TRACK II

ADVANCED DISTRIBUTED-MEMORY PROGRAMMING

DAY 1

2024 IHPCSS

TRACK 2

TEAM

Andrew Kirby

Instructor

Research Scientist
School of Computing
University of Wyoming

TRACK 2

TEAM

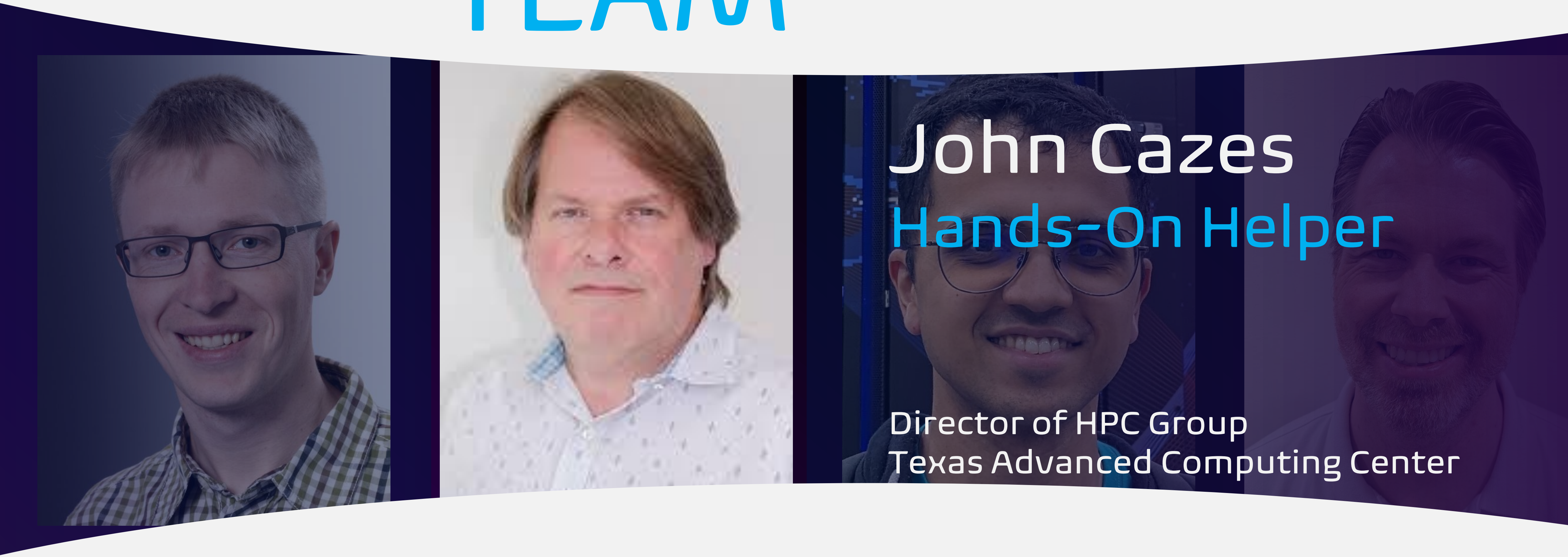
Ilya Zhukov

Hands-On Helper

HPC Application Analyst
Jülich Supercomputing Centre

TRACK 2

TEAM



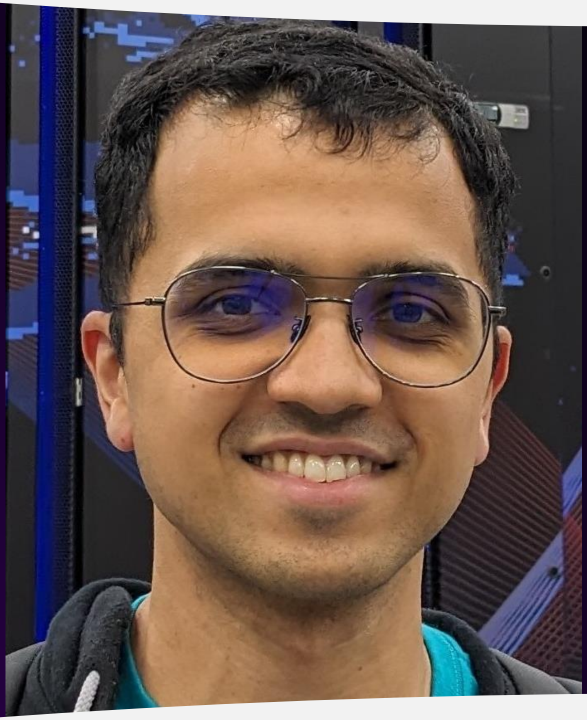
John Cazes

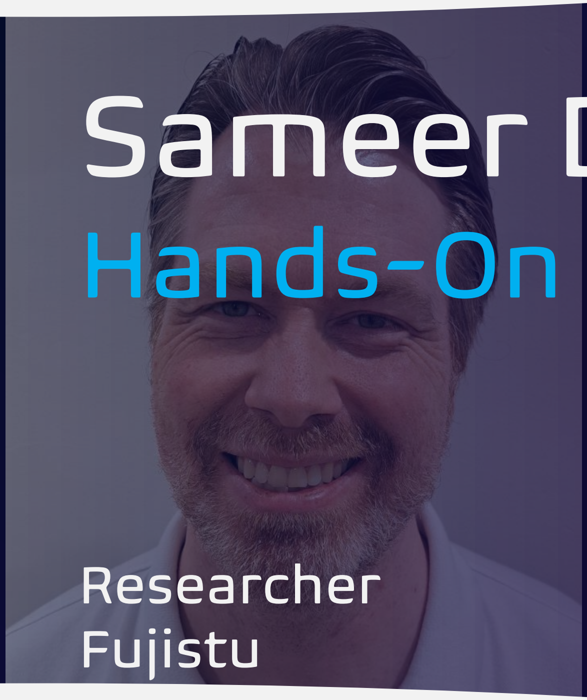
Hands-On Helper

Director of HPC Group
Texas Advanced Computing Center

TRACK 2

TEAM





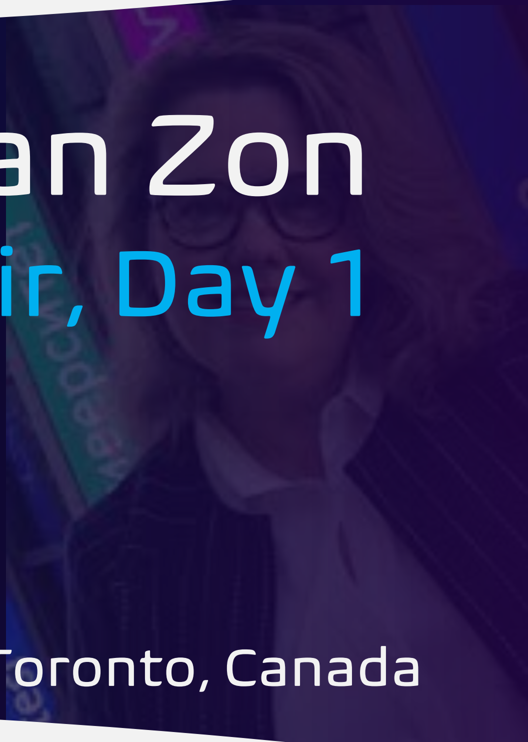
Sameer Deshmukh

Hands-On Helper

Researcher
Fujistu

TRACK 2

TEAM



Ramses van Zon

Session Chair, Day 1

Training Lead
SciNet, University of Toronto, Canada

TRACK 2

TEAM

Matthais Brust

Session Chair, Day 2

Research Associate
University of Luxembourg

TRACK 2

TEAM

Julie Wernert

Evaluator

Lead Cyberinfrastructure and Evaluation
Indiana University

LECTURE CONTENT INSPIRATION

Ludovic Capelli
David Henty

EPCC Archer Tutorials
SC17 MPI Tutorial

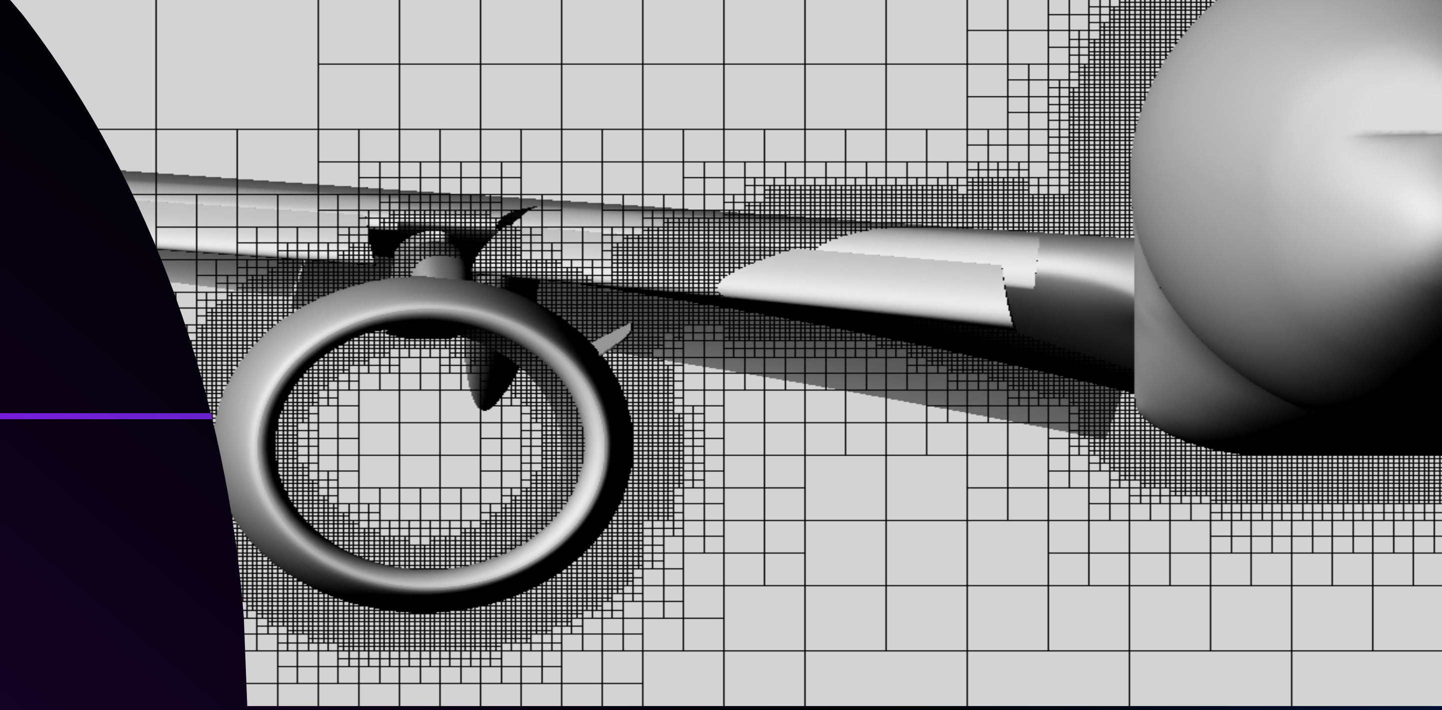
https://web.cels.anl.gov/~thakur/sc17-mpi-tutorial/

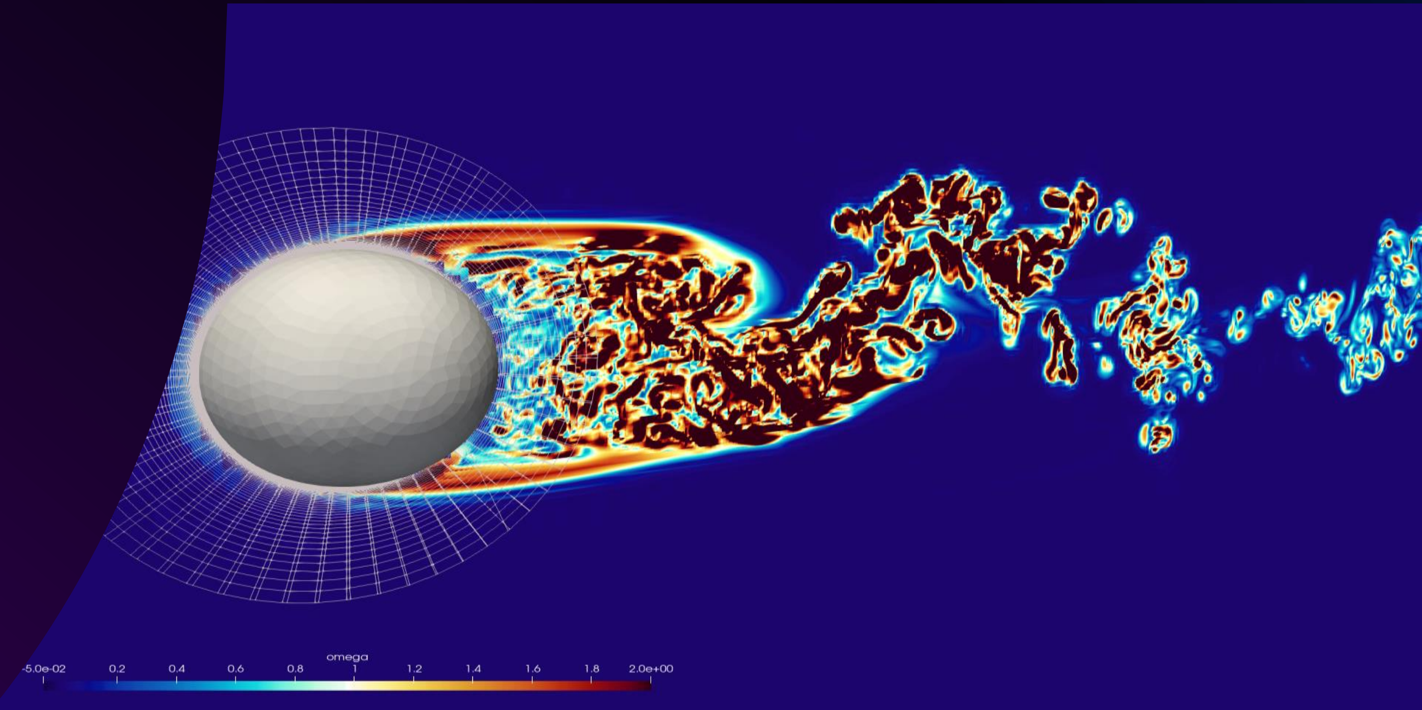
TRACK OVERVIEW

MOTIVATION

MOTIVATION

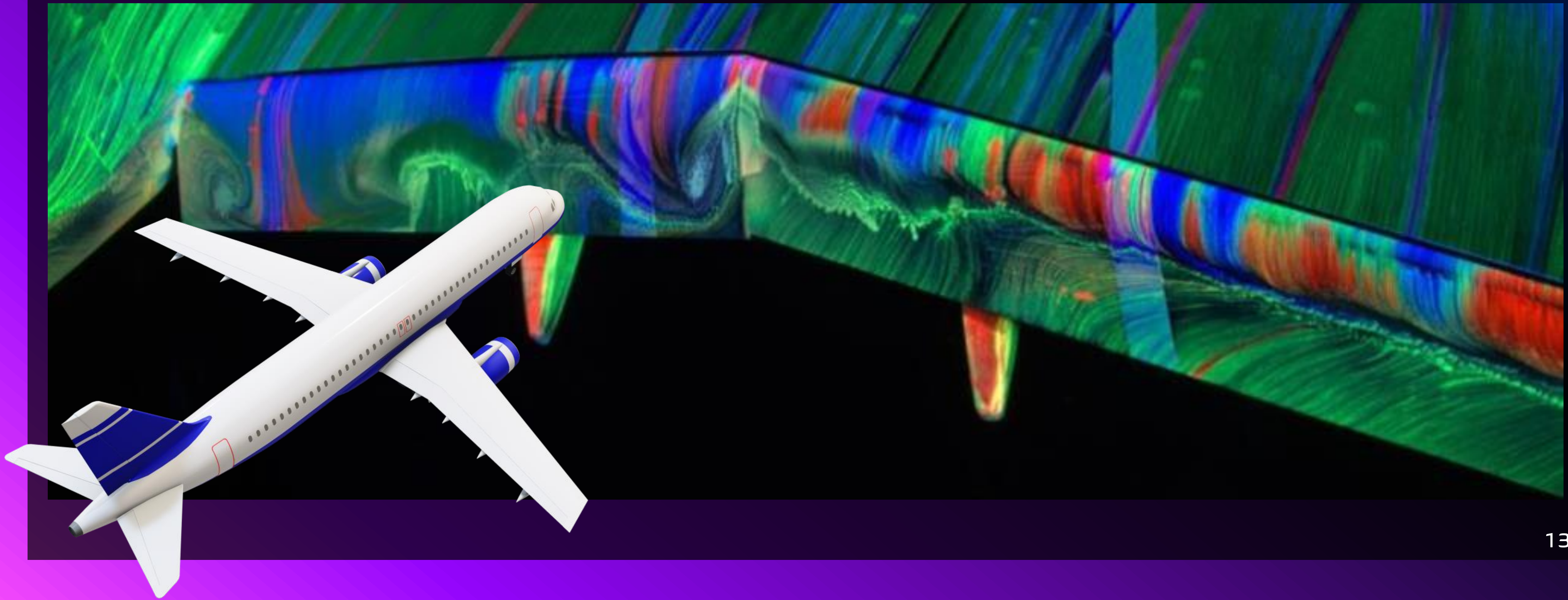
Mesh-Based Applications



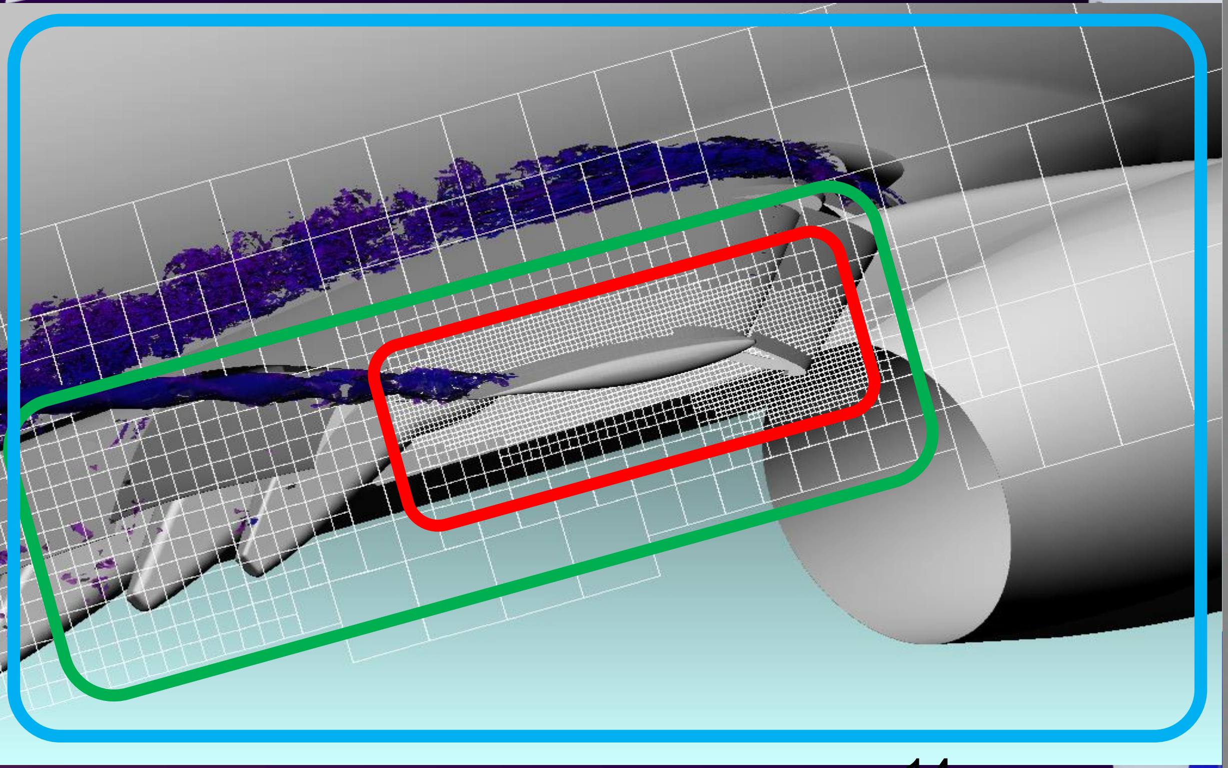


COMPUTATIONAL FLUID DYNAMICS

COMPUTATIONAL FLUID DYNAMICS

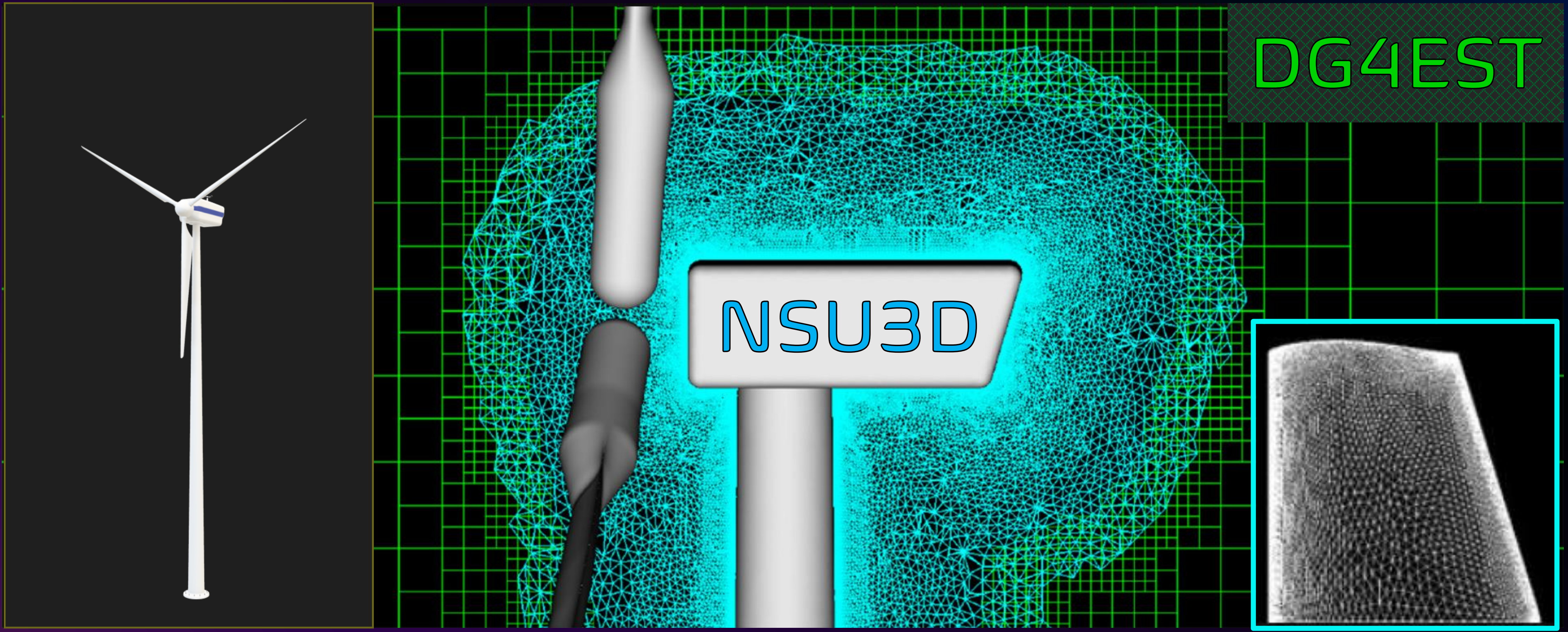


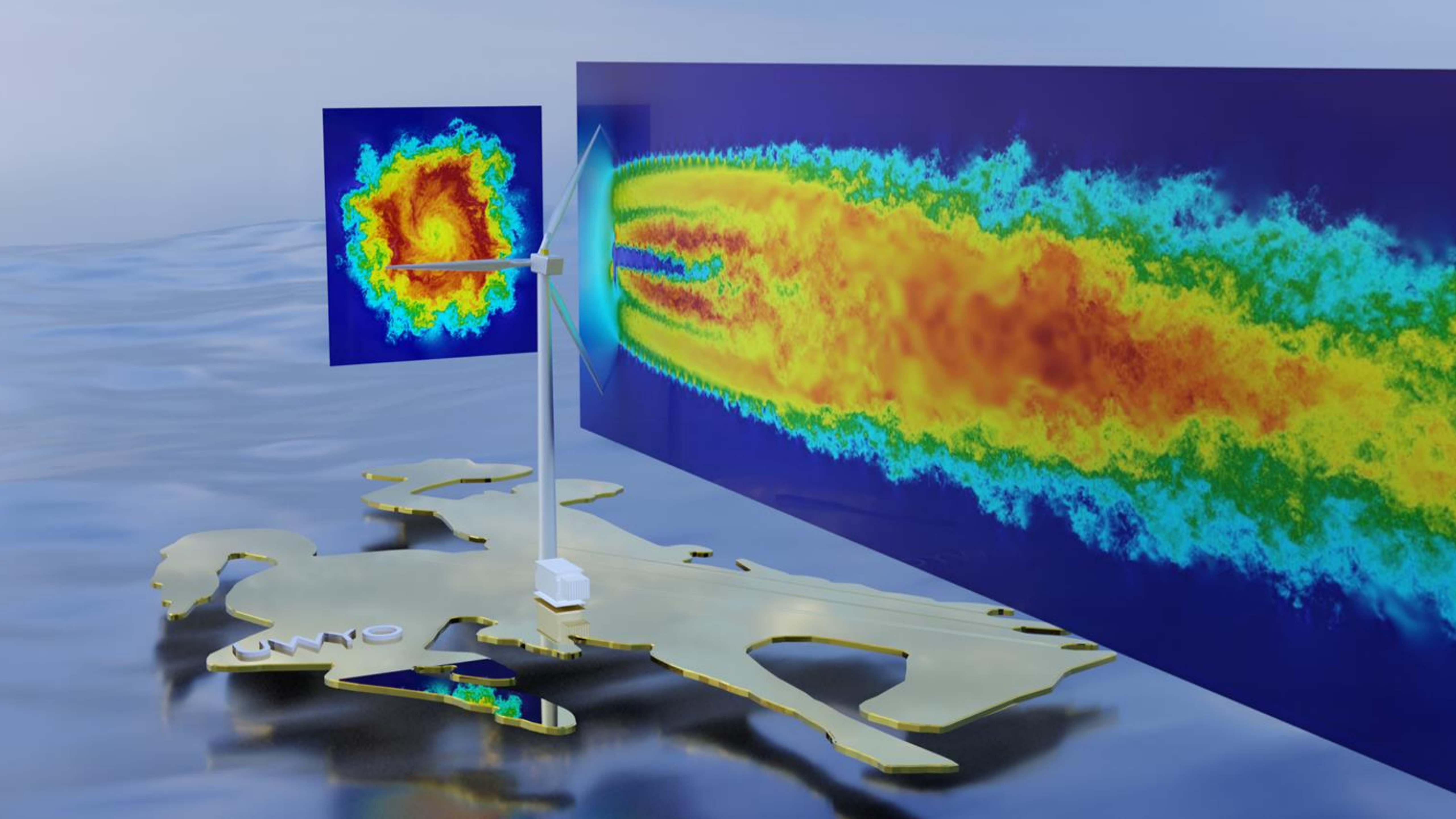

VARIABLE DATA SIZES




290 Million Pts
3rd Order – 27 pts
4th Order – 64 pts
6th Order – 216 pts

NONUNIFORM MESHES / MPI SUBGROUPS




UWYO

TRACK OVERVIEW

- **Section I –** Derived Datatypes & Ops
- **Section II –** Cartesian Virtual Topologies
- **Section III –** Graph Virtual Topologies
- **Section IV –** Neighborhood Collectives
- **Section V –** Graph Partitioning

DAY 1 AGENDA

Section I – Derived Datatypes & Ops

Hands On Activity # 1

Coffee Break (15:30-16:00)

Section II – Cartesian Virtual Topologies

Hands On Activity # 2

SECTION I

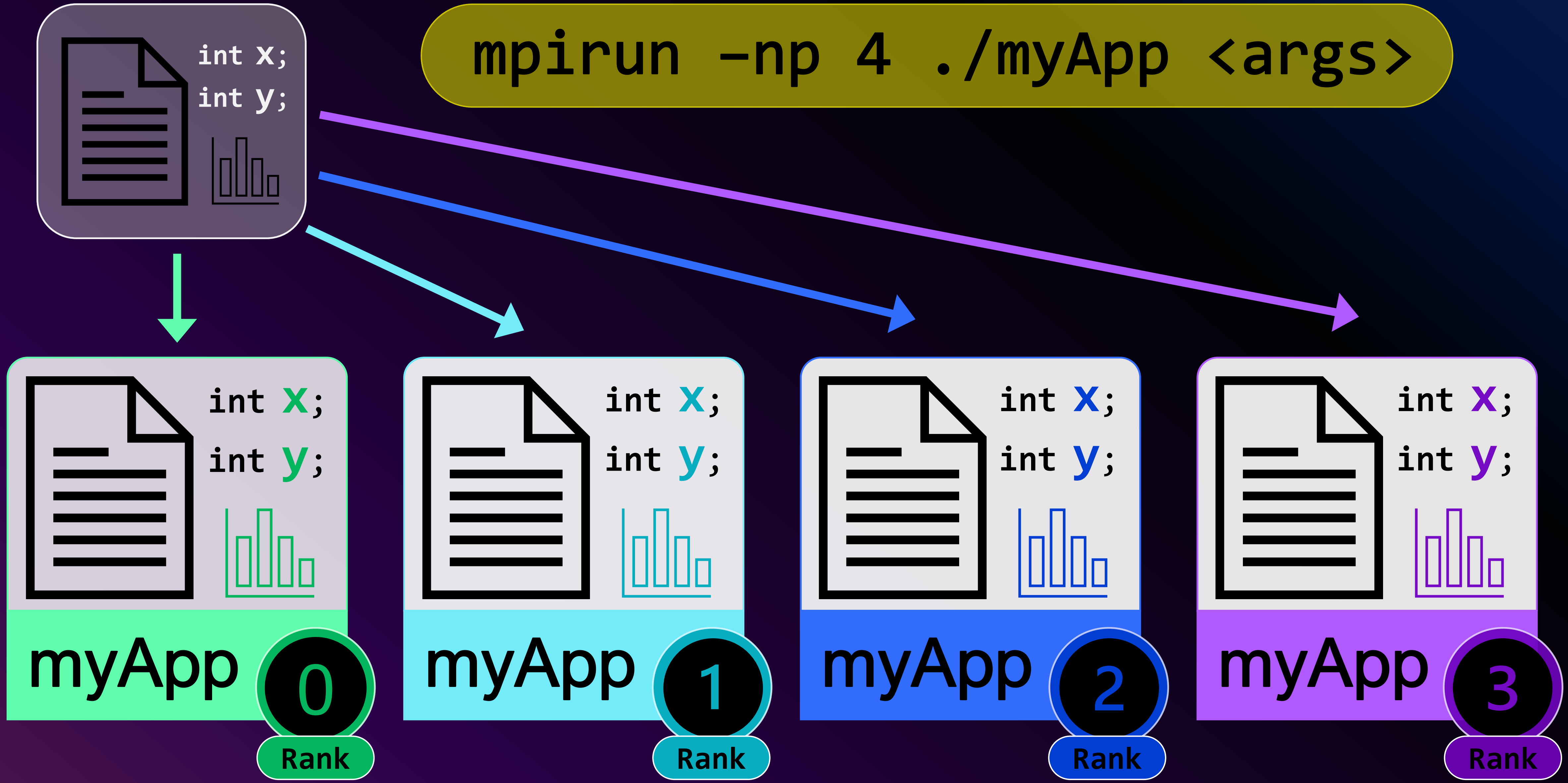
DATATYPES & OPS

```
mpirun -np 4 ./myApp <args>
```

BRIEF REVIEW

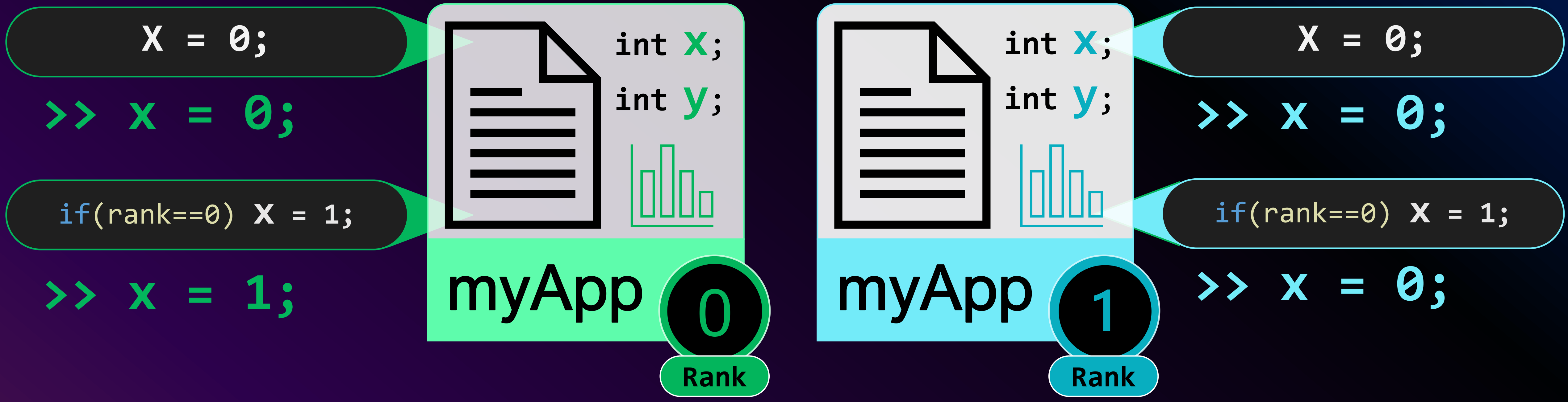
Distributed-Memory Parallel Computing

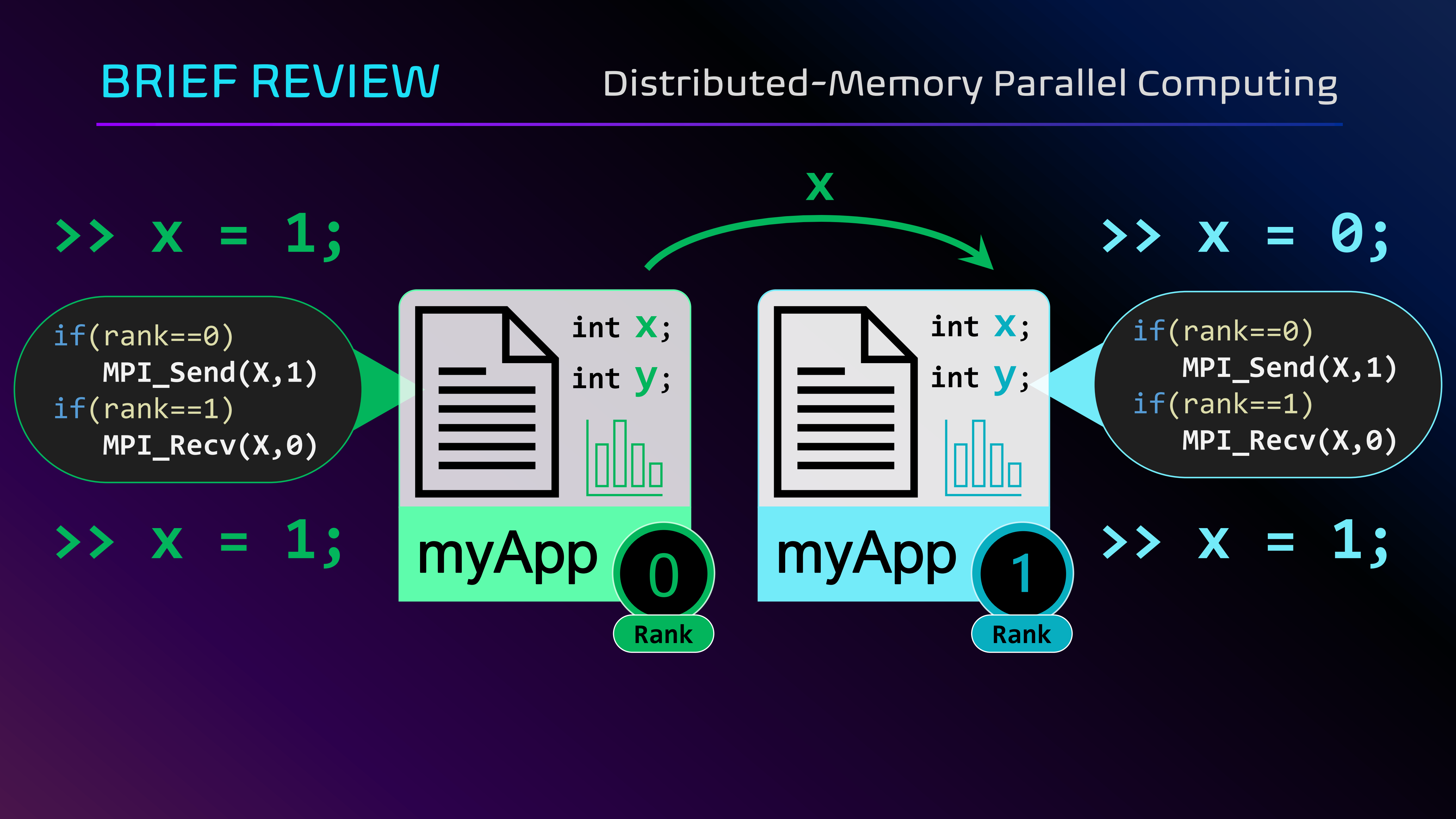
`mpirun -np 4 ./myApp <args>`



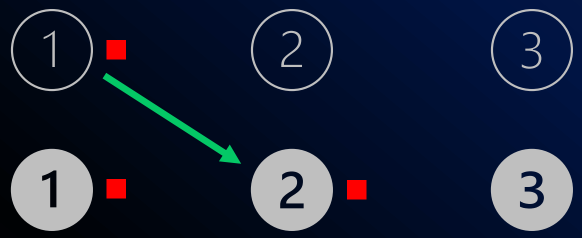
BRIEF REVIEW

Distributed-Memory Parallel Computing




BRIEF REVIEW
Distributed-Memory Parallel Computing
x
>> x = 1;
>> x = 0;
if(rank==0)
MPI_Send(X,1)
if(rank==1)
MPI_Recv(X,0)
int x;
int y;
myApp
0
Rank
int x;
int y;
myApp
1
Rank
if(rank==0)
MPI_Send(X,1)
if(rank==1)
MPI_Recv(X,0)
>> x = 1;
>> x = 1;

MPI SEND/RECV



① RANK

```
int MPI_Send(const void *sbuf,         // [in] Initial address of send buffer
             int count,                 // [in] Maximum number of elements to send
             MPI_Datatype datatype,     // [in] Datatype of each send buffer entry
             int dest,                  // [in] Rank of destination (integer)
             int tag,                   // [in] Message tag (integer)
             MPI_Comm comm);            // [in] Communicator (handle)
```

② RANK

```
int MPI_Recv(void *rbuf,               // [out] Initial address of receive buffer
             int count,                 // [in] Maximum number of elements to recv
             MPI_Datatype datatype,     // [in] Datatype of each recv buffer entry
             int source,                // [in] Rank of source (integer)
             int tag,                   // [in] Message tag (integer)
             MPI_Comm comm,             // [in] Communicator (handle)
             MPI_Status *status);       // [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);
```

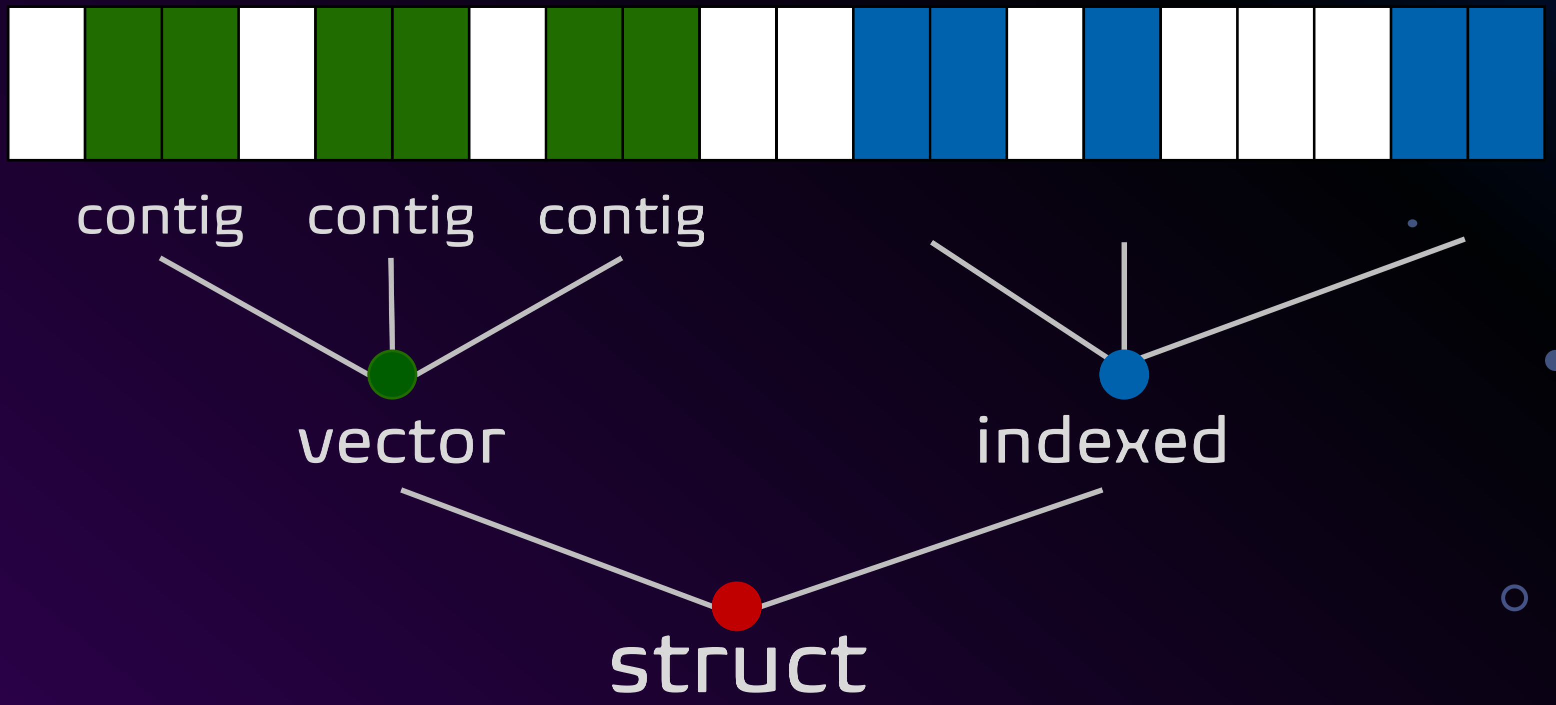
MPI DATA TYPES

We can serialize custom data into a message stream

>> Declarative specification of data layout <<

<u>Derived Data Types</u>

- 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 multi-dimensional array
`MPI_Type_create_darray`	> distribution of a ndim-array into a grid of ndim-logical processes
`MPI_Type_create_struct`	> fully general datatype

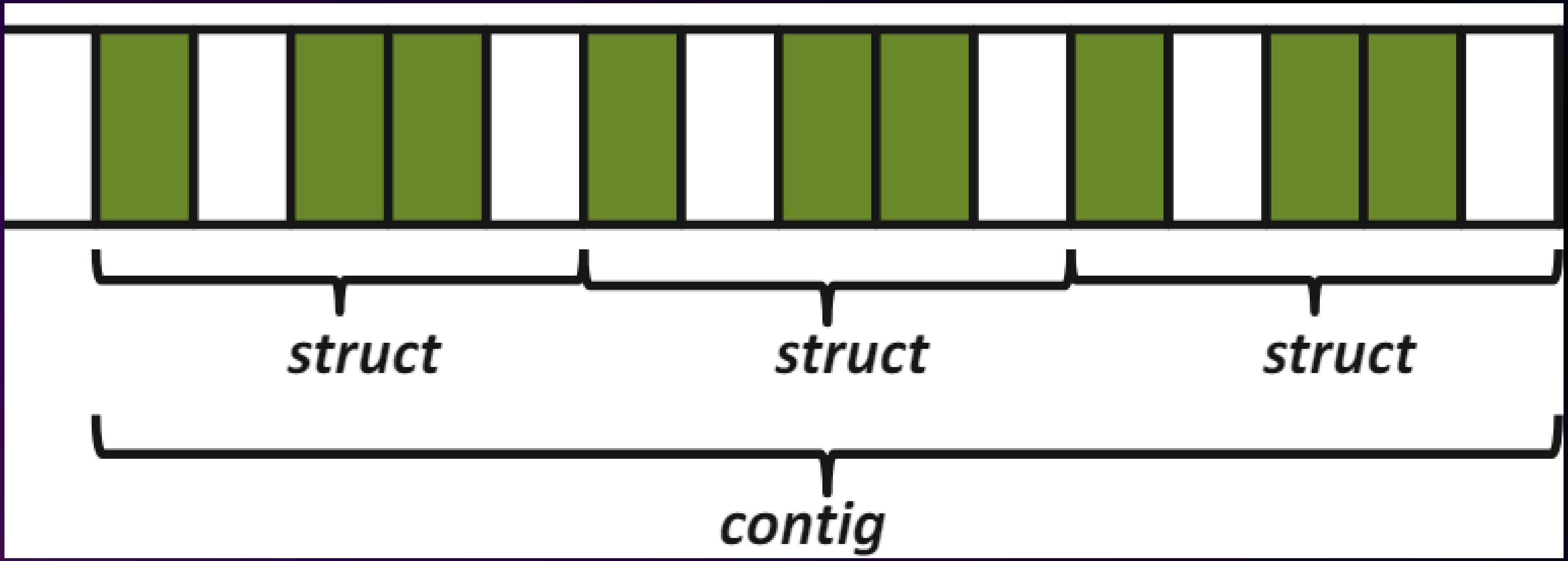
Discussed in Parallel I/O Session (Wednesday)

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);
```

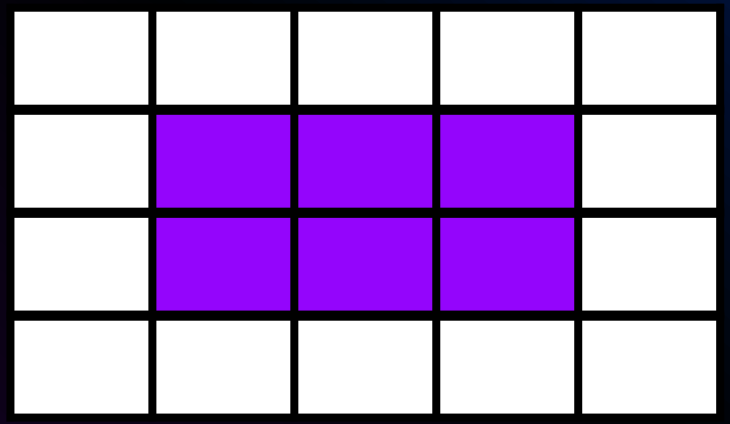
float A[4][4]

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_VECTOR

Creates a vector (strided) datatype.

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

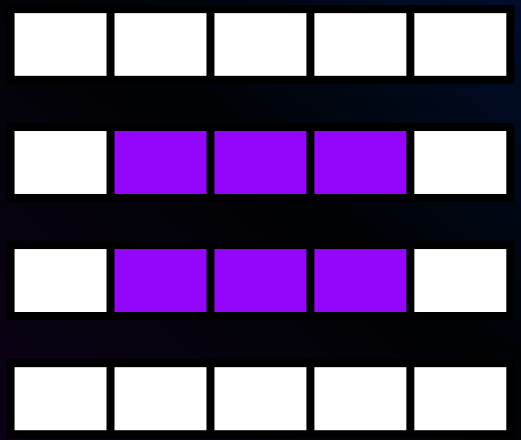


How can we send the subarray?

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);
```

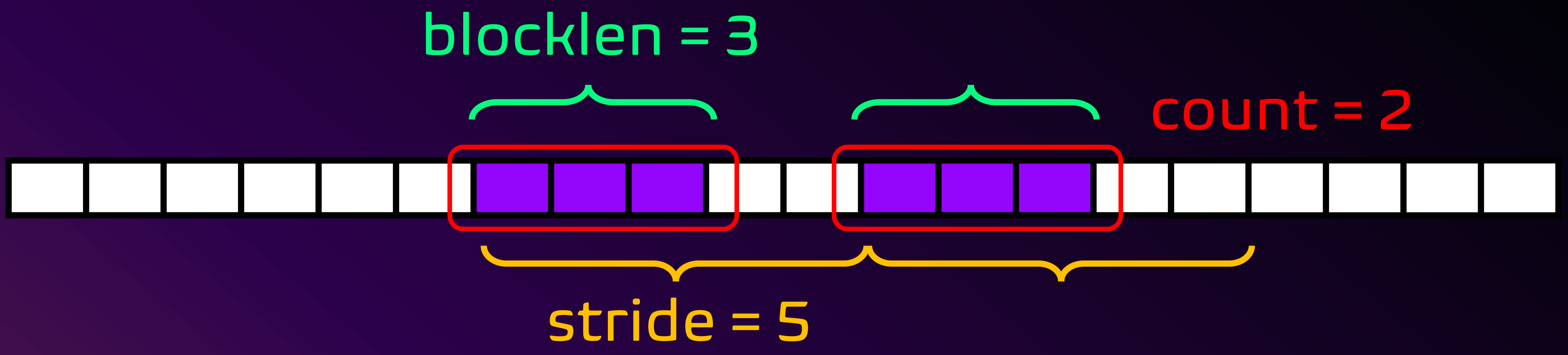
count : Number of blocks.
> "How many chunks of data?"

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);
```

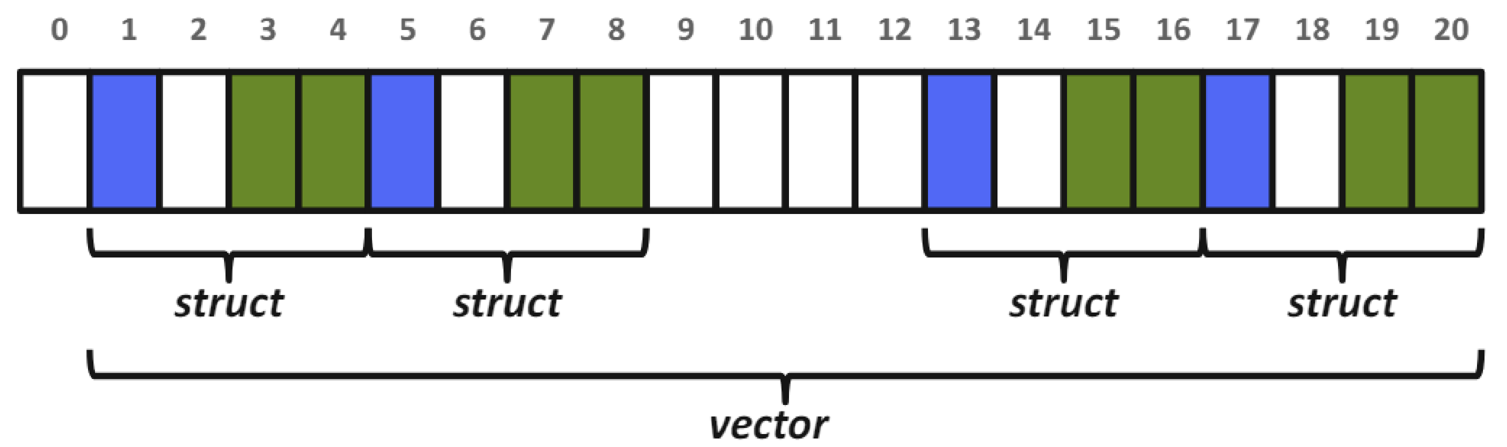
count : Number of blocks.
> "How many chunks of data?"

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).

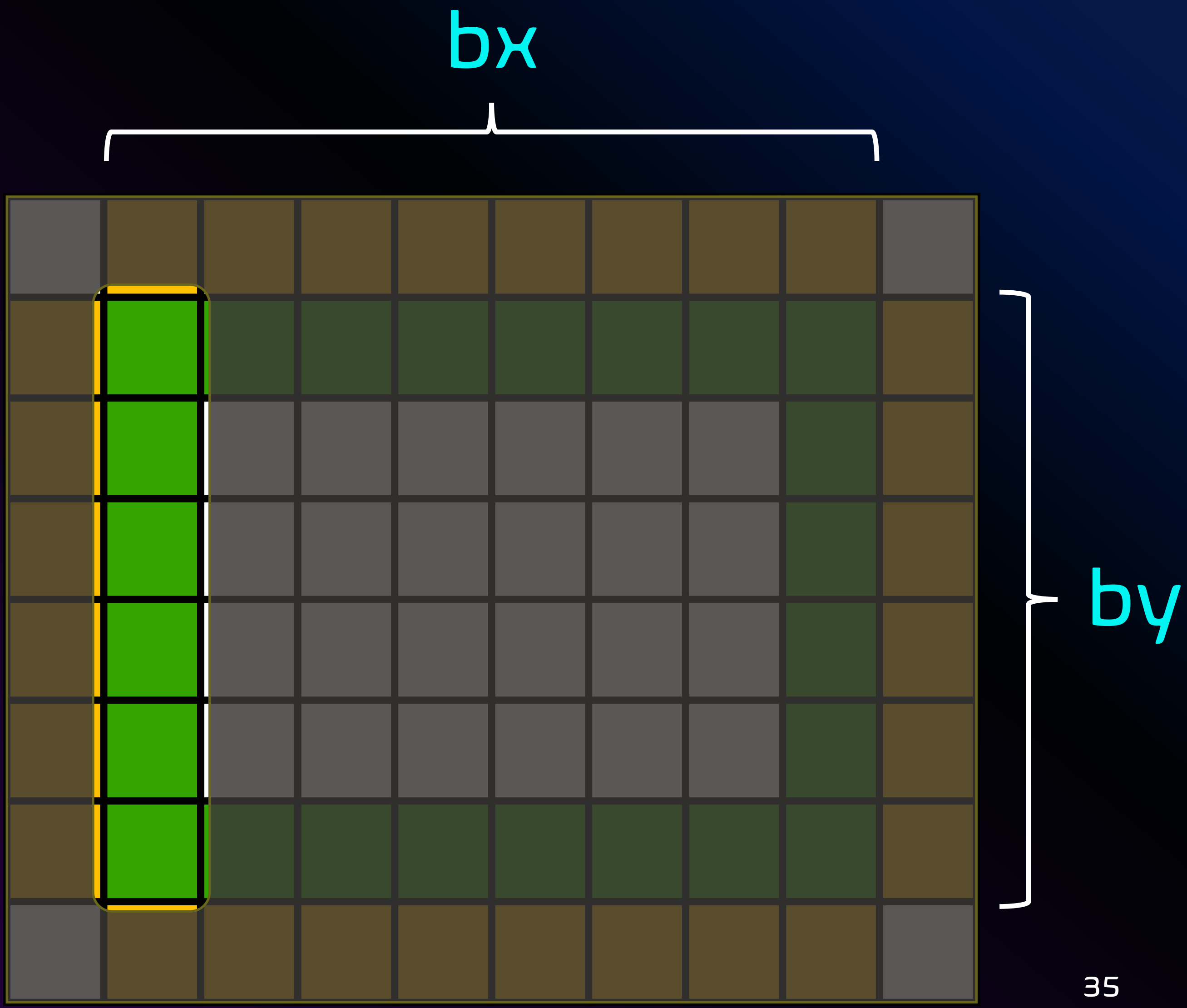


MPI_TYPE_VECTOR

EXAMPLE.3

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

```c
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
```

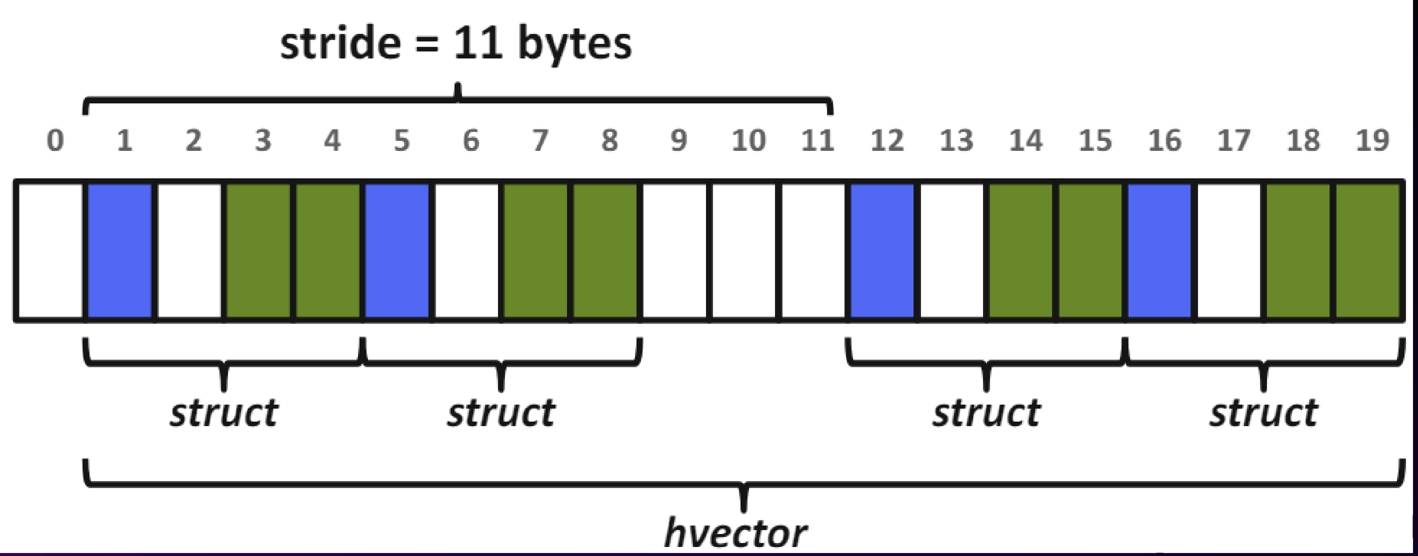


MPI_TYPE_CREATE_HVECTOR (MPI_TYPE_HVECTOR DEPRECATED)

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).

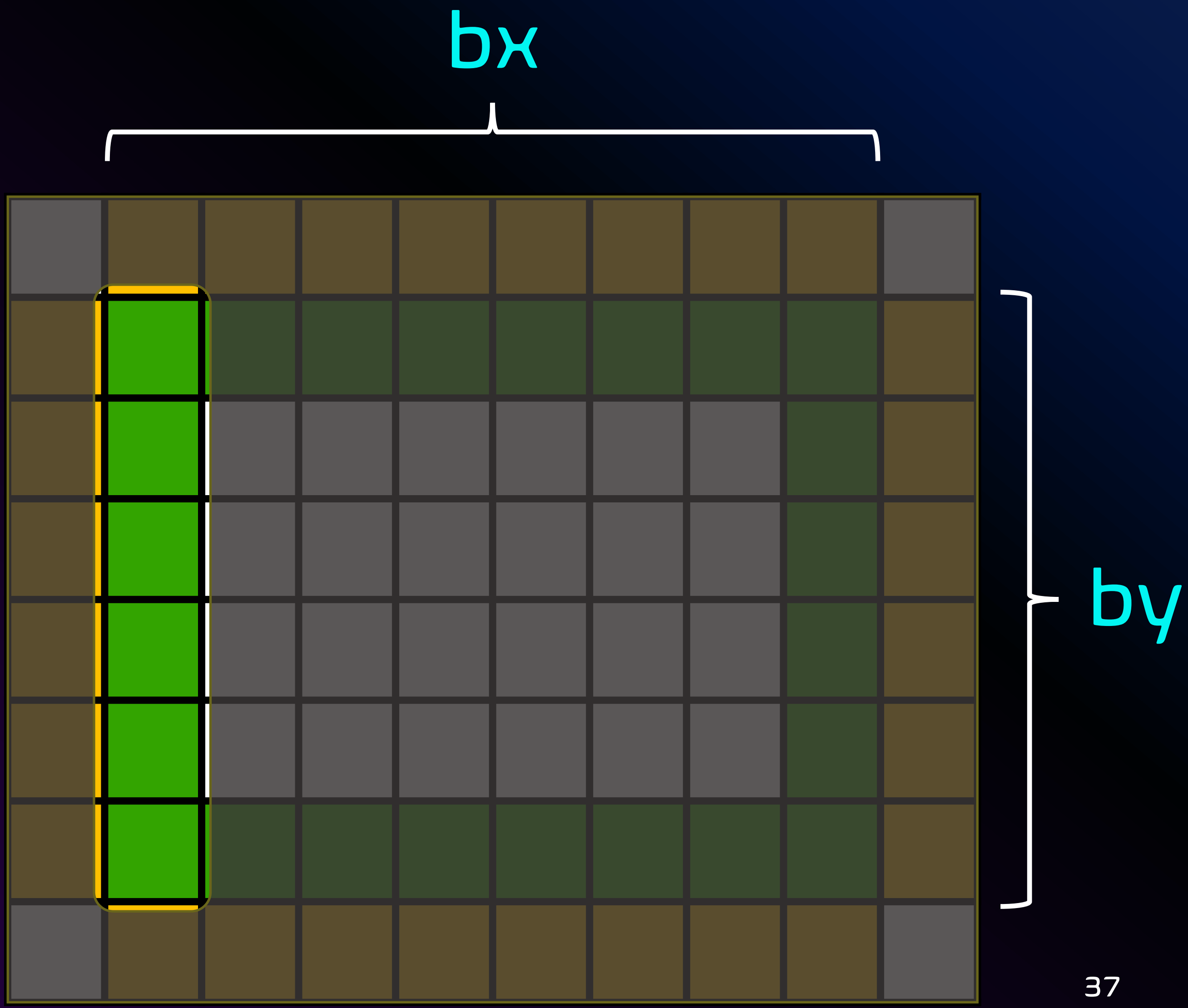


MPI_TYPE_CREATE_HVECTOR

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);
```

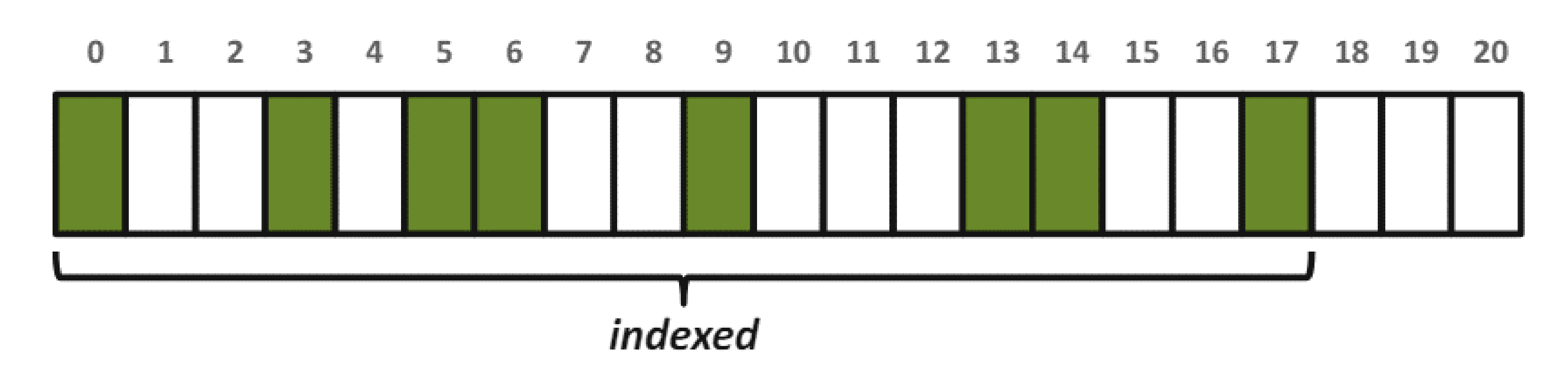
count : Length of array of displacements.

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);
```

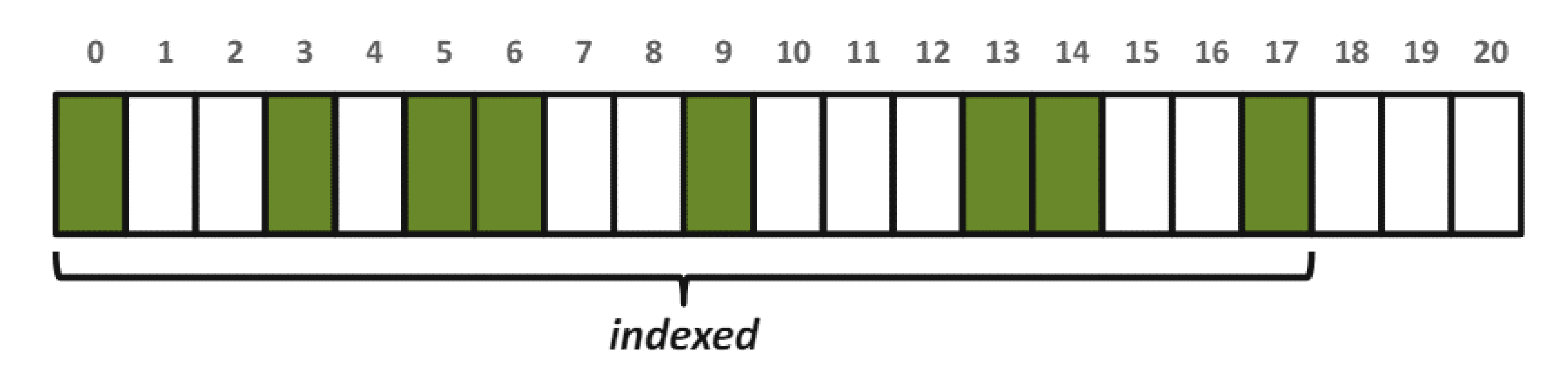
count : Number of data blocks.

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).

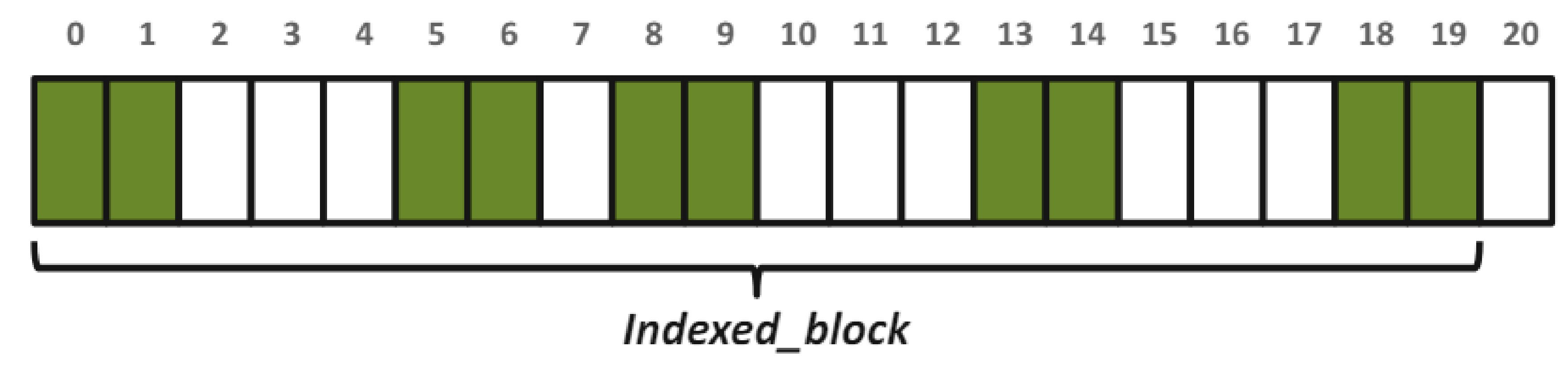


MPI_TYPE_CREATE_INDEXED_BLOCK

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);
```

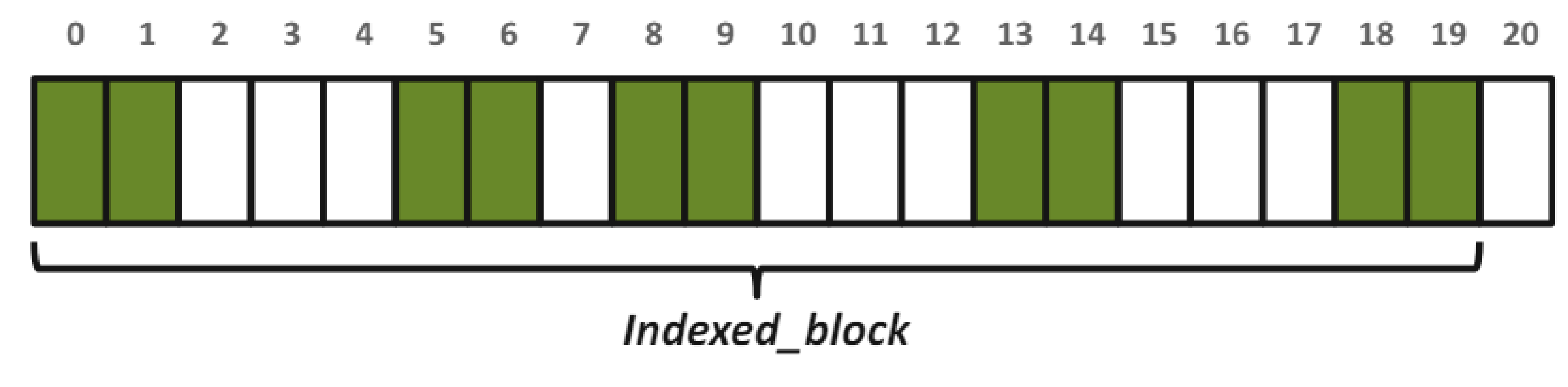
count : Length of array of displacements.
blocklen : Size of block (int).
array_of_displacements : Array of displacements (ints). In units of extent of oldtype.
oldtype : Old datatype (handle).
newtype : New datatype (handle).



MPI_TYPE_CREATE_INDEXED_BLOCK EXAMPLE

```c
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).

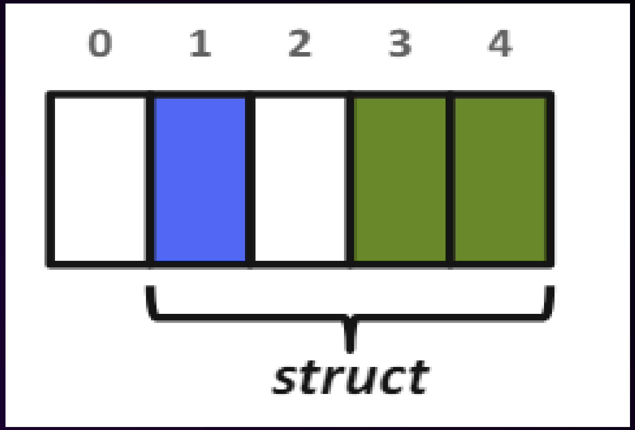
array_of_blocklens : Array of block lengths.

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

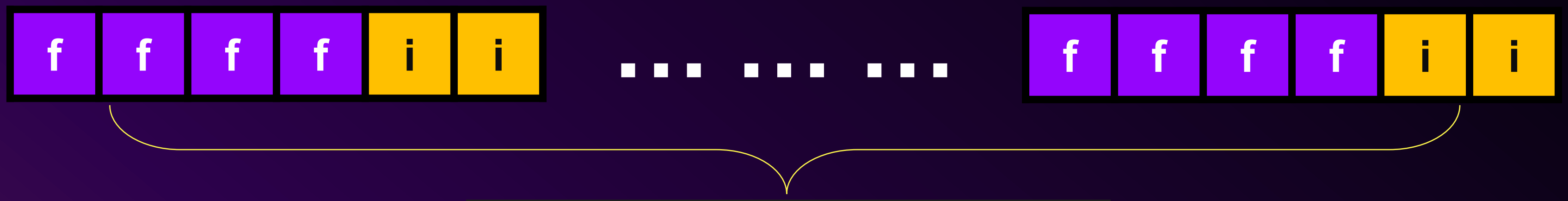
- need byte displacements/offsets

- 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);
```



particles[NELEM];

MIND THE ALIGNMENTS!

C struct may be automatically padded by the compiler!

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

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

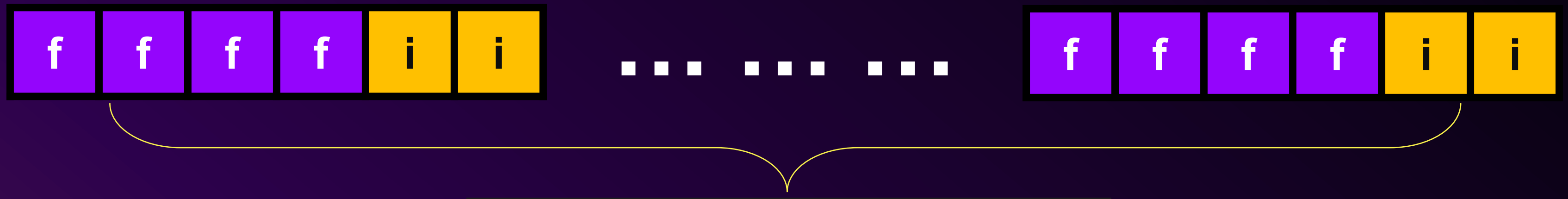
MPI_GET_ADDRESS

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

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

```c
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;
```

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



`particles[NELEM];`

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

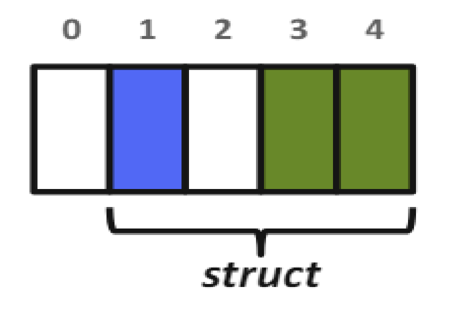
EXAMPLE

```c
/* 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

```c
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;
```

```c
// 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);
...
```




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

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);
```

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.

```c
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).

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

```c
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

20 Minutes

HANDS ON ACTIVITY #1

MPI_Type_contiguous

MPI_Type_vector

MPI_Type_indexed

MPI_Type_create_struct

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

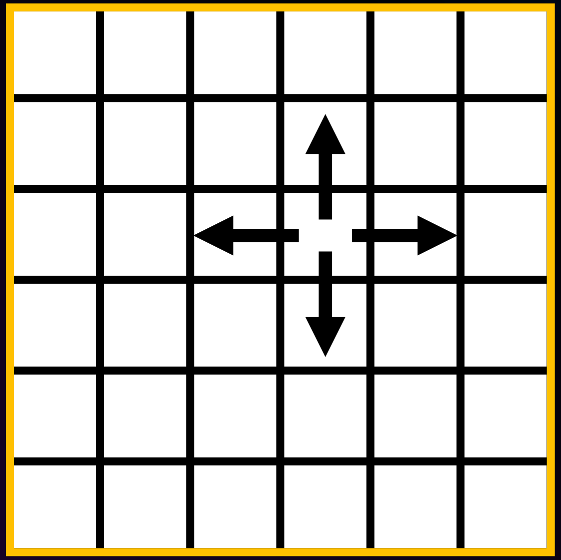
SECTION II

CARTESIAN 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

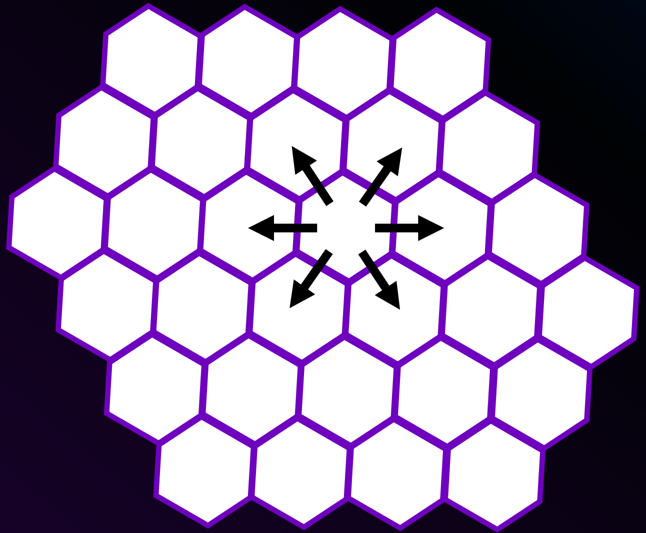


4 Neighbors

STRUCTURED MESHES

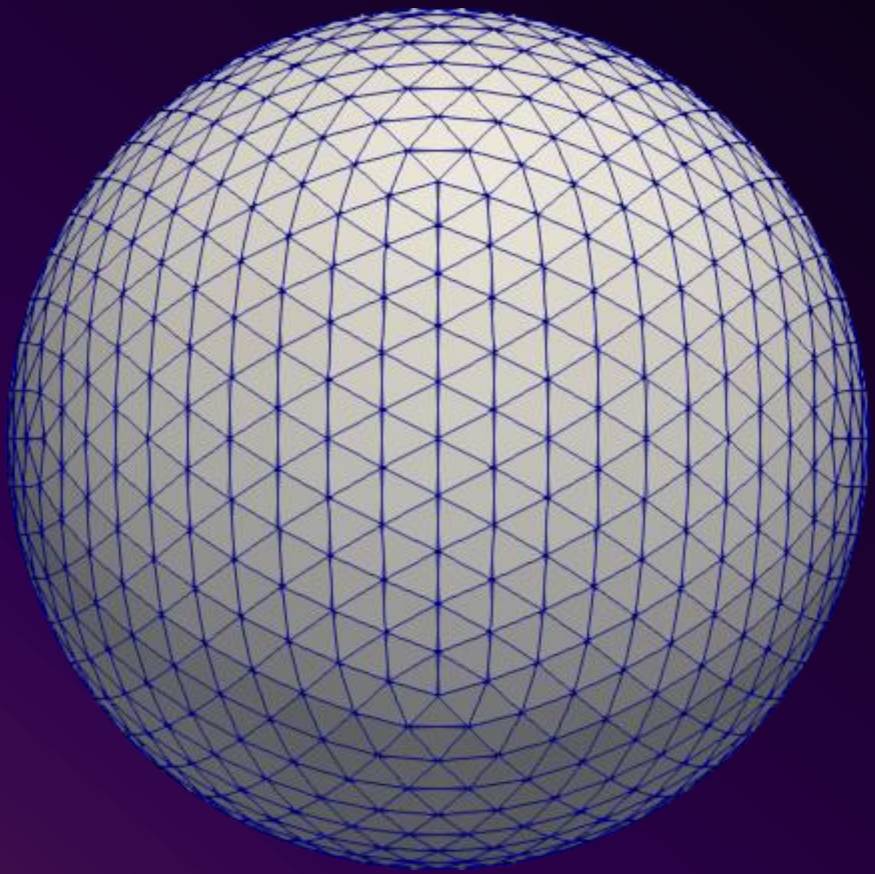
A *structured mesh* is
a mesh where one can find a connectivity pattern that applies to *every* element in the mesh.

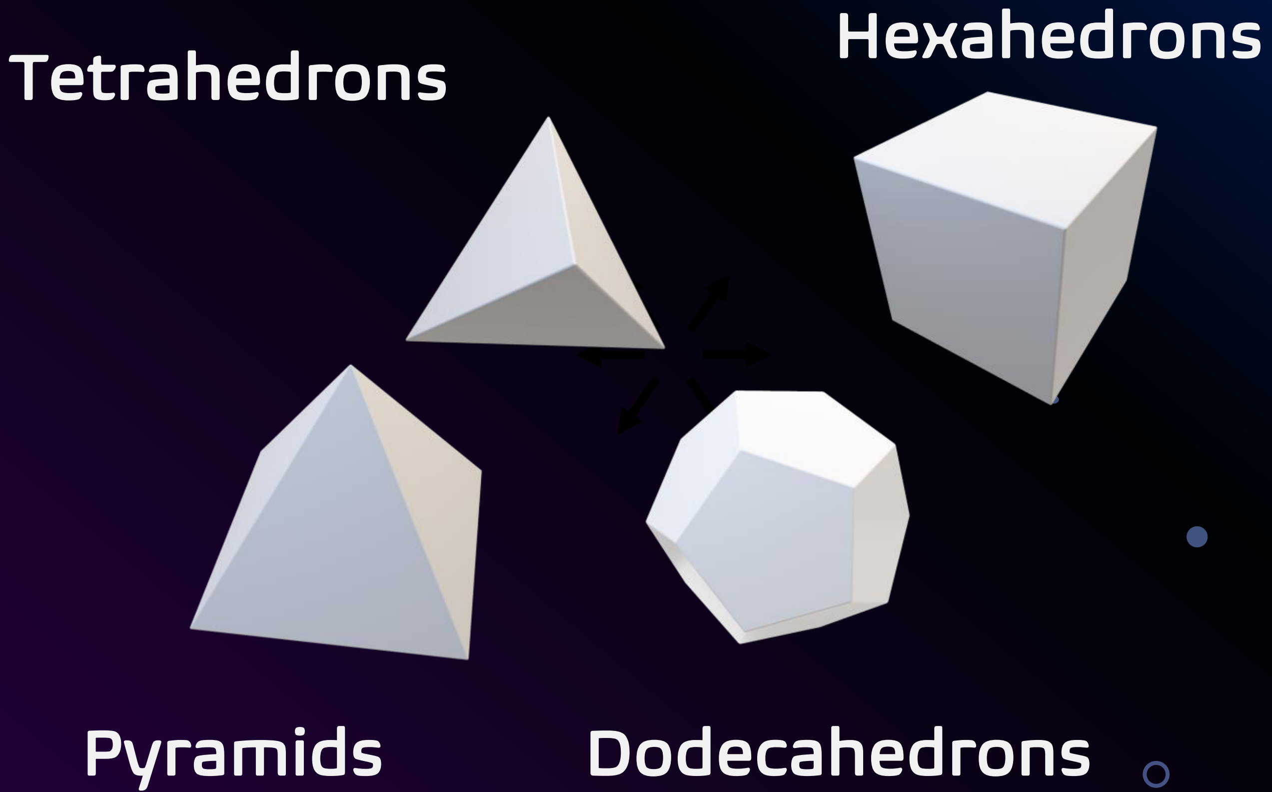
Typical Communication Pattern



6 Neighbors

STRUCTURED MESHES

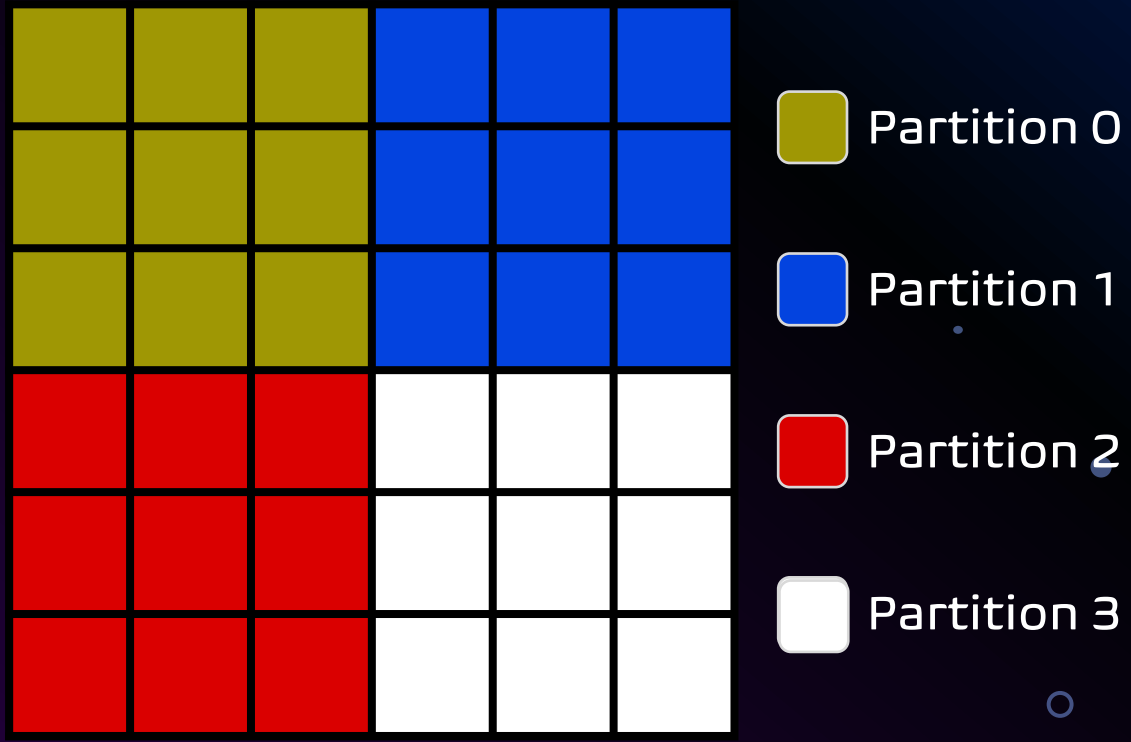




LOAD BALANCING

Basic Partitioning

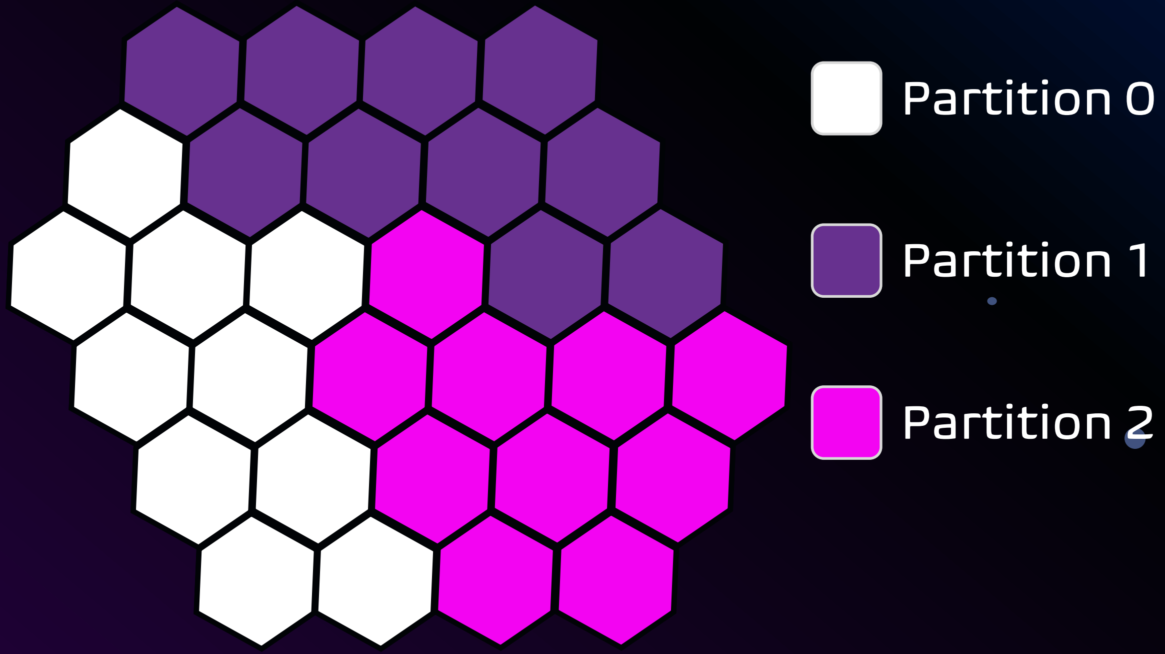
Load balancing a structured mesh typically comes down to partitioning the mesh to have an equal number of elements per Processing Element (PE).



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

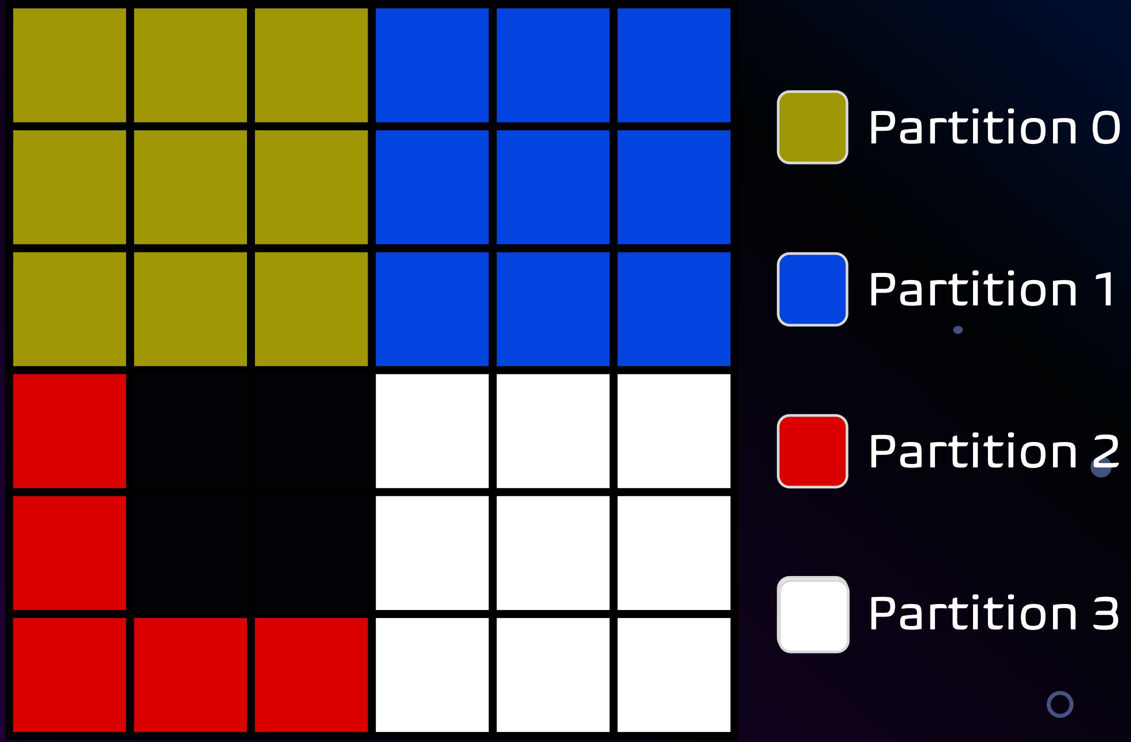
Basic Partitioning

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

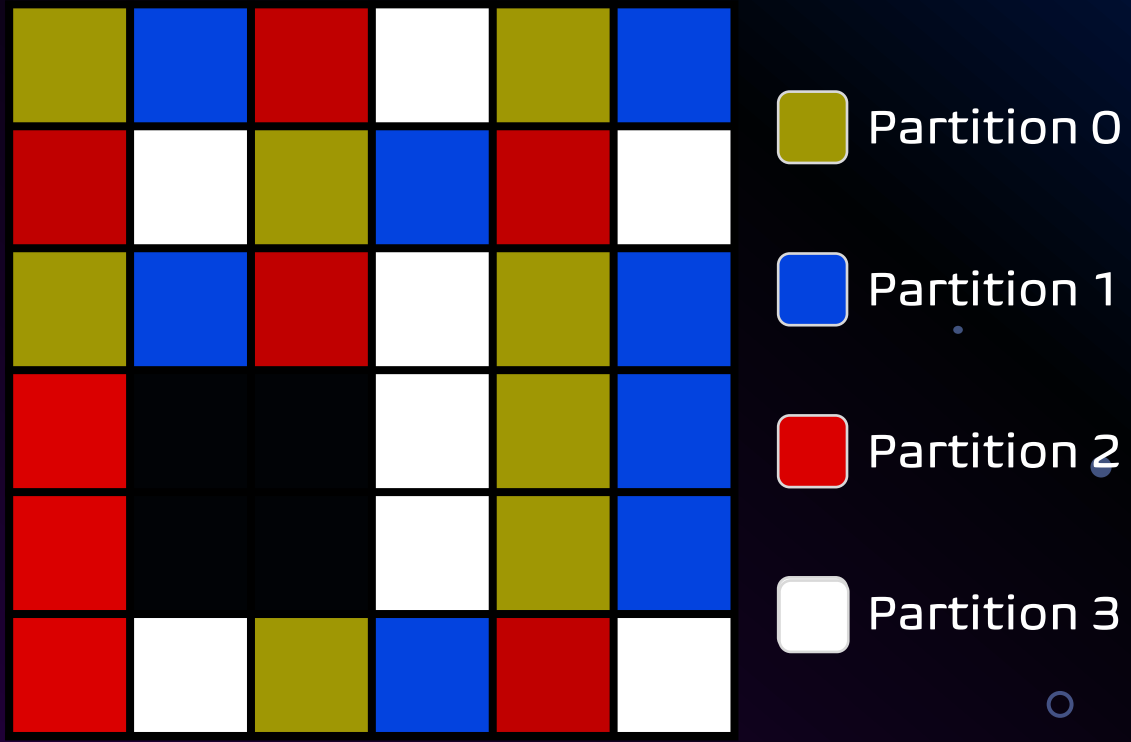
Cyclic Partitioning

Pros

✓ Every MPI process is assigned 8 elements.

Cons

➢ Hindered data locality as elements are rarely consecutive.



LOAD BALANCING: A LITTLE LESS TRIVIAL

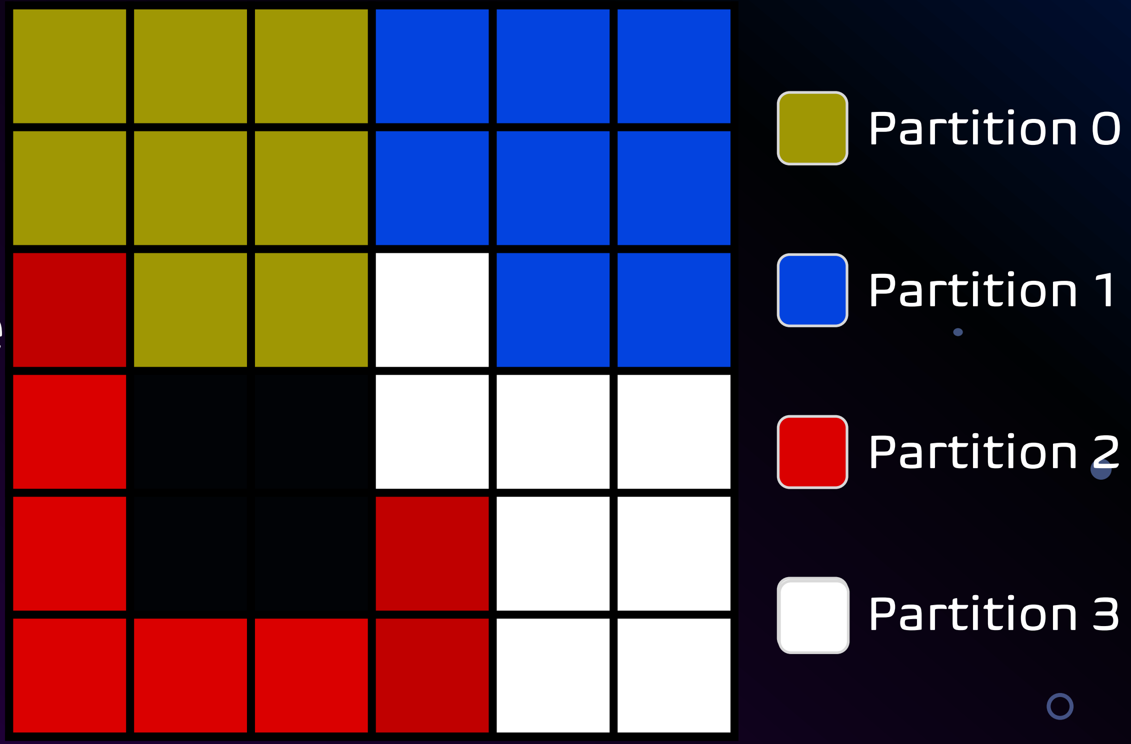
Advanced Partitioning

Pros

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

Cons

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



STRUCTURED MESHES: LIMITATIONS

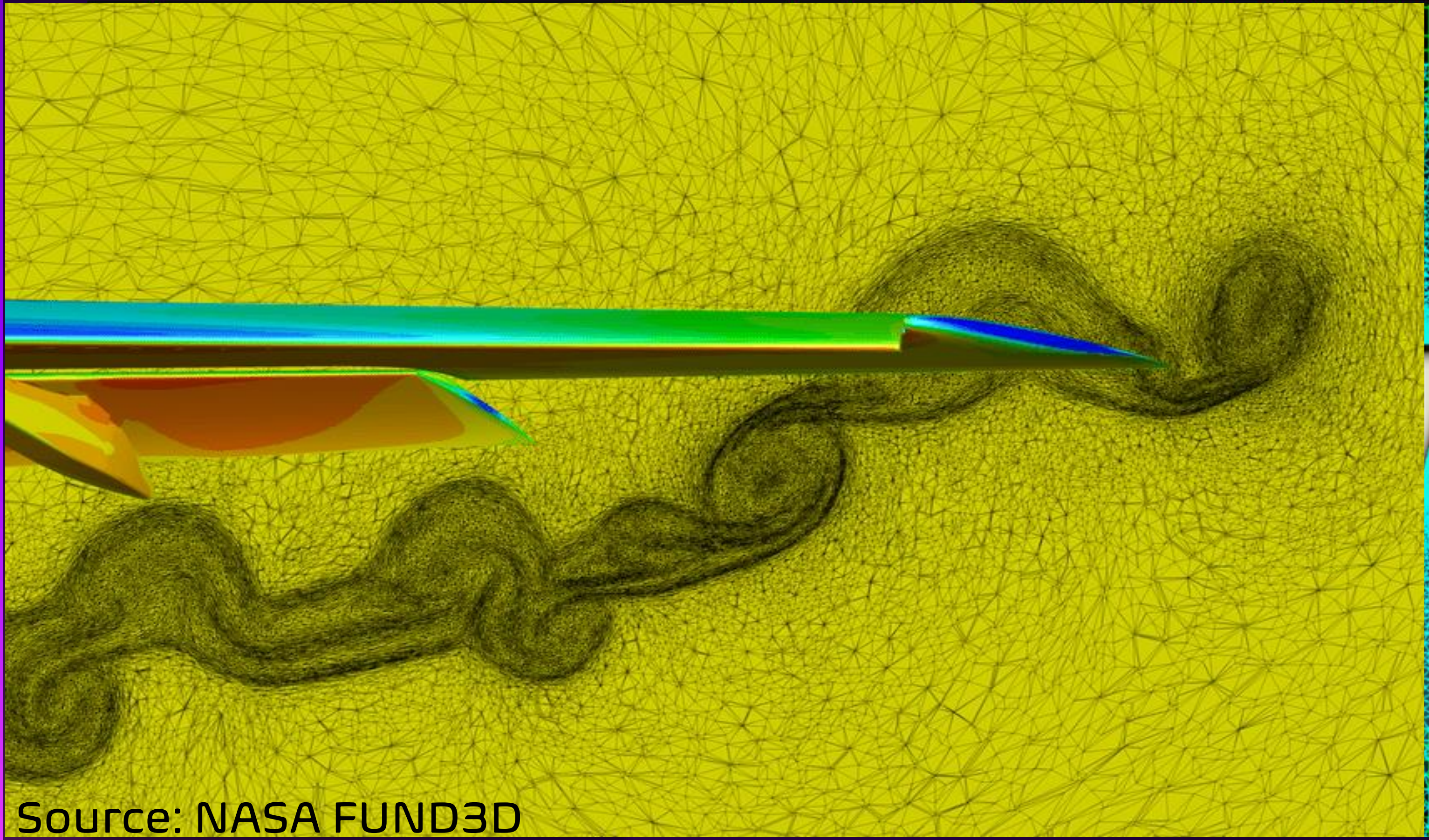
Pros

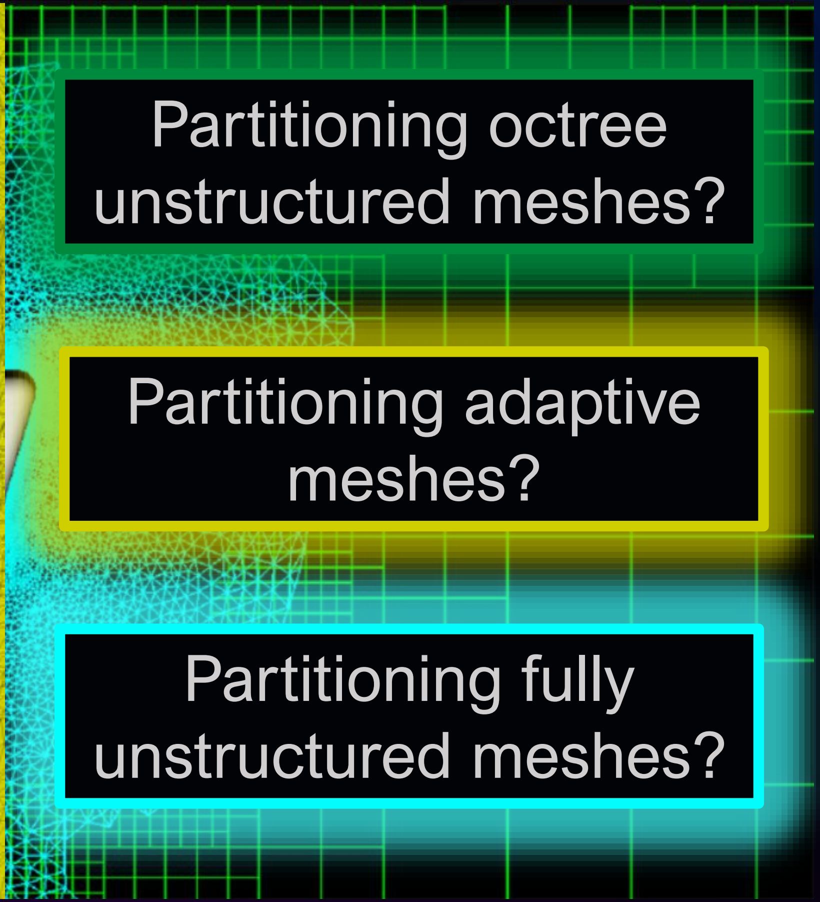
- ✓ Structured meshes have interesting properties that make their storage in memory and indexing, among other aspects, convenient.

Cons

- ➢ However, many real world meshes are more complex and do not decompose into a structured mesh.

UNSTRUCTURED MESHES





Partitioning octree unstructured meshes?

Partitioning adaptive meshes?

Partitioning fully unstructured meshes?

PARTITIONING: HARD.

In general,
user is responsible
for partitioning!

PARTITIONING: POTENTIAL OPTIONS

- ✓ MPI_Dims_create
- ✓ Morton-code orderings.
- ✓ Advanced partitioning strategies exist. ParMETIS is a common solution (Sec. V).

Partitioning Cartesian meshes?

Partitioning octree unstructured meshes?

Partitioning fully unstructured meshes?

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, which can be leveraged by the underlying runtime to improve the placement of MPI processes to hardware.

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

- ✓ Simplifies writing of code.

VIRTUAL TOPOLOGY: FACTORS TO CONSIDER

- MPI process contains one thread, which is eventually run by a hardware thread.
- Certain hardware threads may be on the same core.
- Certain caches may be shared by multiple cores (typically, L3).
- Certain cores may belong to the same processor and thus access the same memory module(s) (c.f.: NUMA region).
- Certain processors may be on different nodes.
- Certain nodes may be on different racks.

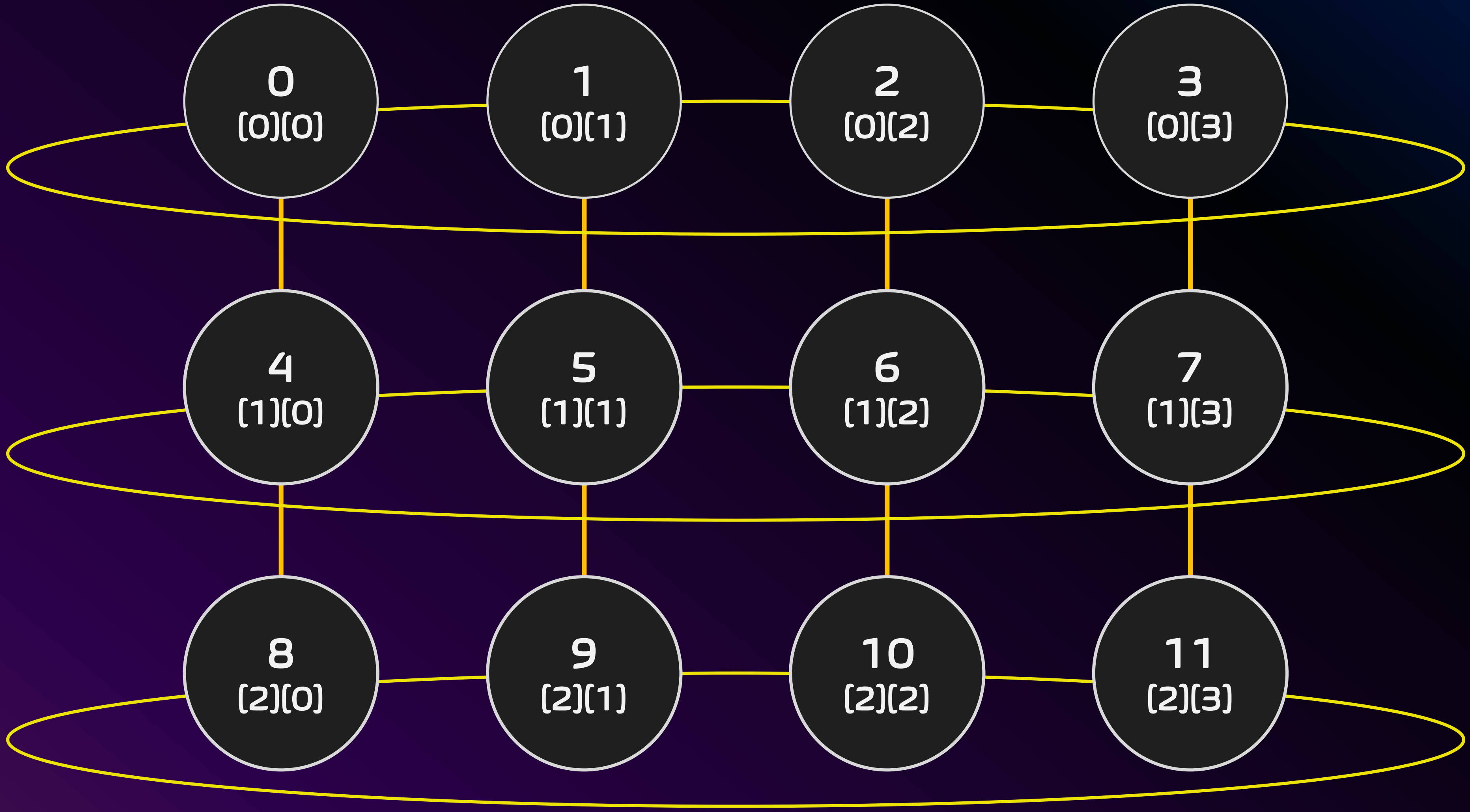
Back to "Why is it useful?"

✓ By declaring a virtual topology, the MPI runtime environment has extra information to leverage for a more efficient placement.

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

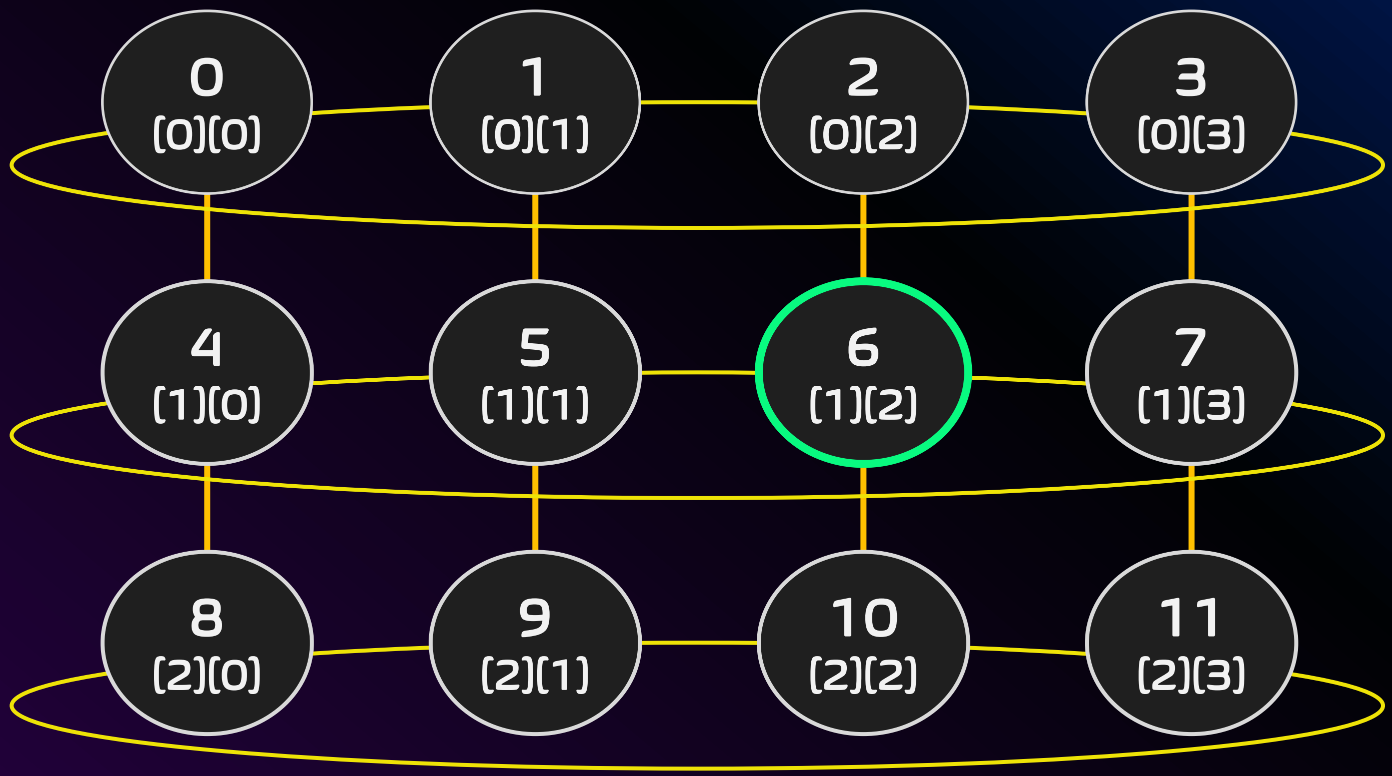
dims before call	Function call	dims on return
(0, 0)	MPI_Dims_create (6, 2, dims);	(3, 2)
(0, 0)	MPI_Dims_create (7, 2, dims);	(7, 1)
(0, 3, 0)	MPI_Dims_create (6, 3, dims);	(2, 3, 1)
(0, 3, 0)	MPI_Dims_create (7, 3, dims);	Erroneous call

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
```

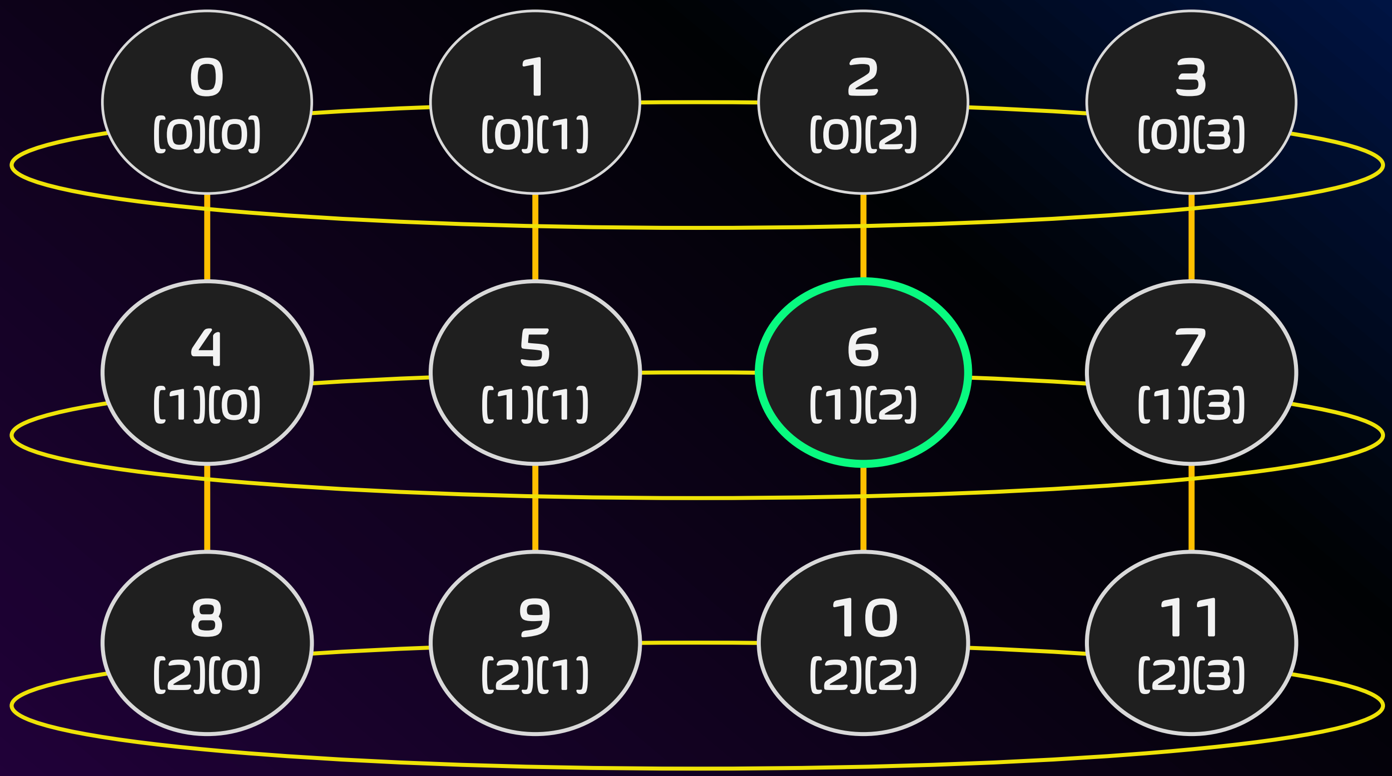


VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

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}
```

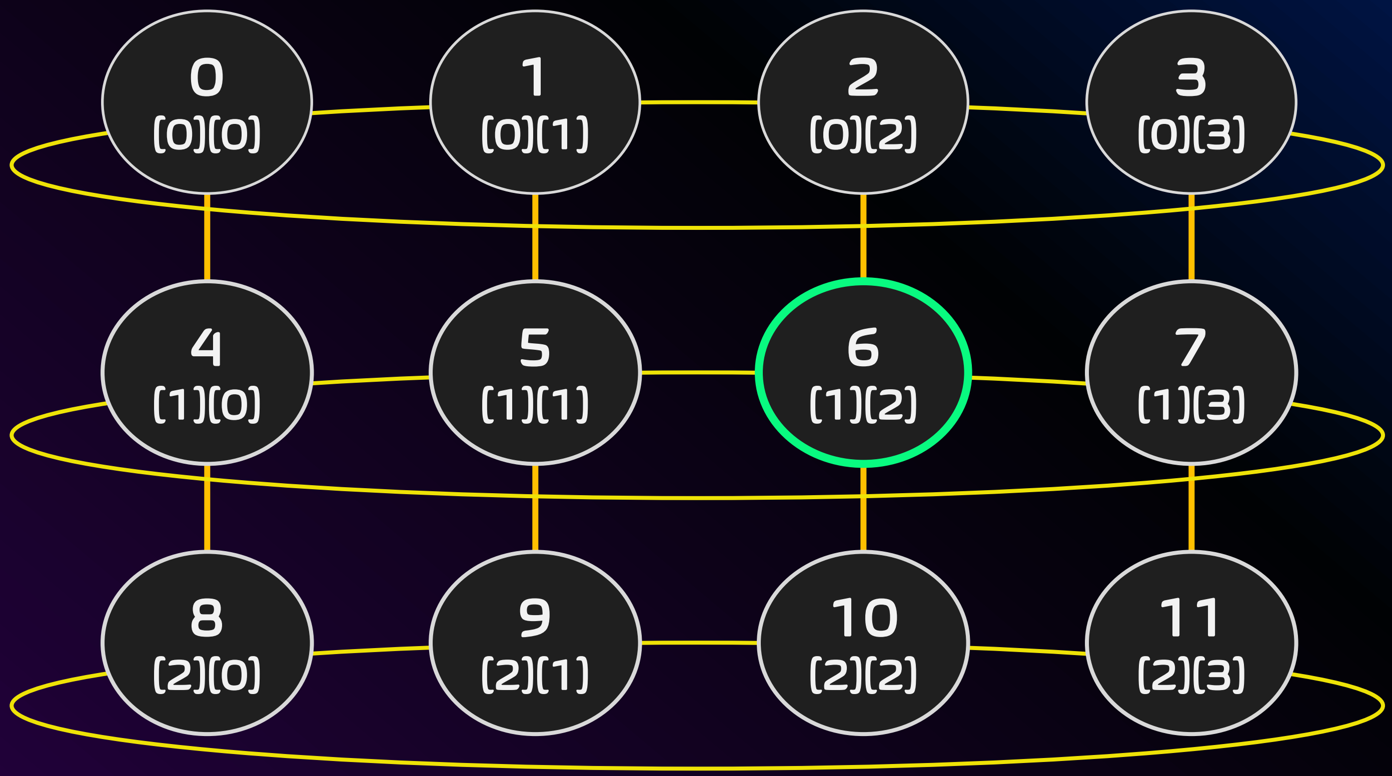


VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

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);
```

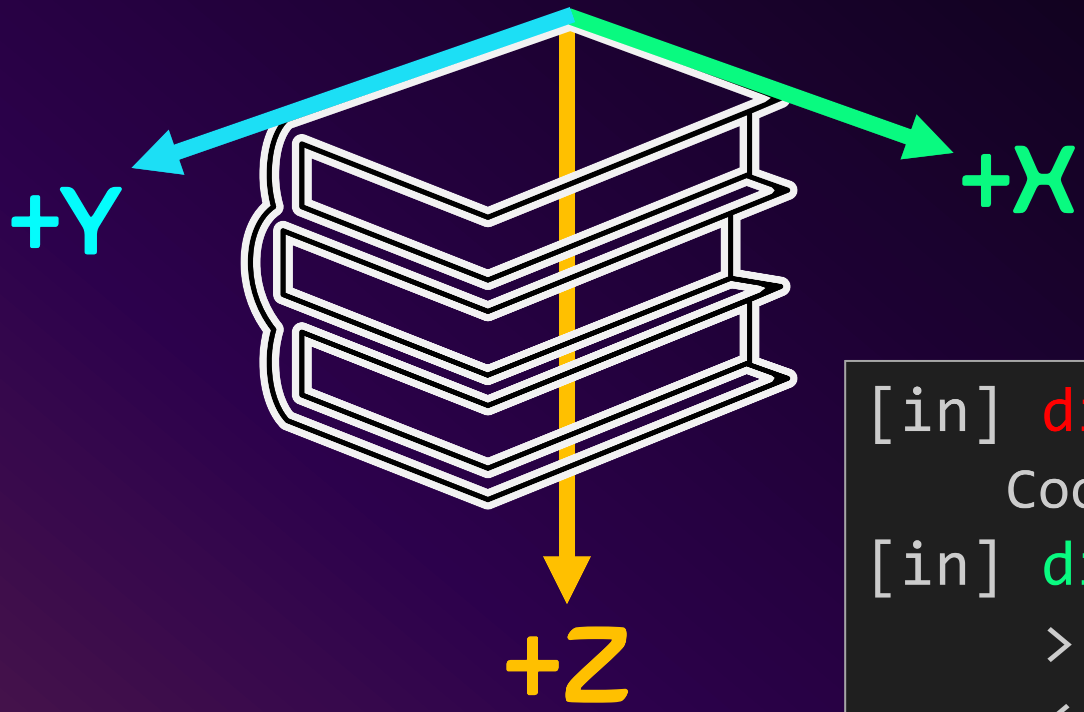
```
[in] direction :
    Coordinate index of shift
[in] displacement :
    > 0: positive shift,
    < 0: negative shift
```

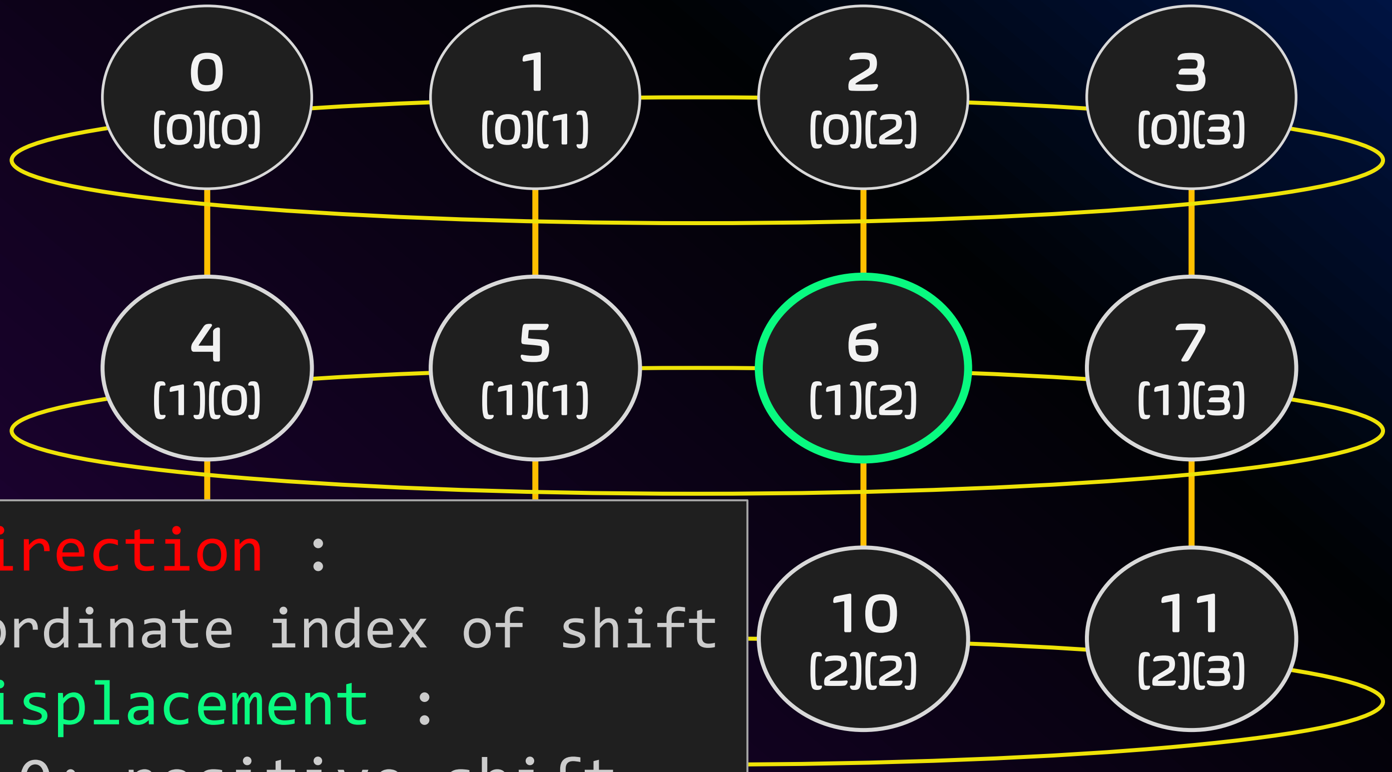


VIRTUAL TOPOLOGY: CARTESIAN MAPPING FUNCTIONS

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

LEXICOGRAPHIC ORDERING





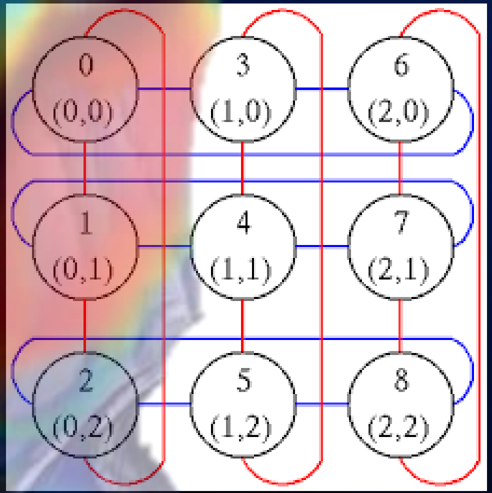

```
[in] direction :
     Coordinate index of shift
[in] displacement :
     > 0: positive shift,
     < 0: negative shift
```

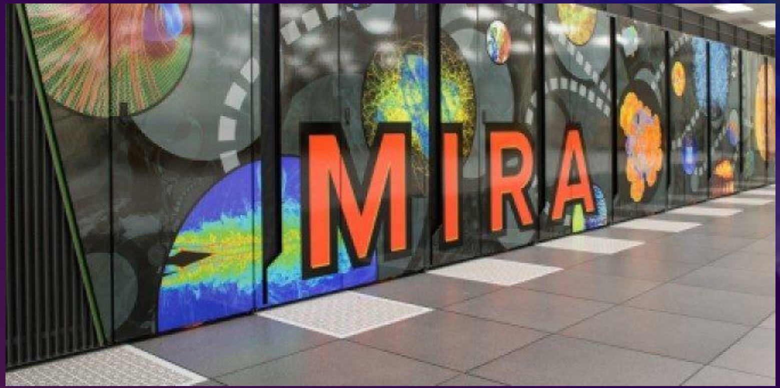
MPI_CART_SHIFT: PARALLEL SCALING

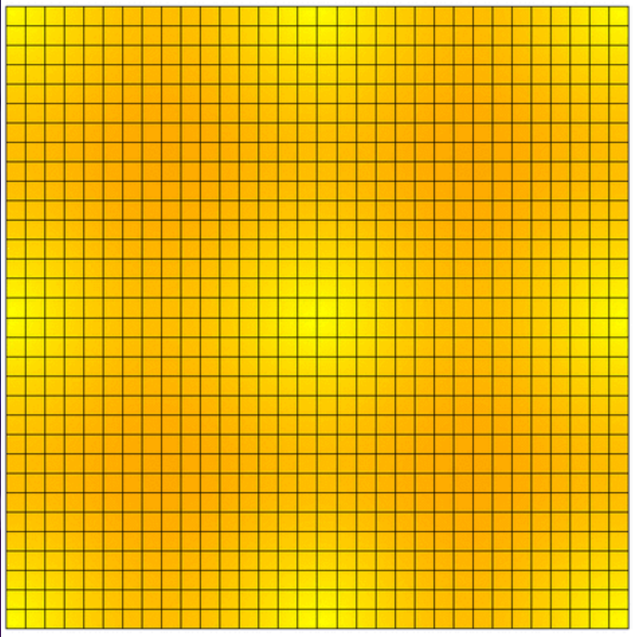
Strong Scaling on ALCF Mira

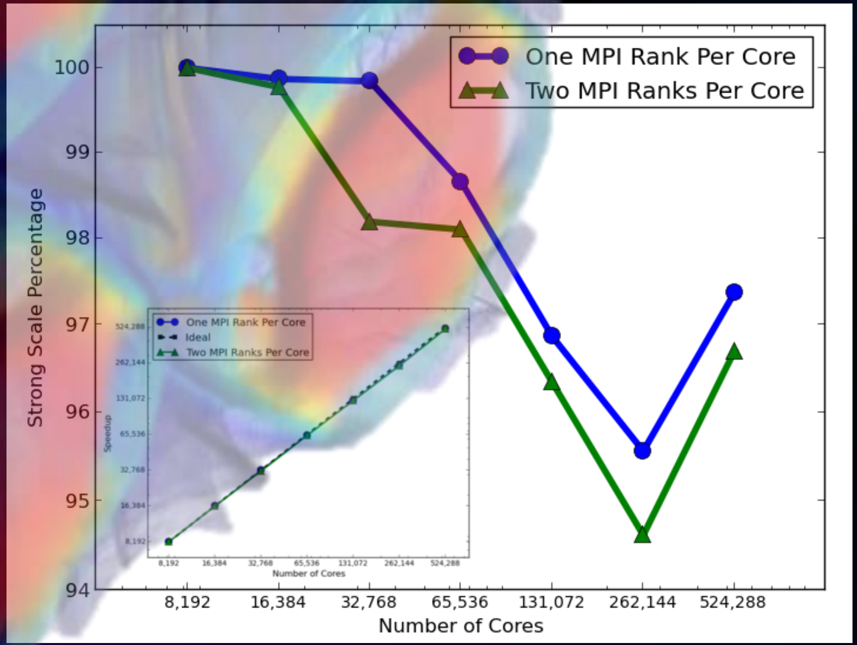
- Taylor-Green Vortex
- Fully periodic
- Periodic Mesh: (512, 512, 512)
- Discontinuous Galerkin: 5^{th}-order
- 16.8 Billion DOFs
  - ➢ 83.9 Billion unknowns

1 Million MPI Ranks





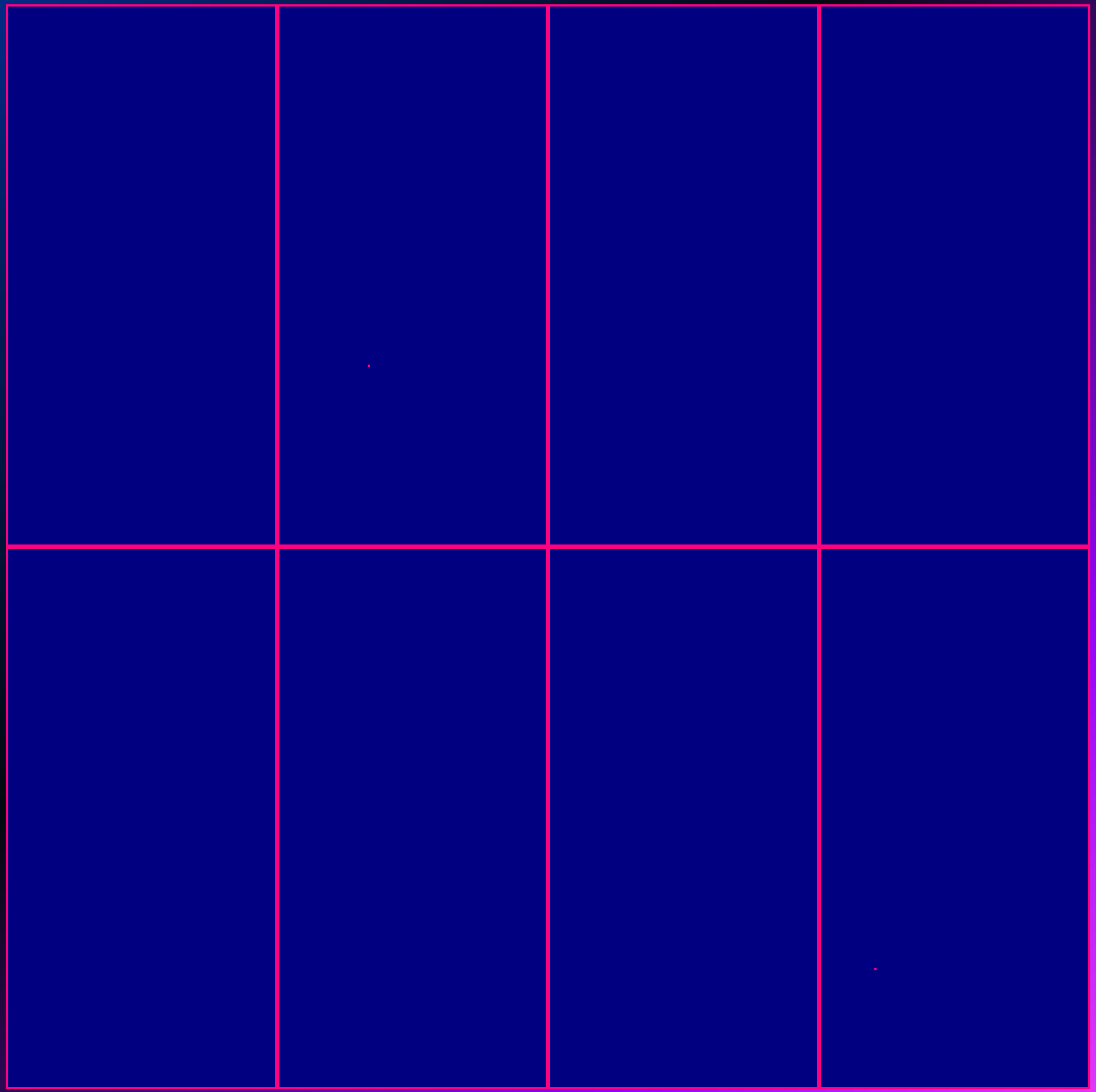




20 Minutes

HANDS ON ACTIVITY #2

MPI_Cart_shift



TRACK II

ADVANCED DISTRIBUTED-MEMORY PROGRAMMING

DAY 2

ANDREW KIRBY

DAY 2 AGENDA

Section III – Graph Virtual Topologies

- Hands On Activity # 3

Coffee Break (10:30-11:00)

Section IV – Neighborhood Collectives

- Hands On Activity # 4

Section V – Graph Partitioning

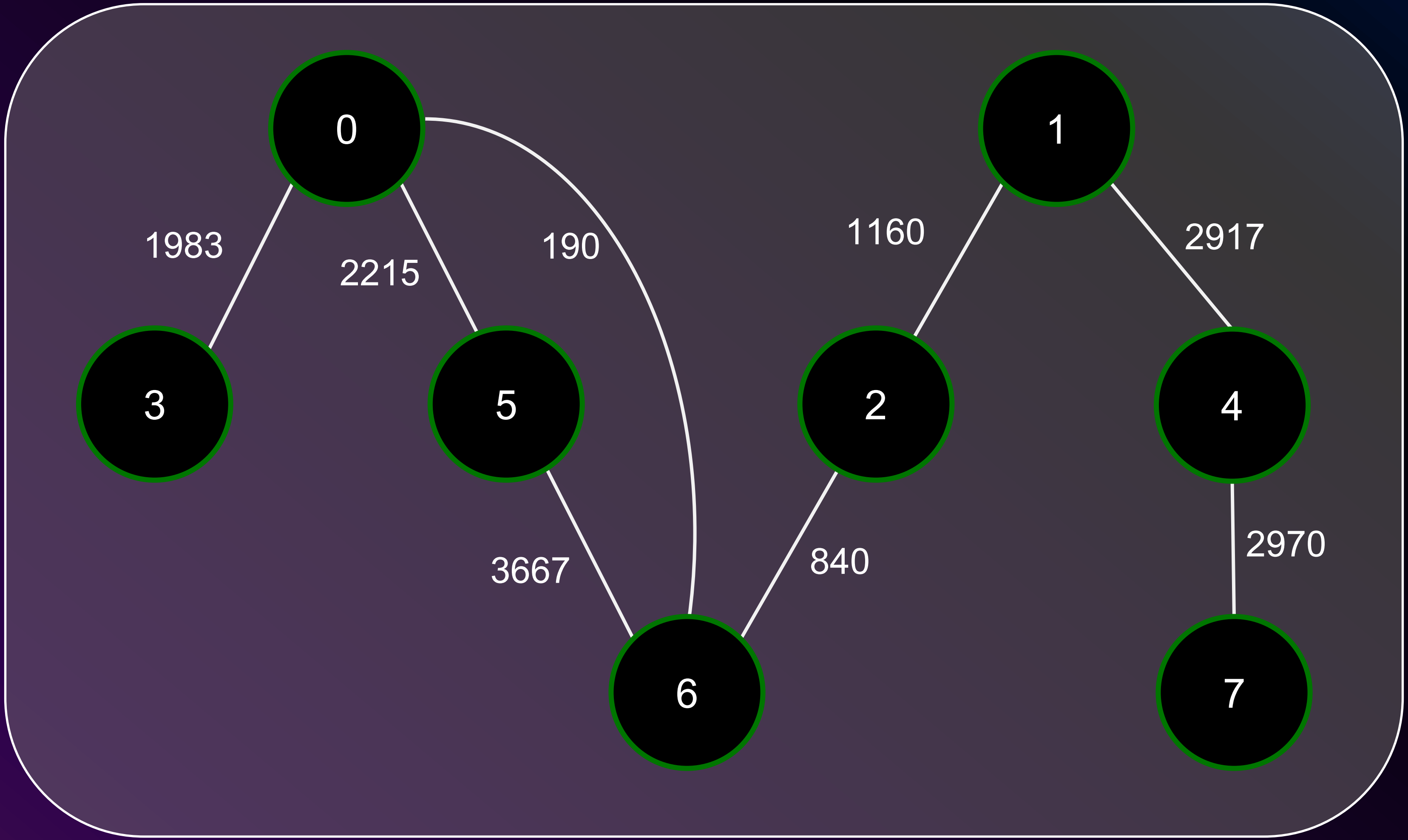
- Hands On Activity # 5

SECTION III

GRAPH TOPOLOGIES

VIRTUAL TOPOLOGY: GRAPHS

Goal: Make a new communicator with Graph Topology information attached.



VIRTUAL TOPOLOGY: GRAPHS

Three ways to create a graph topology:

1. MPI_Graph_create
2. MPI_Dist_graph_create
3. MPI_Dist_graph_create_adjacent

1. MPI_GRAPH_CREATE

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

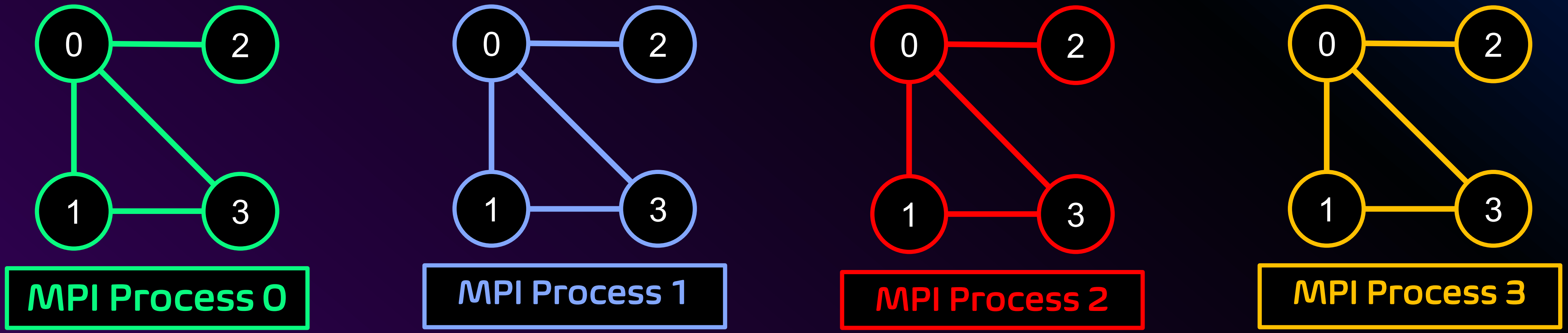
```
int MPI_Graph_create(
    MPI_Comm comm_old,
    int nnodes,
    const int index[],
    const int edges[],
    int reorder,
    MPI_Comm *comm_graph);
```

- comm_old : Input communicator.
- nnodes : Number of nodes in graph.
- index : Array describing node degrees.
- edges : Array describing graph edges.
- reorder : Rank reordering flag.
- comm_graph : Output communicator with graph topology.

When creating a graph topology using *MPI_Graph_create*, every MPI process pass the entirety of the graph topology.

1. MPI_GRAPH_CREATE

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



When creating a graph topology using MPI_Graph_create, every MPI process pass the entirety of the graph topology.

1. MPI_GRAPH_CREATE

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

```
int MPI_Graph_create(
    MPI_Comm comm_old,
    int nnodes,
    const int index[],
    const int edges[],
    int reorder,
    MPI_Comm *comm_graph);
```

index : This parameter contains the total number of neighbors[1] of all MPI processes with a rank lesser than, or equal to, the position considered in indexes.

This design allows MPI to know the total number of entries in the edges parameter.

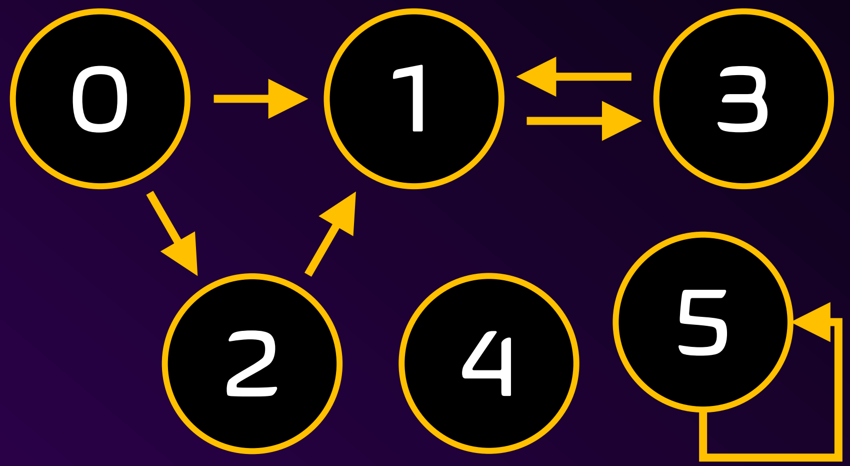
1. MPI_GRAPH_CREATE

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

```
int MPI_Graph_create(
    MPI_Comm comm_old,
    int nnodes,
    const int index[],
    const int edges[],
    int reorder,
    MPI_Comm *comm_graph);
```

edges : This parameter contains **all** edges of the graph, sorted by the source identifier. Edges are **undirected** so you do not have to specify both edges for each connection. Redundant and self edges are also allowed.

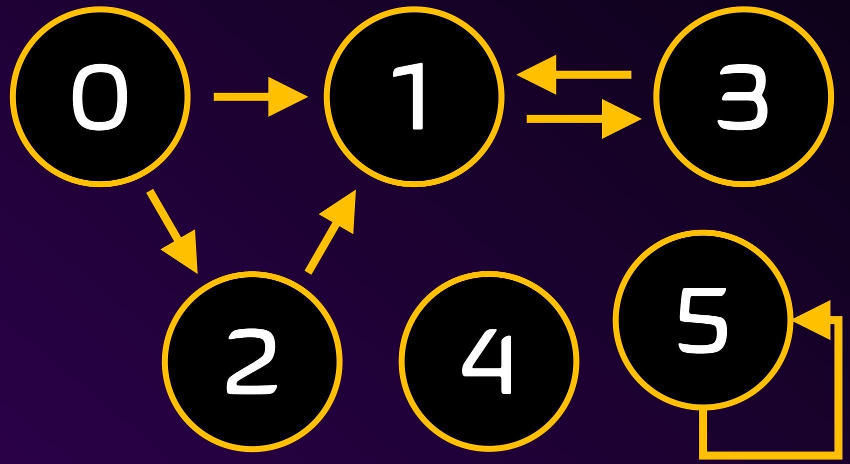
1. MPI_GRAPH_CREATE EXAMPLE



MPI Process	Connected To	Degree	Cumulated Degree
0	1, 2	2	2
1	3	1	3
2	1	1	4
3	1	1	5
4	-	0	5
5	5	1	6

```
This graph topology [using 7 MPI processes]:
 * - Node connecting to a single neighbor (MPI process 1, 2, 3)
 * - Node connecting to multiple neighbors (MPI process 0)
 * - Node connected by a single neighbor (MPI processes 2, 3)
 * - Node connected by multiple neighbors (MPI process 1)
 * - Nodes forming a cycle (MPI processes 1 and 3, MPI process 5)
 * - Node without neighbors (MPI process 4)
 * - Node connected to itself (MPI Process 5)
 * - MPI process not included at all in the graph (MPI process 6)
```

1. MPI_GRAPH_CREATE EXAMPLE

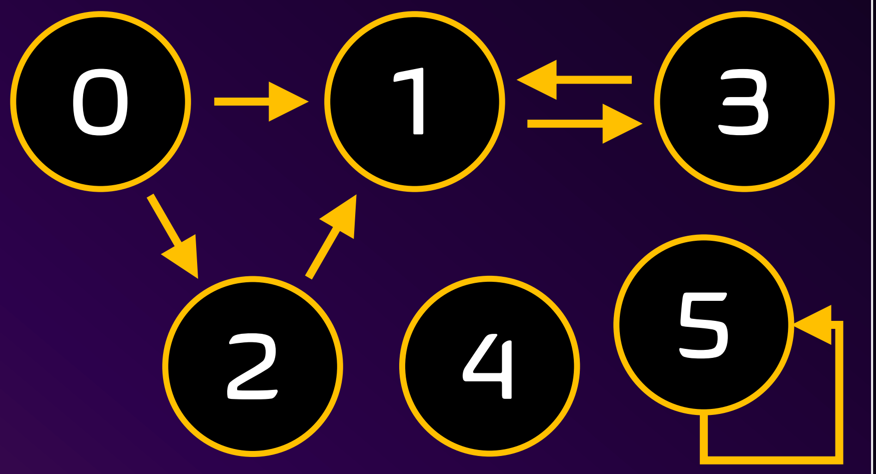


MPI Process	Connected To	Degree	Cumulated Degree
0	1, 2	2	2
1	3	1	3
2	1	1	4
3	1	1	5
4	-	0	5
5	5	1	6

```
// Number of nodes in the graph (MPI process 4 not connected -> excluded)
const int number_of_nodes = 6;

// Declare the total number of neighbors until each MPI process (= the ones before + its own)
int indexes[6] = {2, 3, 4, 5, 5, 6}; // “inclusive scan”

// Declare the endpoint of each edge
int edges[6] = {1, 2, 3, 1, 1, 5};
```

1. MPI_GRAPH_CREATE

EXAMPLE




```c
const int number_of_nodes = 6;
int indexes[6] = {2, 3, 4, 5, 5, 6};
int edges[6] = {1, 2, 3, 1, 1, 5};
int reorder = false;

// Create NEW communicator with graph topology.
MPI_Comm new_communicator;
MPI_Graph_create(MPI_COMM_WORLD,
                 number_of_nodes,
                 indexes,
                 edges,
                 reorder,
                &new_communicator);
```

1. MPI_GRAPH_CREATE WARNING

Image a graph of with **100 billion edges**

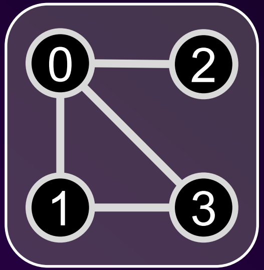
>> requires ~1 TB of RAM *per* MPI process <<

- Every MPI process must store the entire graph.
- Every MPI process passes the entire graph over the network

MPI_Graph_create is **not scalable**.

There is a better way!

2. MPI_DIST_GRAPH_CREATE



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

Every MPI process may specify 0, 1 or more edges.

The edges specified do not have to contain the MPI process that passes them.

2. MPI_DIST_GRAPH_CREATE

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

```
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

num_src_nodes : Number of source nodes for which this process specifies edges.

sources : Array containing the num_src_nodes source nodes for which this process specifies edges (array of non-negative integers).

degrees : Array specifying the number of destinations for each source node in the source node array (array of non-negative integers).

* Note that the Process ID has nothing to do with the Node IDs.

2. MPI_DIST_GRAPH_CREATE

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

```
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

destinations : Destination nodes for the source nodes in the source node array (array of non-negative integers).

weights : Weights for source to destination edges (array of non-negative integers).

info : Hints on optimization and interpretation of weights (handle).

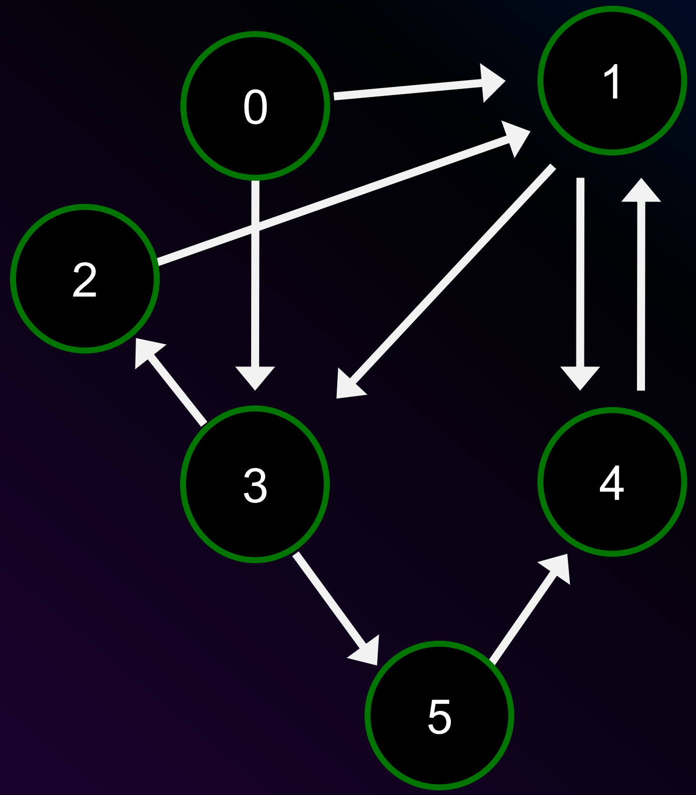
reorder : Ranking may be reordered (true) or not (false) (logical).

* Note that the Process ID has nothing to do with the Node IDs.

2. MPI_DIST_GRAPH_CREATE EXAMPLE

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

```
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```



* Note that the Process ID has nothing to do with the Node IDs.

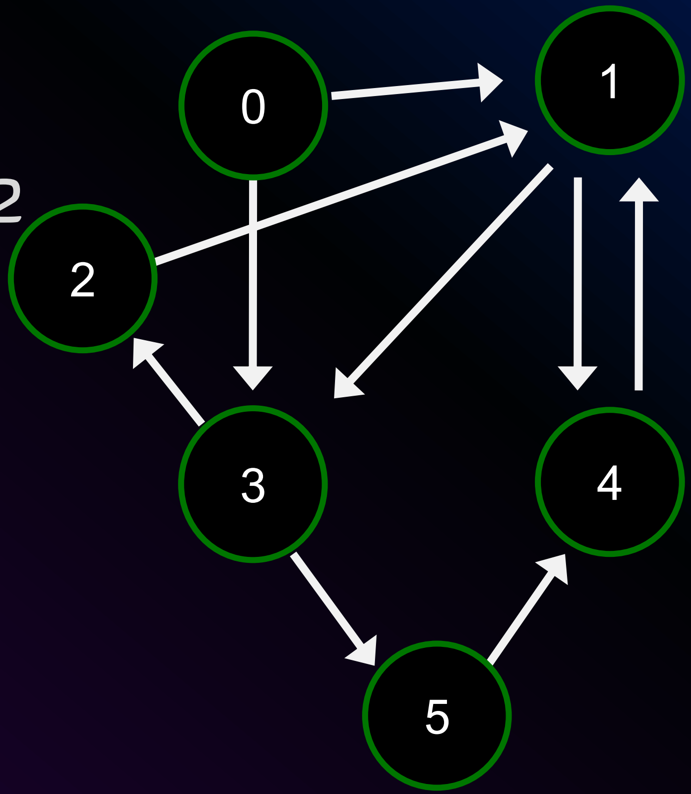
2. MPI_DIST_GRAPH_CREATE EXAMPLE

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

```c
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0*

- num_src_nodes: 2
- Sources(2): {0, 1}



* Note that the Process ID has nothing to do with the Node IDs.

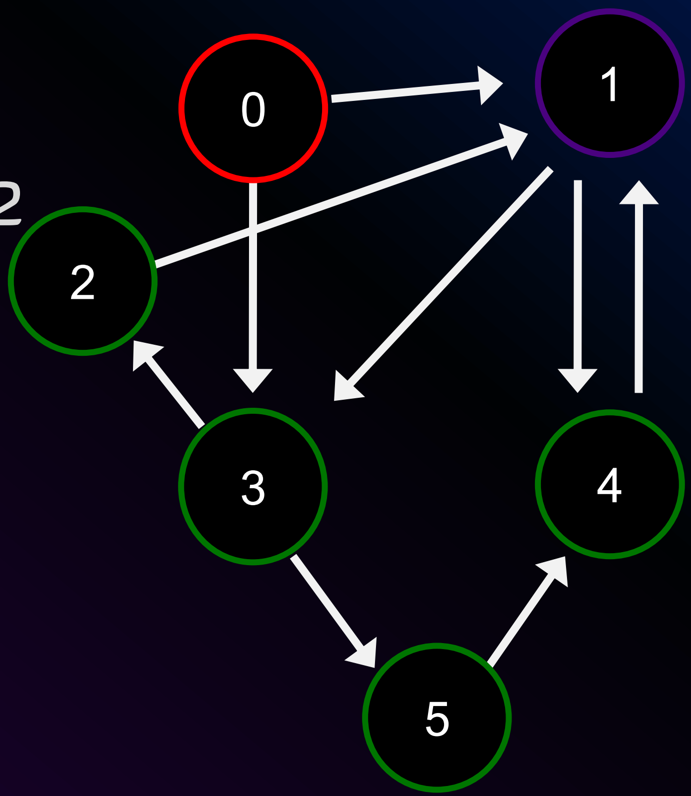
2. MPI_DIST_GRAPH_CREATE EXAMPLE

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

```
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0*

- num_src_nodes: 2
- Sources(2): {0, 1}
- Degrees(2): {2,1}



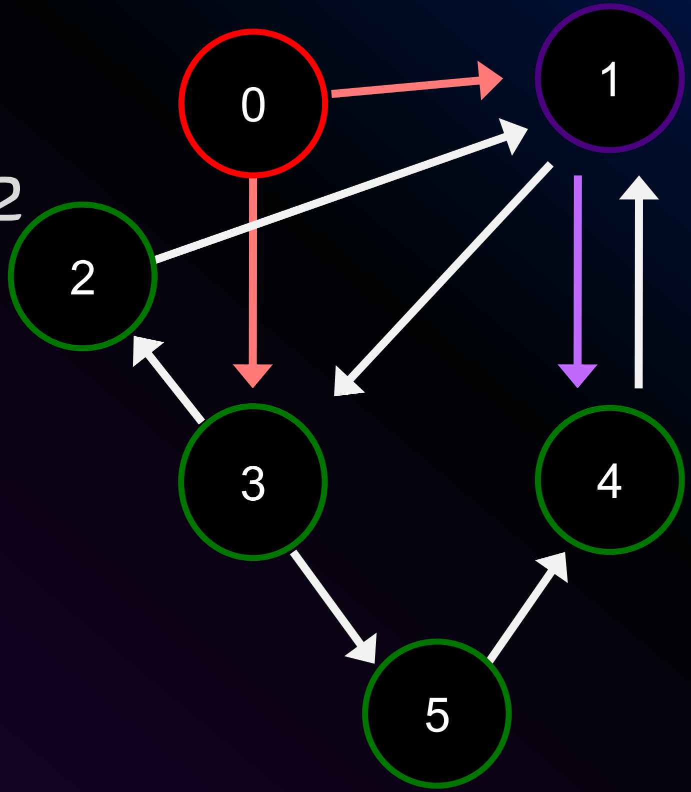
2. MPI_DIST_GRAPH_CREATE EXAMPLE

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

```
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0*

- num_src_nodes: 2
- Sources(2): {0, 1}
- Degrees(2): {2,1}



* Note that the Process ID has nothing to do with the Node IDs.

2. MPI_DIST_GRAPH_CREATE — EXAMPLE

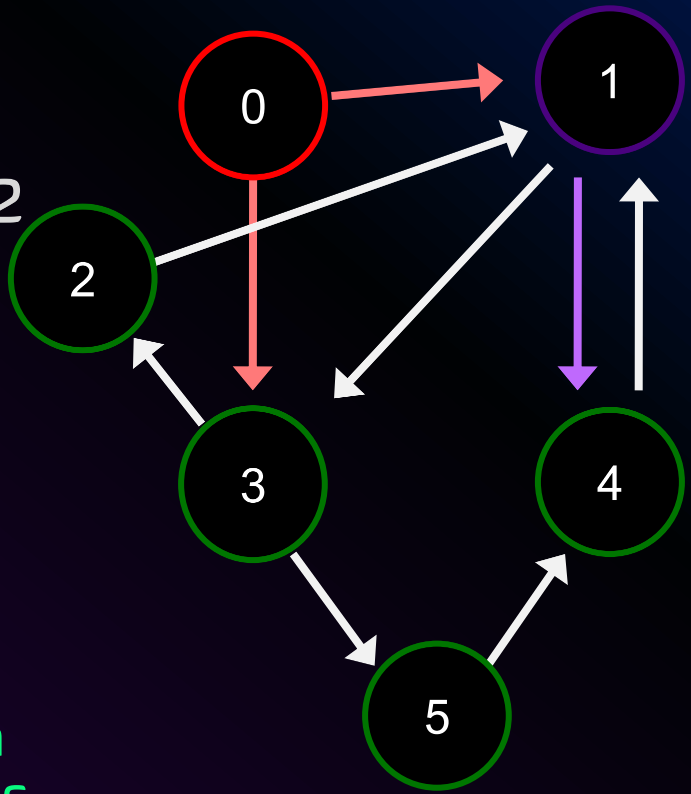
Makes a new communicator to which topology information has been attached.

```
int MPI_Dist_graph_create(
 MPI_Comm comm_old,
 int num_src_nodes,
 const int sources[],
 const int degrees[],
 const int destinations[],
 const int weights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0*

- num_src_nodes: 2
- Sources(2): {0, 1}
- Degrees(2): {2,1}
- Dest(2+1): {3,1,4}

... Other processes then specify remaining edges.



* Note that the Process ID has nothing to do with the Node IDs.

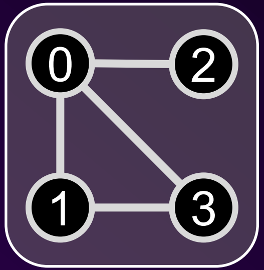
2. MPI_DIST_GRAPH_CREATE

Allows MPI processes to pass any edge of the graph, which offers the best flexibility.

>> Requires global communication <<

Slightly more expensive than next approach (adjacent).

3. MPI_DIST_GRAPH_CREATE_ADJACENT



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

Each MPI process specifies all edges in which it is part of, and solely those edges.

Local Info Only

Direct Edges

Edges Specified Twice

3. MPI_DIST_GRAPH_CREATE_ADJACENT

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

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

indegree : Size of sources and sourceweights arrays.

sources : Ranks of processes for which the calling process is a destination (array of non-negative integers).

sourceweights : Weights of the edges into the calling process (array of non-negative integers).

* Note that the Process ID has nothing to do with the Node IDs.

3. MPI_DIST_GRAPH_CREATE_ADJACENT

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

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

outdegree : Size of destinations and destweights arrays.

destinations : Ranks of processes for which the calling process is a source (array of non-negative integers).

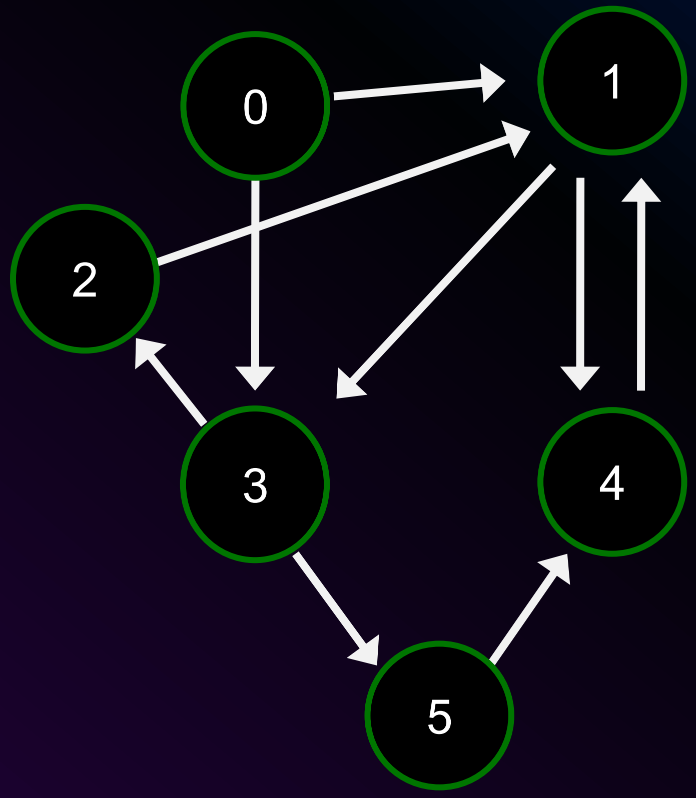
destweights : Weights of the edges out of the calling process (array of non-negative integers).

* Note that the Process ID has nothing to do with the Node IDs.

3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

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

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

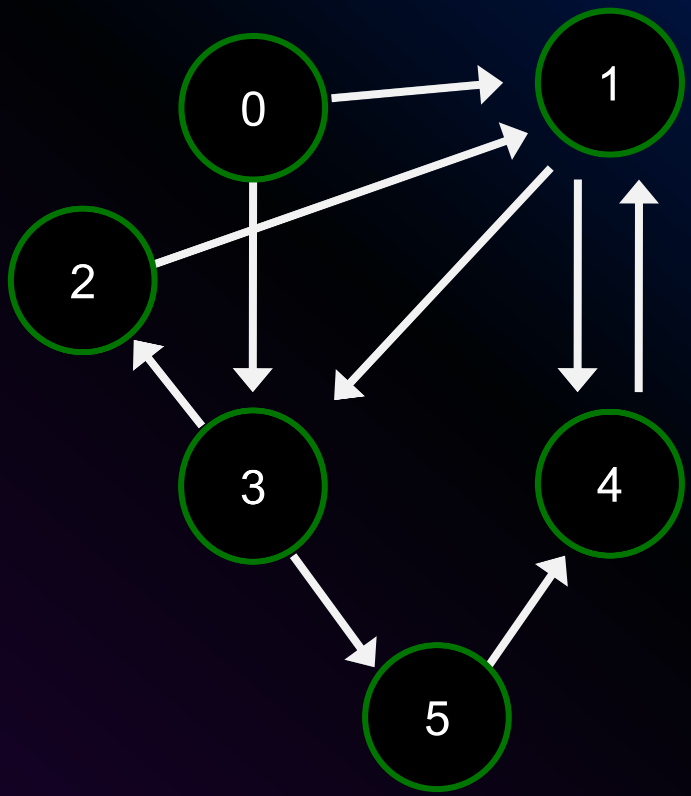


3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

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

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0



3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

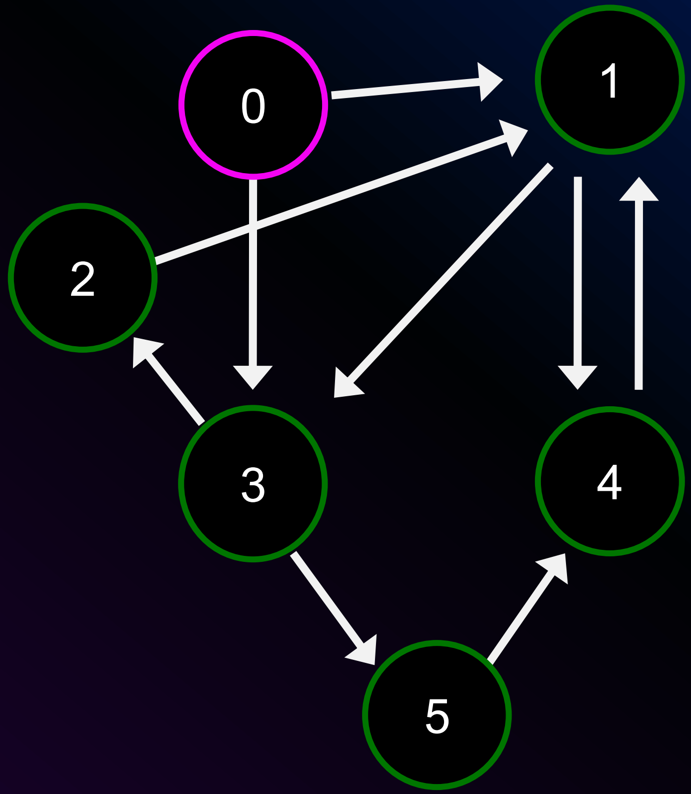
Makes a new communicator to which topology information has been attached.

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0

(map to Node 0) *

- indegree: 0
- outdegree: 2



* For convenience, map each Process to corresponding Graph Node.

3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

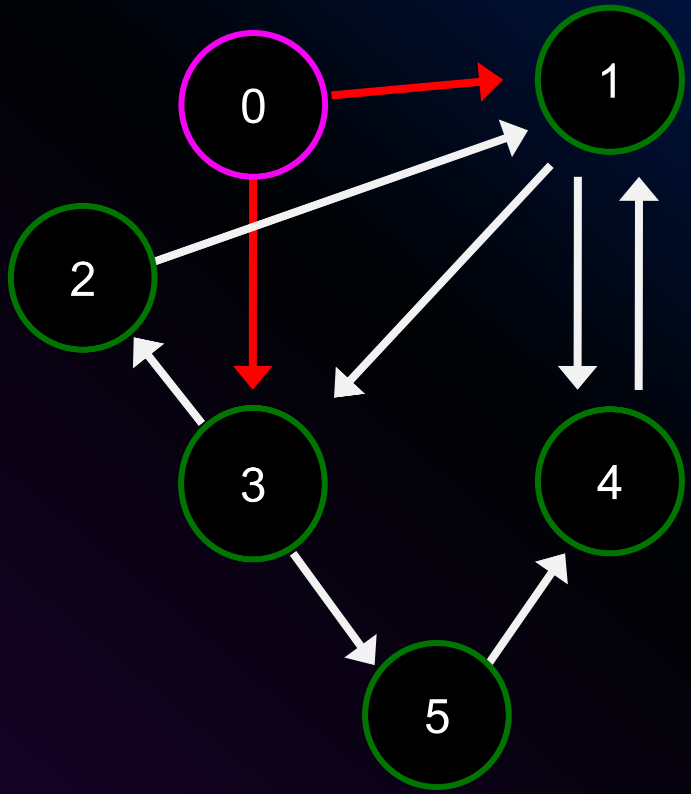
Makes a new communicator to which topology information has been attached.

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0

(map to Node 0) *

- indegree: 0
- outdegree: 2
- dest(2): {3, 1}



* For convenience, map each Process to corresponding Graph Node.

3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

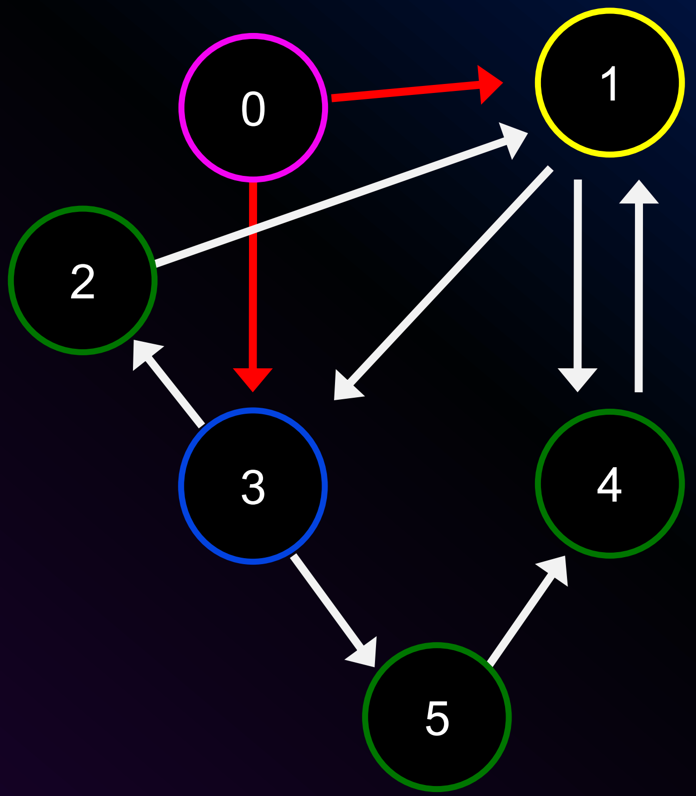
Makes a new communicator to which topology information has been attached.

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 0

(map to Node 0) *

- indegree: 0
- outdegree: 2
- dest(2): {3, 1}



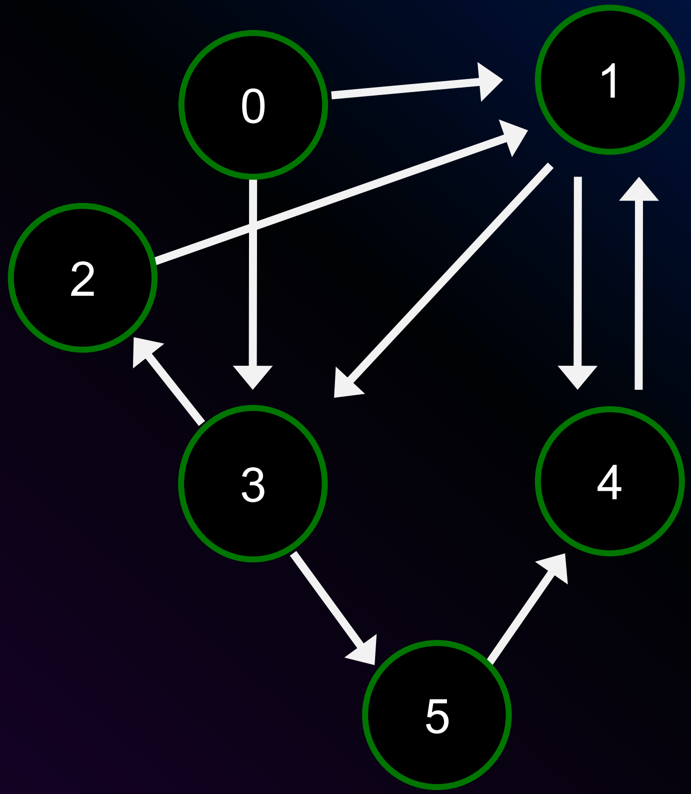
* For convenience, map each Process to corresponding Graph Node.

3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

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

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 1



* For convenience, map each Process to corresponding Graph Node.

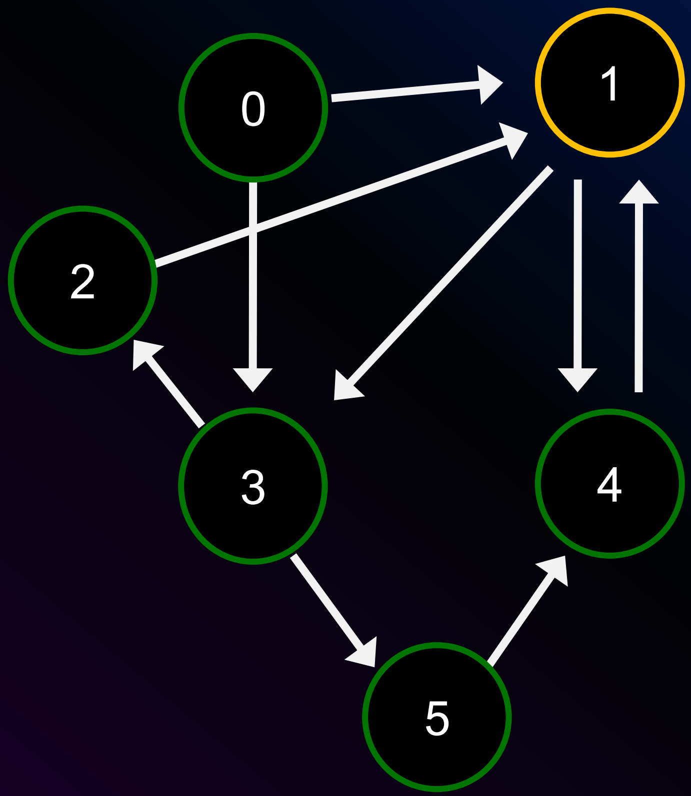
3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

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

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 1

(map to Node 1) *



* For convenience, map each Process to corresponding Graph Node.

3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

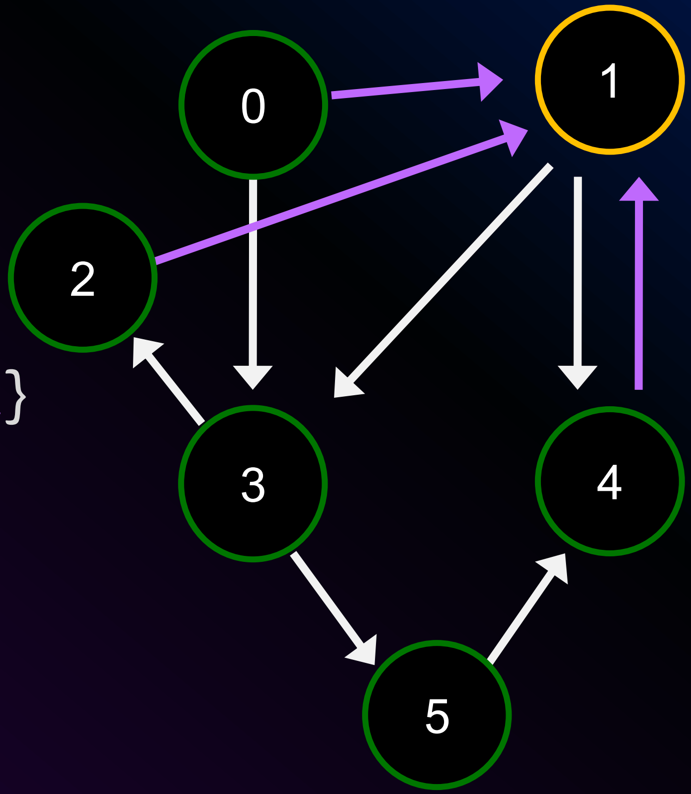
Makes a new communicator to which topology information has been attached.

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 1

(map to Node 1) *

- indegree: 3
- Sources(3): {4,0,2}



* For convenience, map each Process to corresponding Graph Node.

3. MPI_DIST_GRAPH_CREATE_ADJACENT EXAMPLE

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

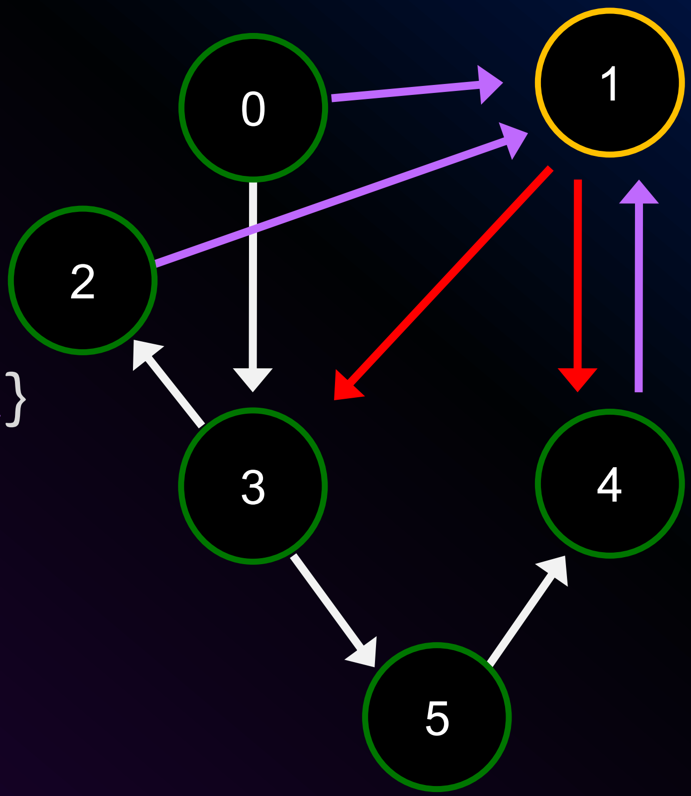
```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm_old,
 int indegree,
 const int sources[],
 const int sourceweights[],
 int outdegree,
 const int destinations[],
 const int destweights[],
 MPI_Info info,
 int reorder,
 MPI_Comm *comm_dist_graph);
```

Process 1

(map to Node 1) *

- indegree: 3
- Sources(3): {4,0,2}
- outdegree: 2
- Dest(2): {3,4}

… Remaining Processes



* For convenience, map each Process to corresponding Graph Node.

VIRTUAL TOPOLOGY: GRAPHS RECAP

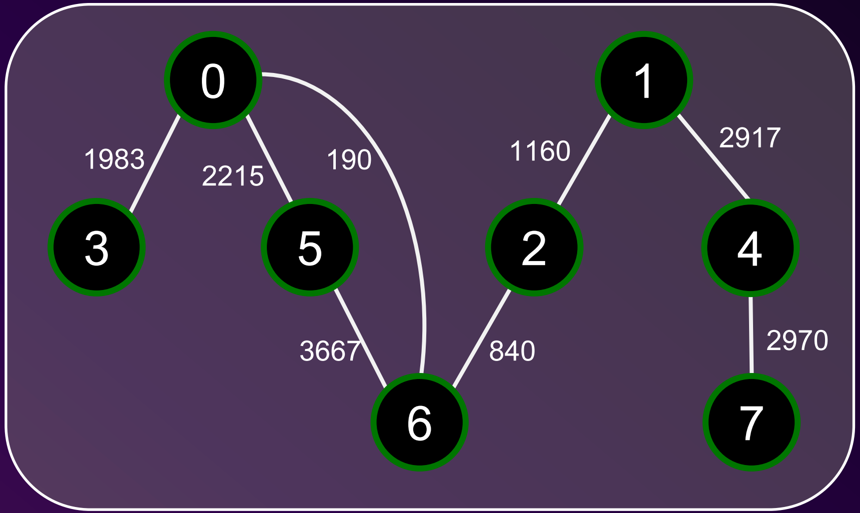
We have three ways to create a graph topology:

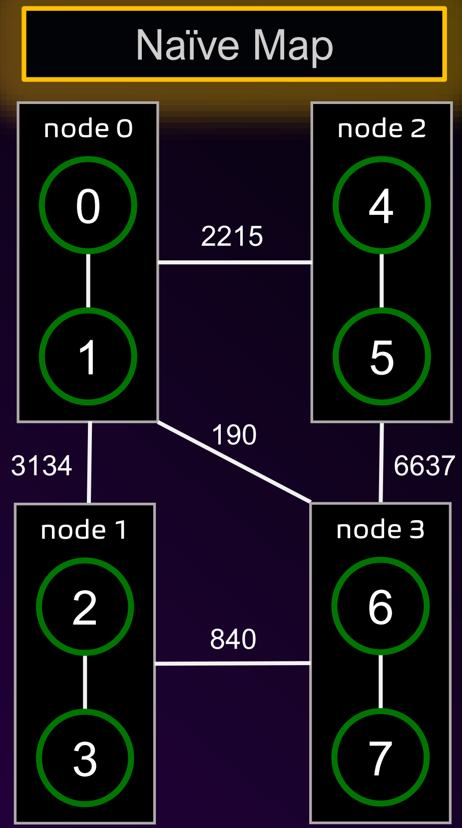
1. MPI_Graph_create
2. MPI_Dist_graph_create
3. MPI_Dist_graph_create_adjacent

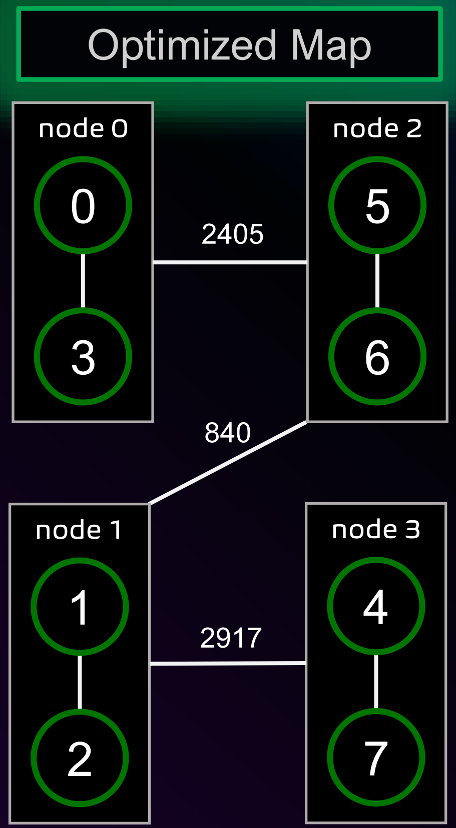
Opportunities to Optimize Performance through Reordering Ranks

VIRTUAL TOPOLOGY: MAPPING BASICS

- MPI supports rank reordering.
  - Change numbering in a given allocation to reduce congestion or dilation.
  - Through may depends on vendor implementation, e.g., OpenMPI, MPICH, Intel MPI.
- Example: On-Node Reordering







DISTRIBUTED GRAPH NEIGHBOR QUERIES

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm,int *indegree,int *outdegree,int *weighted);
```

Returns the number of in and out edges for the calling processes in a distributed graph topology and a flag indicating whether the distributed graph is weighted.

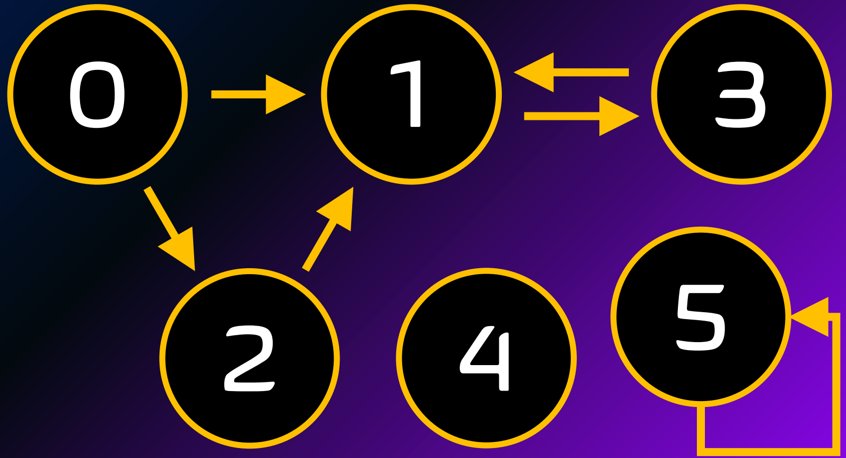
```
MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[], int sourceweights[],
                         int maxoutdegree, int destinations[], int destweights[]);
```

Returns the neighbors of the calling process in a distributed graph topology.

20 Minutes

HANDS ON ACTIVITY #3

MPI_Dist_graph_create



SECTION IV

NEIGHBORHOOD COLLECTIVES

NEIGHBORHOOD COLLECTIVES

Virtual Topologies implement no communication!

- ➢ They are just helper functions.

Collective Communications only cover some patterns

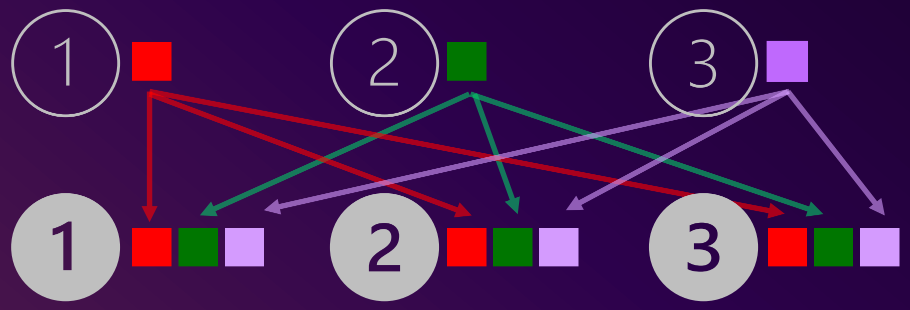
- ➢ E.g., no stencil patterns.

✓ 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);
```



sendbuf : Starting address of send buffer.

sendcount : Number of elements in send buffer (integer).

sendtype : Datatype of send buffer elements (handle).

recvbuf : Starting address of recv buffer.

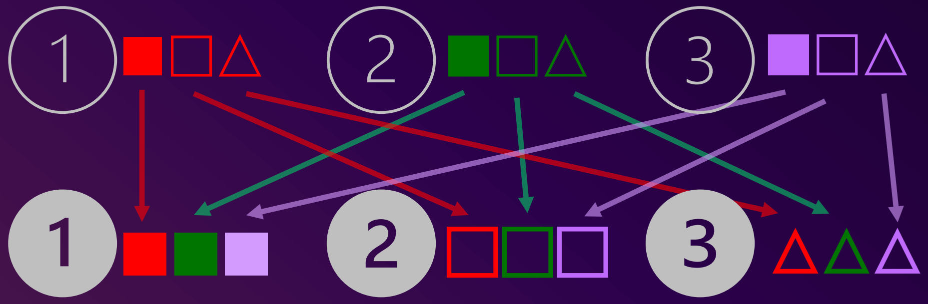
recvcount : Number of elements in send buffer (integer).

recvtype : Datatype of send buffer elements (handle).

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);
```



sendbuf : Starting address of send buffer.

sendcount : Number of elements in send buffer (integer).

sendtype : Datatype of send buffer elements (handle).

recvbuf : Starting address of recv buffer.

recvcount : Number of elements in send buffer (integer).

recvtype : Datatype of send buffer elements (handle).

NEIGHBORHOOD COLLECTIVES: CARTESIAN

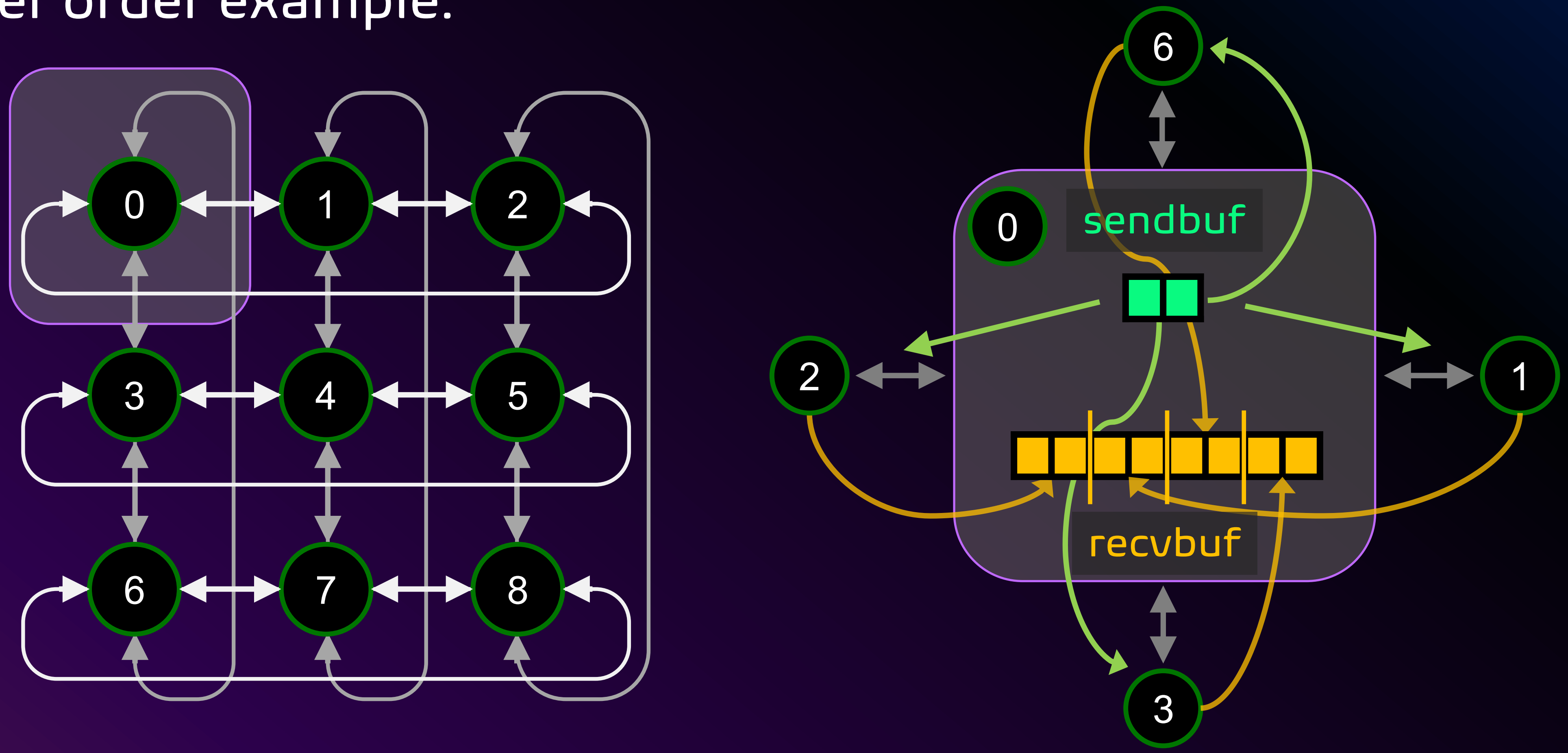
Communicate with direct neighbors in Cartesian topology

- Corresponds to cart_shift with disp = 1
- Collective:
  - all processes in comm must call it, including processes without neighbors
- Buffers are laid out as neighbor sequence:
  - Defined by order of dimensions, first negative, then positive
  - 2*ndims sources and destinations
  - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!

NEIGHBORHOOD COLLECTIVES: CARTESIAN

Communicate with direct neighbors in Cartesian topology

- Buffer order example:



NEIGHBORHOOD COLLECTIVES: GRAPHS

Collective Communication along arbitrary neighborhood

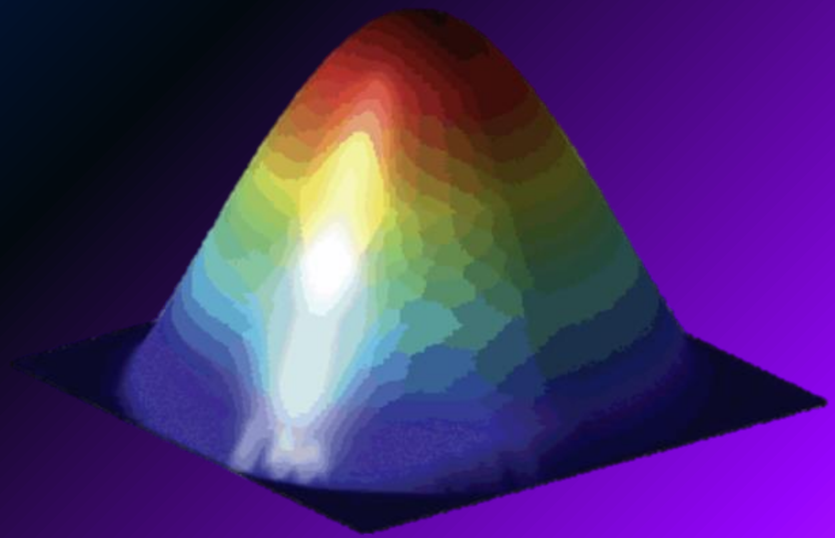
- Order is determined by order of neighbors as returned by (dist_)graph_neighbors.
- Distributed graph is directed, so may have different number of send/recv neighbors.
- Can express dense collective operations.

15 Minutes

HANDS ON ACTIVITY #4

MPI_Ineighbor_alltoallv

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



SECTION V

GRAPH PARTITIONING

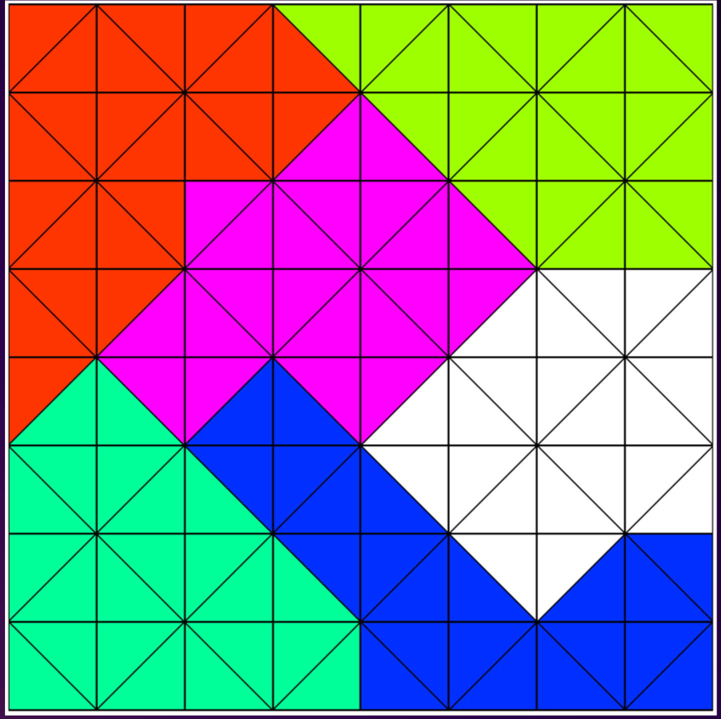
(USING METIS)

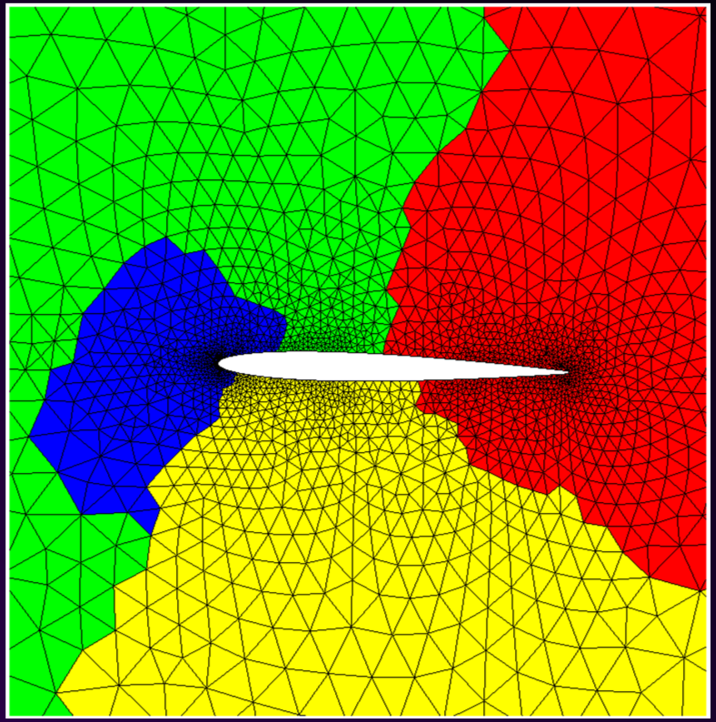
METIS / PARMETIS: GRAPH PARTITIONER

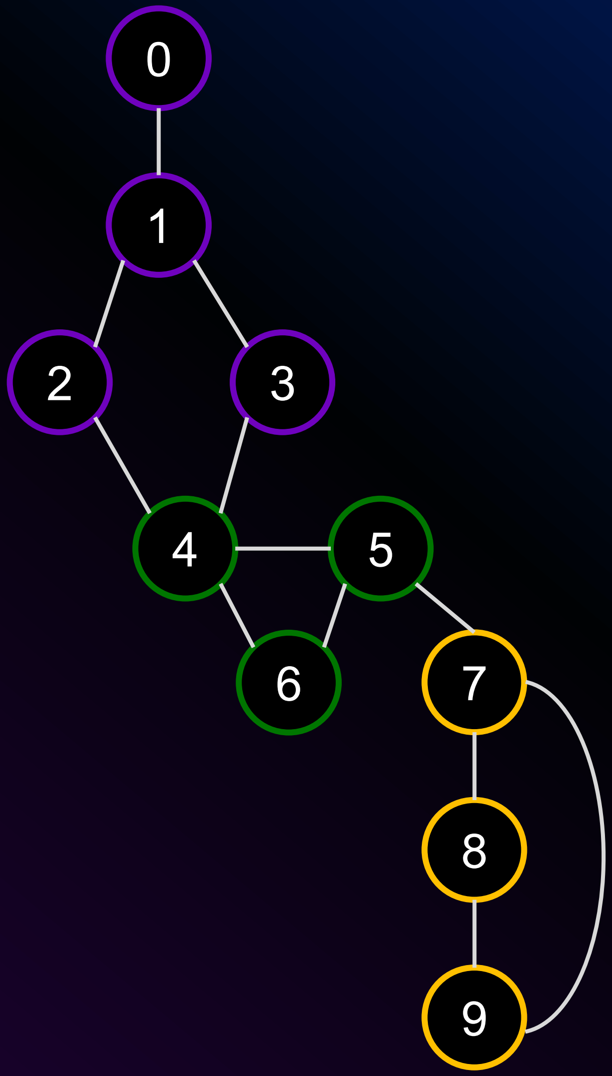
Available from the Karypis Lab

University of Minnesota

- http://glaros.dtc.umn.edu/gkhome/views/metis







METIS' COMMAND-LINE AND LIBRARY INTERFACES.

Operation	Stand-alone Program	API Routine
Partition a graph	gpmetis	METIS_PartGraphRecursive METIS_PartGraphKway
Partition a mesh	mpmetis	METIS_PartMeshNodal METIS_PartMeshDual
Compute a fill-reducing ordering of a sparse matrix	ndmetis	METIS_NodeND
Convert a mesh into a graph	m2gmetis	METIS_MeshToNodal METIS_MeshToDual

Documentation

http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf

METIS: METIS_PARTGRAPHRECURSIVE

Partition a graph into k parts using multilevel recursive bisection.

```
int METIS_PartGraphRecursive(
      idx_t *nvtxs,
      idx_t *ncon,
      idx_t *xadj,
      idx_t *adjncy,
      idx_t *vwgt,      // NULL
      idx_t *vsize,     // NULL
      idx_t *adjwgt,    // NULL
      idx_t *nparts,
      real_t *tpwgts,   // NULL
      real_t ubvec,     // NULL
      idx_t *options,   // NULL
      idx_t *objval,
      idx_t *part);
```

- nvtxs : The number of vertices in the graph.
- ncon : The number of balancing constraints. It should be at least 1 (i.e., set to 1).
- xadj [nvtxs+1] : The 1st adjacency structure.
- adjncy [2*nedges] : The 2nd adjacency structure.
- vwgt : The weights of the vertices.
- vsize : The size of the vertices for computing the total communication volume.
- adjwgt : The weights of the edges.
- nparts : The number of parts to partition.
- tpwgts : The desired weight for each partition/constraint.
- ubvec : The unbalanced load tolerance vector.
- part [nvtxs] : The partition vector of the graph. Use options(METIS_OPTION_NUMBERING) for base 0 or 1.

METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

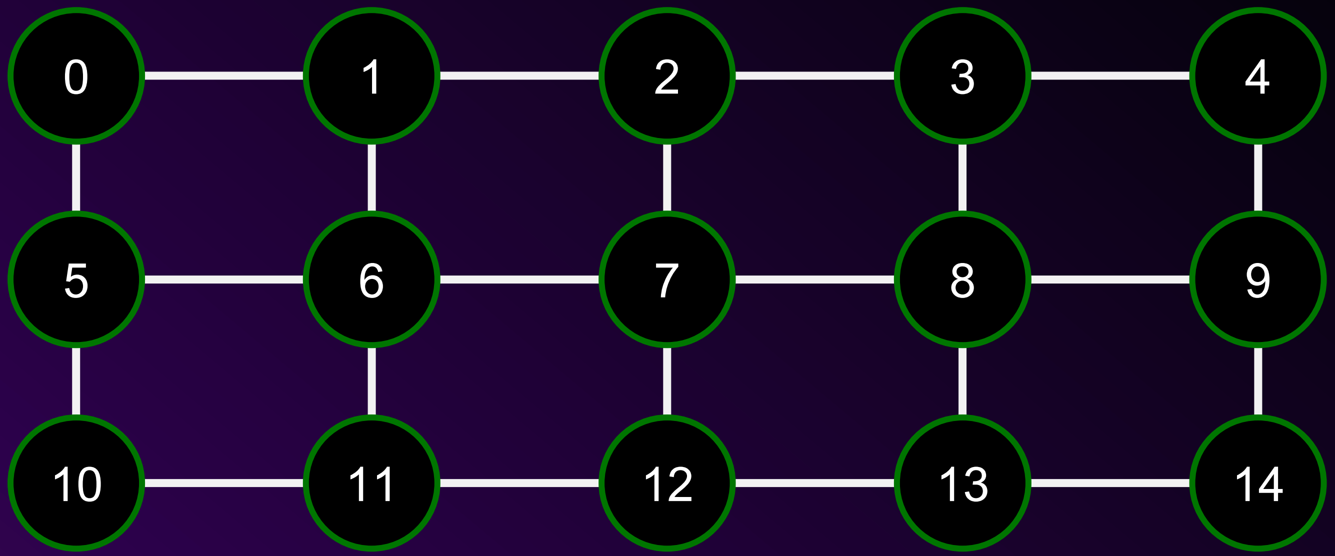
Partition a graph into k parts using multilevel recursive bisection.

xadj - size is nvts+1
adjncy - size is 2*nedges

CSR Format

xadj = [0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44]
adjncy = [1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13]

nvtxs = 15
nedges = 22



METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

xadj – stores the index

adjncy – list of node connected

xadj	0																					
adjncy	1	5																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15

nedges = 22

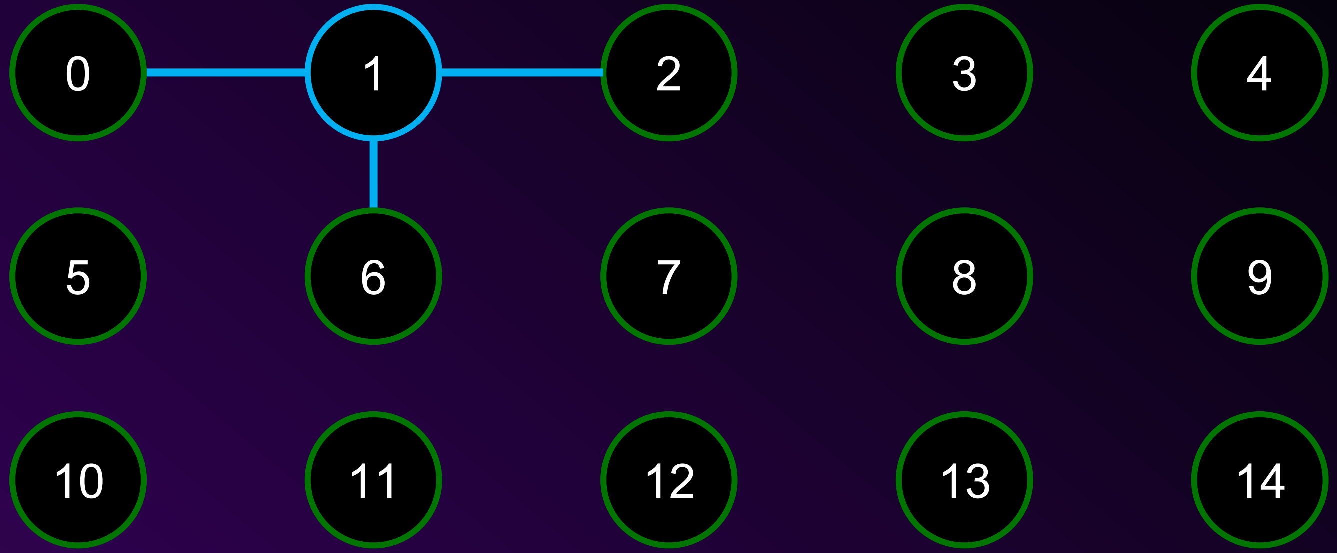
METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

xadj – stores the index
adjncy – list of node connected

xadj	0	2																				
adjncy	1	5	0	2	6																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15
nedges = 22



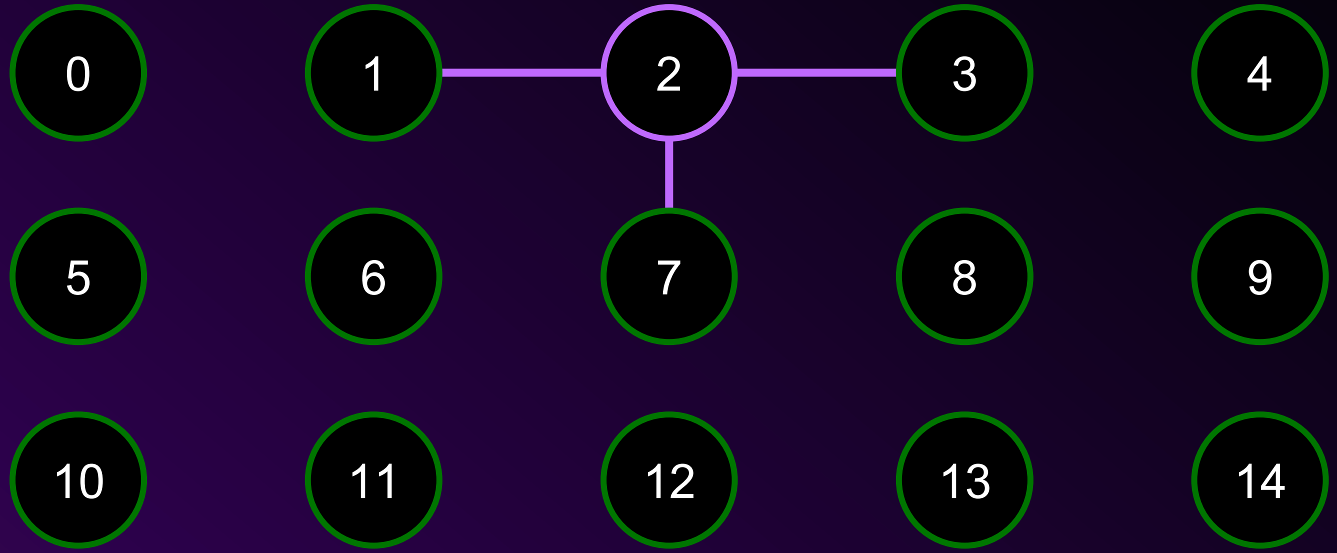
METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

xadj – stores the index
adjncy – list of node connected

xadj	0	2	5																			
adjncy	1	5	0	2	6	1	3	7														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15
nedges = 22



METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

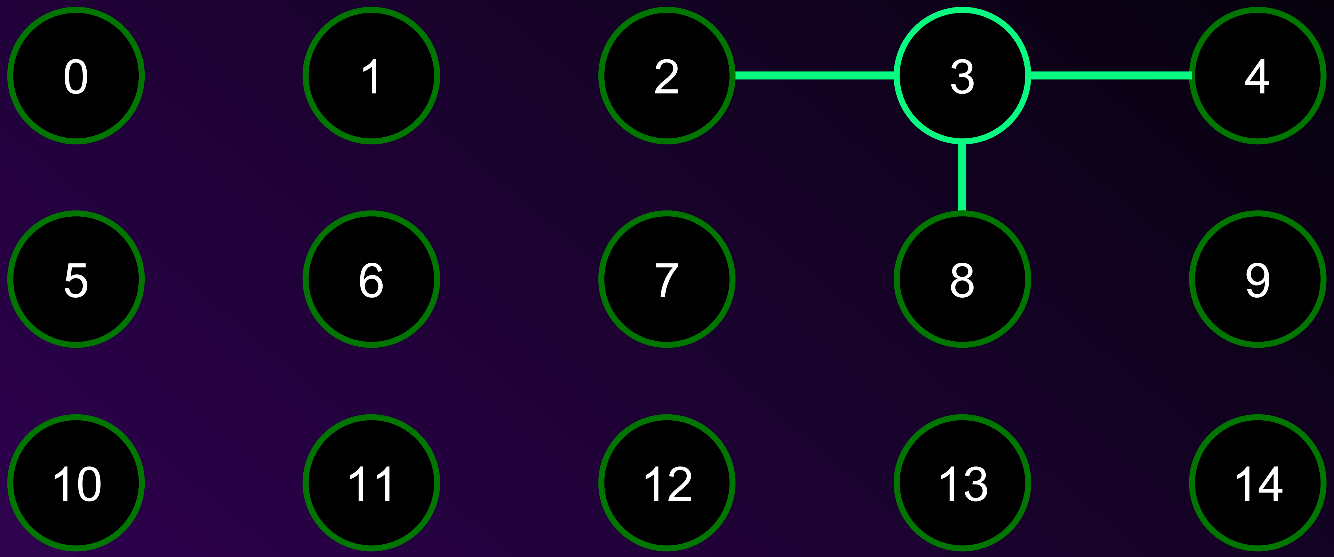
xadj – stores the index

adjncy – list of node connected

xadj	0	2	5	8																		
adjncy	1	5	0	2	6	1	3	7	2	4	8											
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15

nedges = 22



METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

xadj – stores the index
adjncy – list of node connected

xadj	0	2	5	8	11																	
adjncy	1	5	0	2	6	1	3	7	2	4	8	3	9									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15
nedges = 22

METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into **k parts** using multilevel recursive bisection.

xadj – stores the index
adjncy – list of node connected

xadj	0	2	5	8	11	13																
adjncy	1	5	0	2	6	1	3	7	2	4	8	3	9	0	6	10						
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15
nedges = 22

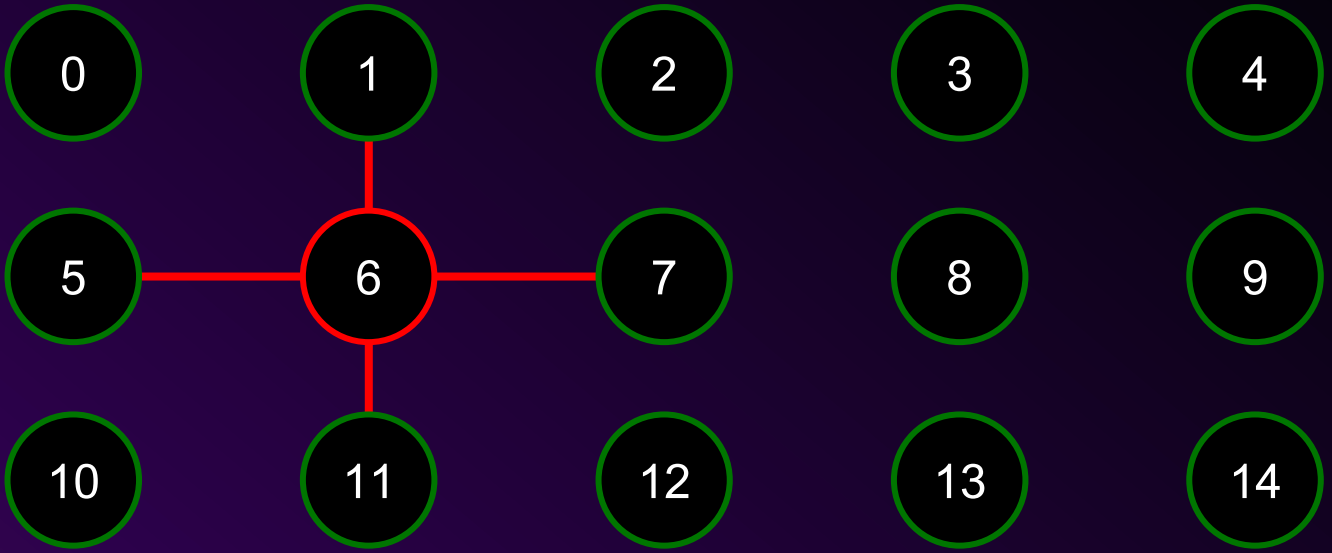
METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

xadj – stores the index
adjncy – list of node connected

xadj	0	2	5	8	11	13	16															
adjncy	1	5	0	2	6	1	3	7	2	4	8	3	9	0	6	10	1	5	7	11		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

nvtxs = 15
nedges = 22



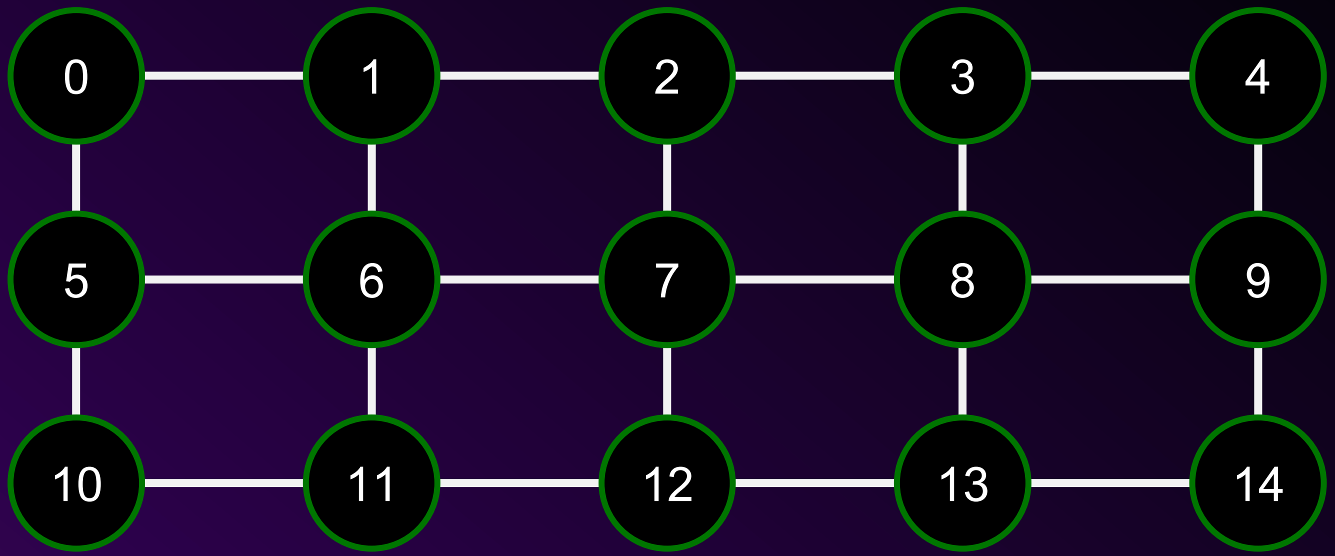
METIS: METIS_PARTGRAPHRECURSIVE EXAMPLE

Partition a graph into k parts using multilevel recursive bisection.

xadj – stores the index
adjncy – list of node connected

xadj	0	2	5	8	11	13	16	...														
adjncy	1	5	0	2	6	1	3	7	2	4	8	3	9	0	6	10	5	7	1	11	...	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

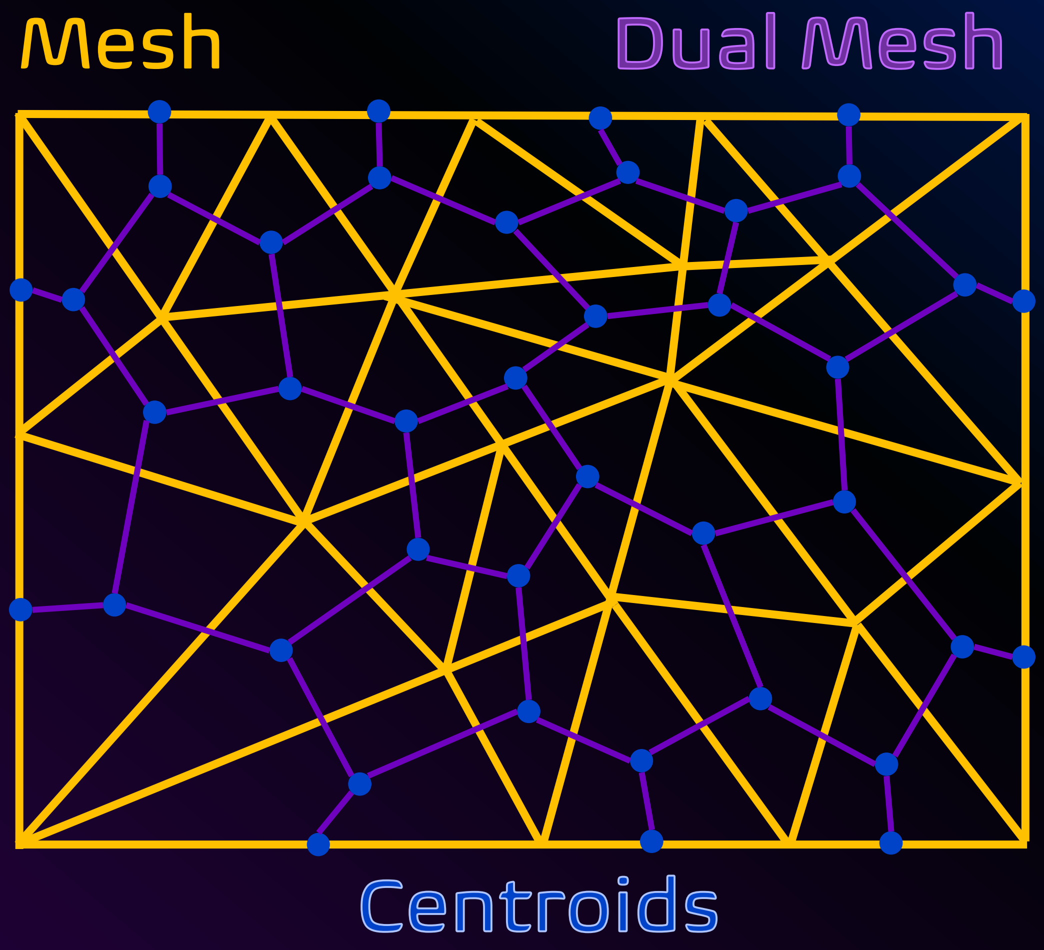
nvtxs = 15
nedges = 22



METIS: METIS_PARTMESHDUAL

Partition mesh into k parts based on a partitioning of the mesh's dual graph.

```
int METIS_PartMeshDual (
        idx_t *ne,
        idx_t *nn,
        idx_t *eptr,
        idx_t *adjncy,
        idx_t *vwgt,      // NULL
        idx_t *vsize,     // NULL
        idx_t ncommon,
        idx_t *nparts,
        idx_t *adjwgt,    // NULL
        idx_t *options, // NULL
        idx_t *objval,
        idx_t *epart,
        idx_t *npart);
```



METIS: METIS_PARTMESHDUAL

Partition mesh into k parts based on a partitioning of the mesh's dual graph.

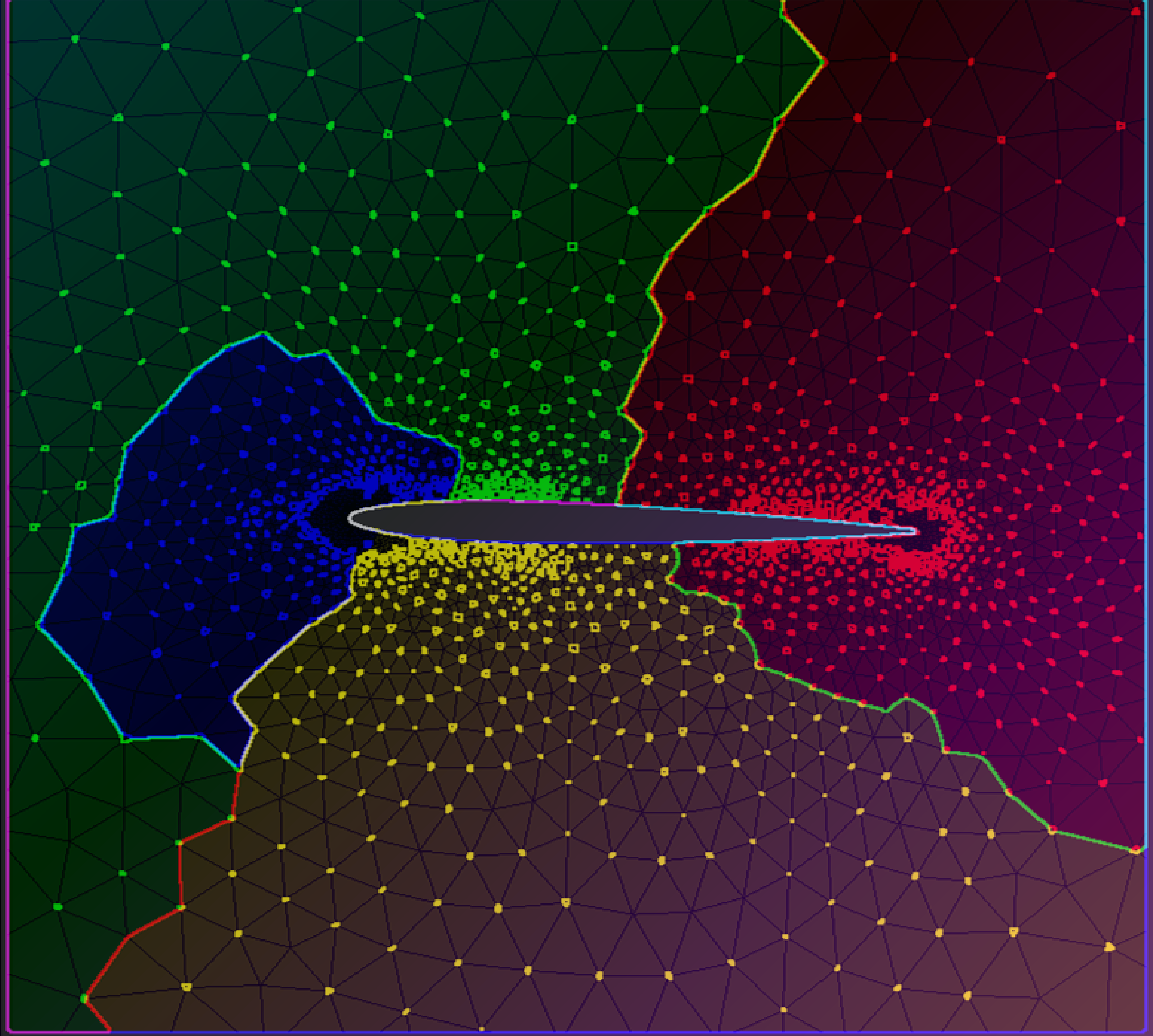
```
int METIS_PartMeshDual (
      idx_t *ne,
      idx_t *nn,
      idx_t *eptr,
      idx_t *adjncy,
      idx_t *vwgt,     // NULL
      idx_t *vsize,    // NULL
      idx_t ncommon,
      idx_t *nparts,
      idx_t *adjwgt,   // NULL
      idx_t *options,  // NULL
      idx_t *objval,
      idx_t *epart,
      idx_t *npart);
```

ne : The number of elements in mesh.
nn : The number nodes is mesh.
eptr : The 1st mesh array structure.
eind : The 2nd mesh array structure.
vwgt : The weights of the elements.
vsize : The size of the elements for computing the total communication volume.
ncommon : Number of nodes to make element face.
nparts : The number of parts to partition.
adjwgt : The weights of the edges.

epart [ne] : The partition vector of the elements.
npart [nn] : The partition vector of the nodes.

HANDS ON ACTIVITY #5

METIS_PartMeshDual



FINAL TAKEAWAYS

What we learned:

- MPI Derived Datatypes and Custom Operations
- MPI Virtual Topologies
- MPI Neighborhood Collectives
- Graph Partitioning