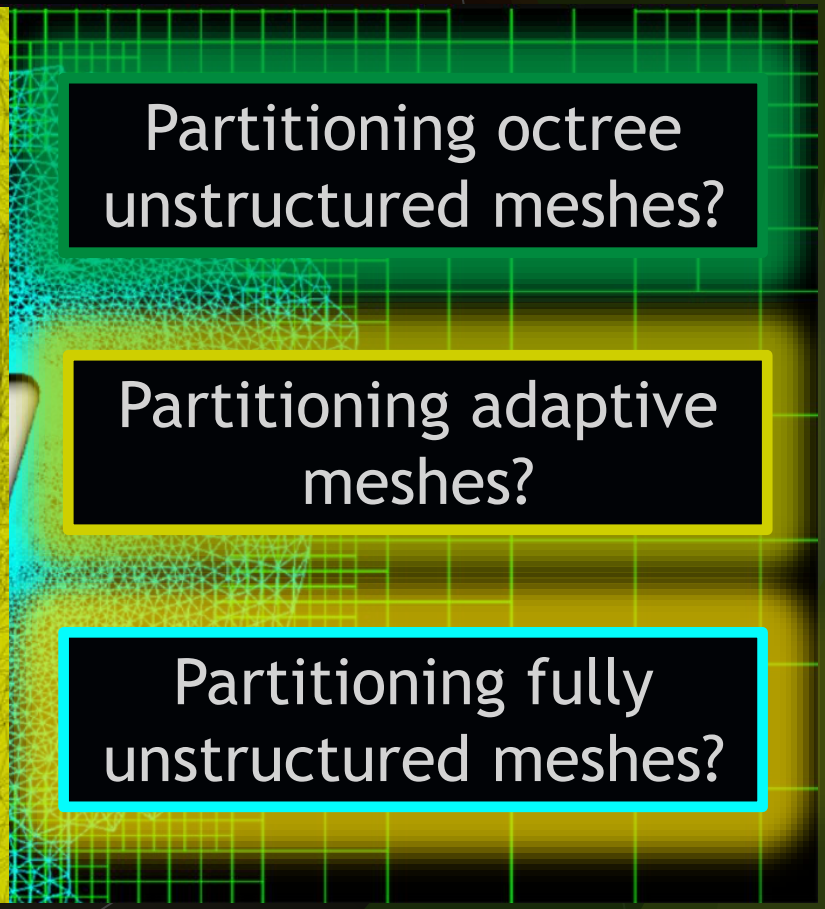
PARTITIONING YOUR PROBLEM: HARD.

In general,
user is responsible for partitioning!

UNSTRUCTURED MESHES



Source: NASA FUN3D



Partitioning octree unstructured meshes?

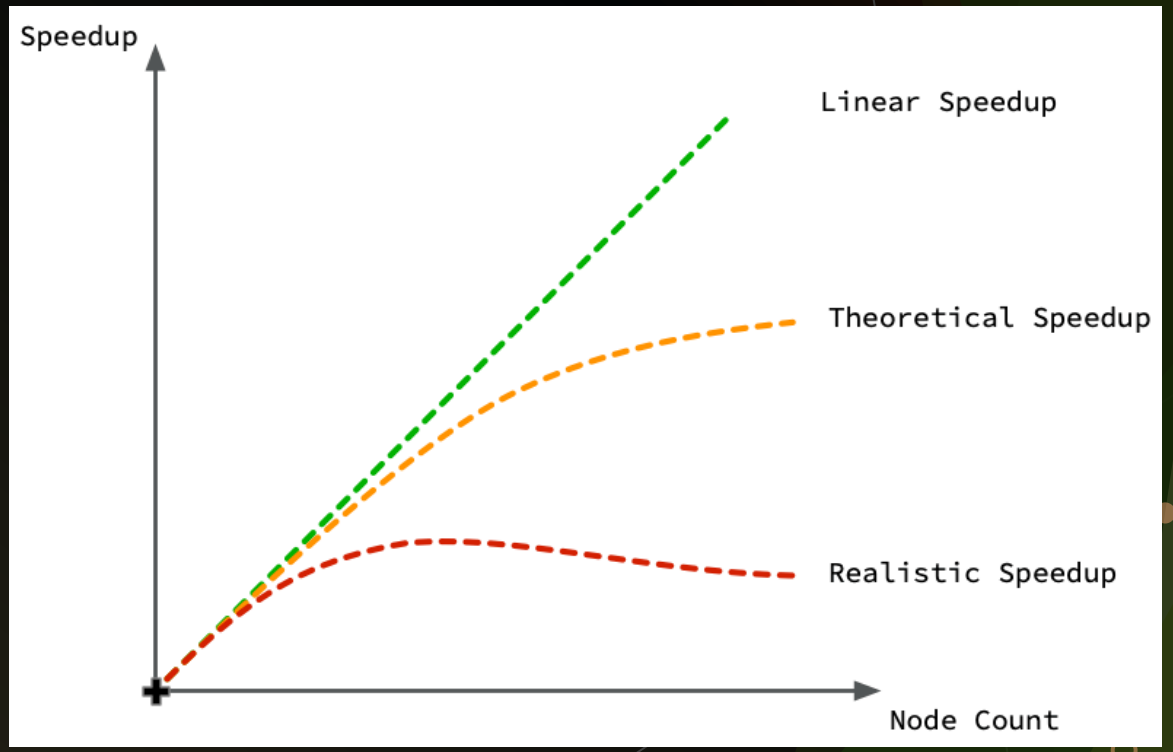
Partitioning adaptive meshes?

Partitioning fully unstructured meshes?

WHY IS IT IMPORTANT? PARALLEL PERFORMANCE

$$\text{speedup} = \frac{\text{wallclock time for serial}}{\text{wallclock time for parallel}}$$

$$\text{parallel efficiency} = \frac{\text{speedup}}{\text{number of cores}}$$



VIRTUAL TOPOLOGY: WHAT? WHY?

What is it?

A virtual topology is the arrangement of MPI processes.

Why is it useful?

- ✓ Provide convenient and intuitive constructions,
  e.g., finding ranks with shifts in a Cartesian virtual topology.
- ✓ Inform the MPI runtime of the communications patterns –
  improve the placement of MPI processes to hardware.

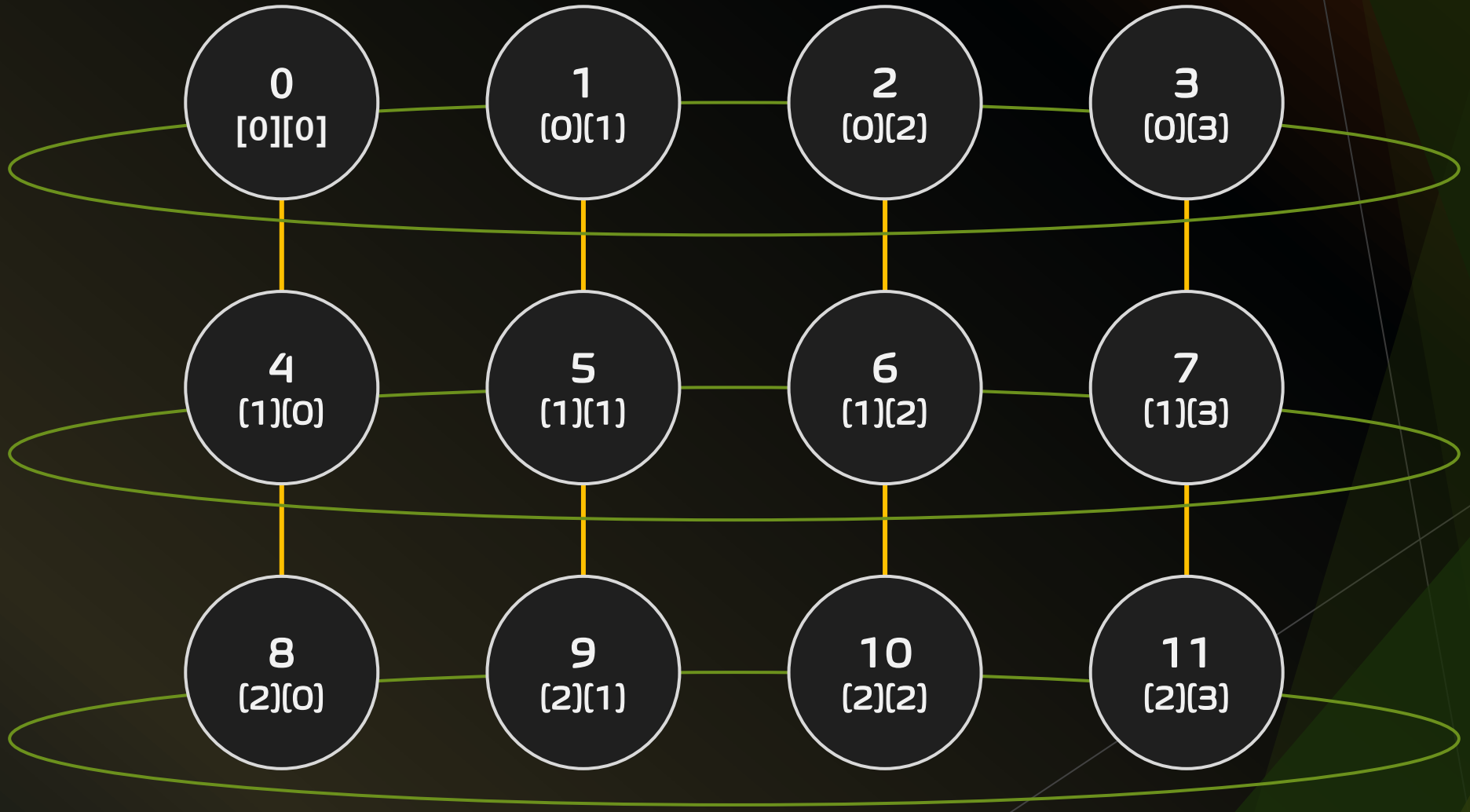
VirTops are not required (c.f.: do communications yourself), but

- ✓ Simplifies writing of code.

Better Performance

VIRTUAL TOPOLOGY: CARTESIAN

Goal: Make a new communicator with Cartesian topology information attached.



MPI_CART_CREATE

Makes a new communicator to which Cartesian topology information has been attached.

```
int MPI_Cart_create(
    MPI_Comm comm_old,
    int ndims,
    const int dims[],
    const int periods[],
    int reorder,
    MPI_Comm *comm_cart);
```

// comm_old:	Input communicator.
// ndims:	Number of dimensions of Cartesian grid (>= 0).
// dims:	Array of size ndims specifying number of processes in each dimension.
// periods:	Array of size ndims specifying whether dim is periodic.
// reorder:	Rank reordering flag.
// comm_cart:	Output communicator with Cart topology.

MPI_DIMS_CREATE

Partitions Cartesian meshes.

Creates a division of processors in a Cartesian grid.

```
int MPI_Dims_create(
    int nnodes,
    int ndims,
    int dims[]);
```

// nnodes: Number of nodes in grid.
// ndims: Number of Cartesian Topology dimensions.
// dims: Array of size ndims specifying number of nodes in each dimension.

- Nonzero values in dims sets the number of processes required in that direction.

WARNING: make sure dims is set to 0 before the call

dims before call	Function call	dims on return
(0, 0)	MPI_Dims_create (6, 2, dims);	(3, 2)
(0, 0)	MPI_Dims_create (7, 2, dims);	(7, 1)
(0, 3, 0)	MPI_Dims_create (6, 3, dims);	(2, 3, 1)
(0, 3, 0)	MPI_Dims_create (7, 3, dims);	Erroneous call

VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

MPI_Cart_rank - Determines process rank in communicator given Cartesian location.

```
int MPI_Cart_rank(
    MPI_Comm comm_cart,
    int coords[],
    int *rank);
```

```
int coords[2] = {1,2};
MPI_Cart_rank(comm_cart,
              coords,
             &rank);
>> rank = 6
```

VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

MPI_Cart_coords - Determines process coords in Cartesian topology given rank in group.

```
int MPI_Cart_coords(
    MPI_Comm comm_cart,
    int rank,
    int maxdims,
    int coords[]);
```

```
int rank = 6, maxdims = 2;
MPI_Cart_coords(comm_cart,
                rank,
                maxdims,
                coords);
>> coords[] = {1,2}
```

VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

MPI_Cart_shift - Returns the shifted source and destination ranks, given a shift direction and amount.

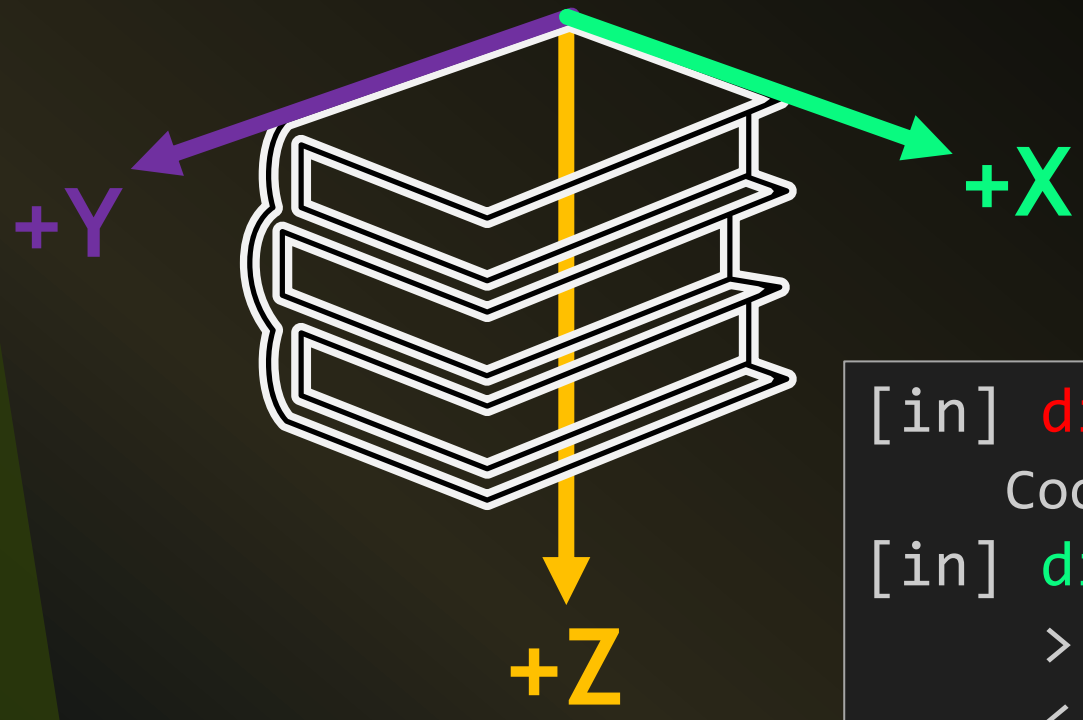
```
int MPI_Cart_shift(
    MPI_Comm comm_cart,
    int direction,
    int displacement,
    int *rank_source,
    int *rank_dest);
```

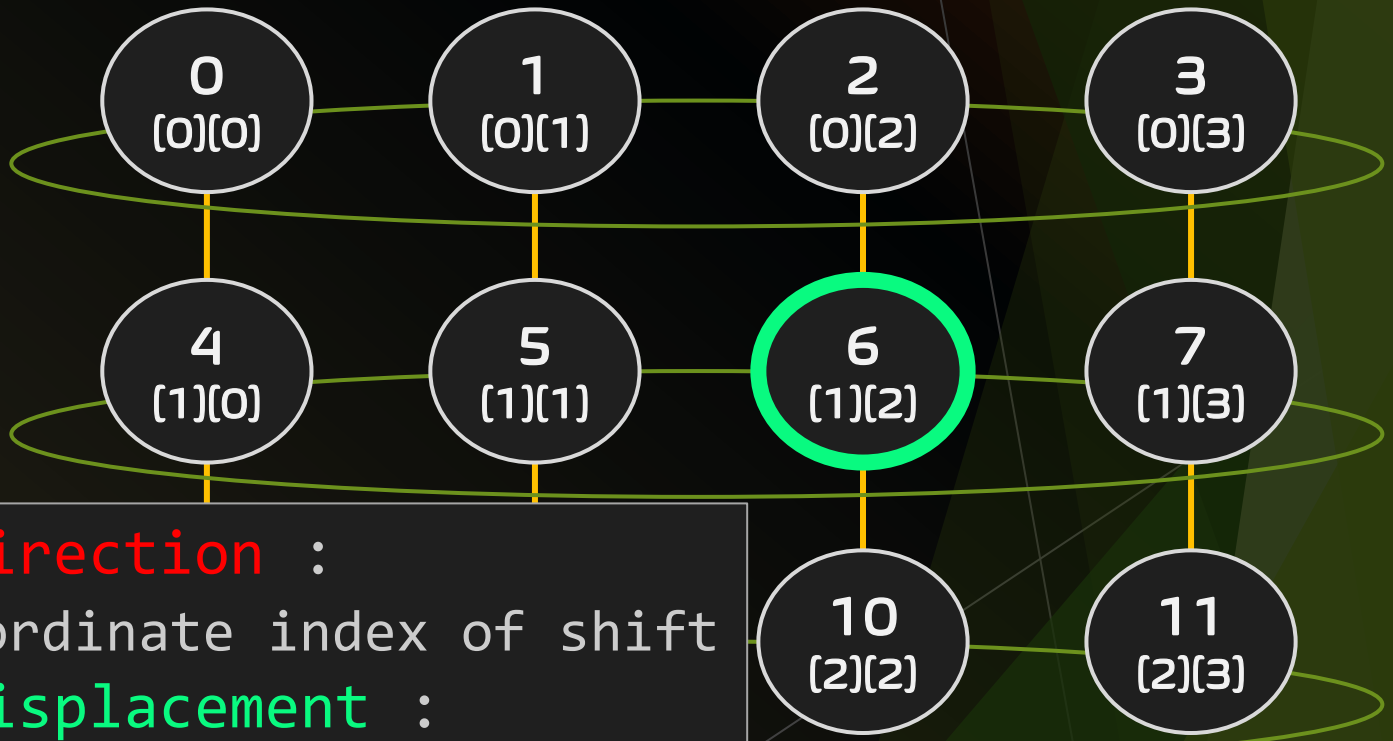
```
[in] direction :
    Coordinate index of shift
[in] displacement :
     > 0: positive shift,
     < 0: negative shift
```

VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

MPI_Cart_shift - Returns the shifted source and destination ranks, given a shift direction and amount.

LEXICOGRAPHIC ORDERING





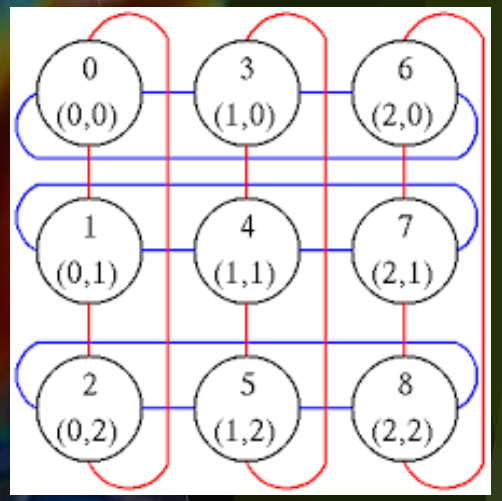

```
[in] direction :
     Coordinate index of shift
[in] displacement :
      > 0: positive shift,
      < 0: negative shift
```

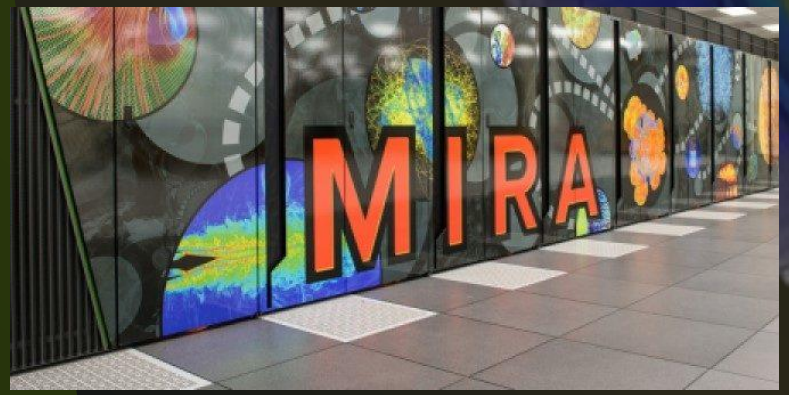
MPI_CART_SHIFT: PARALLEL SCALING

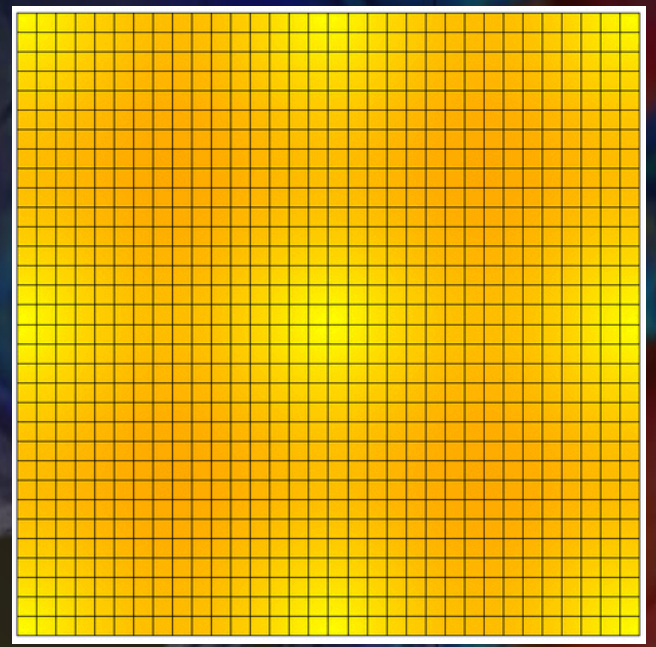
Strong Scaling on ALCF Mira

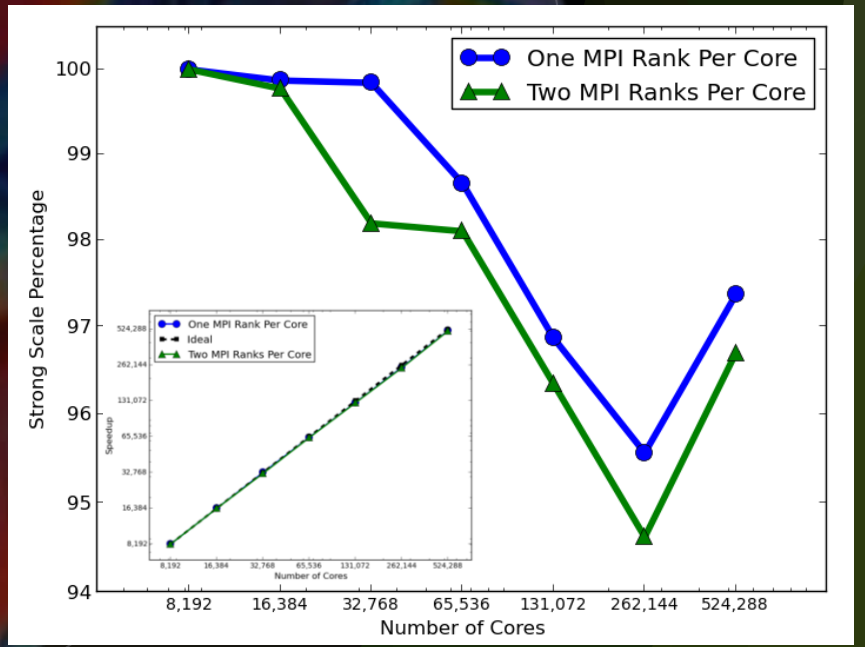
- Taylor-Green Vortex
- Fully periodic
- Periodic Mesh: [512, 512, 512]
- Discontinuous Galerkin: 5th -order
- 16.8 Billion DOFs
  - 83.9 Billion unknowns

1 Million MPI Ranks



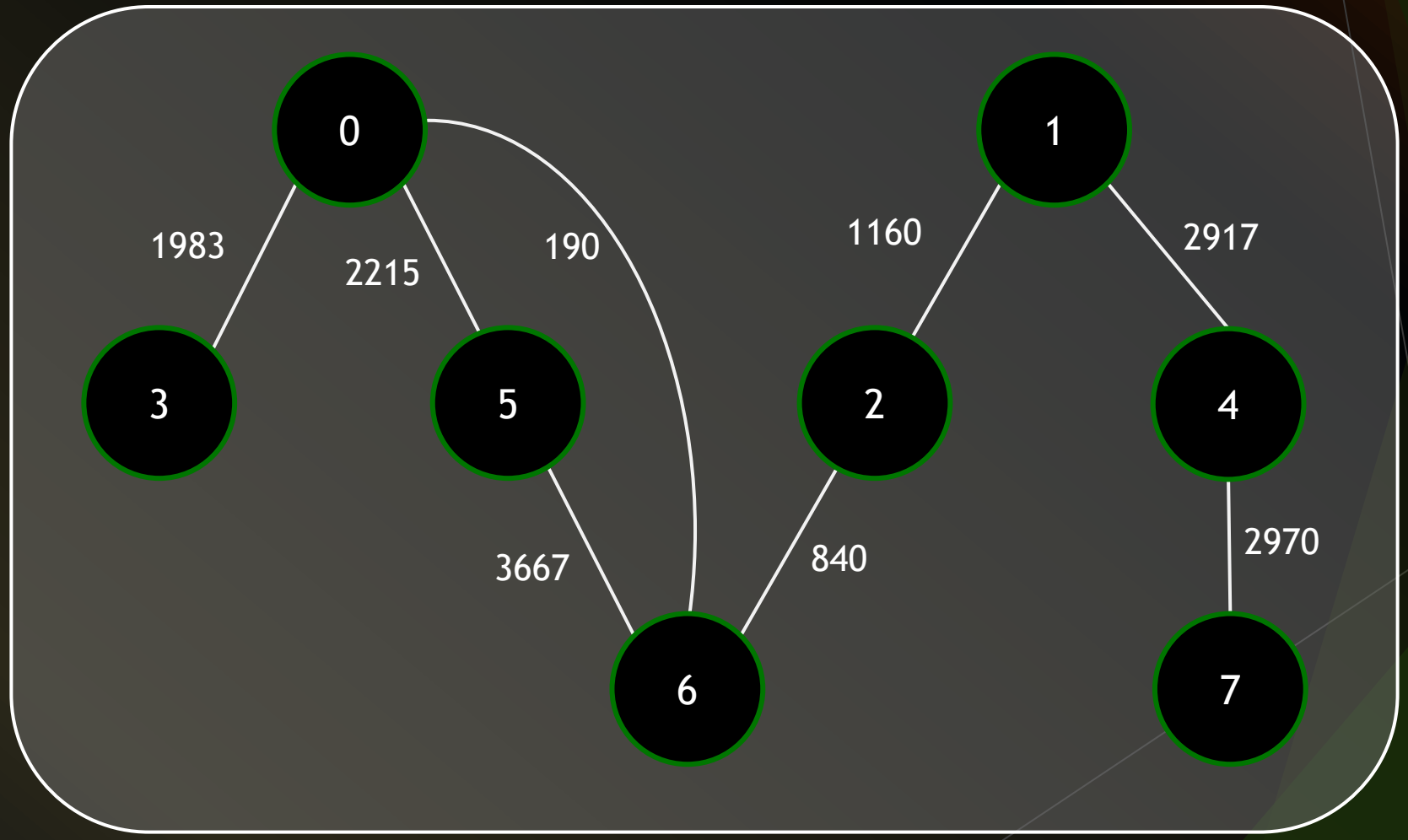






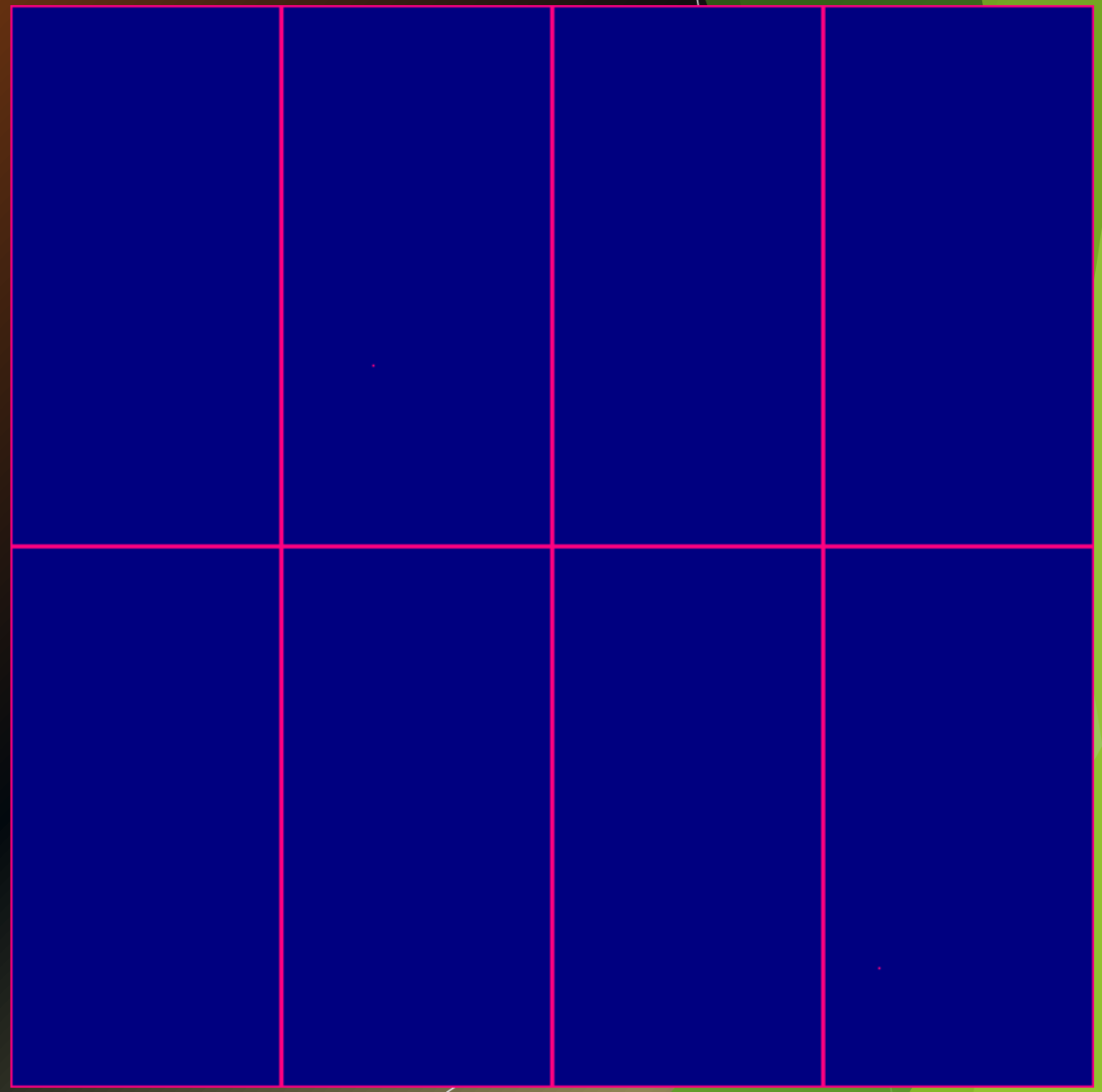
VIRTUAL TOPOLOGY: GRAPHS (FYI)

Can make a new communicator with Graph Topology information attached!



HANDS ON ACTIVITY #2

MPI_Cart_shift

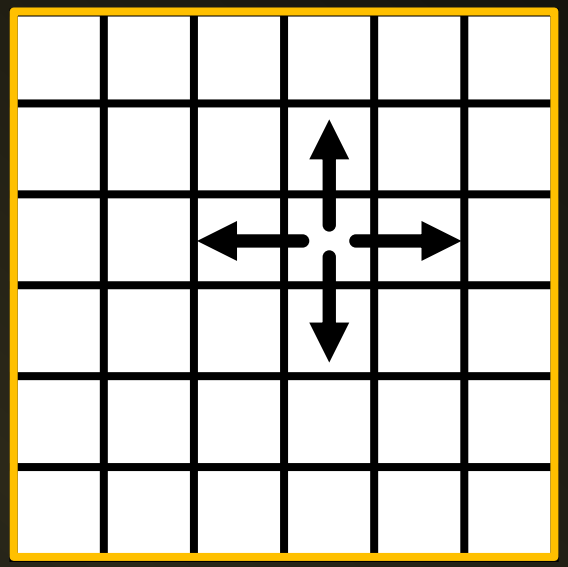


EXERCISE.2 CARTESIAN VIRTUAL TOPOLOGIES

Step 1: cd `exercises/Exercise.2-CartesianTopology/{c or f90}`

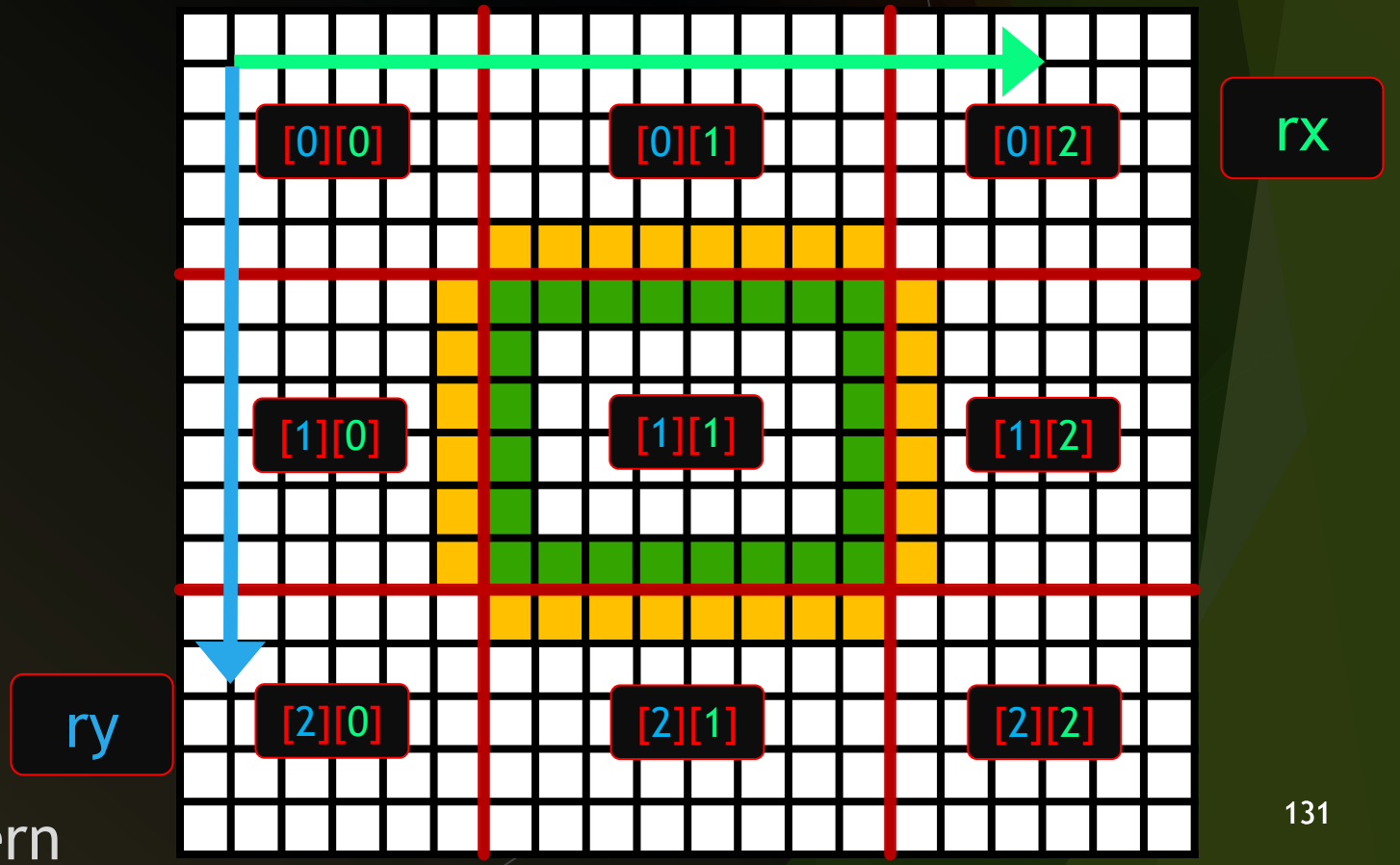
Step 2: Complete the “TODO” tasks in stencil_cart_shift.`{c or f90}`

$$-\Delta u(x,y) = f$$



4 Neighbors

Typical Communication Pattern

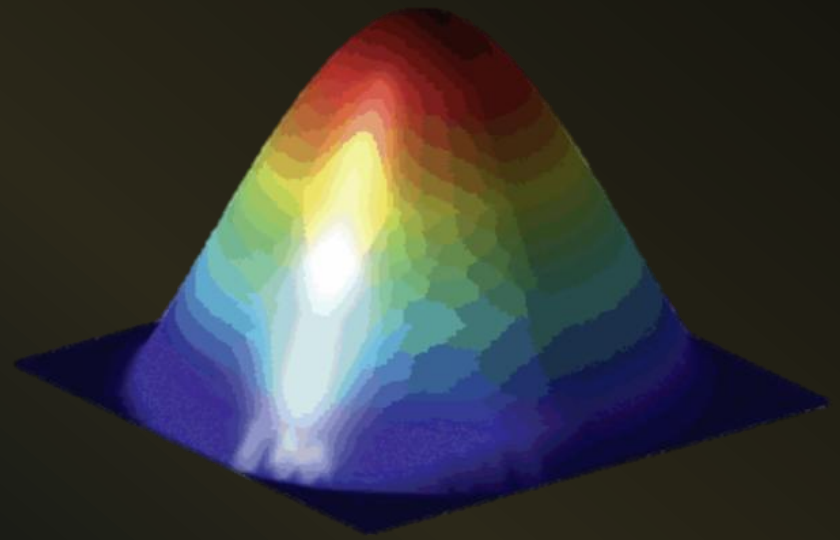


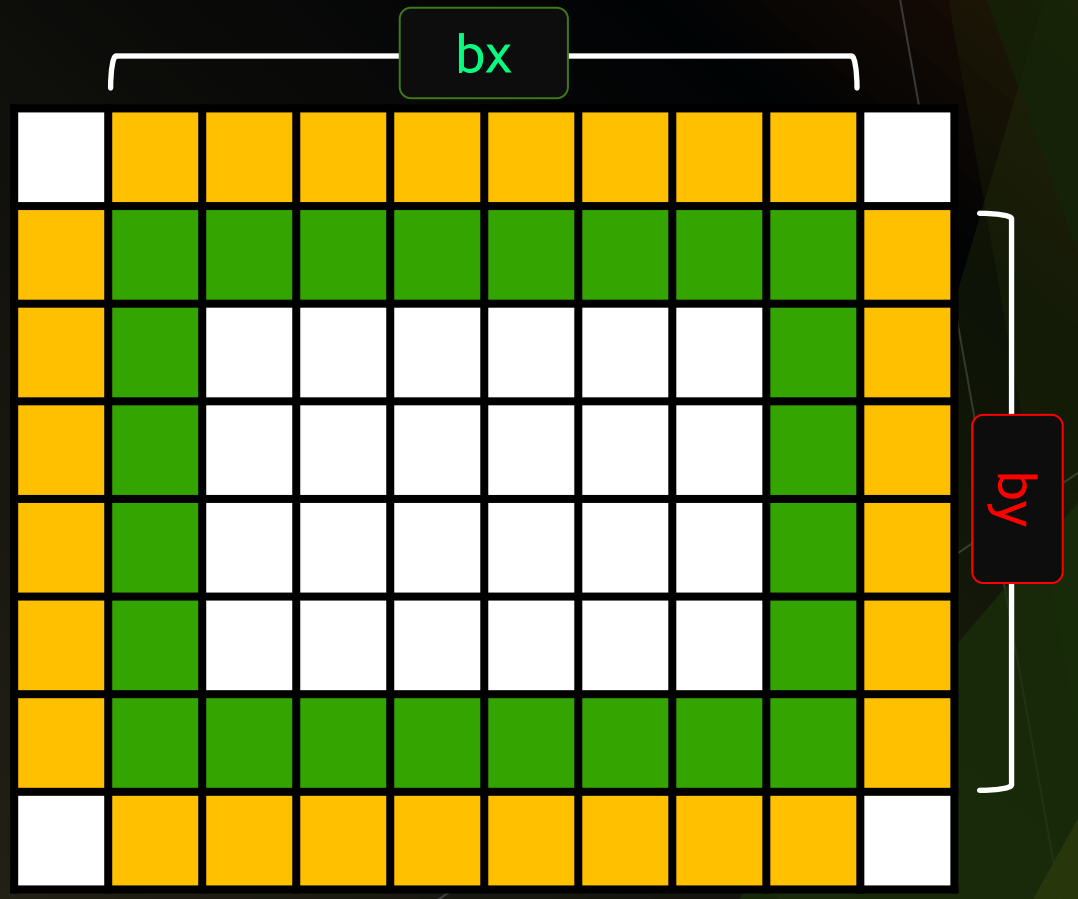
EXERCISE.2 CARTESIAN VIRTUAL TOPOLOGIES

Step 1: cd `exercises/Exercise.2-CartesianTopology/{c or f90}`

Step 2: Complete the “TODO” tasks in stencil_cart_shift.`{c or f90}`

$$-\Delta u(x,y) = f$$

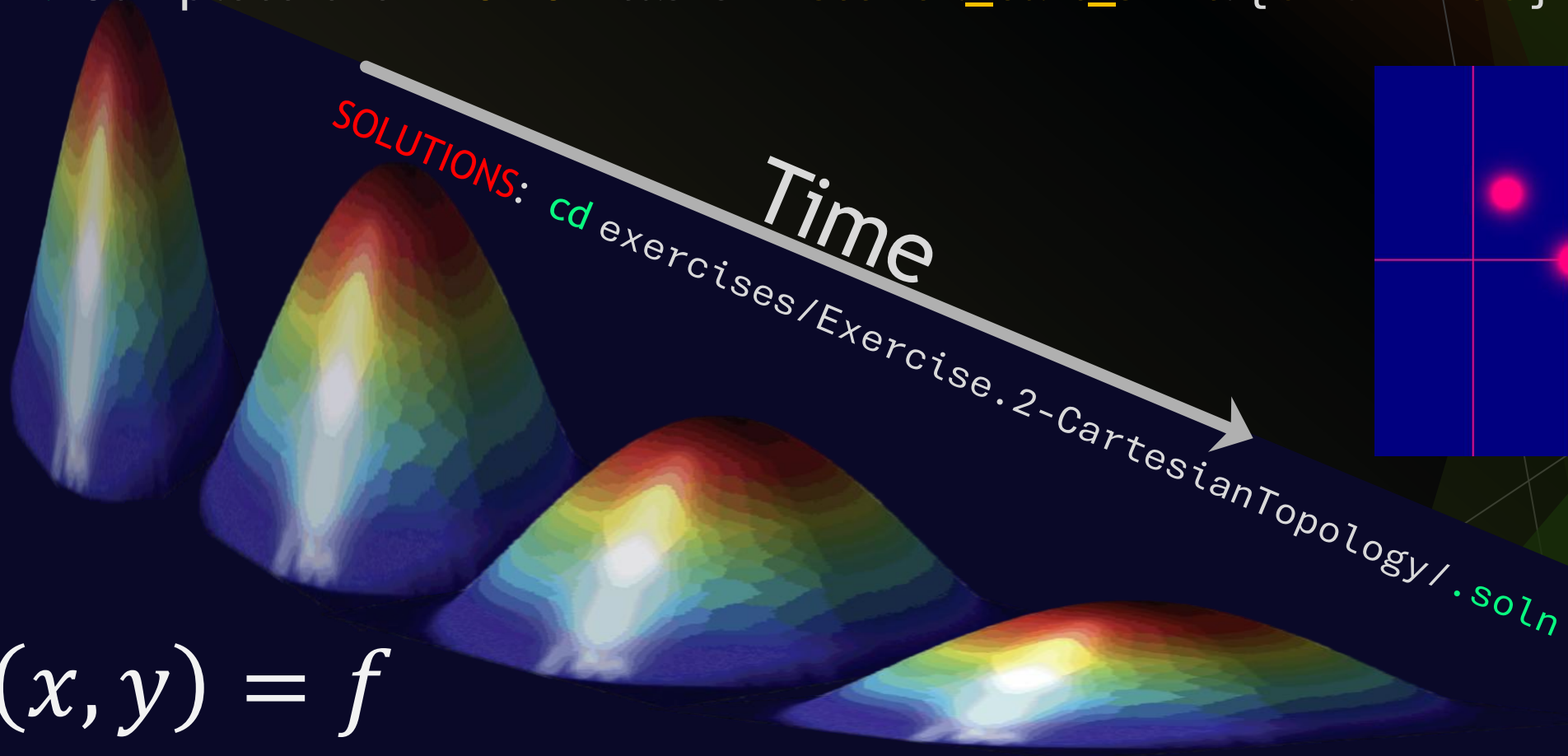


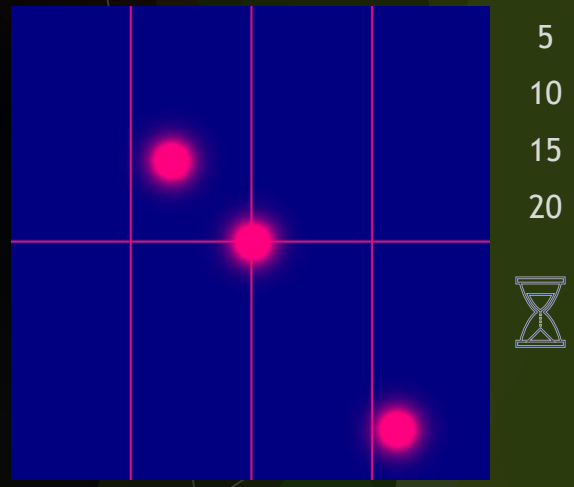


EXERCISE.2 CARTESIAN VIRTUAL TOPOLOGIES

Step 1: cd `exercises/Exercise.2-CartesianTopology/{c or f90}`

Step 2: Complete the “TODO” tasks in stencil_cart_shift.`{c or f90}`





$$-\Delta u(x,y) = f$$

SECTION IV

NEIGHBORHOOD COLLECTIVES

NEIGHBORHOOD COLLECTIVES

Virtual Topologies implement no communication!

- They are just helper functions.

Collective Communications only cover some patterns

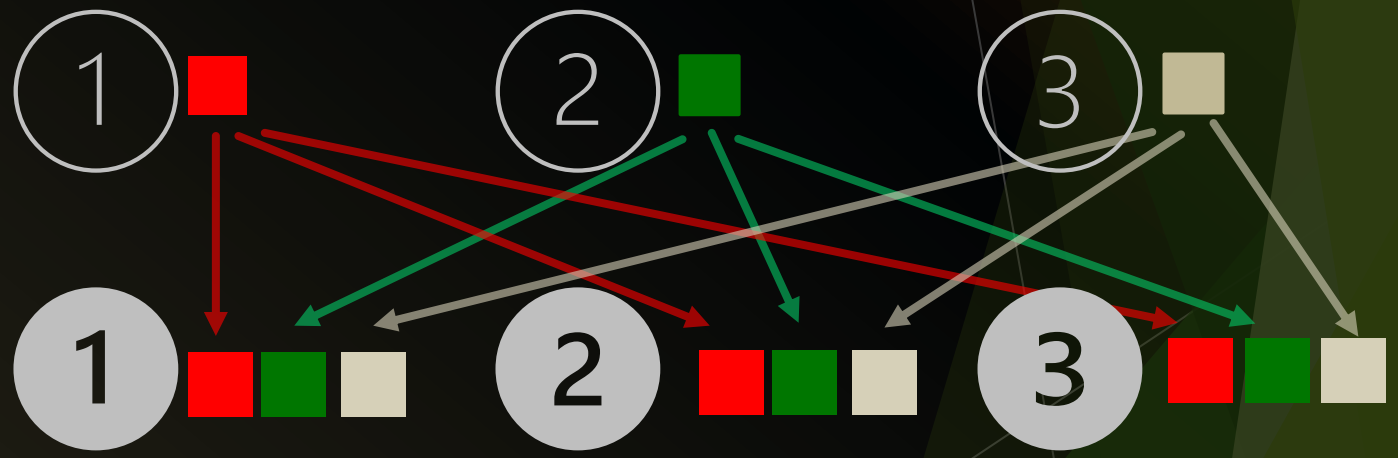
- E.g., no stencil patterns.

✓ Neighborhood collectives add communication functions to process topologies.

MPI_[I]NEIGHBOR_ALLGATHER[V/W]

Gathers and distributes data from and to all neighbors.

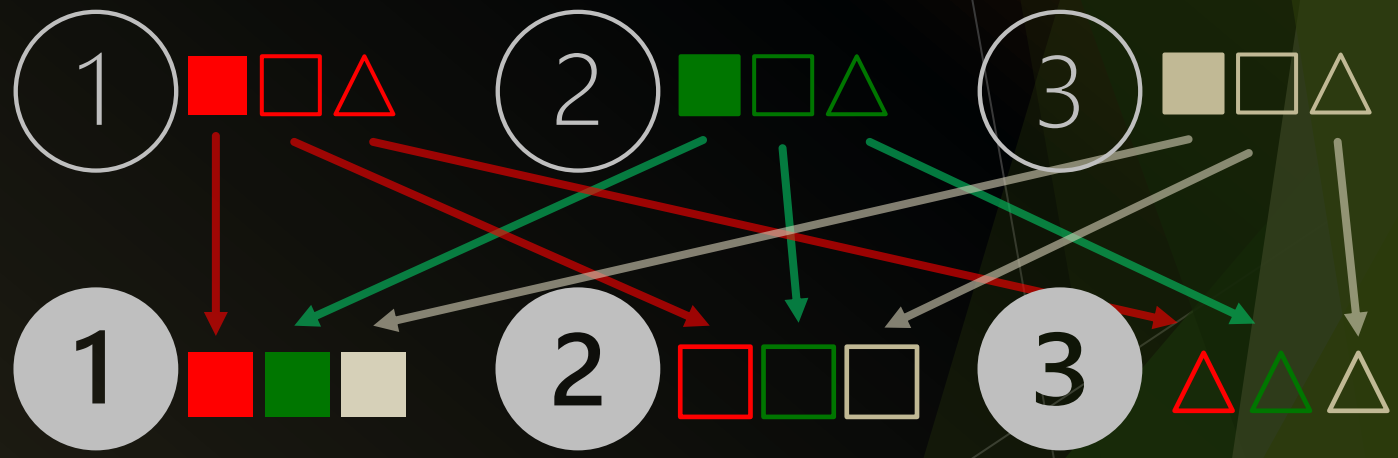
```
int MPI_Neighbor_allgather(
    const void *sendbuf,
    int  sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    MPI_Comm comm);
```



MPI_[I]NEIGHBOR_ALLTOALL[V/W]

All processes send data to neighboring processes in a virtual topology communicator.

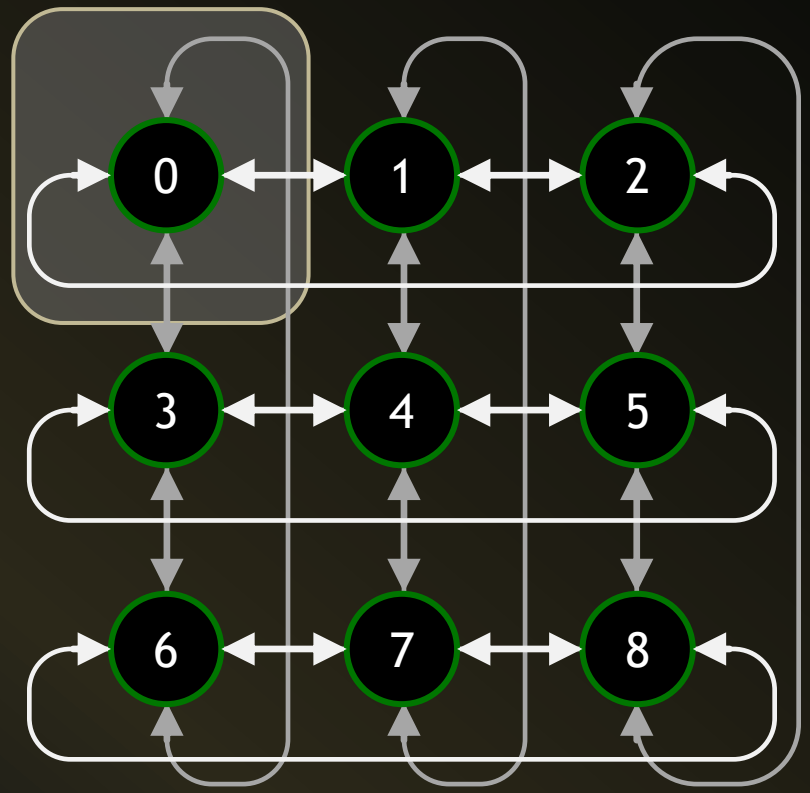
```
int MPI_Neighbor_alltoall(
    const void *sendbuf,
    int  sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    MPI_Comm comm);
```

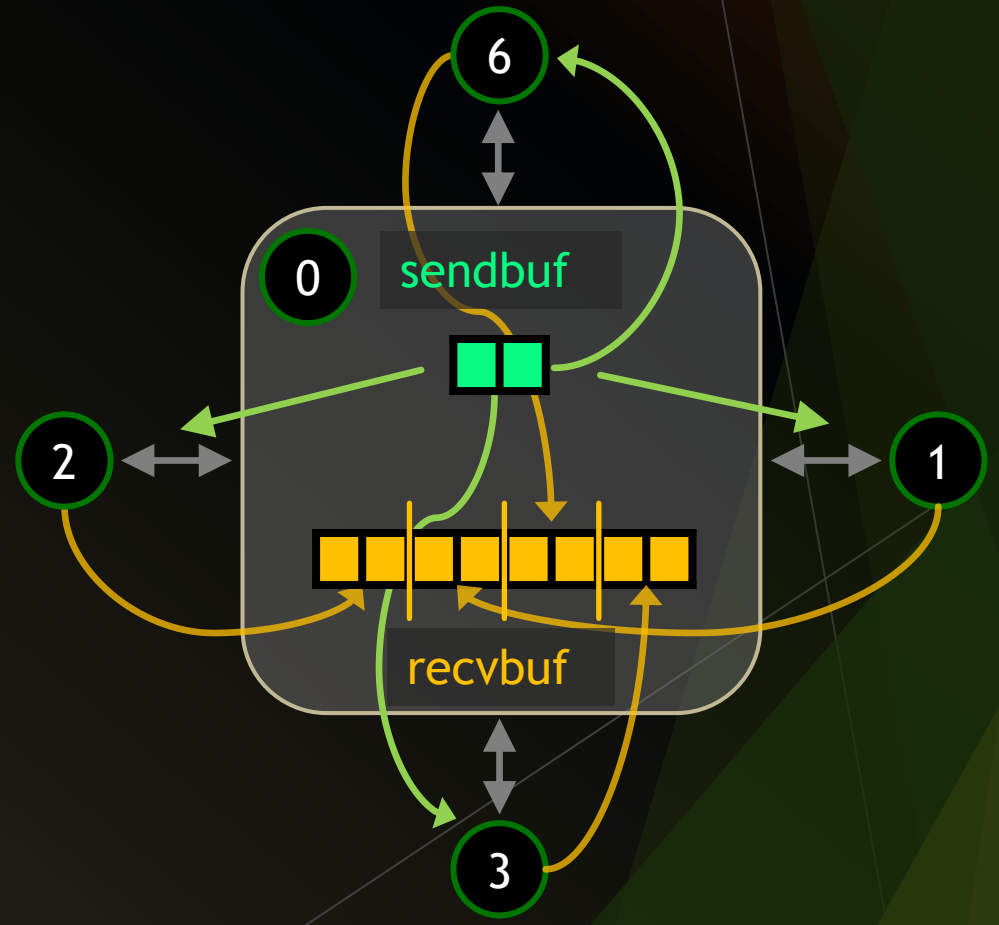


NEIGHBORHOOD COLLECTIVES: CARTESIAN

Communicate with direct neighbors in Cartesian topology

- Buffer order example:

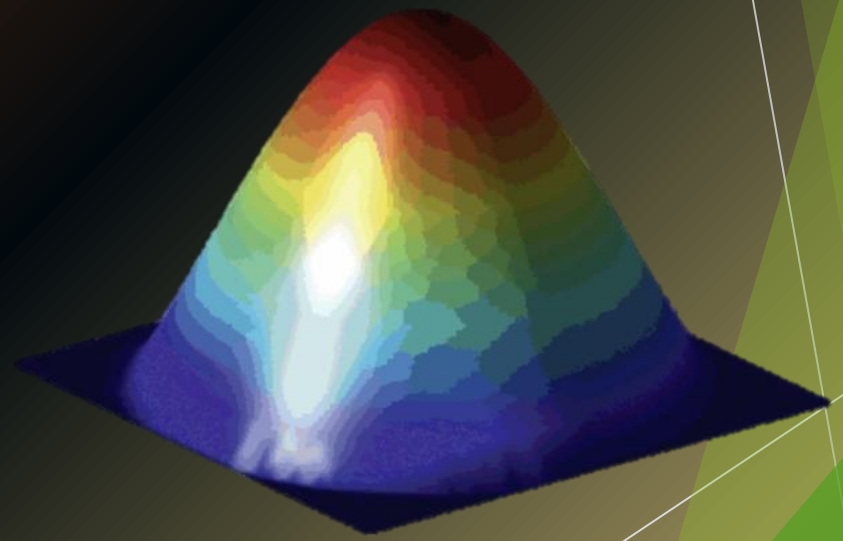




HANDS ON ACTIVITY #3

MPI_Ineighbor_alltoallv

$$-\Delta u(x, y) = f$$

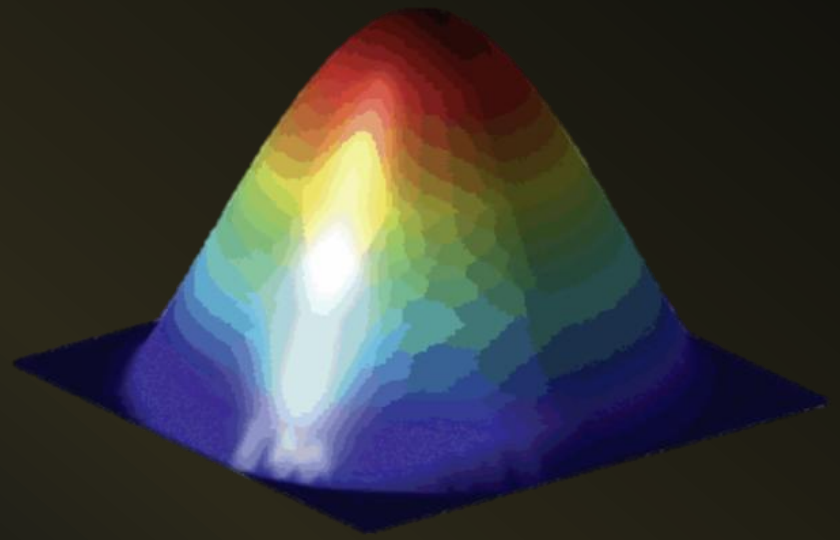


EXERCISE.3 NEIGHBORHOOD COLLECTIVES

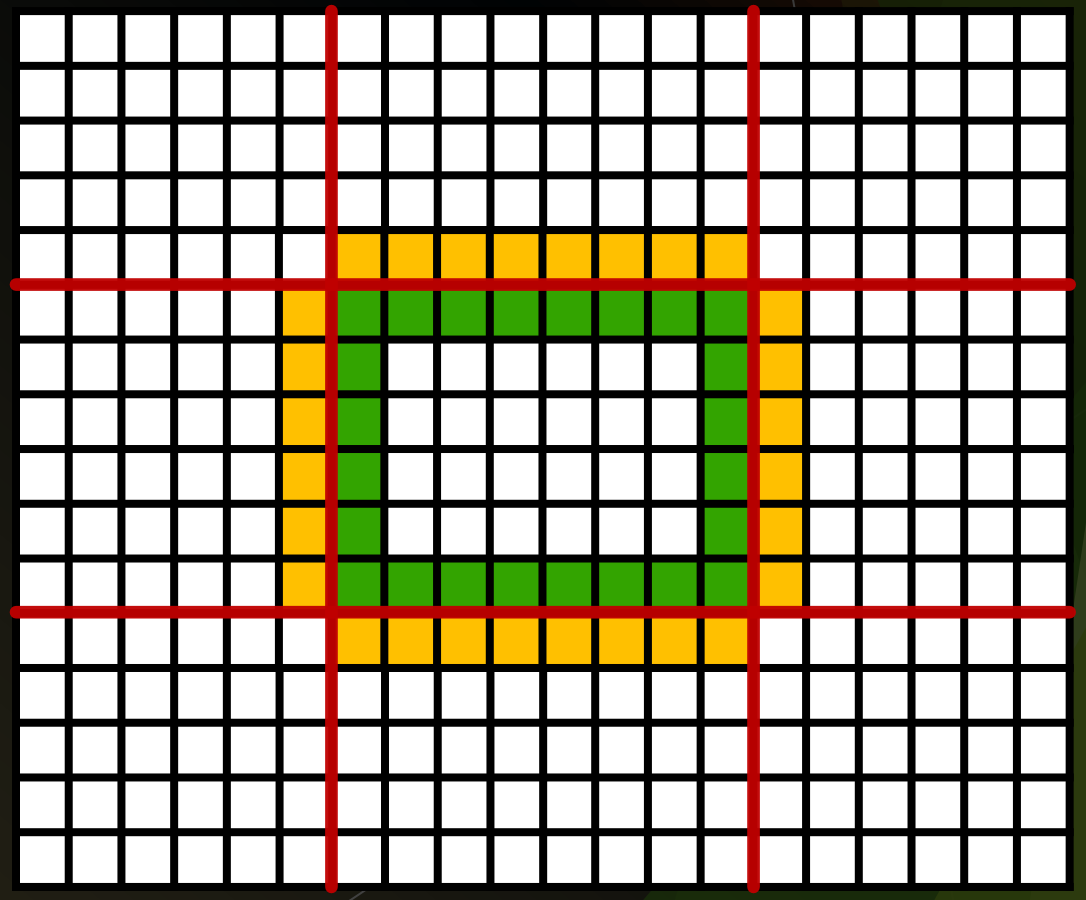
Step 1: cd `exercises/Exercise.3-NeighborhoodCollectives/{C}`

Step 2: Complete the “TODO” tasks in `stencil_mpi_carttopo_neighcolls.{C}`

$$-\Delta u(x,y) = f$$



4 Neighbors
Typical Communication Pattern



EXERCISE.3 MPI_INEIGHBOR_ALLGATHERV

Step 1: cd `exercises/Exercise.3-NeighborhoodCollectives/{c or f90}`

Step 2: Complete the "TODO" tasks in `stencil_mpi_carttopo_neighcolls.{c or f90}`

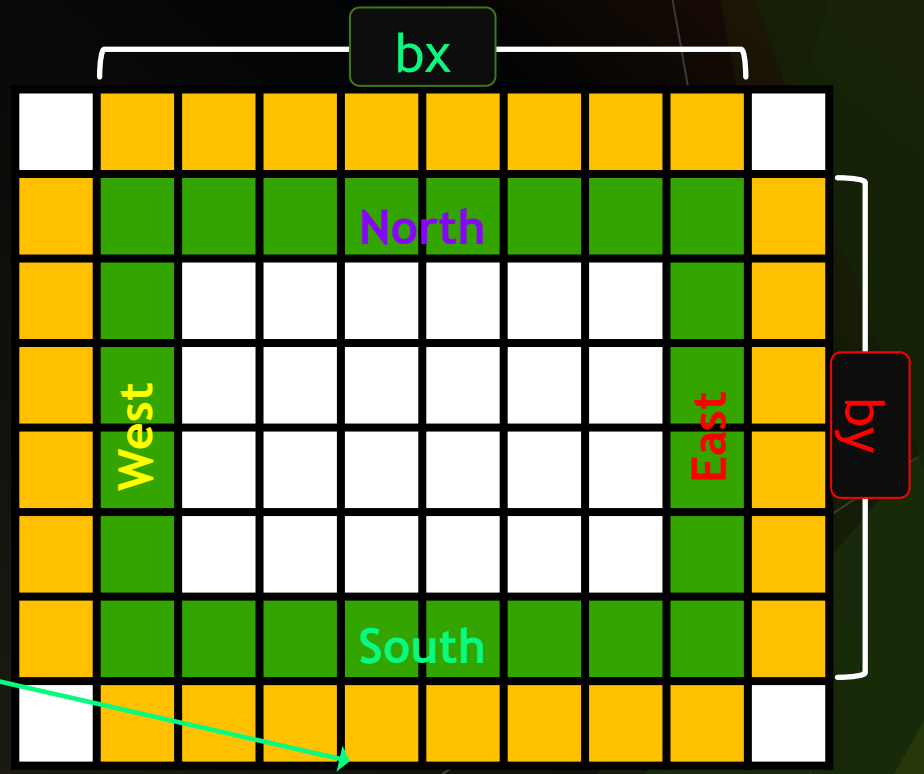
Pack Send Buffer (green cells)

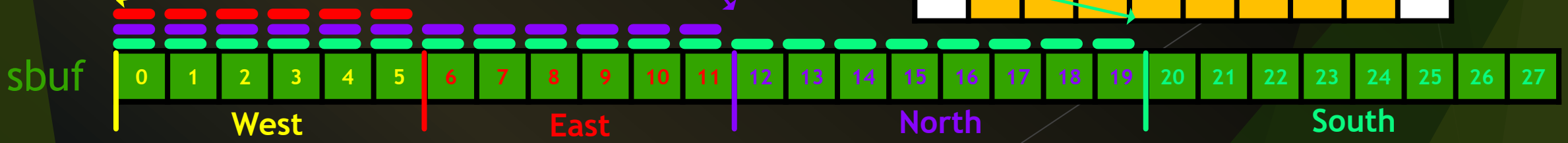
- West, East, North, South

Recall Neighbor Ordering (X-, X+, Y-, Y+)

Counts[4] = {?, ?, ?, ?}

Displacements[4] = {?, ?, ?, ?}

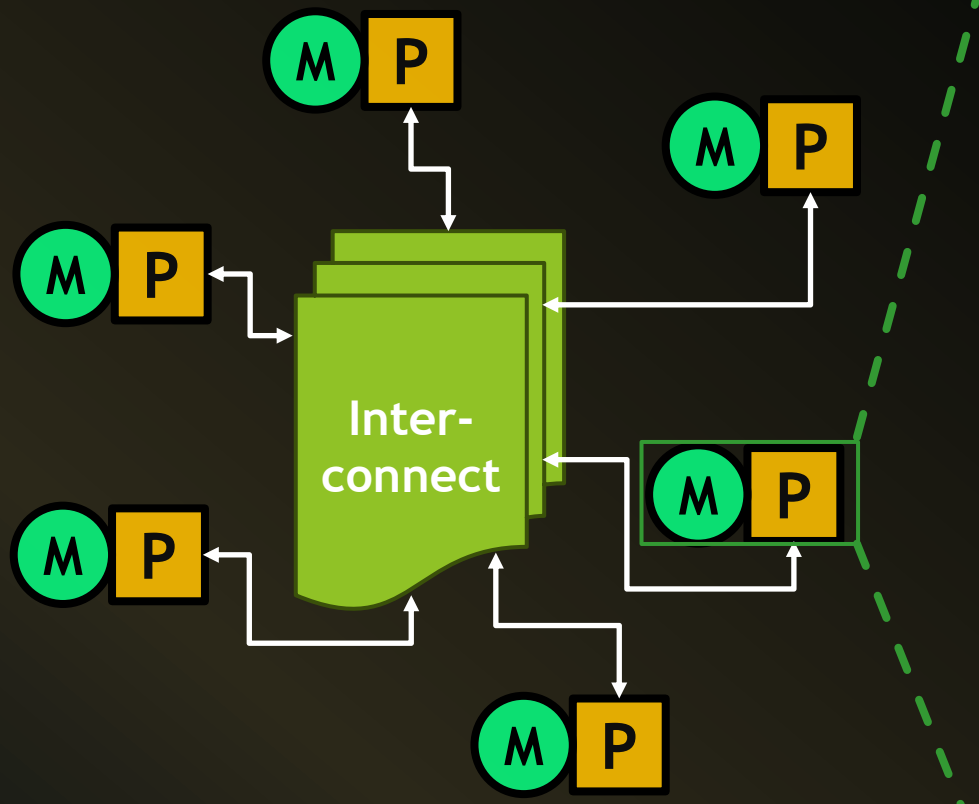




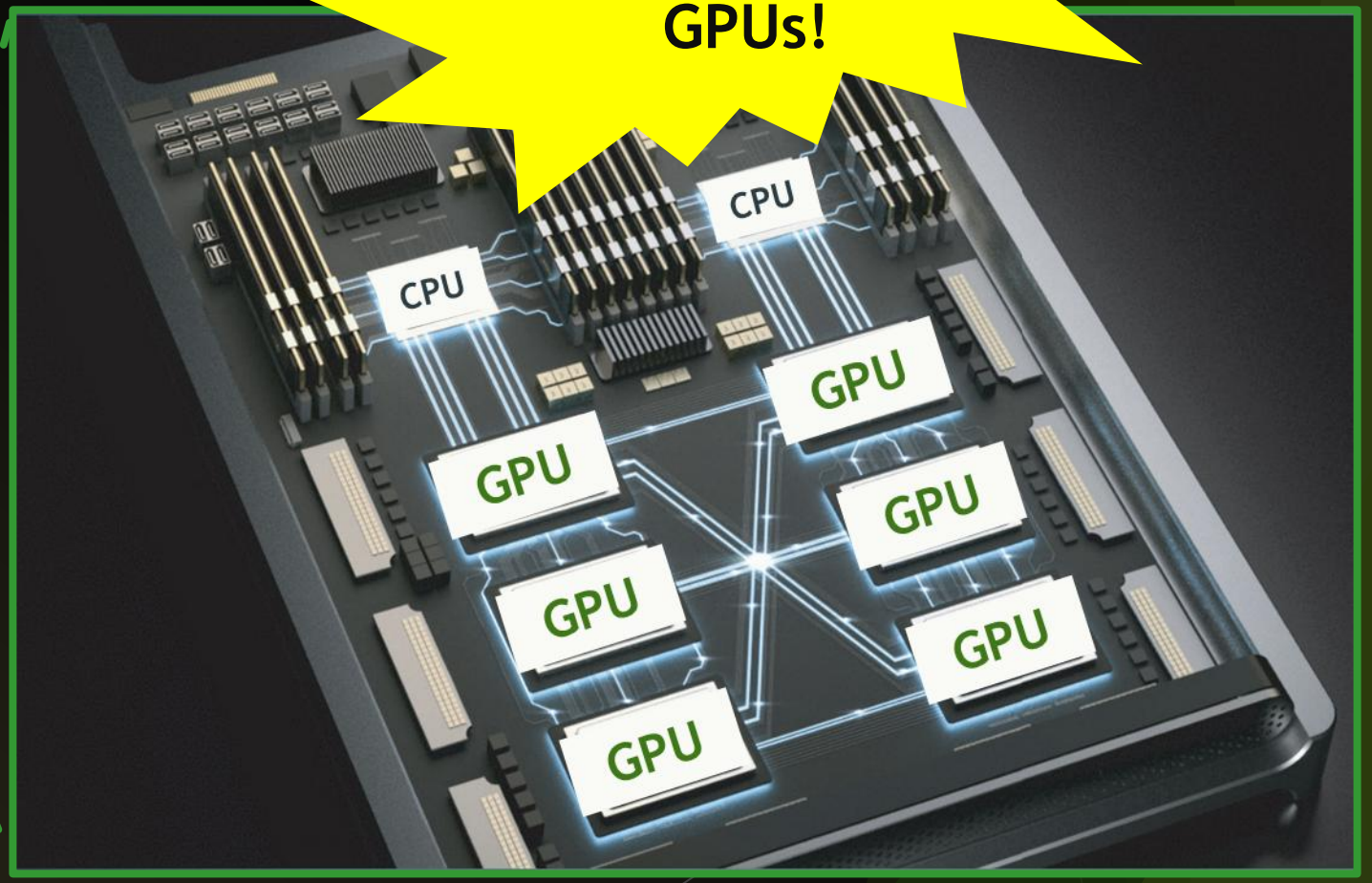
SECTION V

HYBRID PARALLEL PROGRAMMING MPI+X

EXASCALE ARCHITECTURES



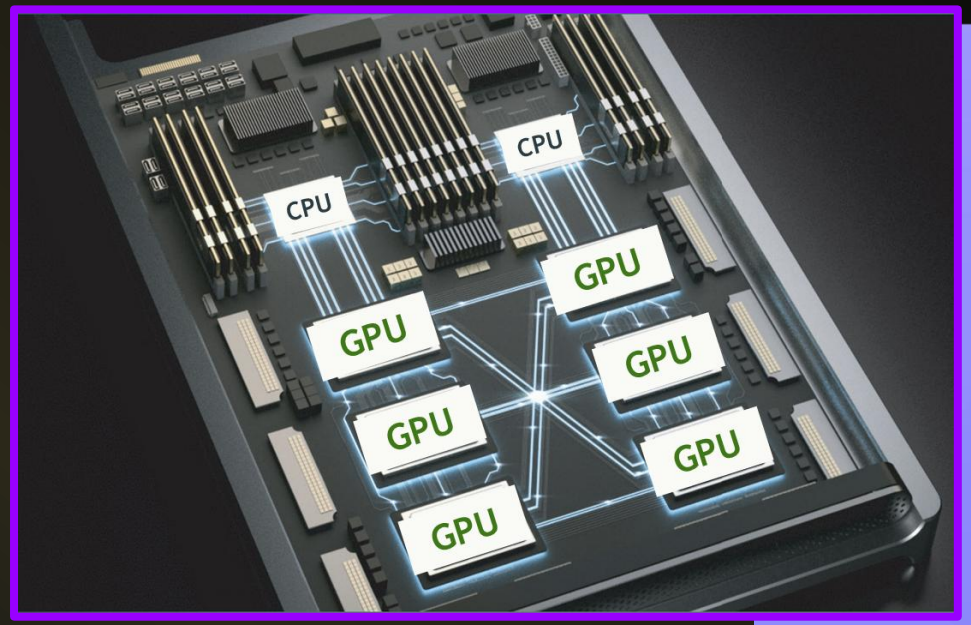
First 66 on Top500 have GPUs!

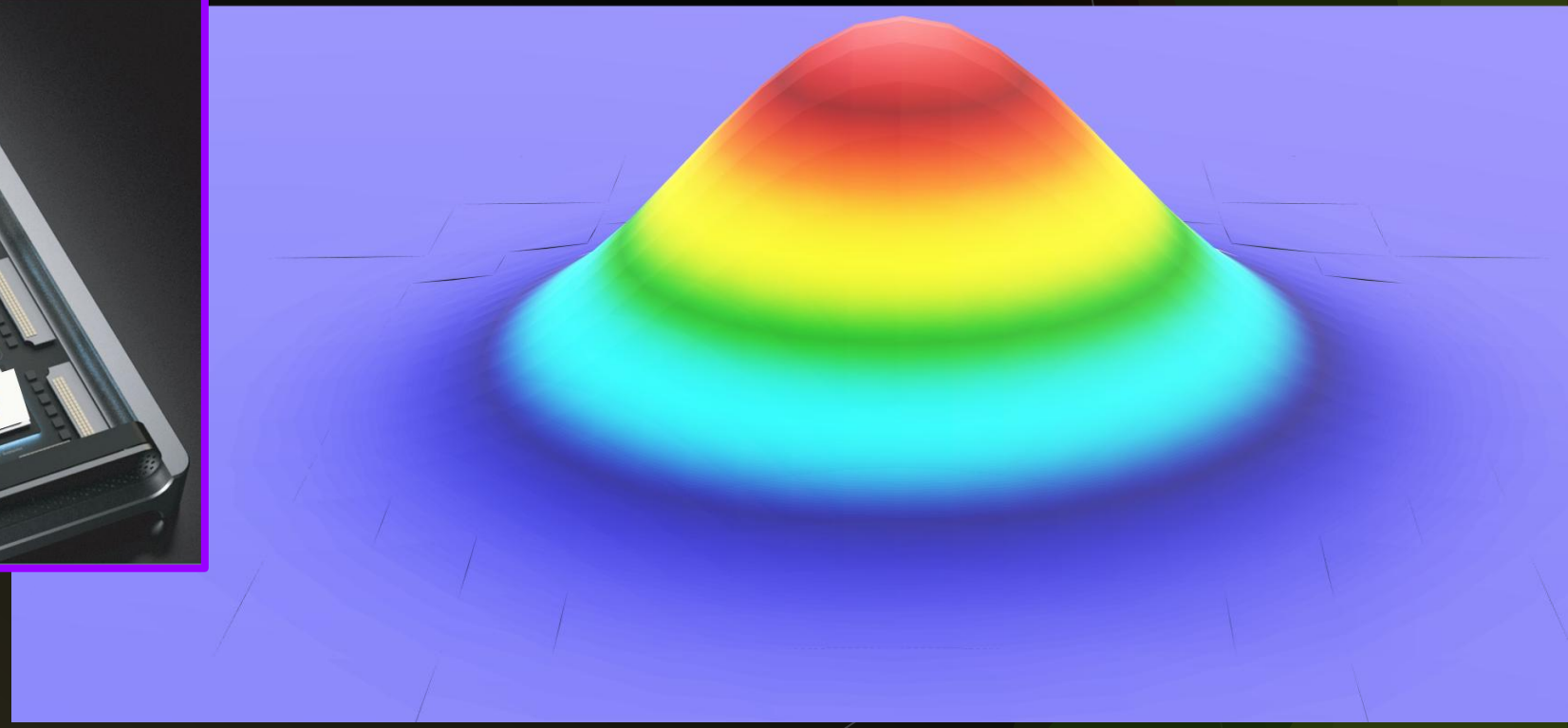


EXASCALE ARCHITECTURES

EXASCALE ARCHITECTURES

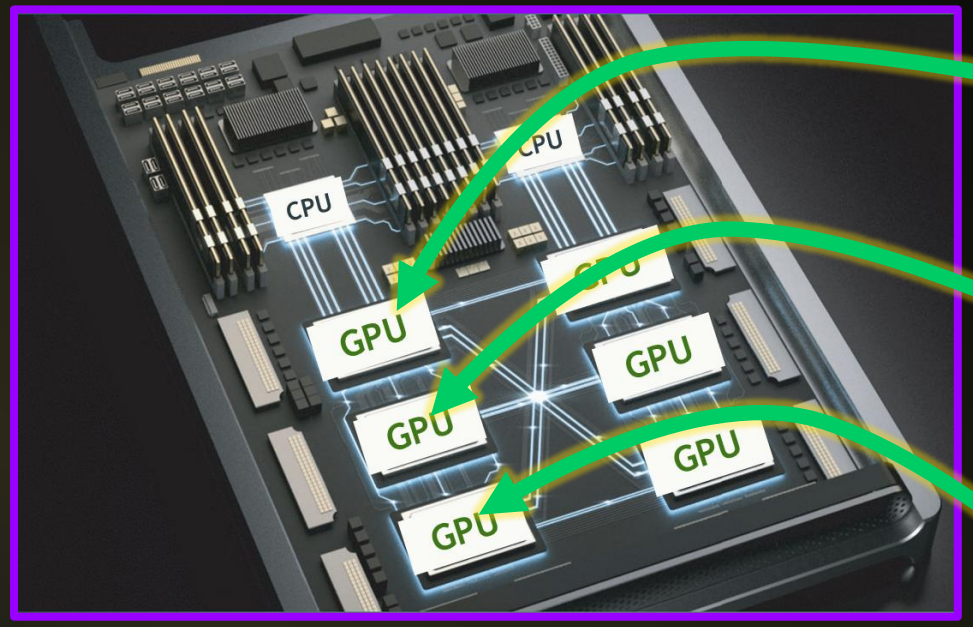
$$-\Delta u(x, y) = f$$

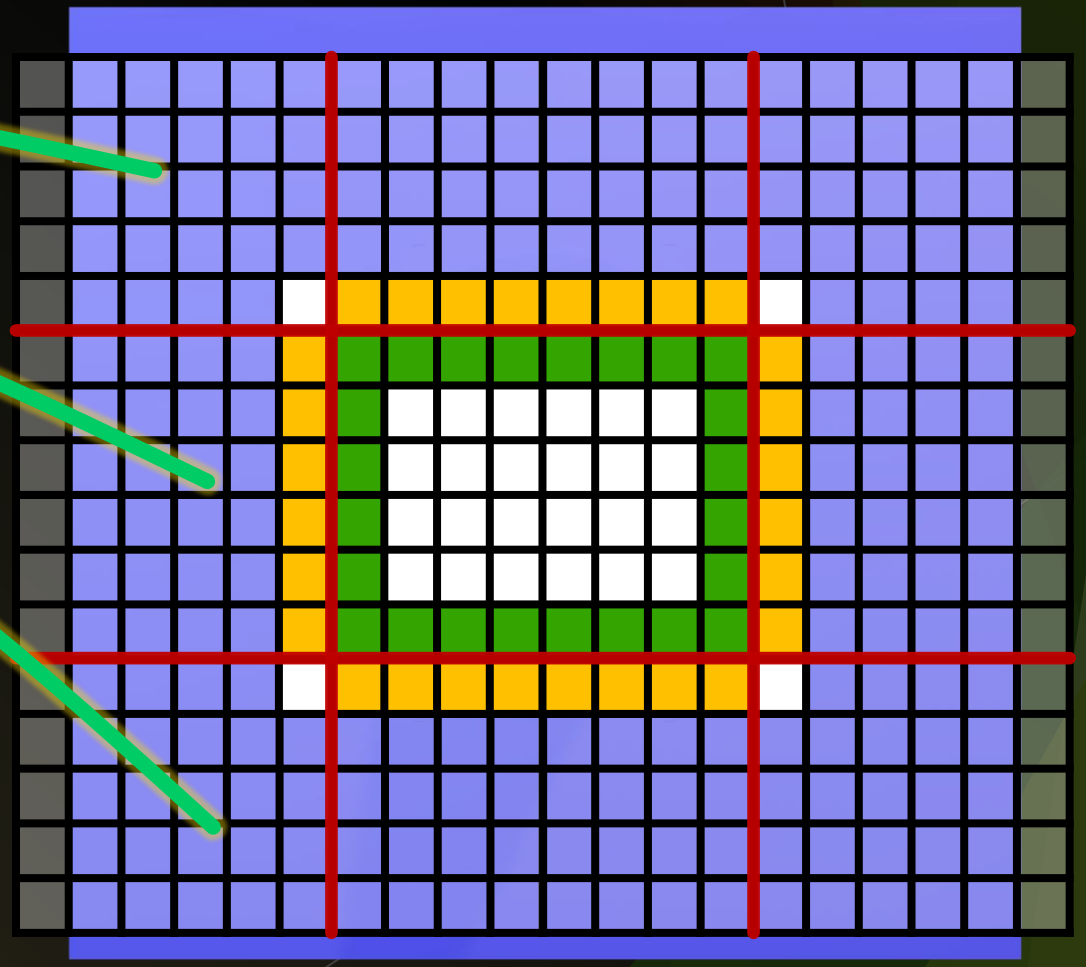




MPI + X

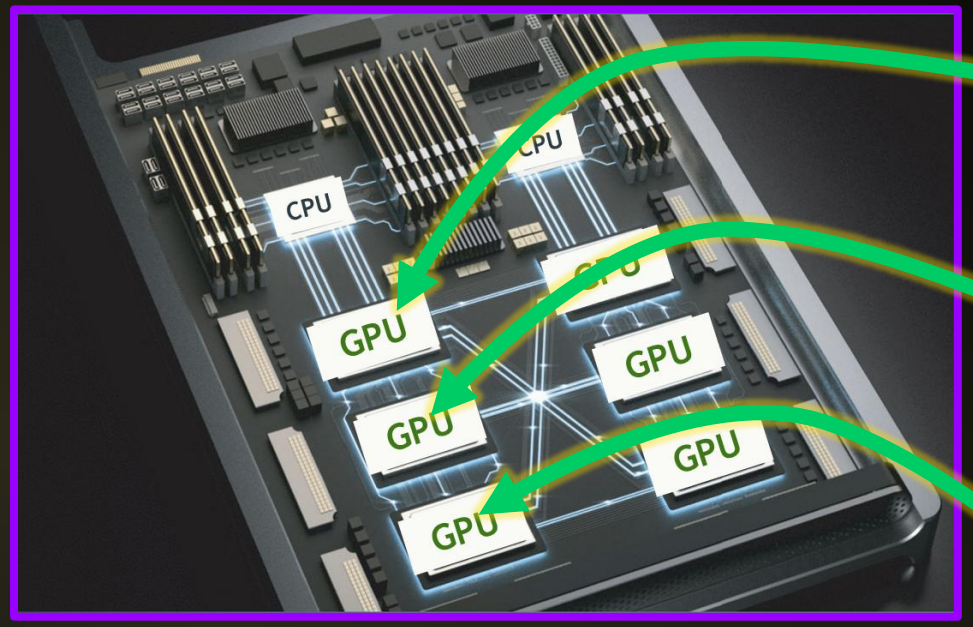
$$-\Delta u(x,y) = f$$

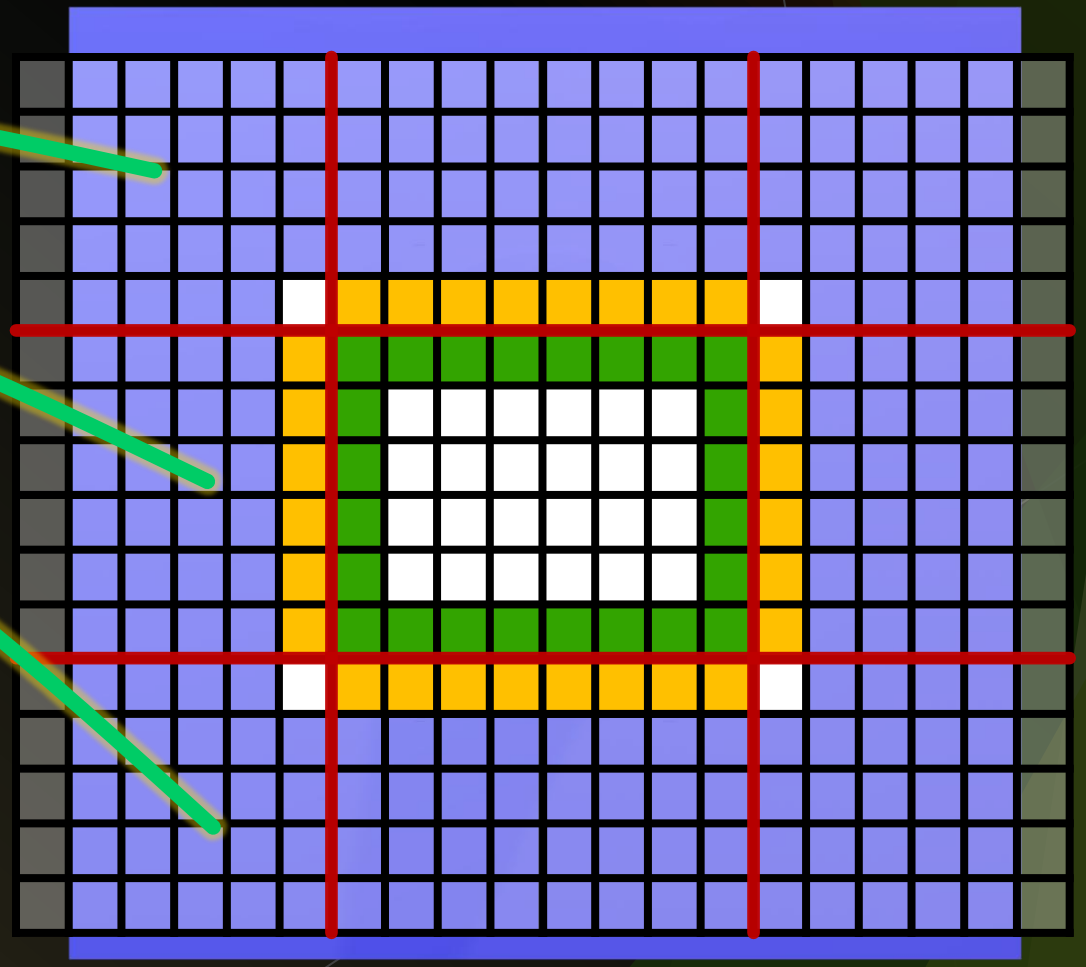




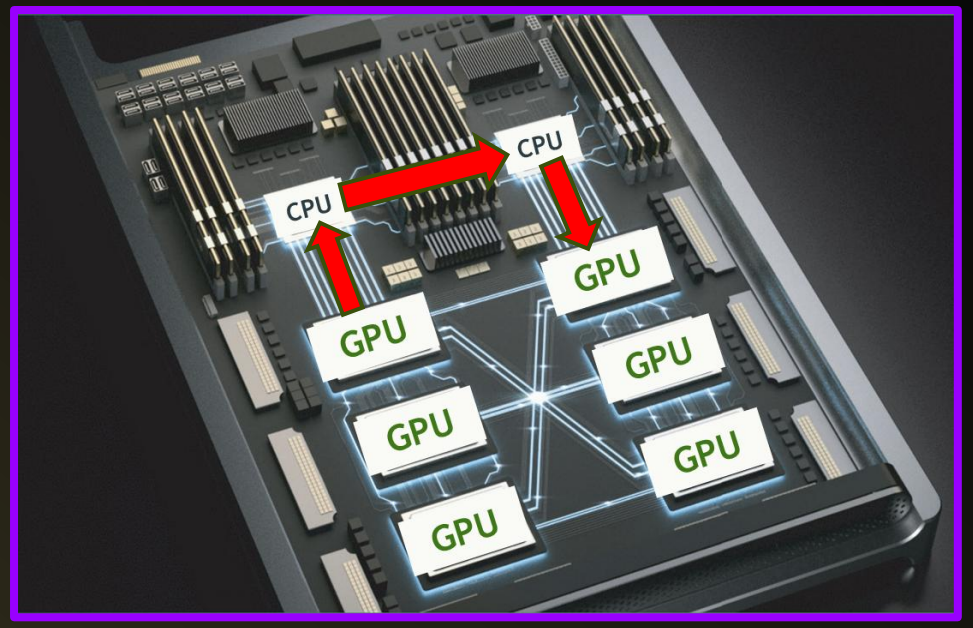
MPI + X

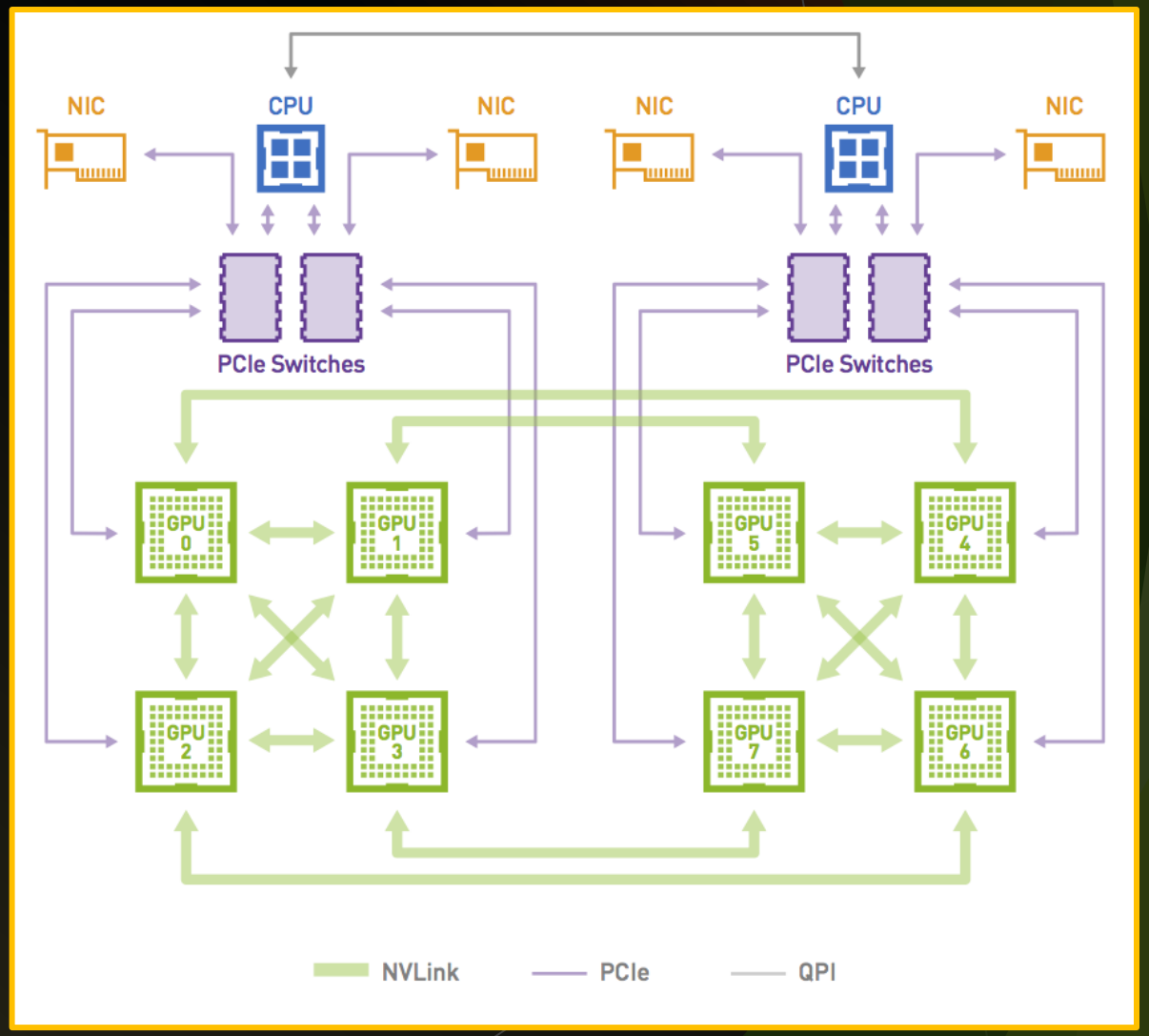
$$-\Delta u(x,y) = f$$





COMMUNICATION BETWEEN GPUS

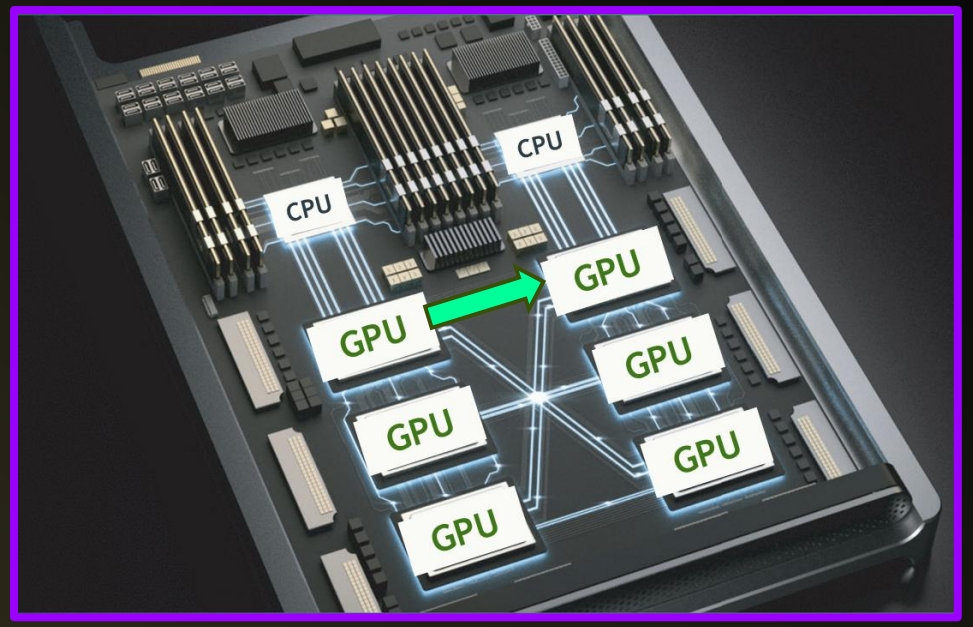


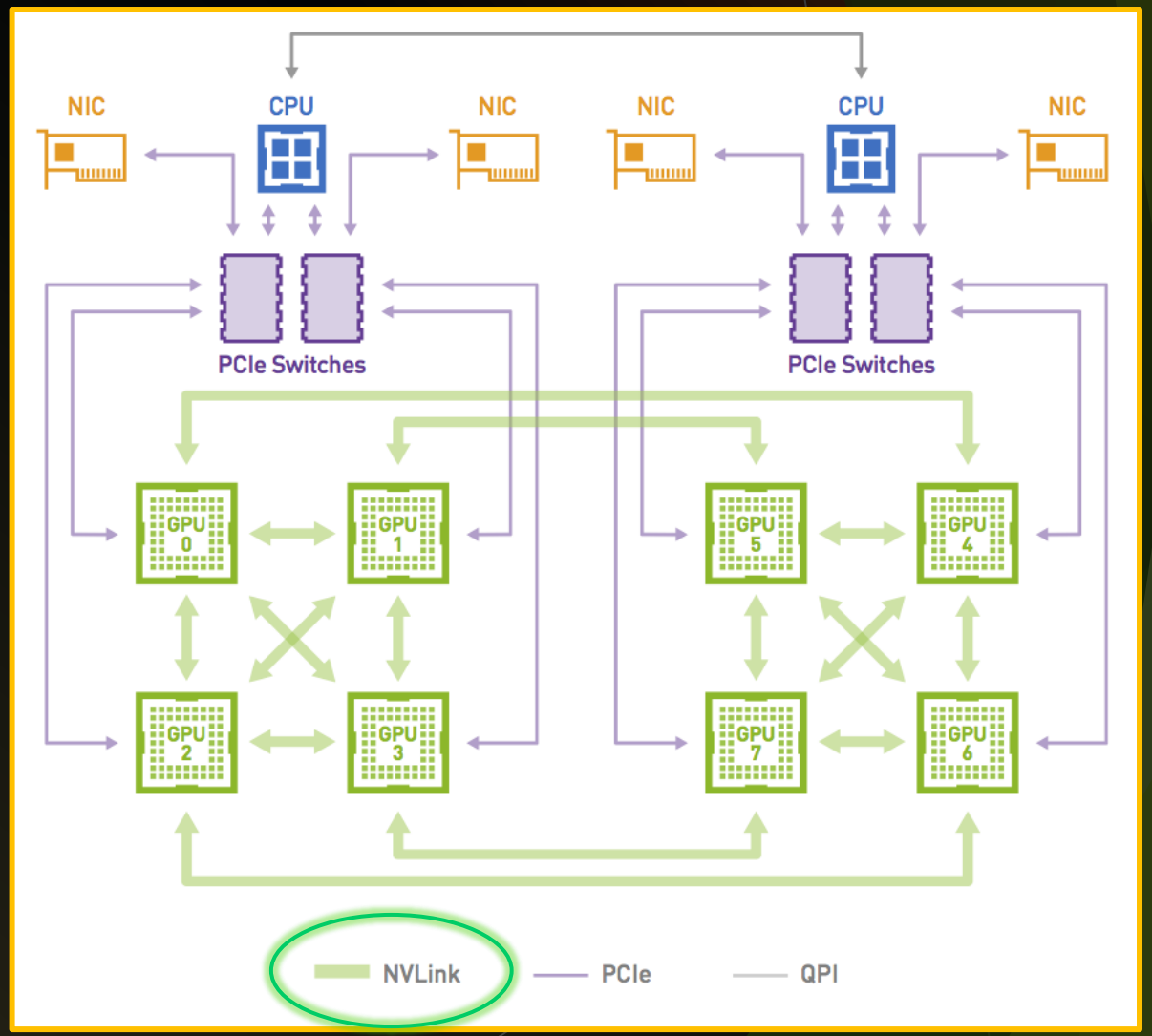


Naïve Method (staged approach)

1. Copy data from GPU to CPU
2. Use MPI to transfer data CPU to CPU
3. Copy data from CPU to GPU

COMMUNICATION BETWEEN GPUS

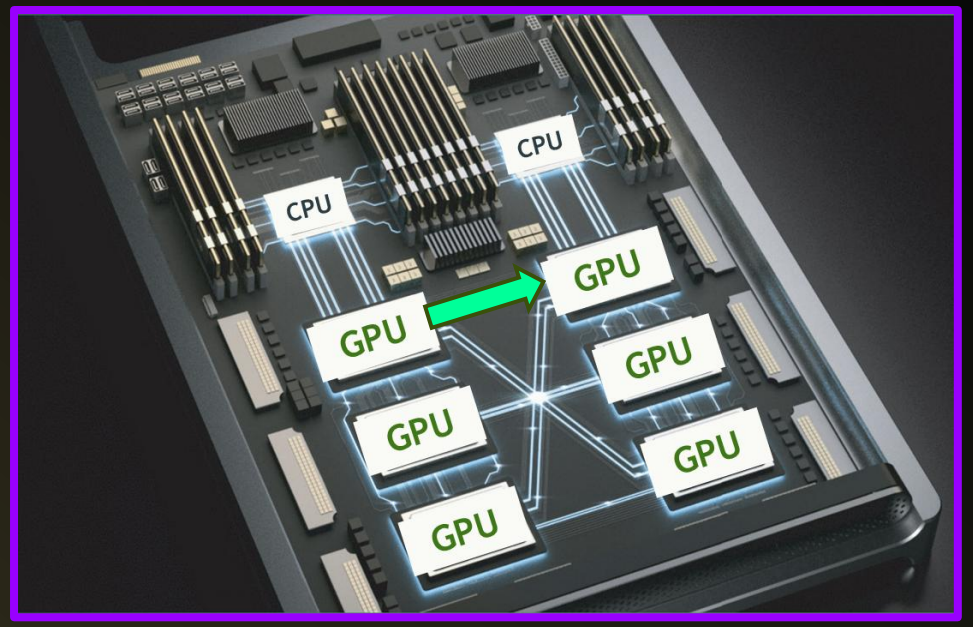




GPU-Aware

1. Copy data from GPU to GPU
   - Pass device memory pointer in MPI functions!
   - Must check system if supported

COMMUNICATION BETWEEN GPUS



GPU-Aware

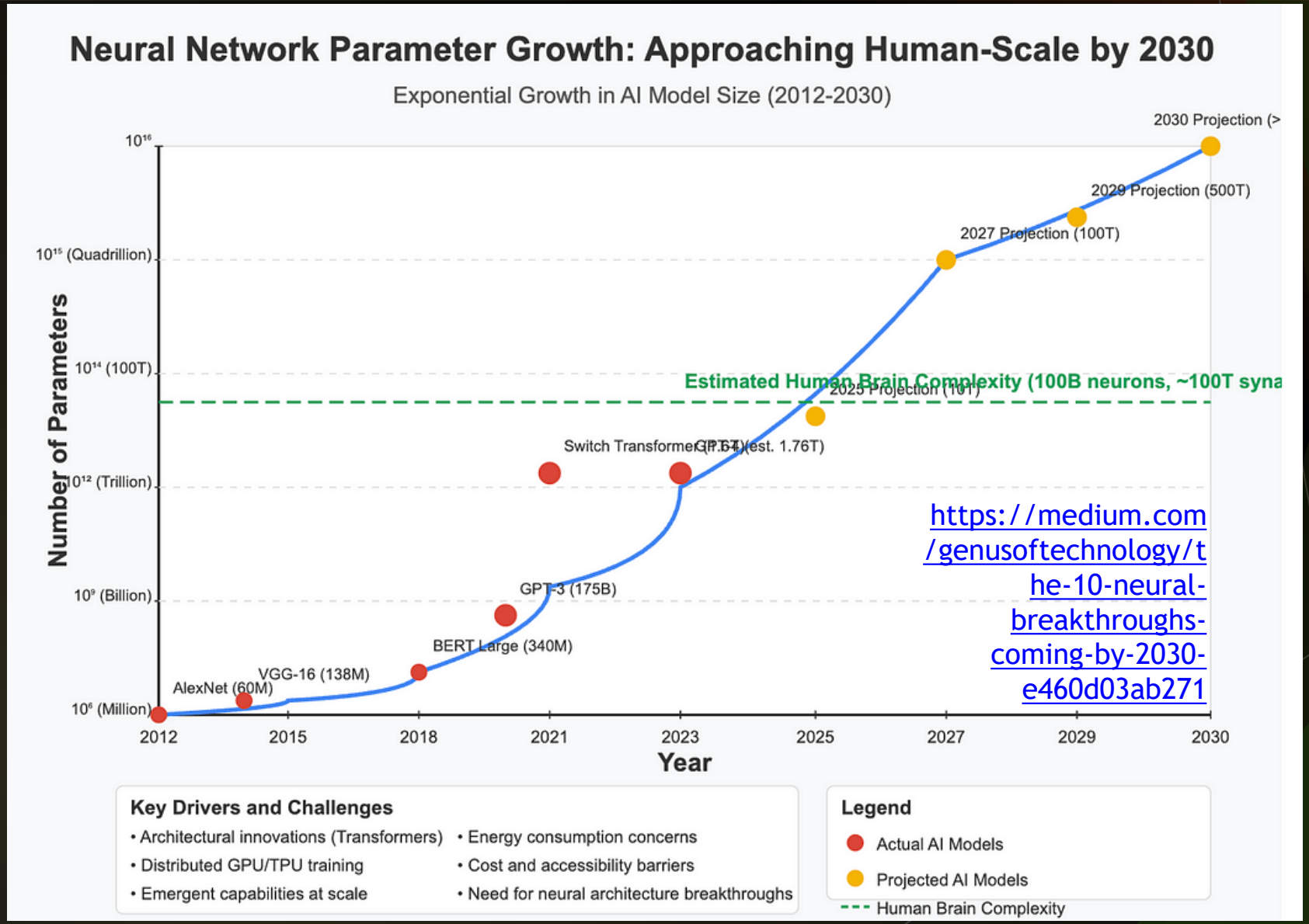
1. Copy data from GPU to GPU
   - Pass device memory pointer in MPI functions!
   - Must check system if supported

Talk to
Daniel Howard!

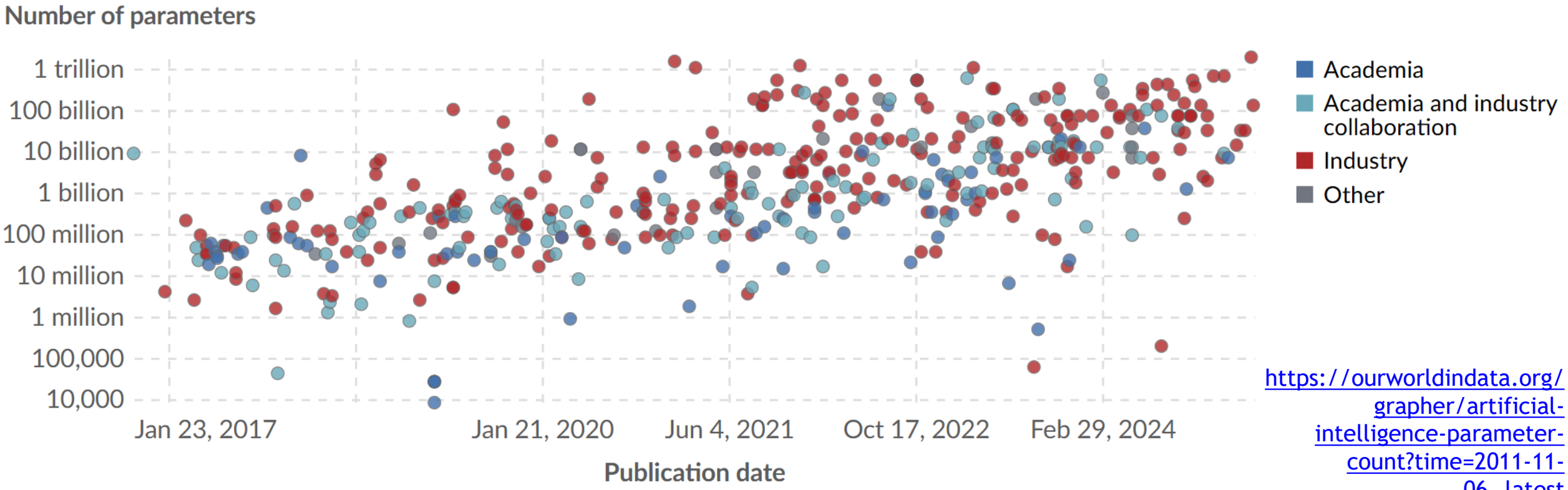
HANDS ON ACTIVITY #4

Neural Networks on GPUs

NEURAL NETWORKS - TRILLION PARAMETERS?!

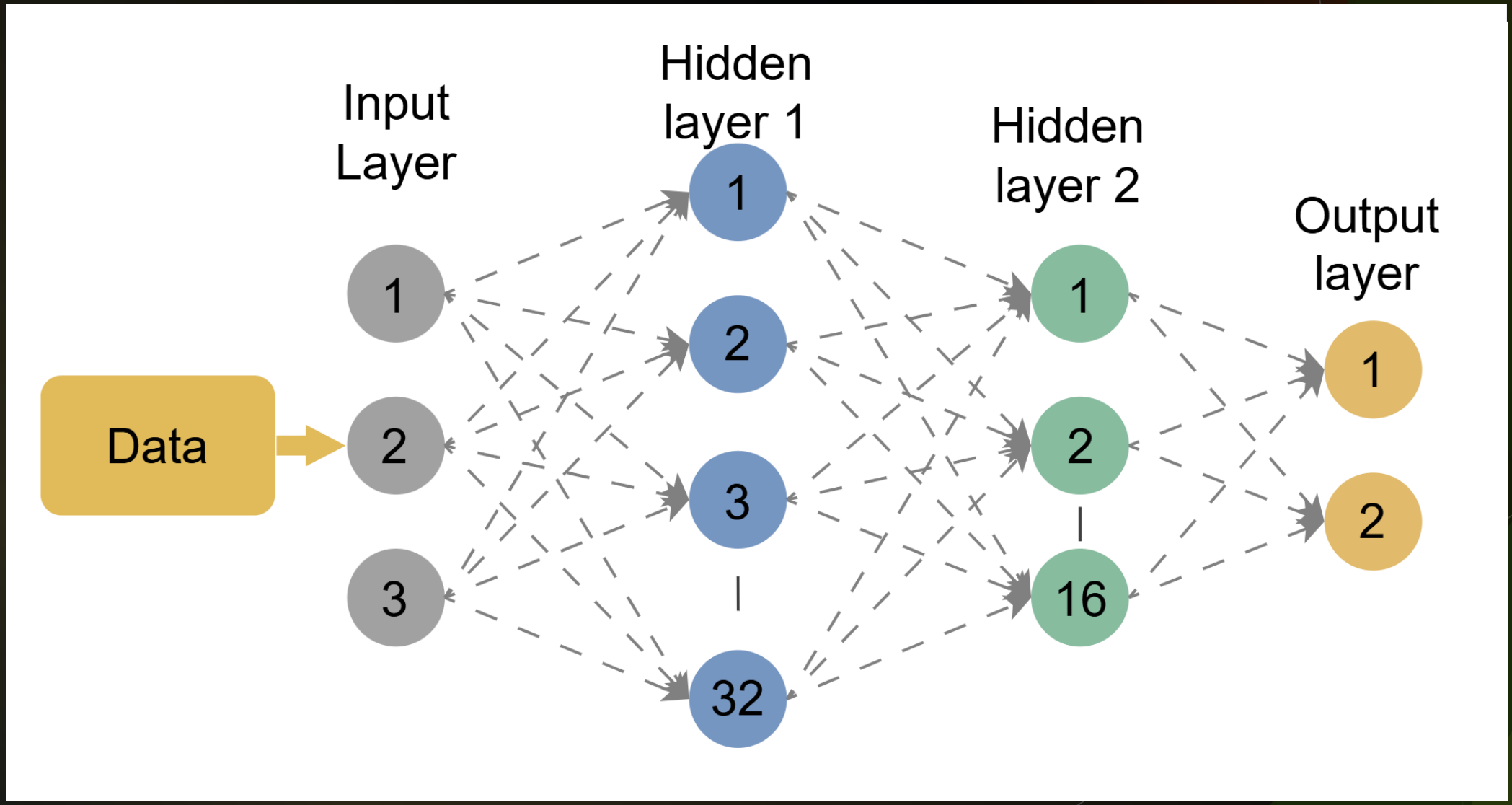


NEURAL NETWORKS - TRILLION PARAMETERS?!

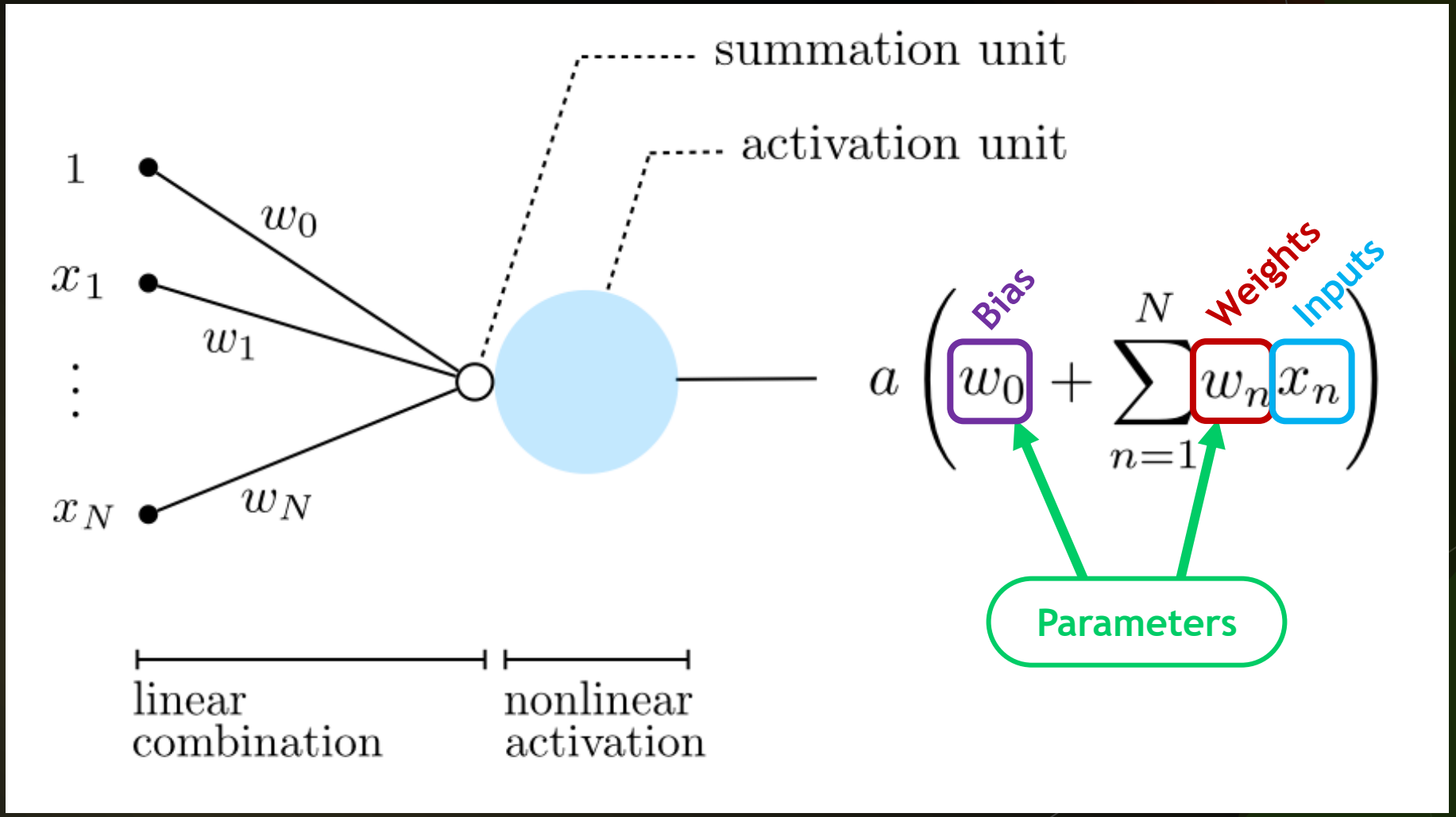


https://ourworldindata.org/grapher/artificial-intelligence-parameter-count?time=2011-11-06..latest

MULTI-LAYER PERCEPTRON (MLP)



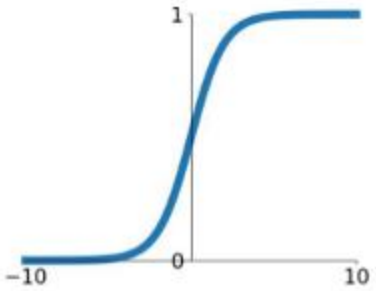
MULTI-LAYER PERCEPTRON (MLP)



ACTIVATION FUNCTIONS

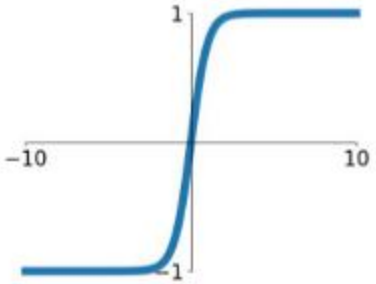
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



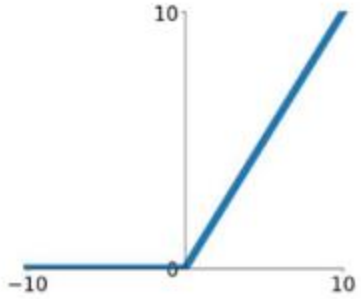
tanh

$$\tanh(x)$$



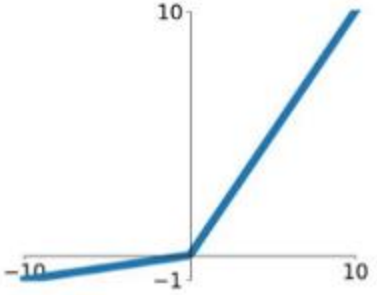
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

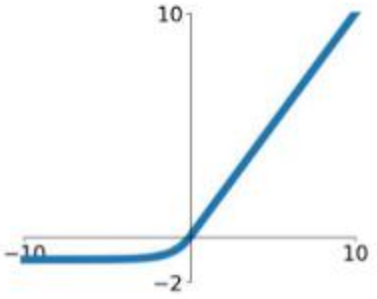


Maxout

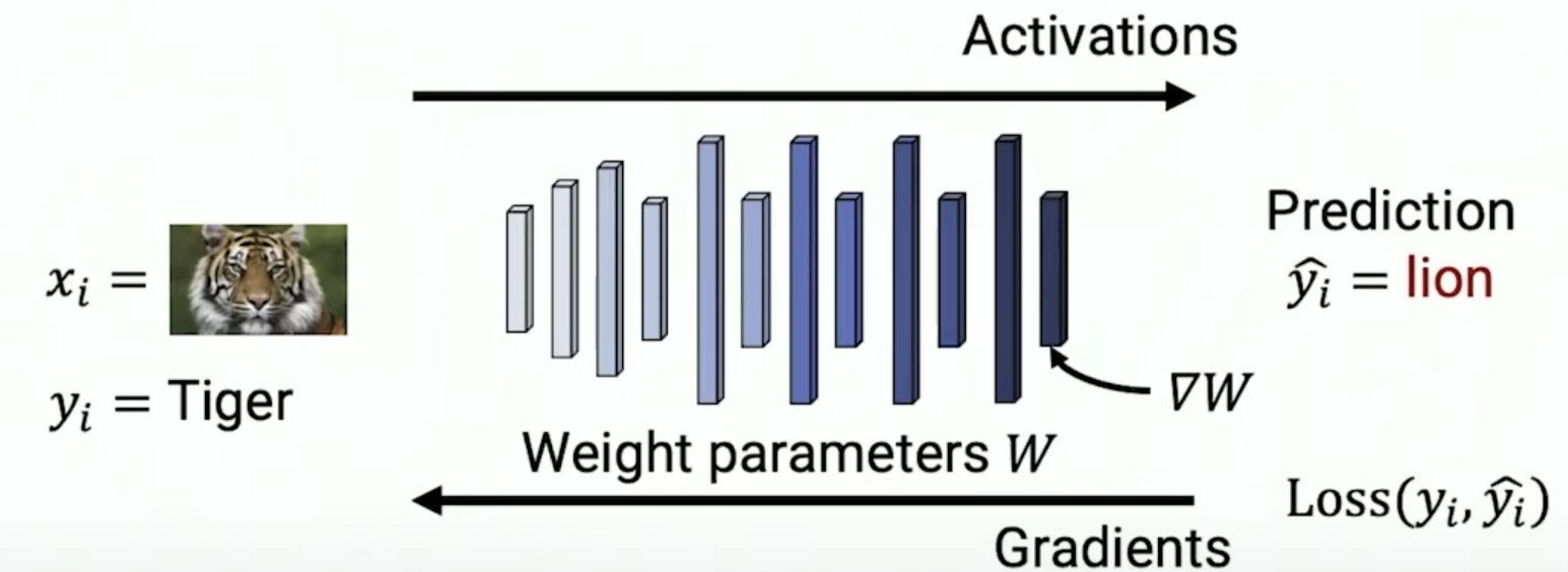
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



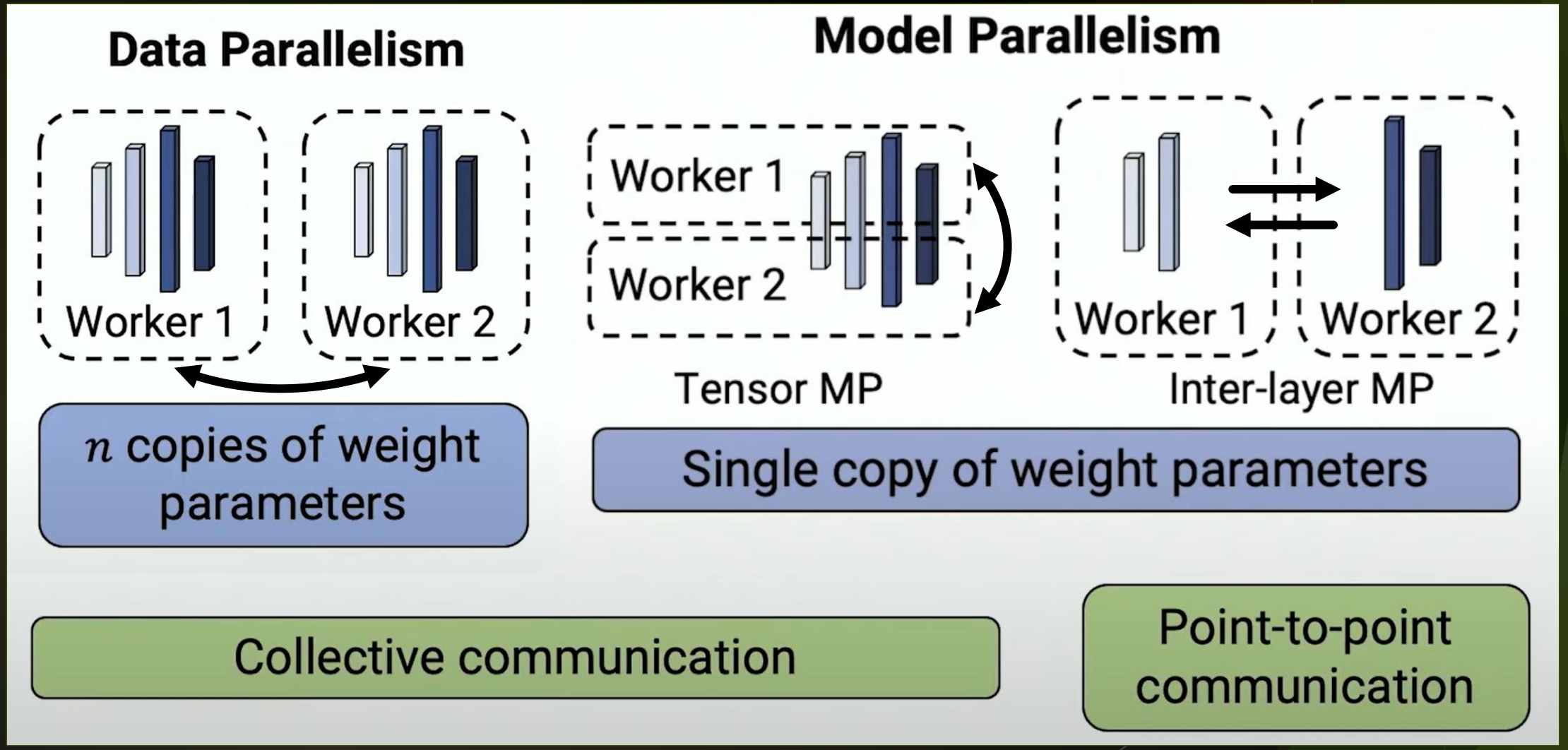
DEEP NEURAL NETWORK (DNN) TRAINING



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

PARALLELIZING DNN TRAINING

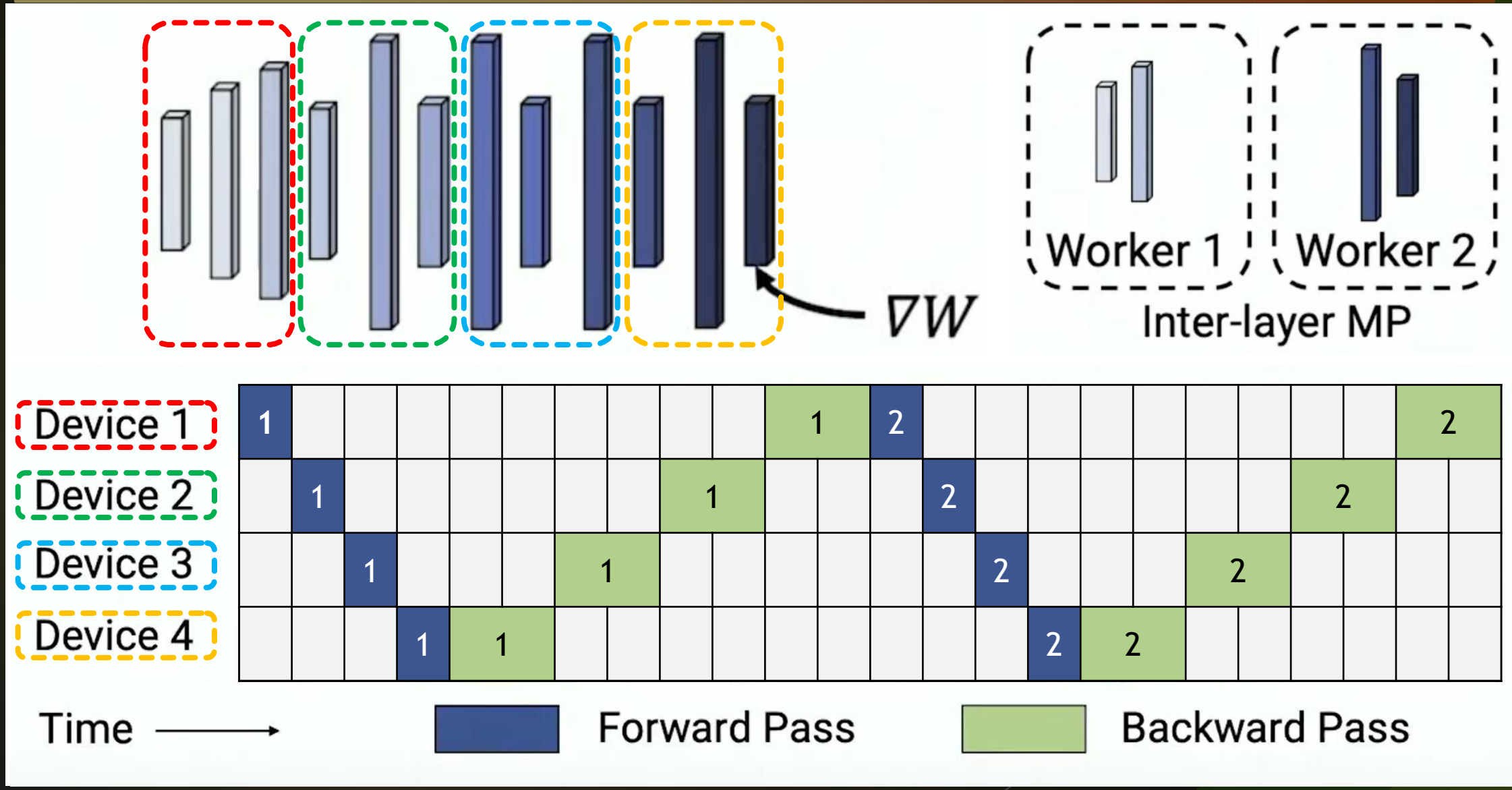


https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

INTER-LAYER MODEL PARALLELISM

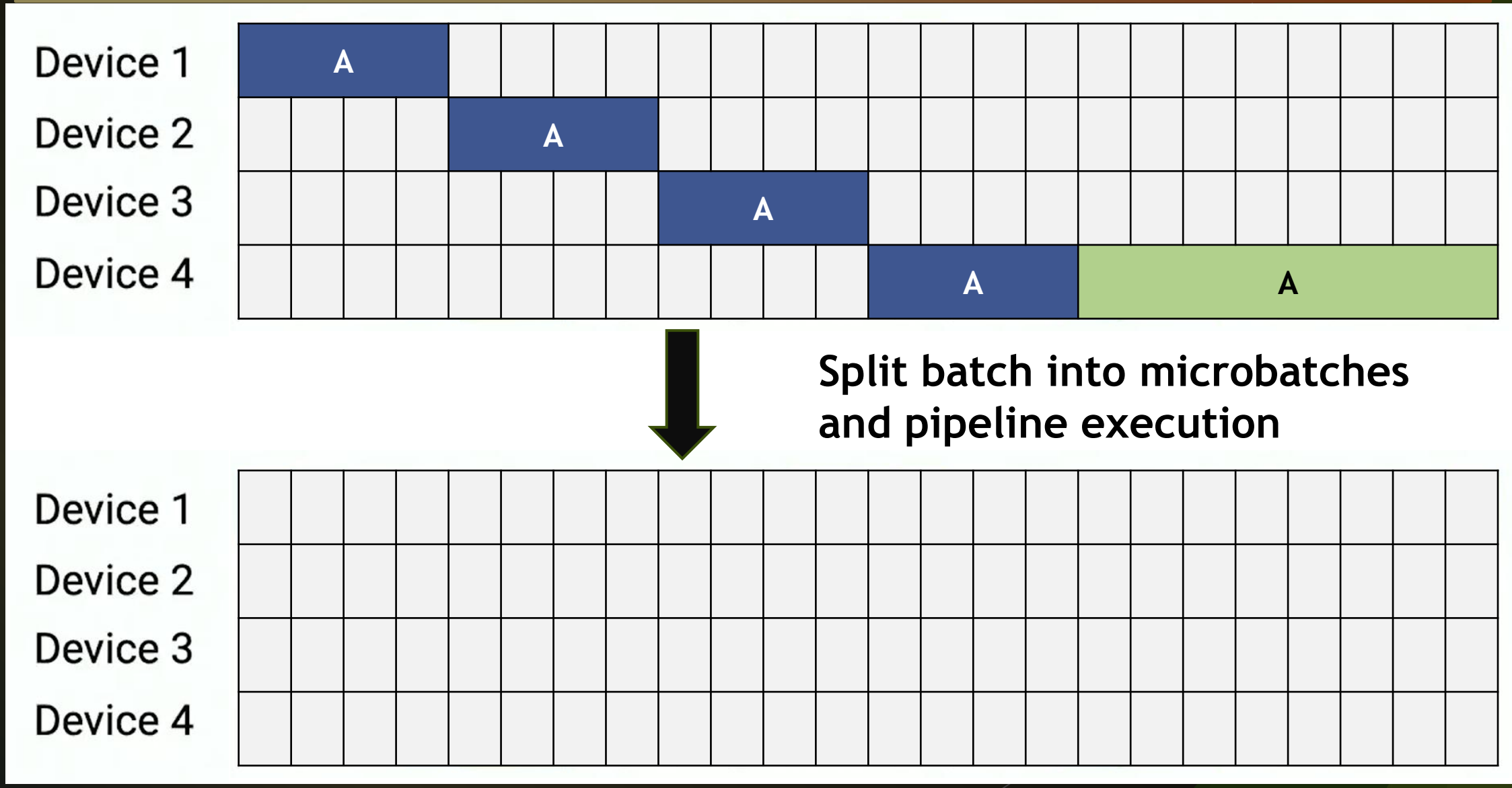
Low throughput due to poor resource utilization



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

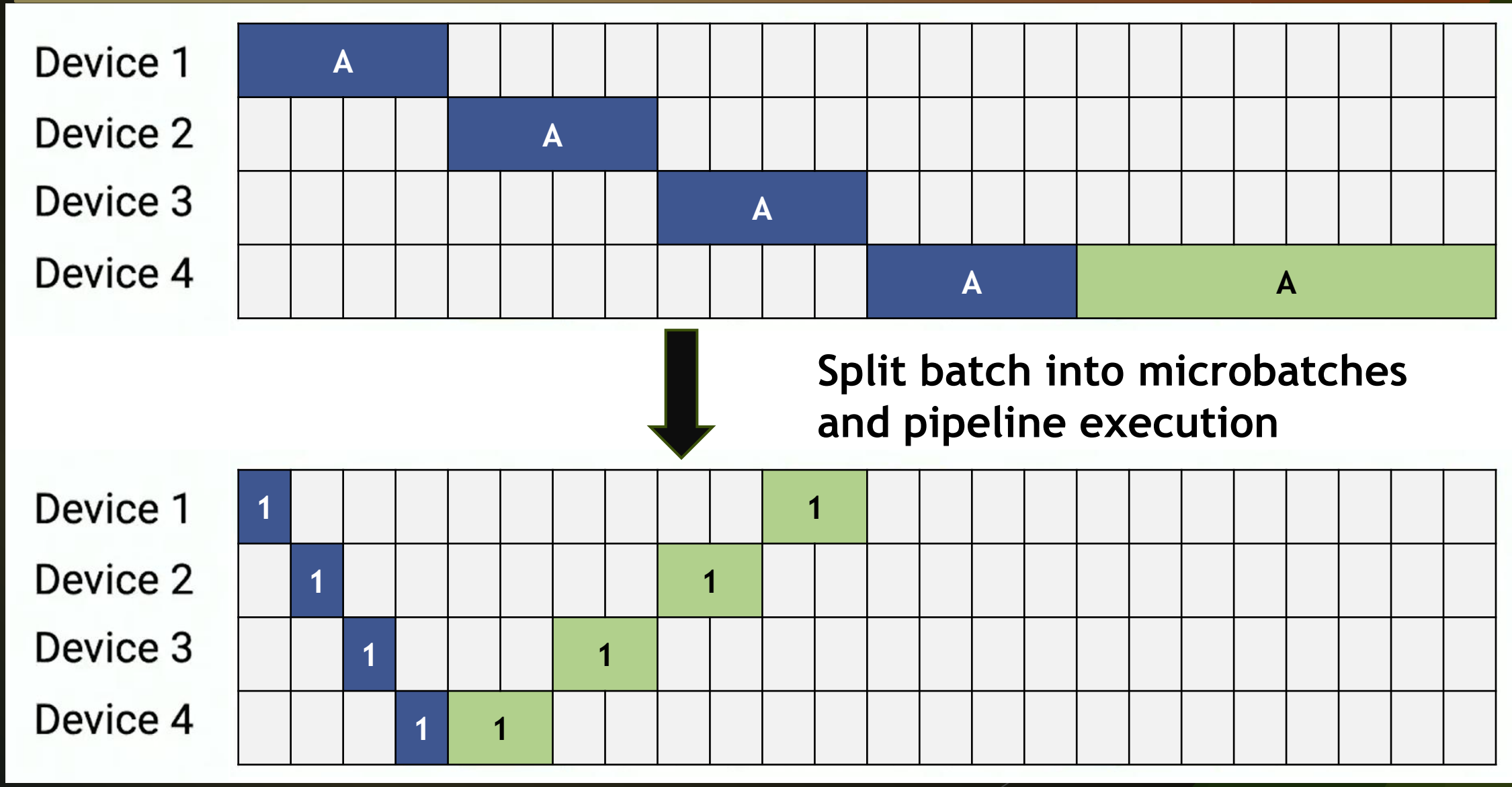
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

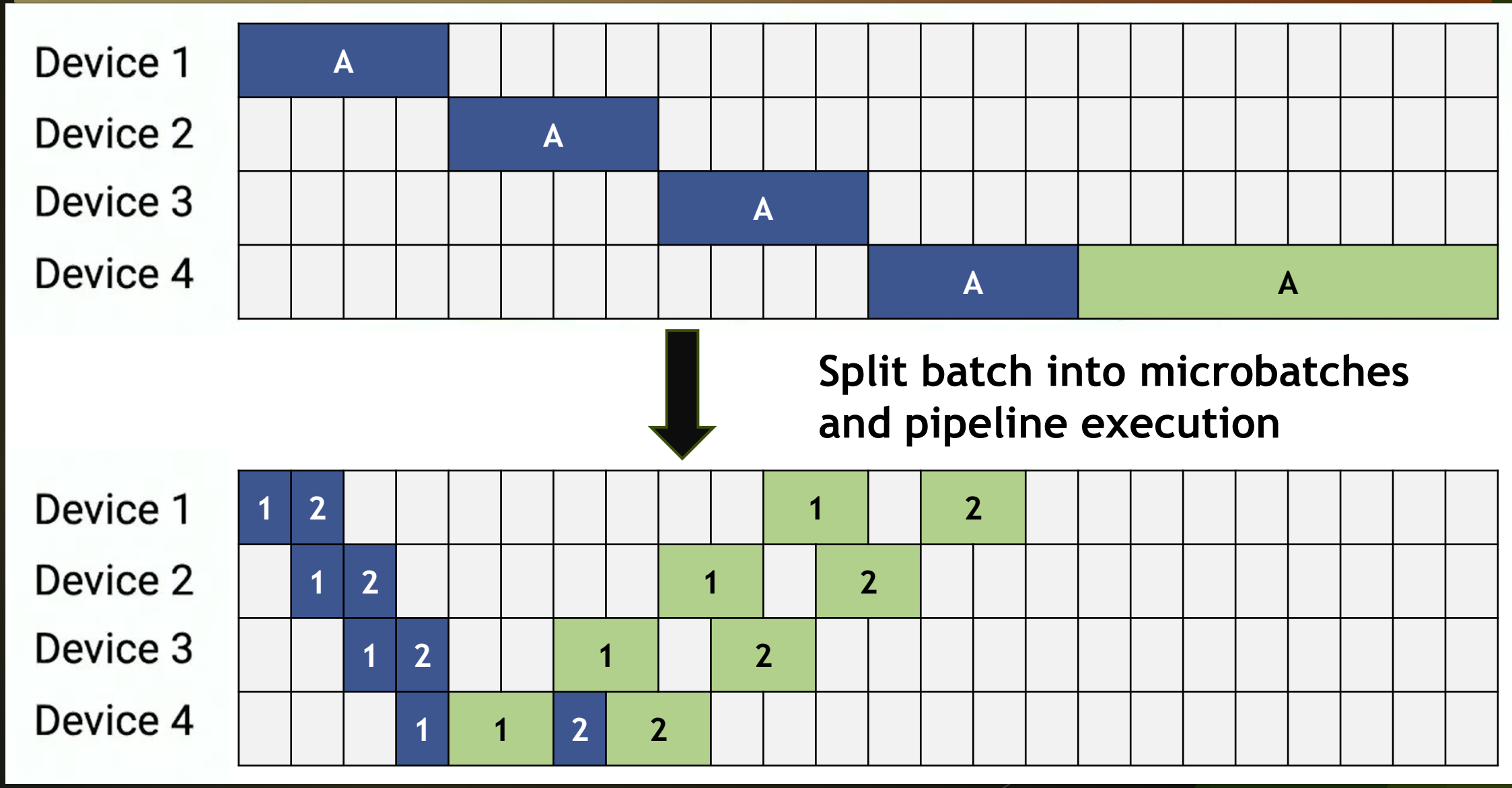
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

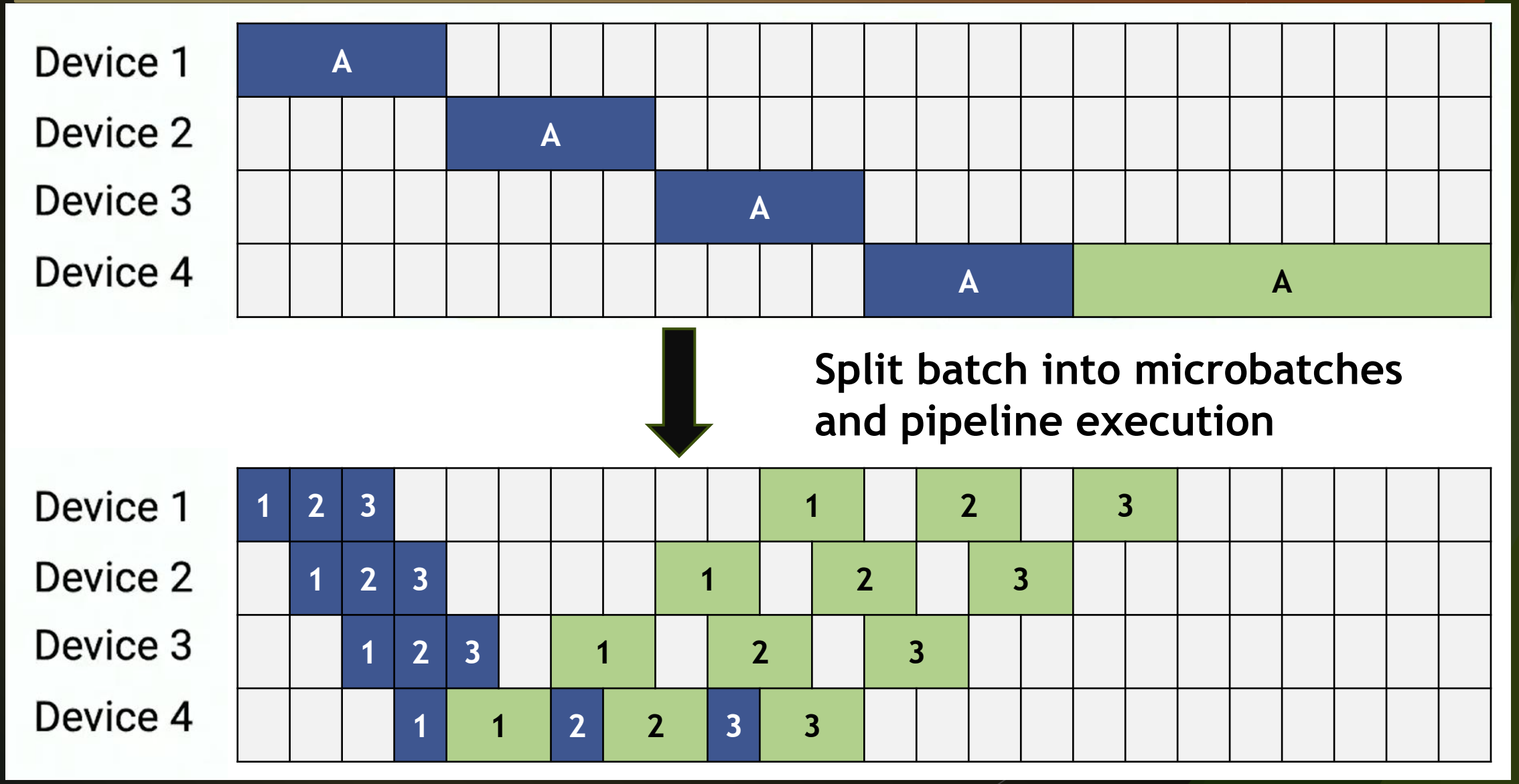
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

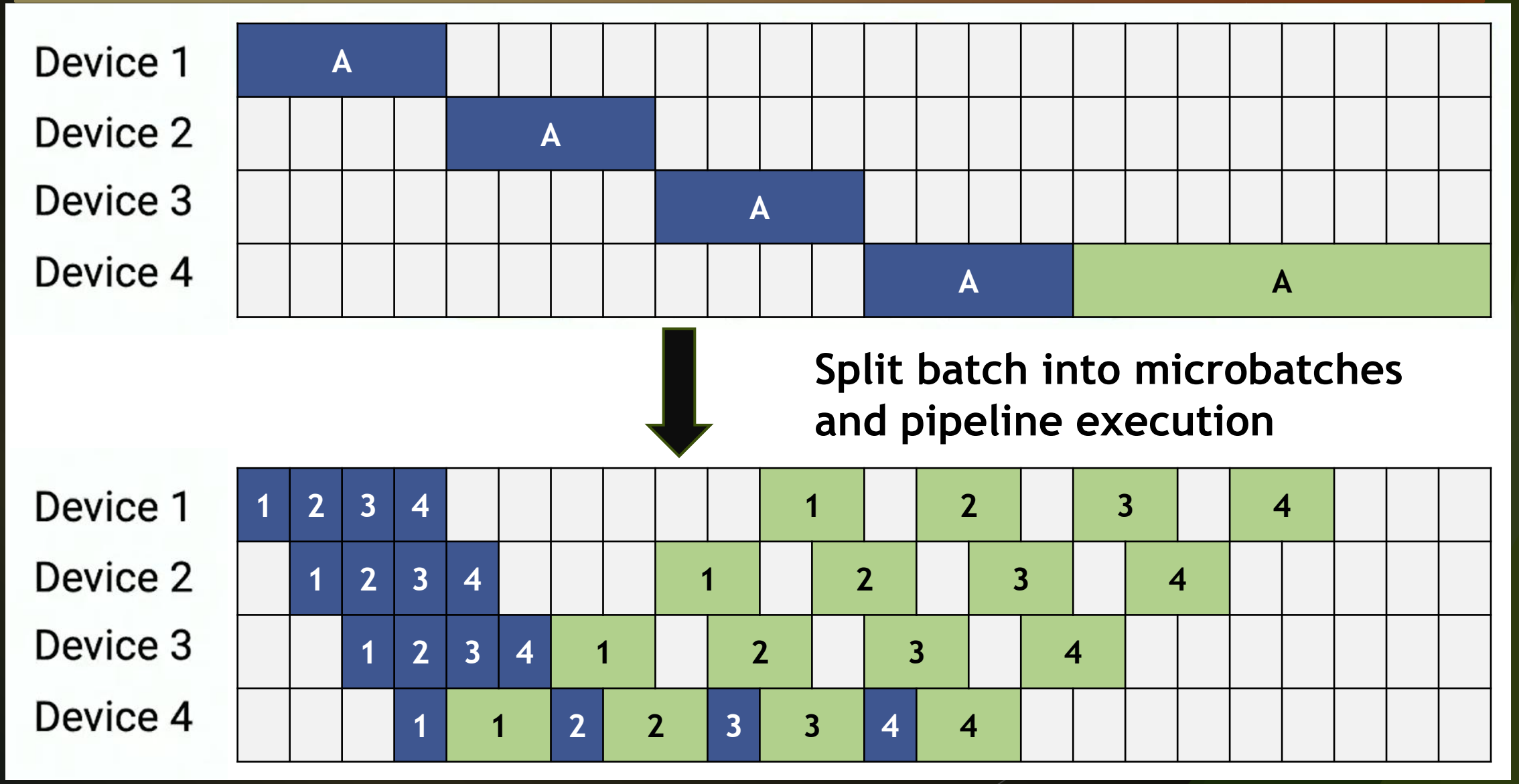
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

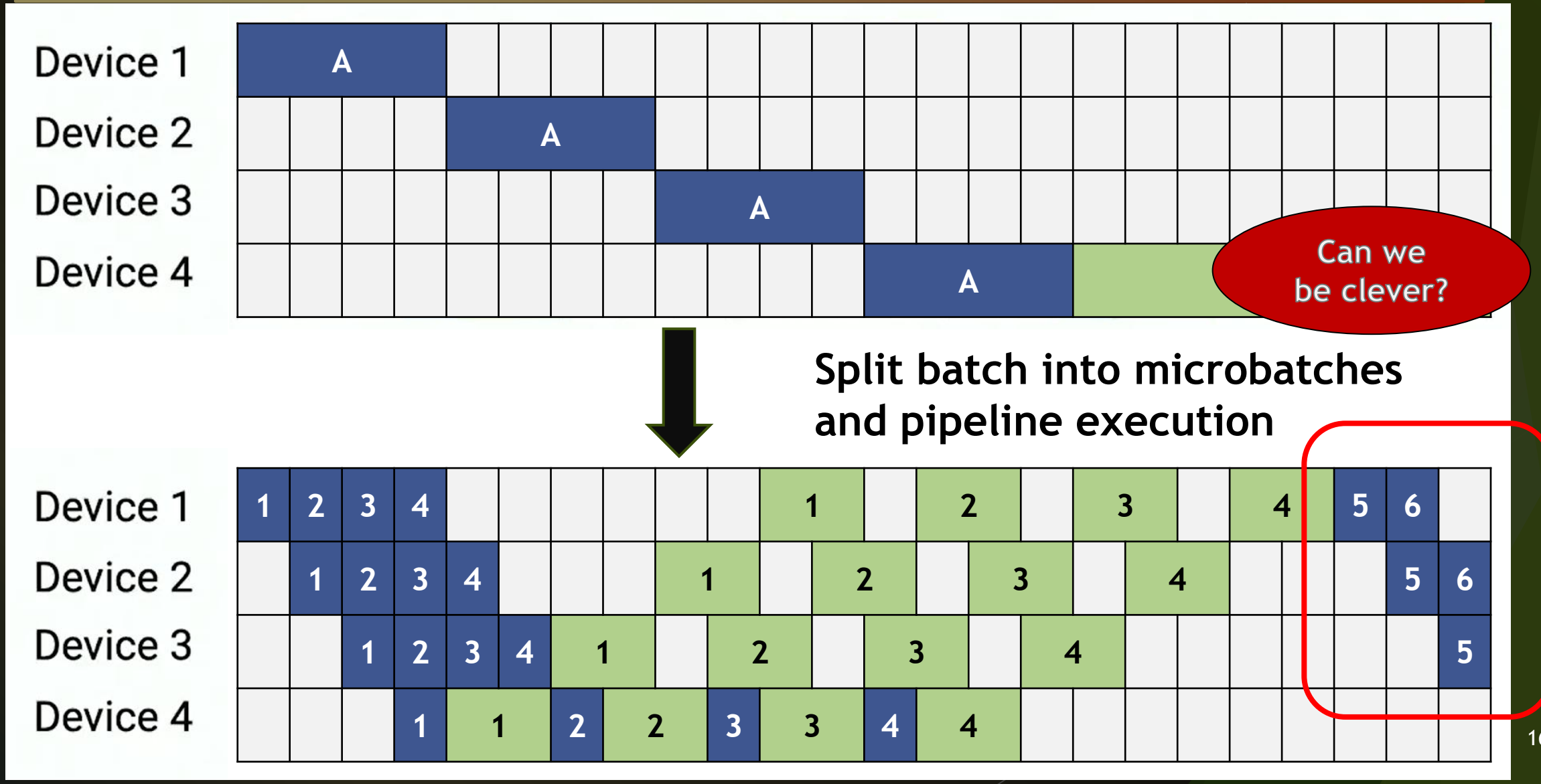
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

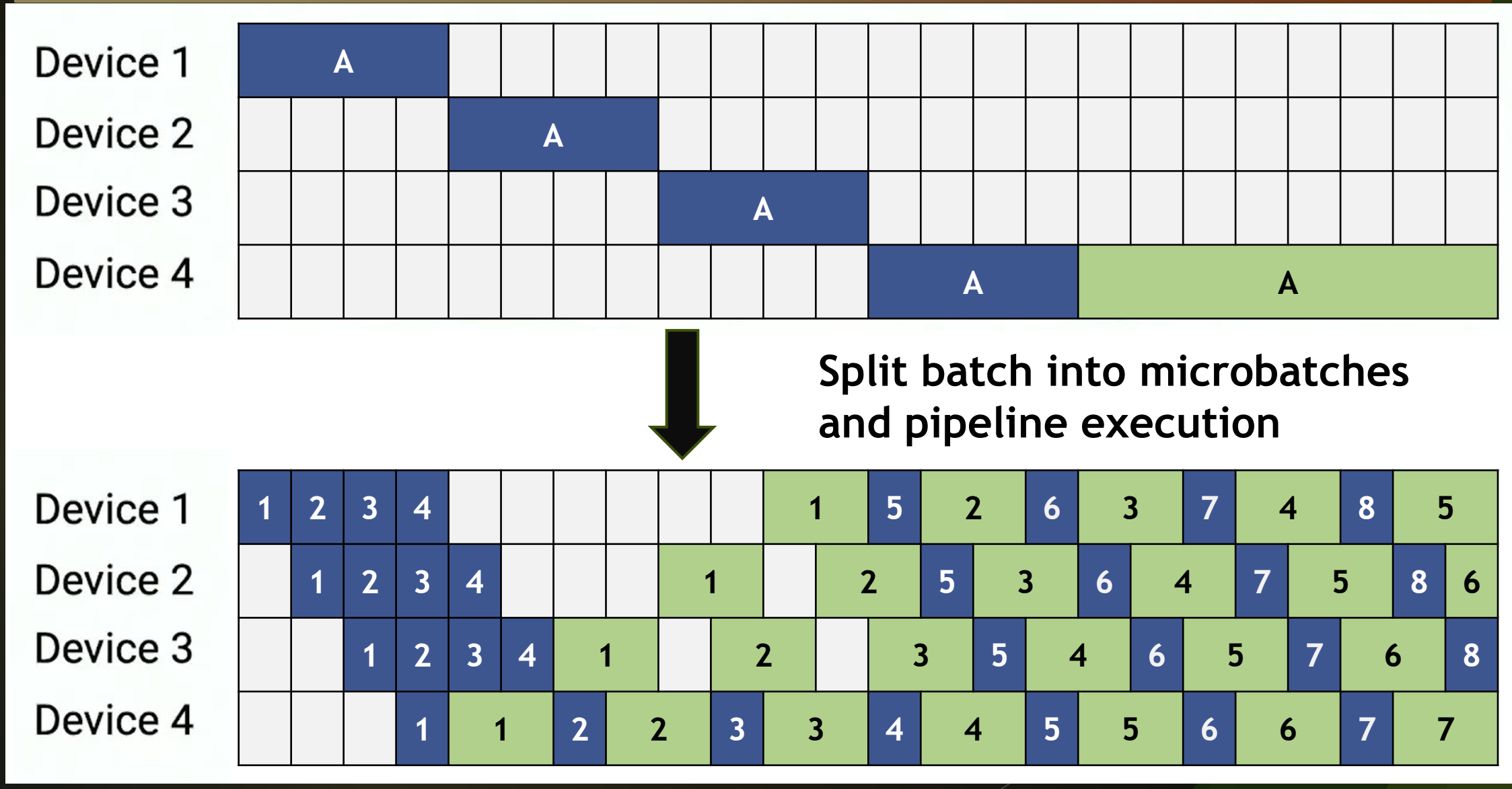
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

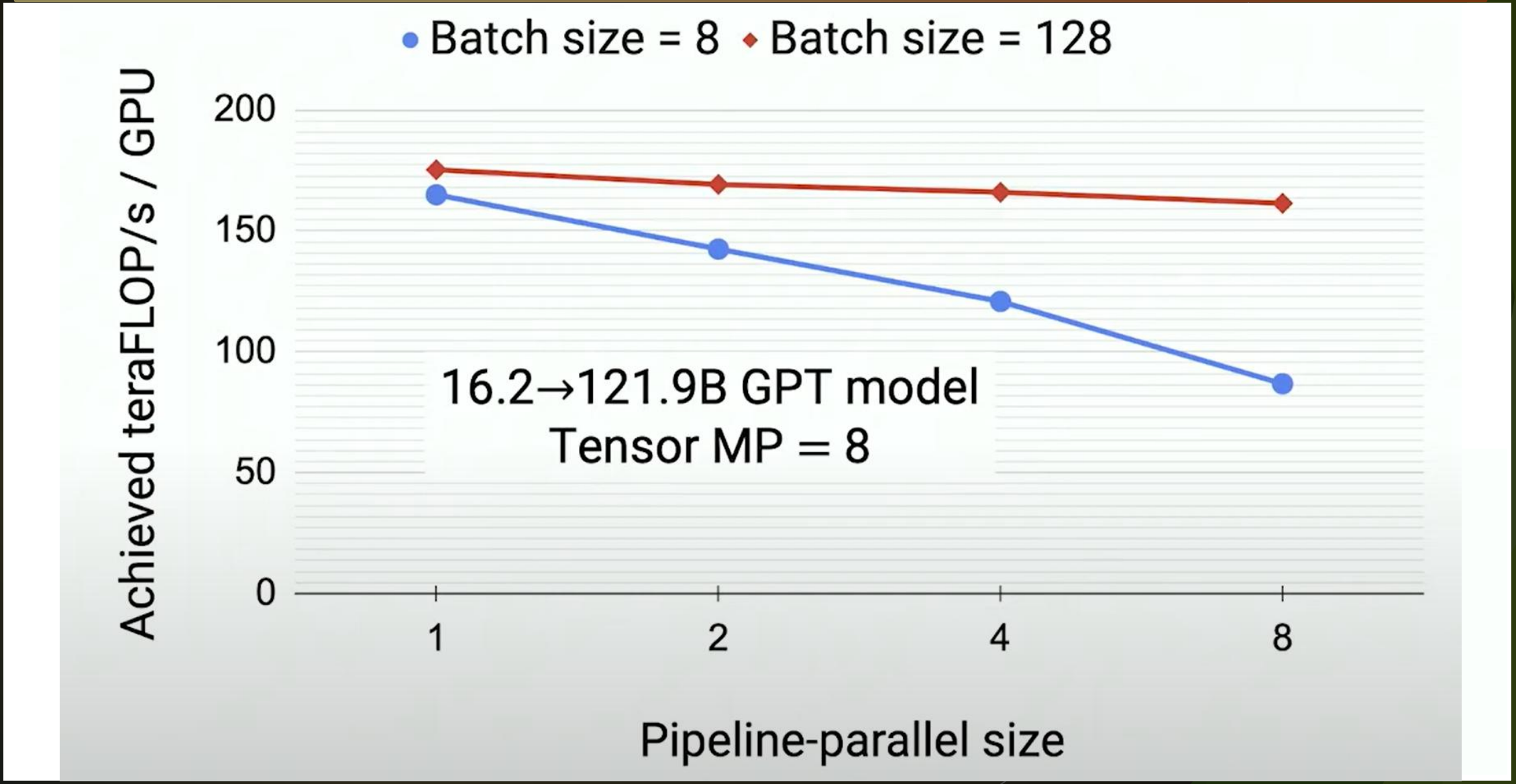
PIPELINE PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

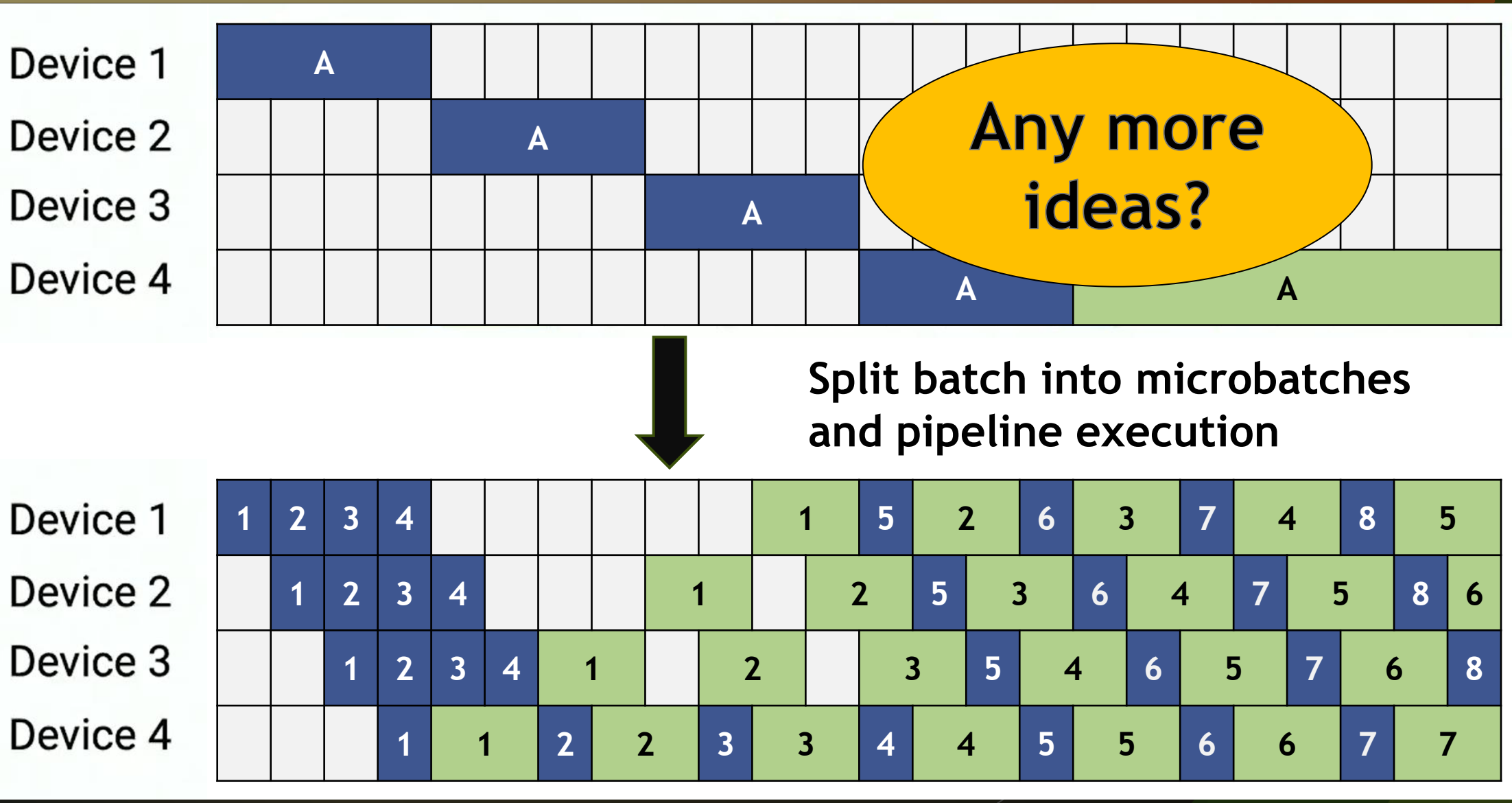
INTER-LAYER MODEL PARALLELISM



https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

INTER-LAYER MODEL PARALLELISM



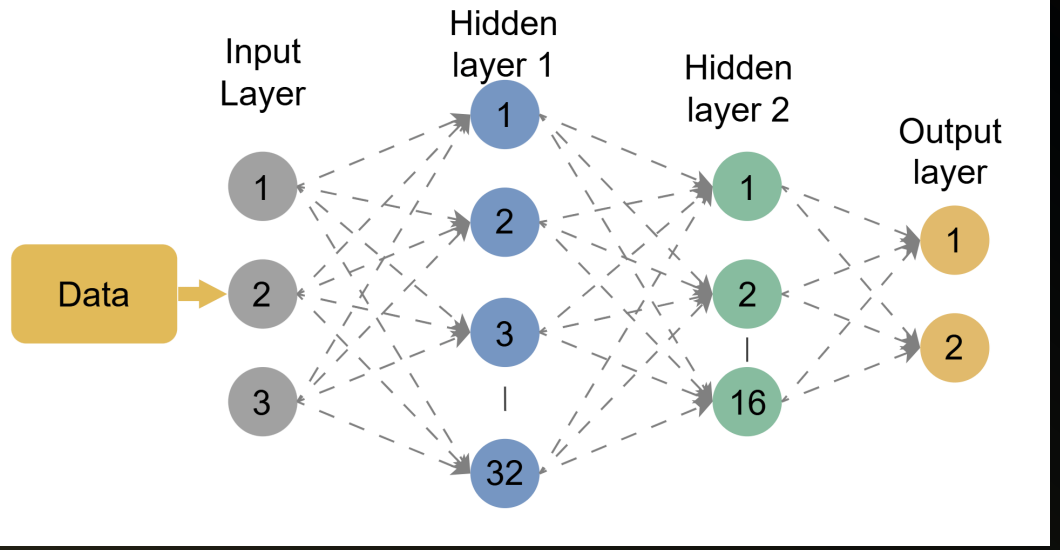
https://www.youtube.com/watch?v=MO_CESBOXgo

https://arxiv.org/abs/2104.04473

EXERCISE.4 PIPELINE MULTI-GPU PARALLELISM

Input Data
1. Living Area
2. Number of bedrooms
3. Number of bathrooms
4. Age of
5. Parking Spaces
6. Number of Floors
7. Land Lot Size
8. Distance to City
9. Neighborhood Crime Rate
10. School Rating
11. Has pool?
12. Has fireplace?
13. Neighborhood Code
14. Year Renovated

Predicting Housing Prices



182,945 Parameters

```
# Define the full model architecture
# (8 hidden layers + output)
self.full_layers = [
    DenseClass(14, 512, lr),  ReLUClass(),
    DenseClass(512, 256, lr), ReLUClass(),
    DenseClass(256, 128, lr), ReLUClass(),
    DenseClass(128, 64, lr),  ReLUClass(),
    DenseClass(64, 32, lr),   ReLUClass(),
    DenseClass(32, 16, lr),   ReLUClass(),
    DenseClass(16, 8, lr),    ReLUClass(),
    DenseClass(8, 4, lr),     ReLUClass(),
    DenseClass(4, 1, lr)
]
```

Step 1: cd `exercises/Exercise.4-Hybrid/`

Step 2: Complete "TODO" tasks in mpiMGR_ASSIGNMENT.py

SOLUTIONS: cd `exercises/Exercise.4-Hybrid/.soln`

FINAL TAKEAWAYS

What we learned:

- Review of Message Passing Concepts
- MPI Derived Datatypes and Custom Operations
- MPI Virtual Topologies
- MPI Neighborhood Collectives
- Hybrid-Parallel Programming