Introduction to MPI Part 2

Yuwu Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
May 31, 2018

Topics to be covered today

- ❑ **Review introduction to MPI part 1**
- ❑ **User defined data types**
- ❑ **Collective communication**

Introduction to MPI, Part 2

Review MPI part 1

MPI Program Basics

- ❑ **In MPI, data is shared among processes using ____**
  - a) shared memory
  - b) distributed memory ☺
  - c) both of shared and distributed memory
- ❑ **What commands did we use in the MPI part1?**
- ❑ **Which MPI compiler did we use?**
- ❑ **Initialize communications**
  - ➢ MPI_INIT initializes the MPI environment
  - ➢ MPI_COMM_SIZE returns the number of processes
  - ➢ MPI_COMM_RANK returns this process's index (rank)
- ❑ **Exit in a "clean" fashion when MPI communication is done**
  - ➢ MPI_FINALIZE

Point-to-point communication

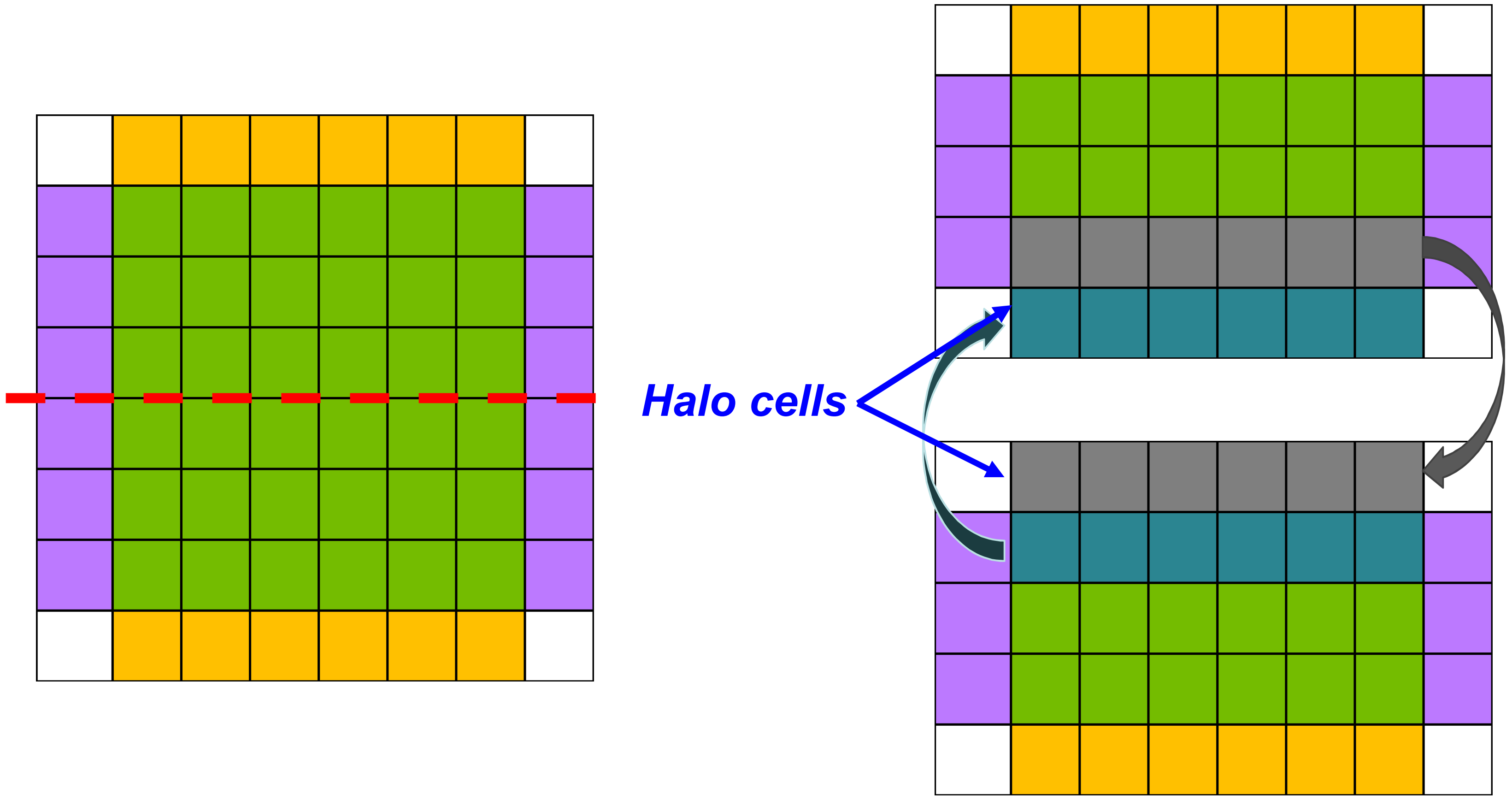
- **Blocking send/receive**
  - The sending process calls the MPI_SEND function
  - C: int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
  - Fortran: MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
  - The receiving process calls the MPI_RECV function
  - C: int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status);
  - Fortran: Fortran: MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS, IERR)
- **A MPI message consists of two parts**
  - Message itself: data body
  - Message envelope: routing info
- **status: information of the message that is received**

Introduction to MPI, Part 2

User Defined Data Types

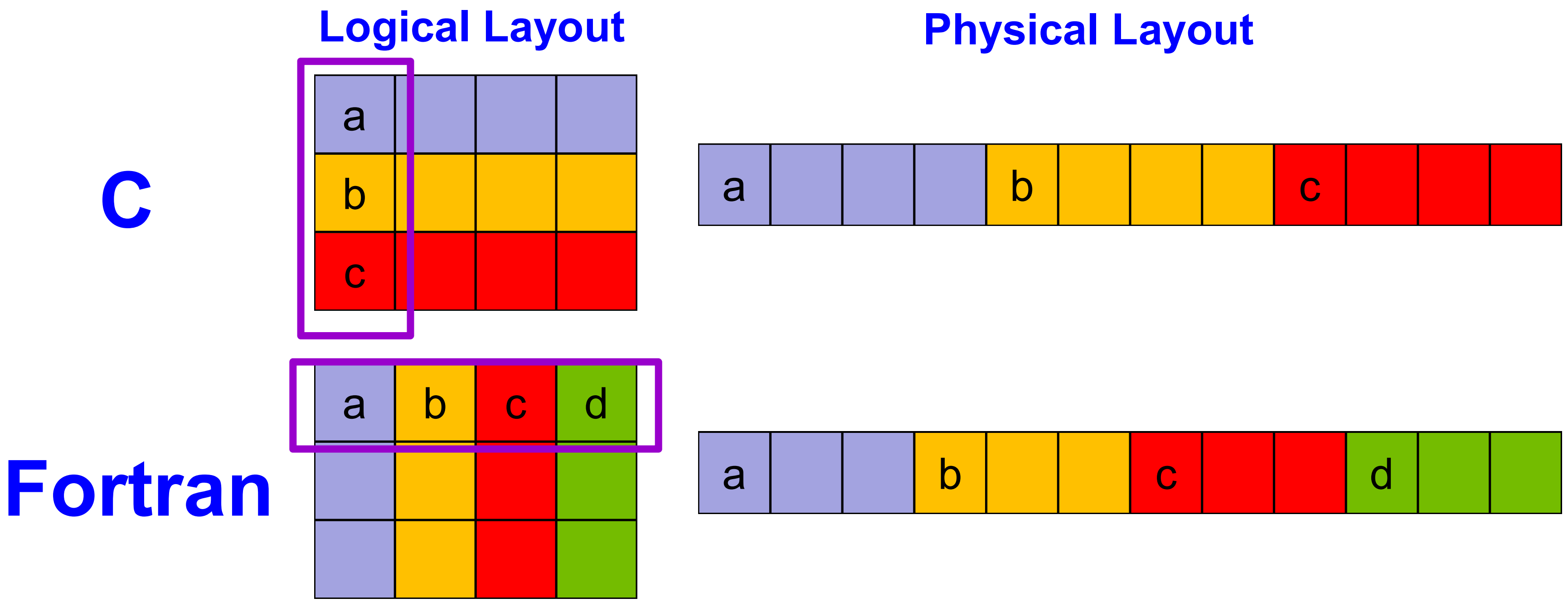
Laplace solver 1D Decomposition

- Why did we divide the domain in rows in C?



Sending a Matrix Column in C or Row in Fortran

- **Column of a matrix is not contiguous in memory in C**
- **Several options for sending a column (row) in C (Fortran):**
  - Use several send commands for each element of a row
  - Copy entire matrix to some temporary buffer and send that with one send command
- **We can create a matching datatype and send all data with one send command**



Summary of Elementary Data Types

- **MPI provides many predefined datatypes for each language binding:**

MPI Data Type	C Data Type
`MPI_CHAR`	signed char
`MPI_SHORT`	signed short int
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short
`MPI_UNSIGNED`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double
`MPI_BYTE`	
`MPI_PACKED`	

MPI Data Type	Fortran Data Type
`MPI_INTEGER`	INTEGER
`MPI_REAL`	REAL
`MPI_REAL8`	REAL*8
`MPI_DOUBLE_PRECISION`	DOUBLE PRECISION
`MPI_COMPLEX`	COMPLEX
`MPI_LOGICAL`	LOGICAL
`MPI_CHARACTER`	CHARACTER(1)
`MPI_BYTE`	
`MPI_PACKED`	

Recall:

```
MPI_Send(sendbuf, count, MPI_INT,
int dest, tag, MPI_COMM_WORLD)
```

More on MPI Datatypes

- **MPI datatypes are used for communication purposes**
  - Datatype tells MPI where to take the data when sending and where to put data when receiving
- **MPI datatypes must match the language data type of the data array.**
- **MPI datatypes are handles and cannot be used to declare variables.**
- **MPI datatypes tell MPI how to:**
  - read actual memory values from the send buffer
  - write actual memory values into the receive buffer
  - convert between machine representations in heterogeneous environments
  - MPI_BYTE is used to send and receive data as-is without any conversion
- **Elementary datatypes (`MPI_INT, MPI_REAL,` ...)**
  - Different types in Fortran and C, correspond to languages basic types
  - Enable communication using contiguous memory sequence of identical elements (e.g. vector or matrix)

Why Derived Data Types?

- **Use elementary datatypes as building blocks**
- **Enable communication of**
  - Non-contiguous data with a single MPI call, e.g. rows or columns of a matrix
  - Heterogeneous data (structs in C, types in Fortran)
- **Provide higher level of programming & efficiency**
  - Code is more compact and maintainable
  - Communication of non-contiguous data is more efficient

Advantages of using Derived Datatypes

- **User-defined datatypes can be used both in point-to-point communication and collective communication**
- **The datatype instructs where to take the data when sending or where to put data when receiving**
  - Non-contiguous data in sending process can be received as contiguous or vice versa

Procedure creating user-defined datatypes

- **A new datatype is created from existing ones with a datatype constructor**
  - Several routines for different special cases
- **A new datatype must be committed before using it**

```
MPI_Type_commit(newtype)
// newtype is the new datatype to commit
```

- **A type should be freed after it is no longer needed**

```
MPI_Type_free(newtype)
// newtype is the datatype for decommission
```

- **User defined datatypes can be nested, e.g., one can use the new type to define another user defined type**

Datatype constructors

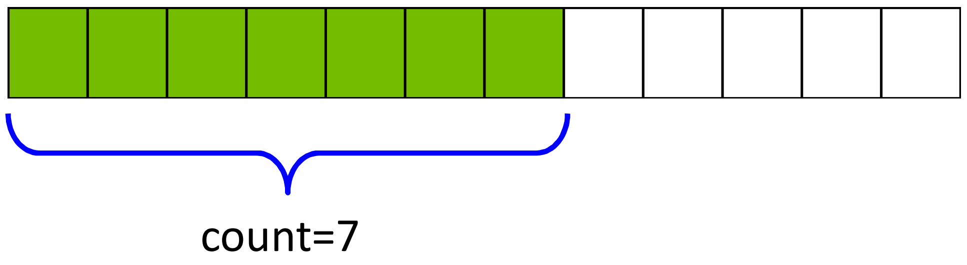
Function Name	Notes
`MPI_Type_contiguous`	Contiguous datatypes
`MPI_Type_vector`	Regularly spaced datatype
`MPI_Type_indexed`	Variably spaced datatype
`MPI_Type_create_subarray`	Subarray within a multi-dimensional array
`MPI_Type_create_hvector`	Like vector, but uses bytes for spacings
`MPI_Type_create_hindexed`	Like index, but uses bytes for spacings
`MPI_Type_create_struct`	Fully general datatype

Derived Data Type: Contiguous

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
    MPI_Datatype *newtype) // C/C++
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER    COUNT, OLDTYPE, NEWTYPE, IERROR ! Fortran
```

- **Creates a contiguous datatype:**
  - count: Replication count (nonnegative integer).
  - oldtype: Old datatype (handle).
  - newtype: New datatype (handle)
- **Example:**

```
MPI_Type_contiguous(7, MPI_FLOAT, &my_contigous_type);
```

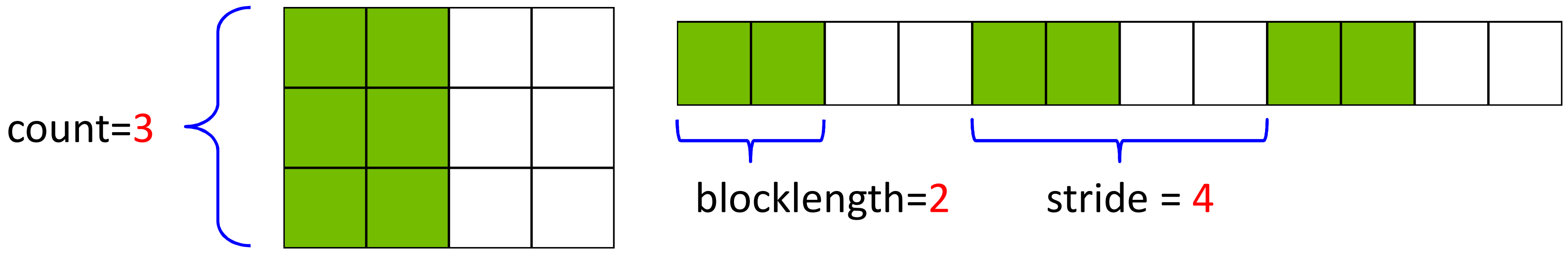


Derived Data Type: Vector

```
int MPI_Type_vector(int count, int blocklength, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype) // C/C++
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER    COUNT, BLOCKLENGTH, STRIDE, OLDTYPE
    INTEGER    NEWTYPE, IERROR ! Fortran
```

- **Allows replication of a data type into locations that consist of equally spaced blocks.**
  - count: Number of blocks (nonnegative integer).
  - blocklength: Number of elements in each block (nonnegative integer).
  - stride: Number of elements between start of each block (integer).
  - oldtype: Old datatype (handle).
  - newtype: New datatype (handle)
- **Example:**

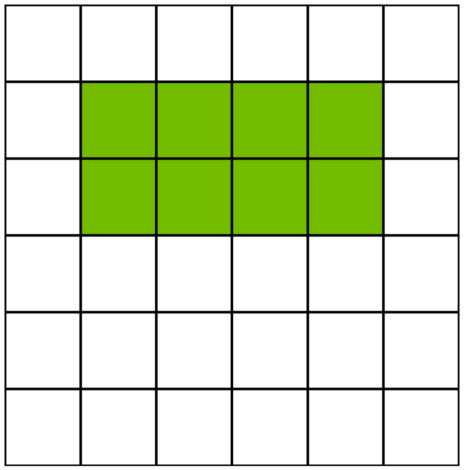
```
MPI_Type_vector(3,2,4,MPI_FLOAT,&my_vector_type);
```



Derived Data Type: MPI_Type_create_subarray

```
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
        const int array_of_subsizes[], const int array_of_starts[], int
        order, MPI_Datatype oldtype, MPI_Datatype *newtype) // C/C++
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
    ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR ! Fortran
```

- **Creates a data type describing an n-dimensional subarray of an n-dimensional array.**



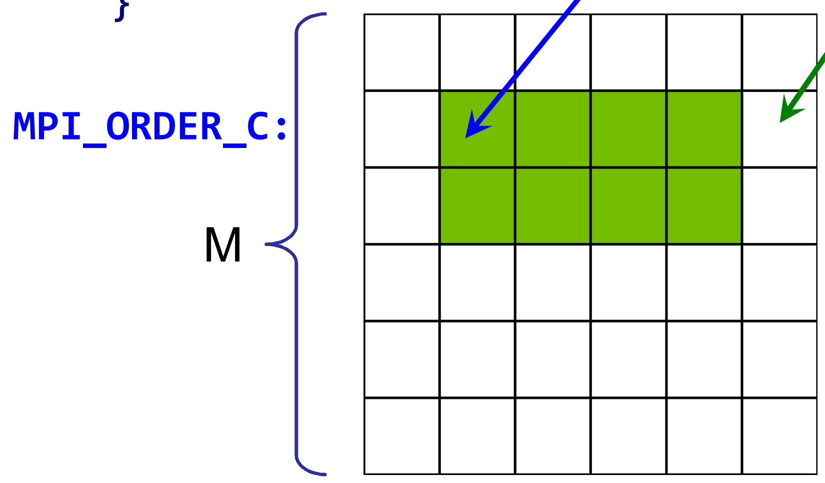
The parameters we might think of:

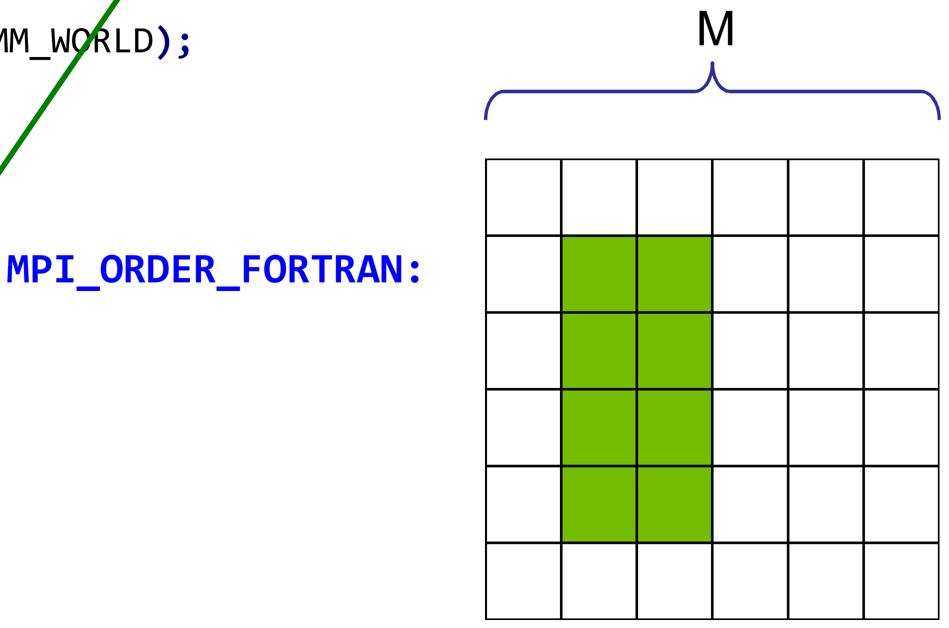
- ndims: Number of array dimensions
- `sizes[]`: of the full array in each dimension
- `subsizes[]`: of the subarray array in each dimension
- `starts[]`: Starting coordinates of the subarray in each dimension.

MPI_Type_create_subarray example

```
/*mpi_mt_1blk_subarray.c*/
#define M 6
if (rank==0) {
    int sizes[2]={M,M};
    int subsizes[2]={2,4}; // defines the sub-region
    int offset[2]={1,1}; // defines the starting location
    MPI_Datatype sub_mat;
    MPI_Type_create_subarray(2,sizes,subsizes,offset,MPI_ORDER_C,MPI_FLOAT,&sub_mat);
    MPI_Type_commit(&sub_mat);
    MPI_Send(a,1,sub_mat,1,0,MPI_COMM_WORLD);
    MPI_Type_free(&sub_mat);
}
```

Do *not* try to transpose matrix using MPI_ORDER_C/MPI_ORDER_FORTRAN.





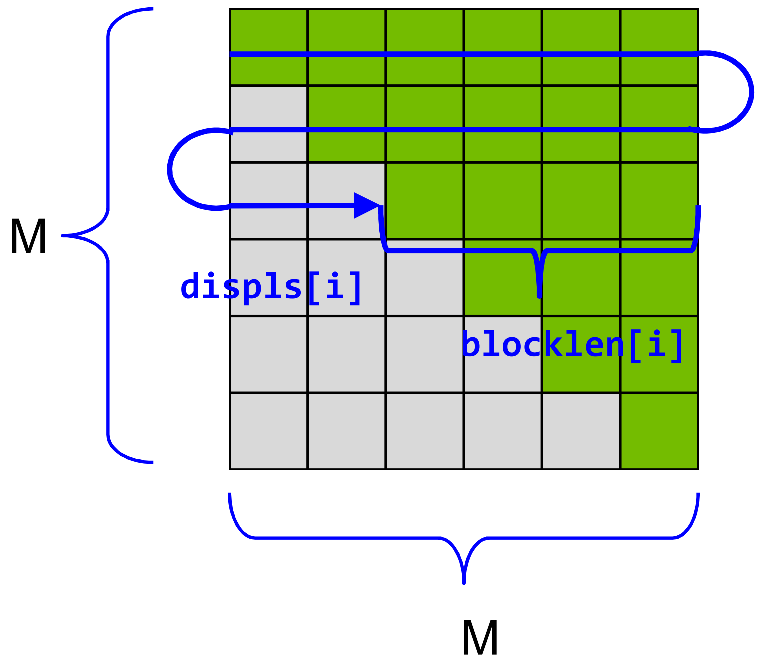
Derived Data Type: Indexed

```
int MPI_Type_indexed(int count, const int array_of_blocklengths[],
    const int array_of_displacements[], MPI_Datatype oldtype,
    MPI_Datatype *newtype)
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
        ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
    INTEGER    COUNT, ARRAY_OF_BLOCKLENGTHS(*)
    INTEGER    OLDTYPE, NEWTYPE
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
    INTEGER    IERROR
```

- **Creates a new type from blocks comprising identical elements. The size and displacements of the blocks may vary.**
  - count: Number of blocks
  - array_of_blocklengths: Number of elements per block (array of nonnegative integers).
  - array_of_displacements: Displacement for each block, in multiples of oldtype extent
  - oldtype: Old datatype (handle).
  - newtype: New datatype (handle).

MPI_Type_indexed example: Send/Recv Triangle Matrix

```
/*mpi_udt_tri.c*/
#define M=6
float a[M][M];
/*.........*/
int blocklen[M],displs[M];
for (i=0;i<M;i++) {
    blocklen[i]=M-i;
    displs[i]=M*i+i;
}
// define the index type
MPI_Datatype upper_tri;
MPI_Type_indexed(M,blocklen,displs,MPI_FLOAT,&upper_tri);
MPI_Type_commit(&upper_tri);
// send from rank 0, recv at rank 1
if (rank==0)
    MPI_Send(a,1,upper_tri,1,0,MPI_COMM_WORLD);
else //rank==1
    MPI_Recv(a,1,upper_tri,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```



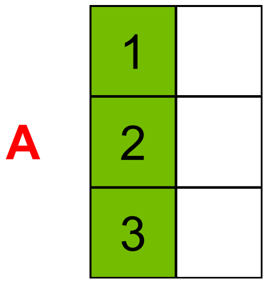
Derived Data Type: Struct

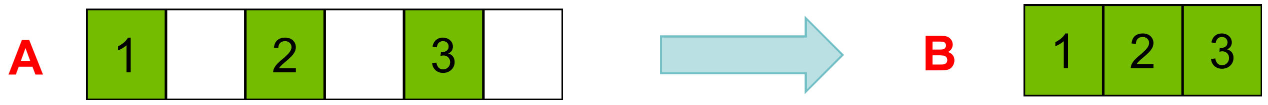
```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
    const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_types[],
    MPI_Datatype *newtype) // C/C++
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
        ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER    COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*),
    INTEGER NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

- **Creates a structured data type. Allows a new data type that represents arrays of types, with different block length, displacement and type**
  - count: Number of blocks (integer).
  - `blocklengths[]:` Number of elements in each block (array of integers).
  - `displacements[]:` Byte displacement of each block (array of integers).
  - `types[]:` Type of elements in each block (array of handles to data-type objects).
  - newtype: New data type (handle).

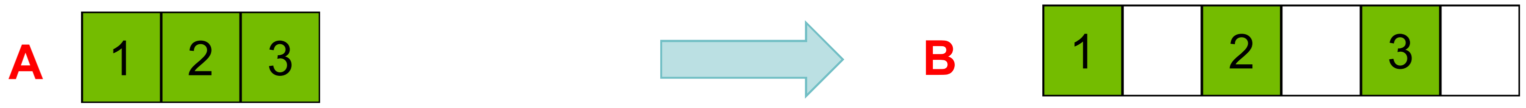
From non-contiguous to contiguous data

```
if (myrank==0) { /* mpi_vector.c */
    MPI_Type_vector( , , ,MPI_FLOAT,&newtype);
    MPI_Type_commit(&newtype);
    MPI_Send(A, , , 1,0,MPI_COMM_WORLD);
} else {
    MPI_Recv(B, , ,0,0,MPI_COMM_WORLD)
}
```



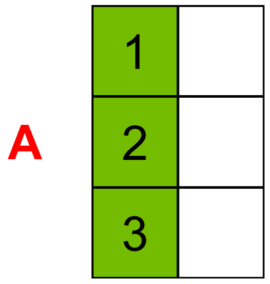


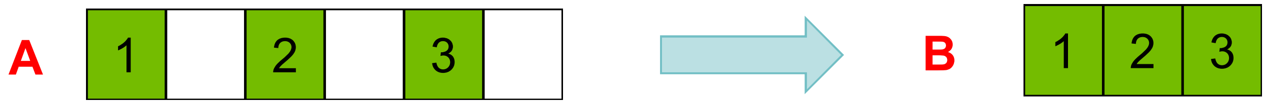
```
if (myrank==0) {
    MPI_Send(A, , , 1,0,MPI_COMM_WORLD);
} else {
    MPI_Type_vector( , , ,MPI_FLOAT,&newtype);
    MPI_Type_commit(&newtype);
    MPI_Recv(B, , , 0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
```



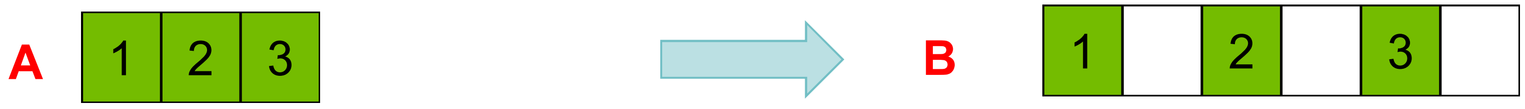
From non-contiguous to contiguous data

```
if (myrank==0) { /* mpi_vector.c */
    MPI_Type_vector(3,1,2,MPI_FLOAT,&newtype);
    MPI_Type_commit(&newtype);
    MPI_Send(A,1,newtype,1,0,MPI_COMM_WORLD);
} else {
    MPI_Recv(B,3,MPI_FLOAT,0,0,MPI_COMM_WORLD)
}
```



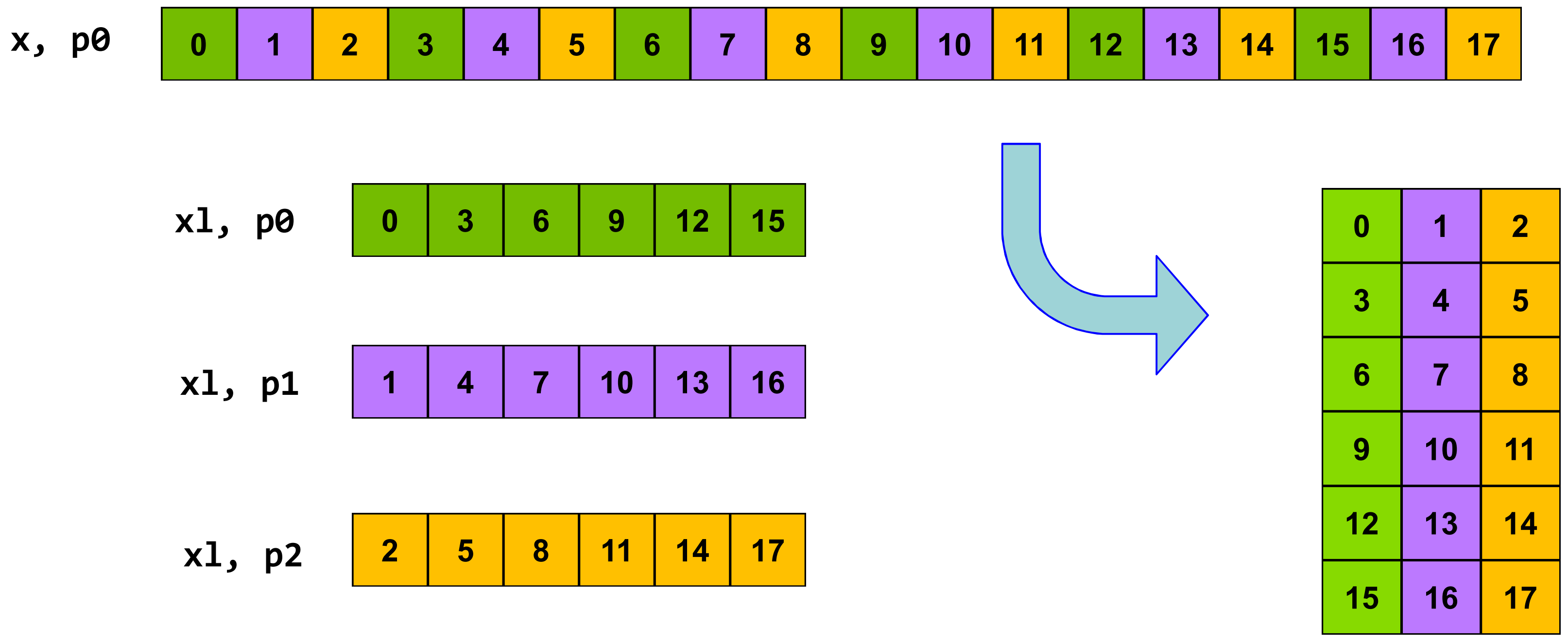


```
if (myrank==0) {
    MPI_Send(A,3,MPI_FLOAT,1,0,MPI_COMM_WORLD);
} else {
    MPI_Type_vector(3,1,2,MPI_FLOAT,&newtype);
    MPI_Type_commit(&newtype);
    MPI_Recv(B,1,newtype,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
```



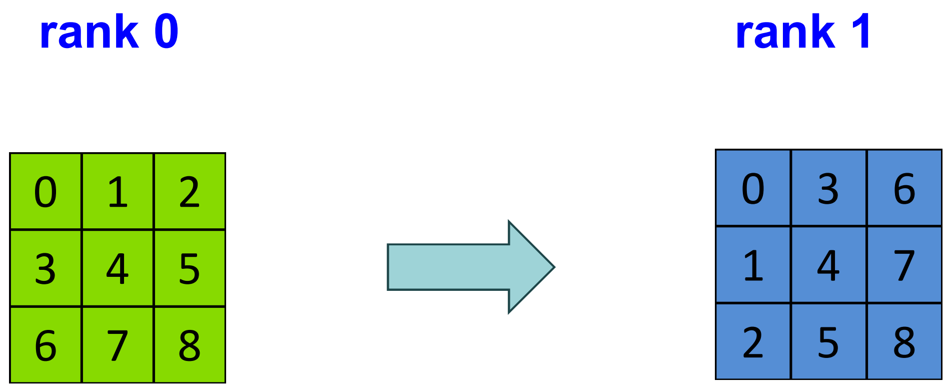
Exercise 3a Distribute 1D array

- Let processor 0 have an array x of length N*P, where P is the number of processors. Elements 0,P,2P,...,N*P should go to processor zero, 1,P + 1,2P + 1,... to processor 1, et cetera.
  - Code this as a sequence of send/recv calls, using a vector datatype for the send, and a contiguous buffer for the receive, below example N=6, P=3



From one type to another type of data? See next few slides

- **Think of the following problem:**
  - Convert a C array from row major to column major of (vice versa for Fortran array)
    - We want rank 0 sends an MxM array
    - Then rank 1 receives the MxM array in column major order

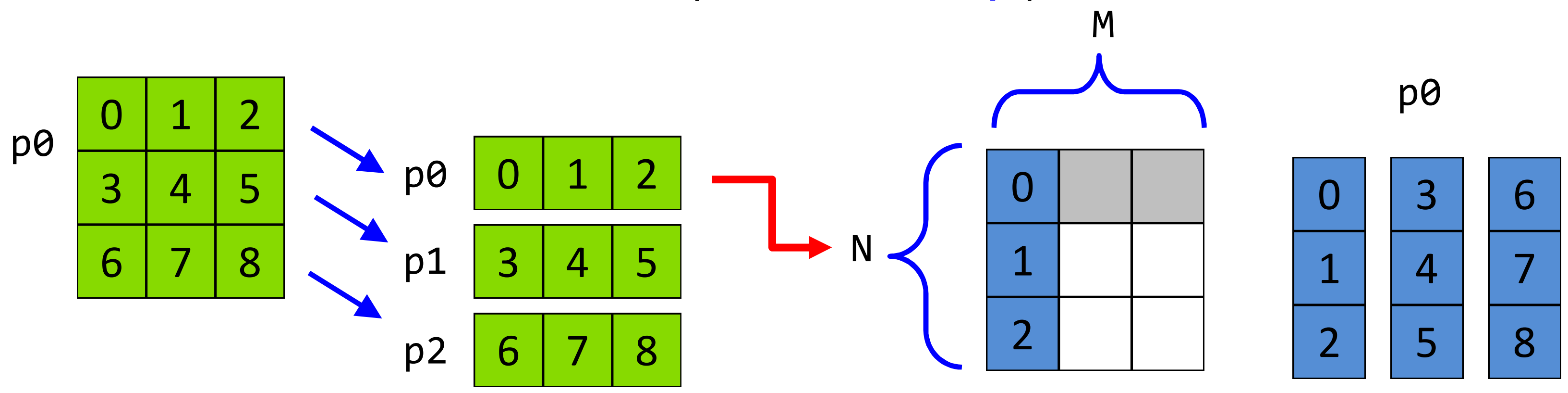


How to achieve this?

- **An intuitive solution:**
  - Root process decompose rows, send 1 row to each process, then each process send its row to root process using a user defined type, so that the root process assemble it in columns

```
MPI_Datatype myvct;
MPI_Type_vector(N,1,M,MPI_INT,&myvct);
MPI_Type_commit(&myvct);
```

  - What is the potential problem here?
    - If we want to decompose rows into ***np*** parts?



MPI_Type_create_hvector

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

- **Creates a vector (strided) data type with offset in bytes.**
- **The same with `MPI_Type_Vector`, except that the unit of the `stride` is byte instead of `old_type`**
  - More flexible than the vector type
  - We can use `MPI_Type_get_extent` to decide the extent (in bytes) of an MPI data type

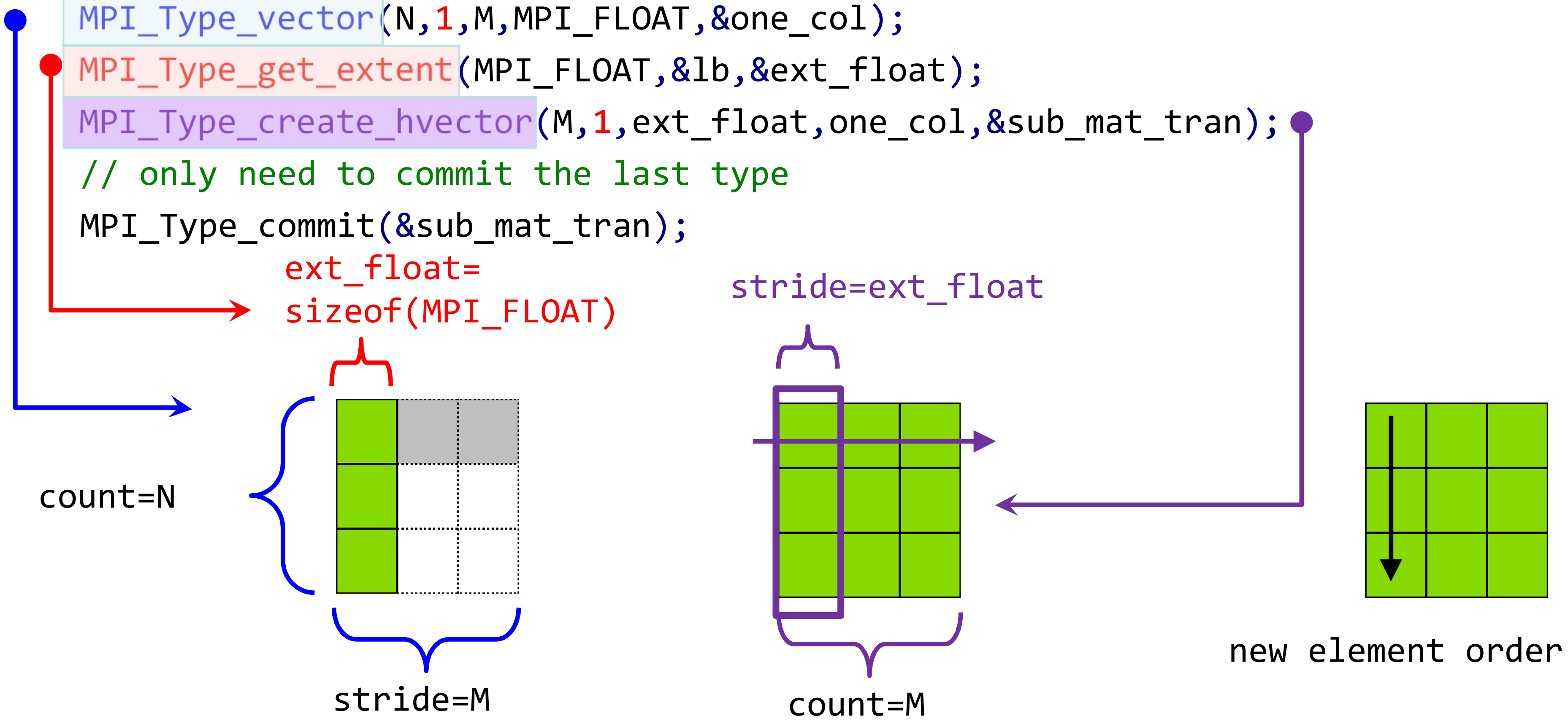
```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
            MPI_Aint *extent)
```

- **Returns the lower bound and extent of a data type.**
  - lb: Lower bound of data type (integer).
  - extent: Data type extent in bytes(integer).

Create Nested Derived Data Type

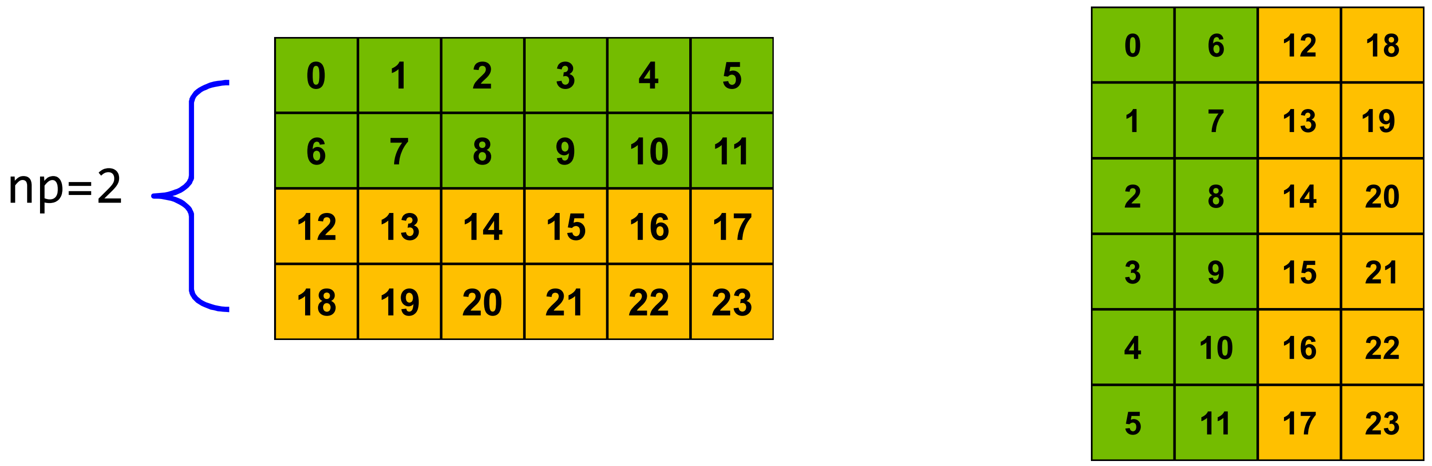
- The following code creates a column major array in C using nested user defined type:

```
MPI_Datatype one_col,sub_mat_tran;  /*mpi_mt_1blk.c*/
MPI_Aint lb,ext_float; //a data type that can hold memory addresses
MPI_Type_vector(N,1,M,MPI_FLOAT,&one_col);
MPI_Type_get_extent(MPI_FLOAT,&lb,&ext_float);
MPI_Type_create_hvector(M,1,ext_float,one_col,&sub_mat_tran);
// only need to commit the last type
MPI_Type_commit(&sub_mat_tran);
```



Exercise 3b Matrix Transposition

- **Goal: write a MPI program that transposes a MxN (*M≠N*) matrix in parallel**
  1. Rank 0 sends the MxN matrix, rank 1 receives in transposed order by a user defined type. (hint: refer to the slides change from row major to column major by defining a user defined type with `MPI_Type_hvector`)
  2. Decompose the MxN matrix by np rows in C (or columns in Fortran), each process send result back to root process using a user defined type, for simplicity, M is a multiple of np.

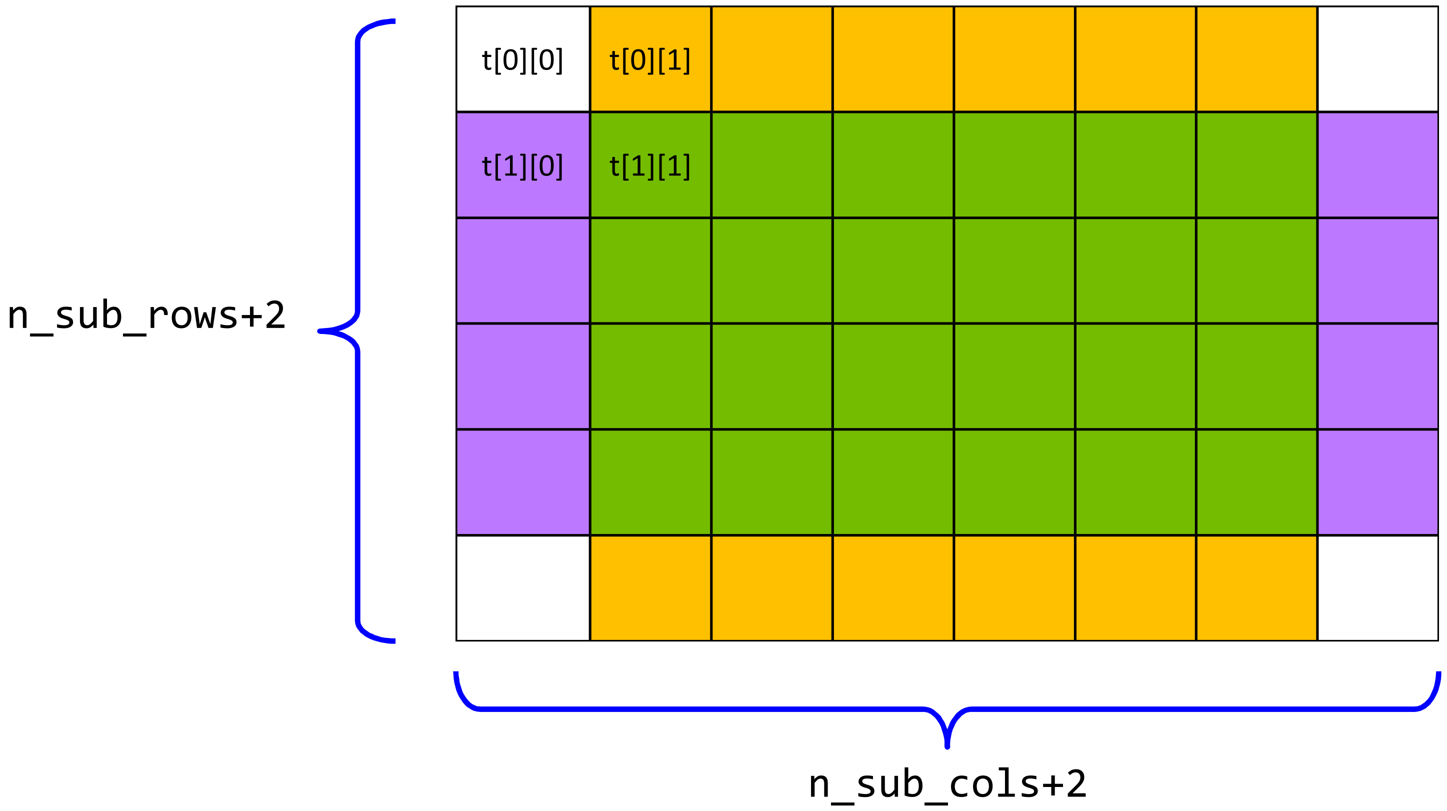


0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

0	6	12	18
1	7	13	19
2	8	14	20
3	9	15	21
4	10	16	22
5	11	17	23

Exercise 3c: Laplace Solver Version 2

- **Goal: Modify the Laplace solver in a two-dimensional decomposition**
  - A template of the 2D Laplacian solver has been provided, change the boundary exchange portion of the code with user-defined type.

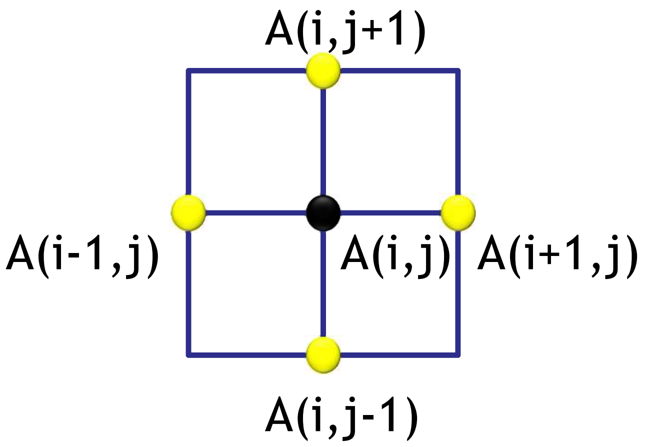


Laplace solver Jacobi Iteration

- **Solve Laplace equation in 2D:**
  - Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

$$\nabla^2 f(x,y) = 0$$

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$



Graphical representation for Jacobi iteration

Array: told

	1.0	1.0	1.0	1.0	1.0	1.0	
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	

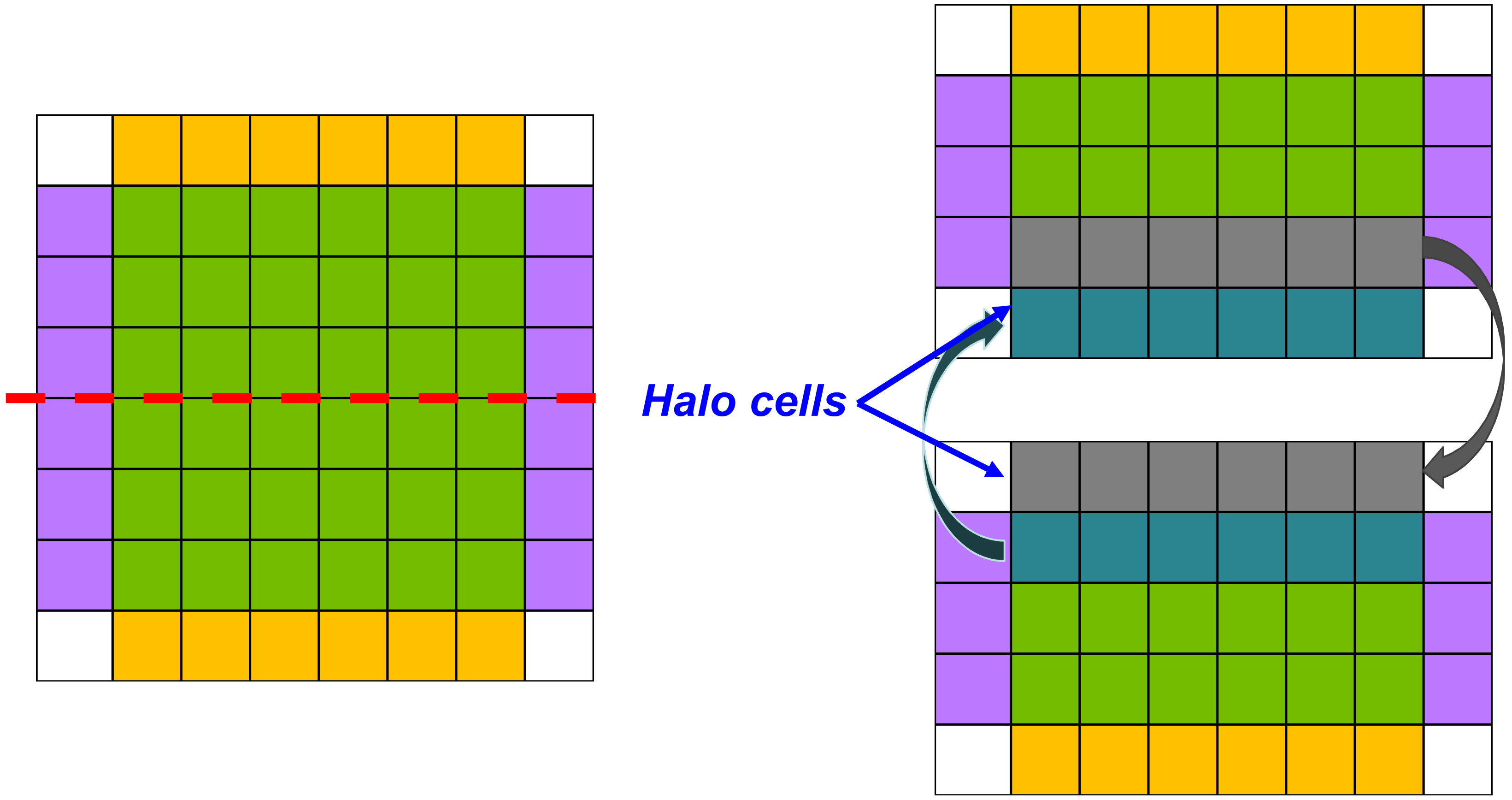
$$t_{1,1} = 0.25 \times (1.0 + 1.0 + 0.0 + 0.0) = 0.5$$

$$dt = \max(t_{i,j} - told_{i,j}) = 0.5 - 0 = 0.5$$

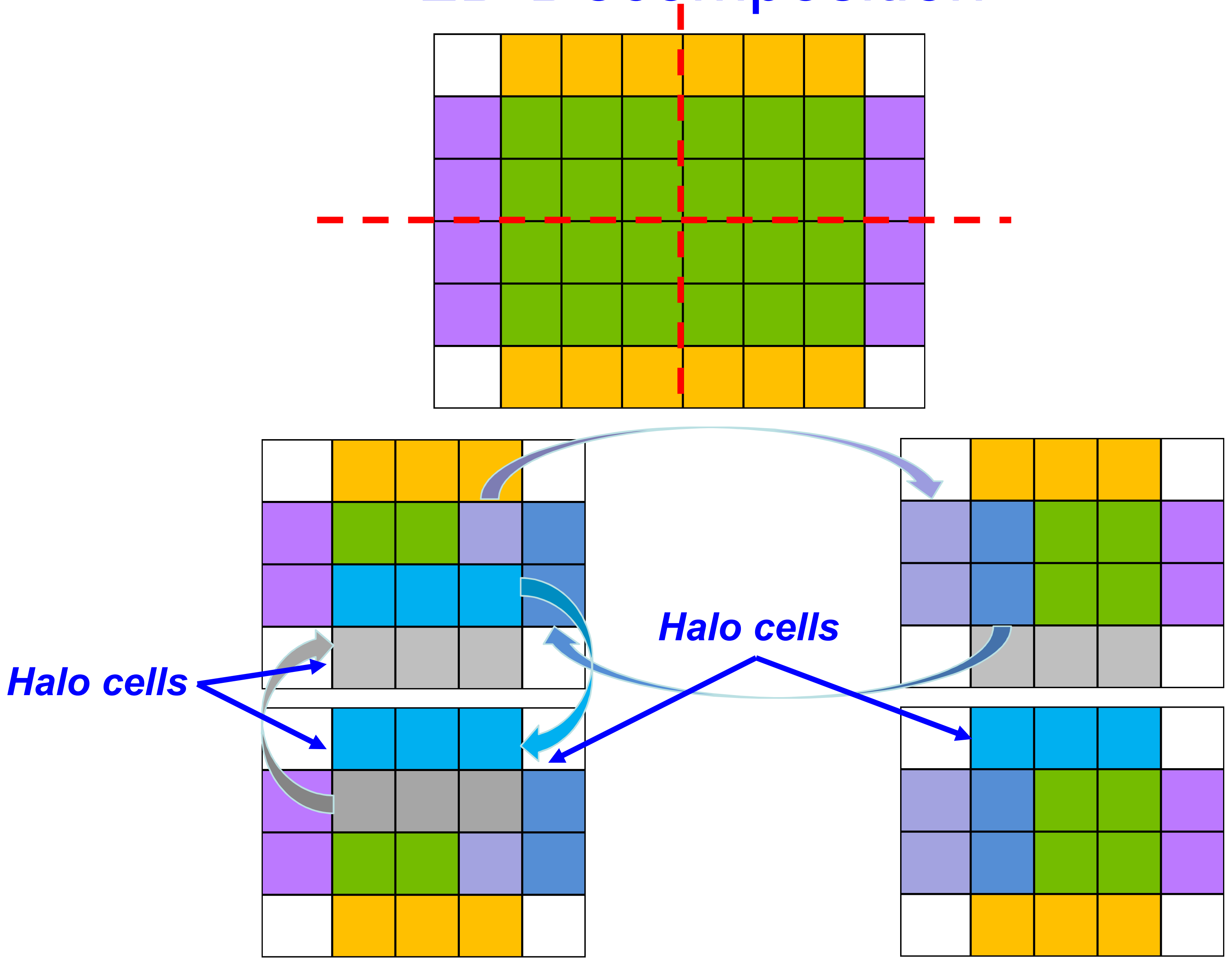
Array: t (tnew)

	1.0	1.0	1.0	1.0	1.0	1.0	
1.0	0.5	0.375	0.344	0.336	0.334	0.333	0.0
1.0	0.375	0.188	0.133	0.117	0.113	0.112	0.0
1.0	0.344	0.133	0.066	0.046	0.04	0.038	0.0
1.0	0.336	0.117	0.046	0.023	0.016	0.013	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	

Laplace solver 1D Decomposition



2D Decomposition



Introduction to MPI, Part 2

Collective Communication

Collective Communication

- **Collective communications involve all processes in a communicator**
  - One to all, all to one and all to all
- **Three types of collective communications**
  - Data movement
  - Collective computation
  - Synchronization

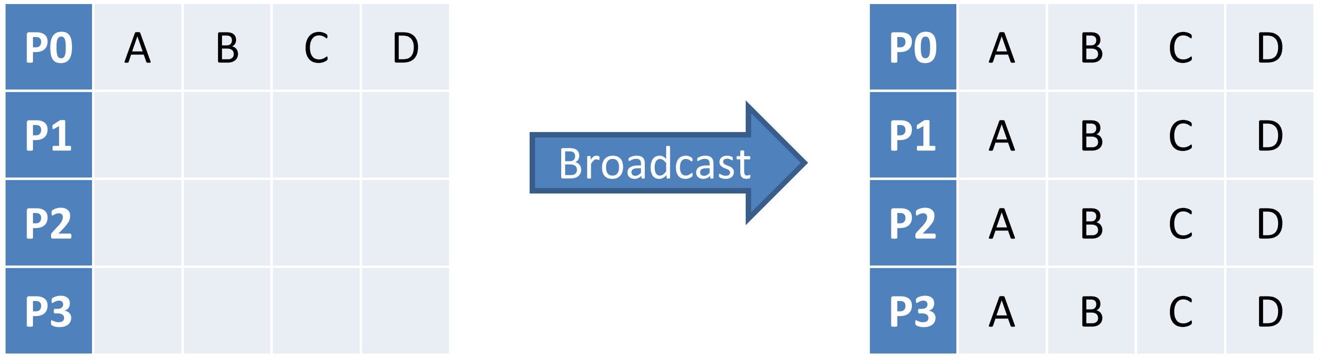
Collective vs. Point-to-point

- ❑ **More concise program**
  - ➢ One collective operation can replace multiple point-to-point operations
- ❑ **Optimized collective communications usually are faster than the corresponding point-to-point communications**

Data Broadcast: Data Movement

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm) // C/C++
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type>    BUFFER(*)
    INTEGER    COUNT, DATATYPE, ROOT, COMM, IERROR ! Fortran
```

- **Broadcast copies data from the memory of one processor to all processes, itself included**
  - One to all operation



Collective Operations: Data Broadcast

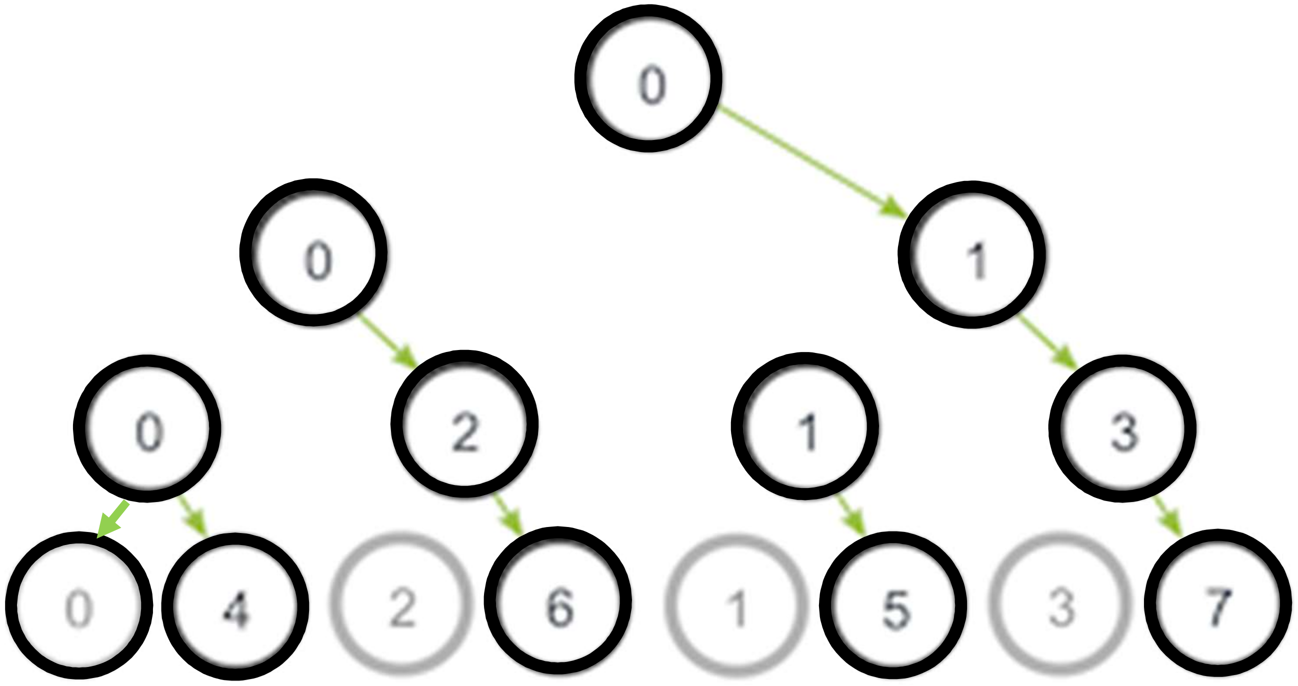
```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm) // C/C++
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) ! Fortran
```

- **MPI broadcast Parameters:**
  - buffer: data to be sent at root; place to put the data in all other ranks
  - count: number of data elements
  - datatype: elements' datatype
  - root: source rank; all ranks must specify the same value
  - comm: communication context
- **Notes:**
  - in all ranks, data is an output argument
  - in rank root, data is also an input argument
  - MPI_Bcast completes only after all ranks in comm have made the call

An Naive Implementation of Broadcast using Point to Point Communication

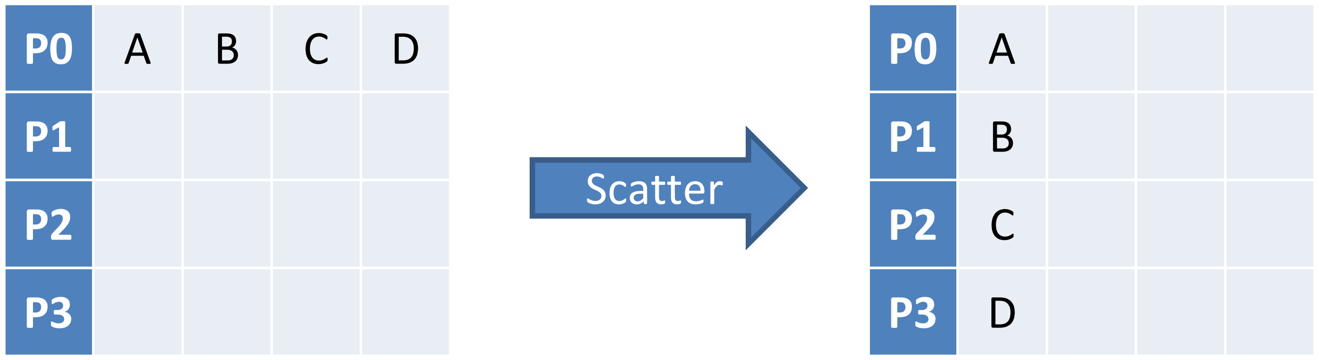
```c
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(...);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(...);
  }
}
```

Better Solution: tree-based hierarchical communication, O(log(#ranks))



Scatter: Data Movement

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm) // C/C++
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR) ! Fortran
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT
    INTEGER    COMM, IERROR
```



Collective Operations: Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm) // C/C++
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
    RECVTYPE, ROOT, COMM, IERROR) ! Fortran
```

- **Scatter takes an array of elements and distributes the elements in the order of process rank**
  - One to all operation
- **MPI scatter Parameters:**
  - sendbuf: data to be distributed
  - sendcount: size of each chunk in data elements
  - sendtype: source datatype
  - recvbuf: buffer for data reception
  - recvcount: number of elements to receive
  - recvtype: receive datatype
  - root: source rank
  - comm: communication context

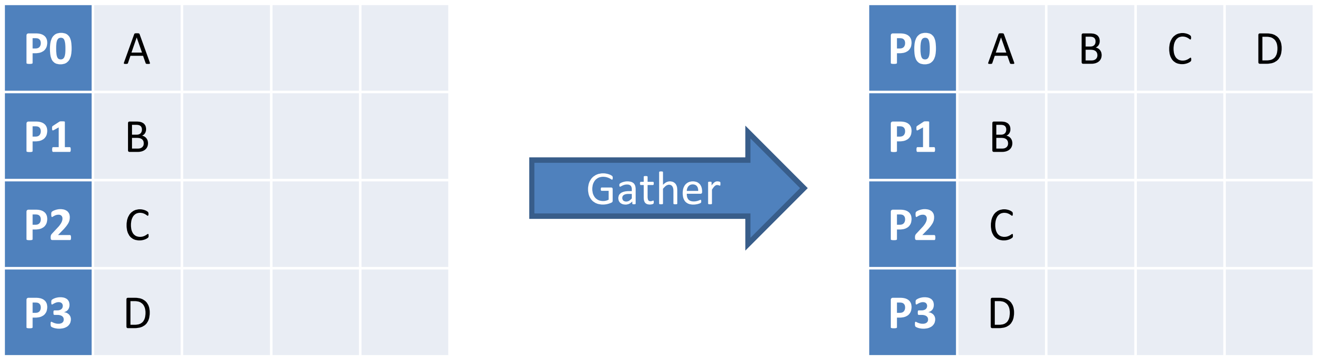
Collective Operations: Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm) // C/C++
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
    RECVTYPE, ROOT, COMM, IERROR) ! Fortran
```

- **Notes:**
  - sendbuf must be large enough in order to supply sendcount elements of data to each rank in the communicator
  - data chunks are taken in increasing order of receiver's rank
  - root also sends one data chunk to itself
  - for each chunk the amount of data sent must match the receive size, i.e. if sendtype == recvtype holds, then sendcount == recvcount must hold too

Gather: Data Movement

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm) // C/C++
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR) ! Fortran
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT
    INTEGER    COMM, IERROR
```



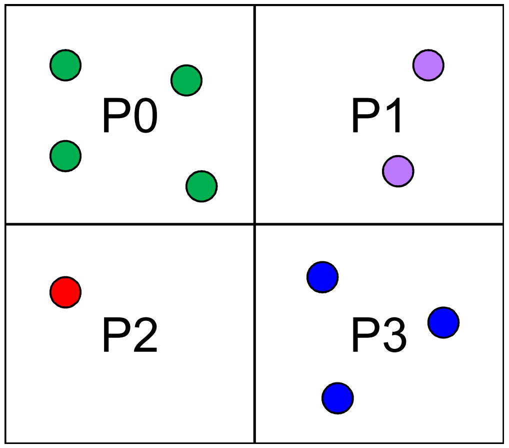
Collective Operations: Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm) // C/C++
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR) ! Fortran
```

- **The opposite operation of MPI_Scatter:**
  - recvbuf must be large enough to hold recvcount
  - elements from each rank
  - root also receives one data chunk from itself
  - data chunks are stored in increasing order of receiver's rank
  - for each chunk the receive size must match the amount of data sent

Varying message collectives

- **MPI_Gatherv and MPI_Scatterv are the variable-message-size versions of MPI_Gather and MPI_Scatter which permit a varying count of data from each process, and to allow some flexibility in where the gathered data is placed on the root process.**
- **The "v" variants**
  - MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv, MPI_Alltoallv
  - What does the "v" stand for?
    - varying – sizes, relative locations of messages
- **We will discuss the usage of these functions later via examples.**
- **Typical scenario:**

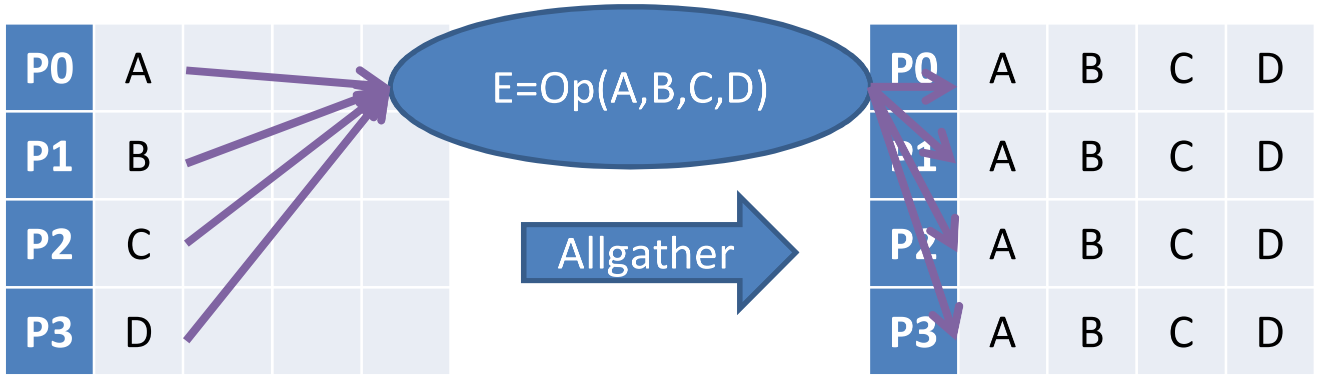


Collective Operations: Gather to All

```
int MPI_Allgather(const void *sendbuf, int  sendcount,
     MPI_Datatype sendtype, void *recvbuf, int recvcount,
     MPI_Datatype recvtype, MPI_Comm comm) // C/C++
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, COMM, IERROR) ! Fortran
```

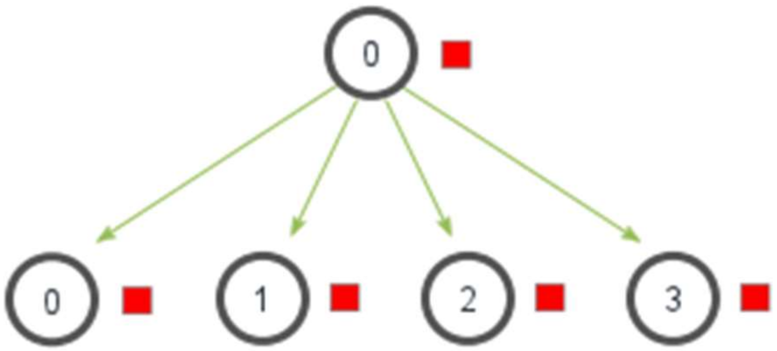
❑ **Note:**

- **rootless operation – all ranks receive a copy of the gathered data**
- each rank also receives one data chunk from itself
- data chunks are stored in increasing order of sender’s rank
- for each chunk the receive size must match the amount of data sent
- equivalent to MPI_Gather + MPI_Bcast, but possibly more efficient

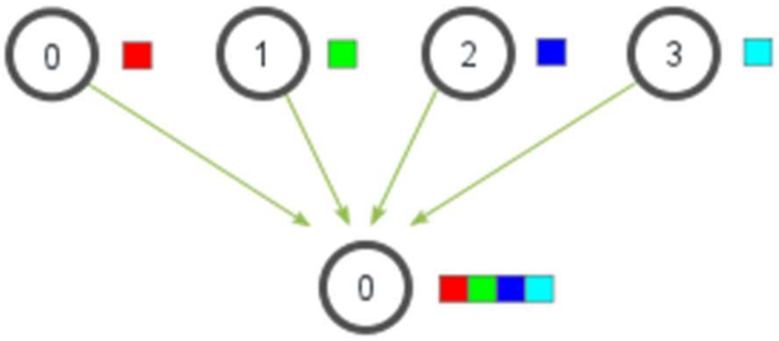


Difference between Bcast, Scatter and Gather

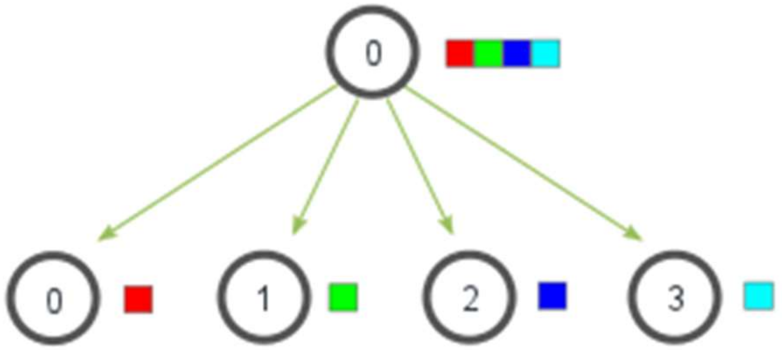
MPI_Bcast



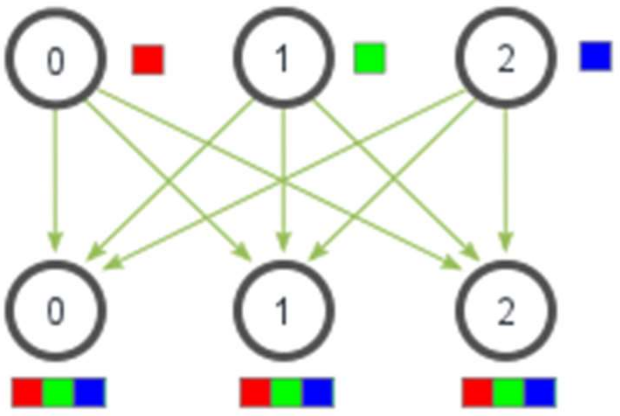
MPI_Gather



MPI_Scatter



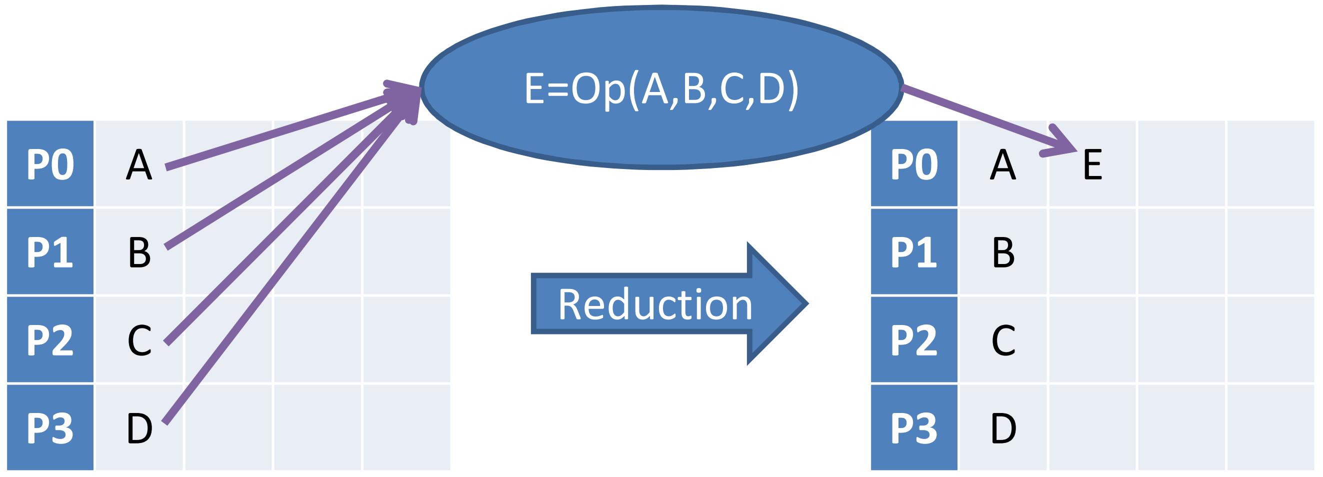
MPI_Allgather



MPI_Scatter/Gather Example: Average of random number array

```c
// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);
// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);
// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);
// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
}
```

Collective Computation: Reduction



- MPI reduction collects data from each process, reduces them to a single value, and store it in the memory of one process
  - All to one operation
- **Syntax:**

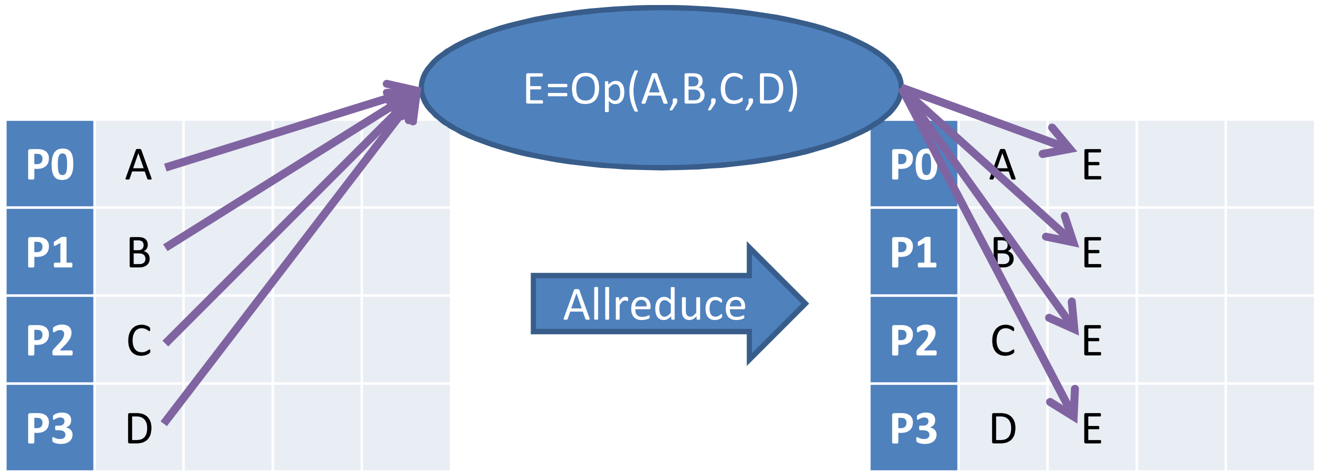
```
// C/C++
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
! Fortran
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Reduction Operation

- Summation and production
- Maximum and minimum
- Max and min location
- Logical
- Bitwise
- User defined

`MPI_MAX`	Returns the maximum element.
`MPI_MIN`	Returns the minimum element.
`MPI_SUM`	Sums the elements.
`MPI_PROD`	Multiplies all elements.
`MPI_LAND`	Performs a logical and across the elements.
`MPI_LOR`	Performs a logical or across the elements.
`MPI_BAND`	Performs a bitwise and across the bits of the elements.
`MPI_BOR`	Performs a bitwise or across the bits of the elements.
`MPI_MAXLOC`	Returns the maximum value and the rank of the process that owns it.
`MPI_MINLOC`	Returns the minimum value and the rank of the process that owns it.

Collective Computation: Allreduce

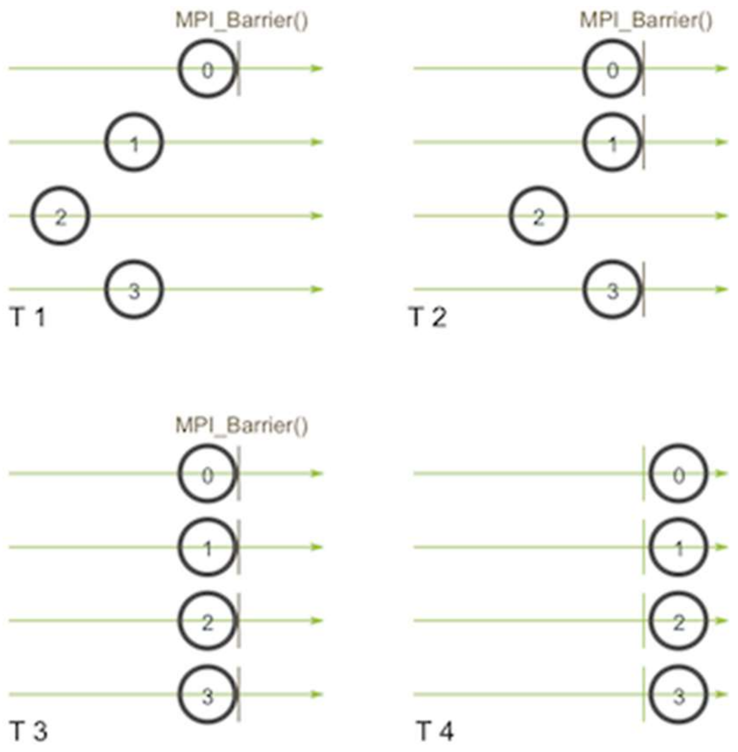


- **MPI_Allreduce collects data from each process, reduces them to a single value, and store it in the memory of EVERY process**
    - All to all operation
- **Syntax:**

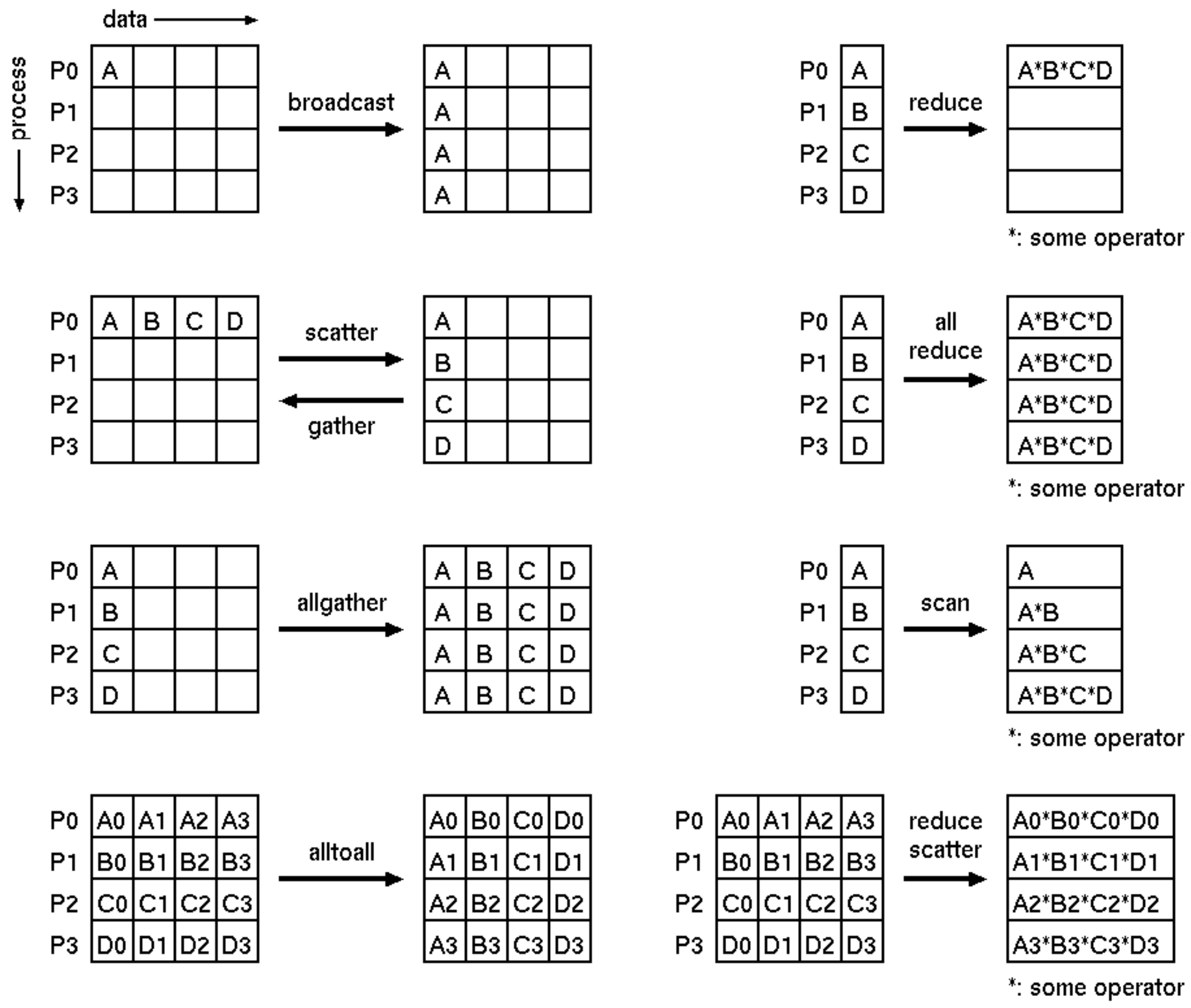
```
// C/C++
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
! Fortran
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    COUNT, DATATYPE, OP, COMM, IERROR
```

Synchronization

- **MPI_Barrier (Communicator)**
  - Blocks processes in a group until all processes have reached the same synchronization point
  - Synchronization is collective since all processes are involved
  - Could cause significant overhead, so do *NOT* use it unless absolutely necessary



Other Collective Communications



Source: Practical MPI Programming, IBM Redbook

Collective Operations: Caveats

- **All ranks in the communicator must call the MPI collective operation for it to complete successfully:**
  - both data sources (root) and data receivers have to make the same call

```
if (rank==0)
    MPI_Bcast(sendbuf,...);
```

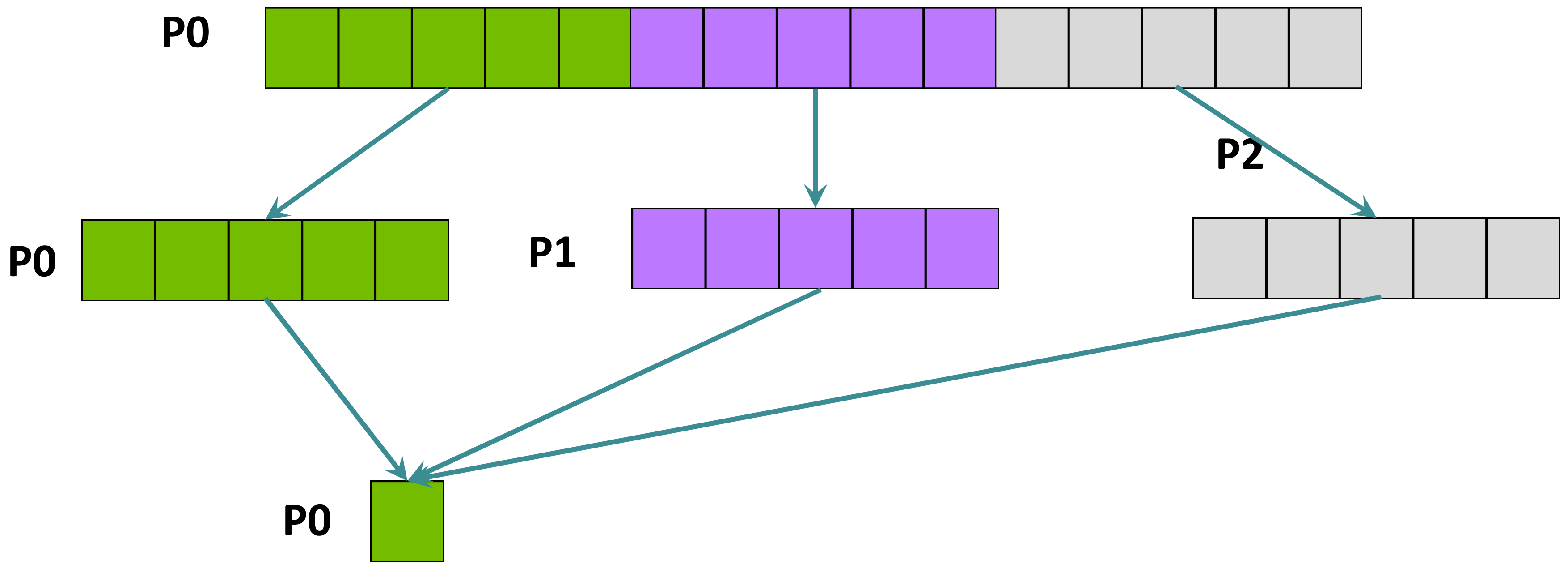
X

  - observe the significance of each argument
- **Multiple collective operations have to be called in the same sequence by all ranks**

- **One cannot use MPI_Recv to receive data sent by MPI_Scatter**
- **One cannot use MPI_Send to send data to MPI_Gather**

Exercise 4a: Find Global Maximum

- Goal: Scatter an array to each process from root rank, find global maximum with appropriate collective communication function(s)

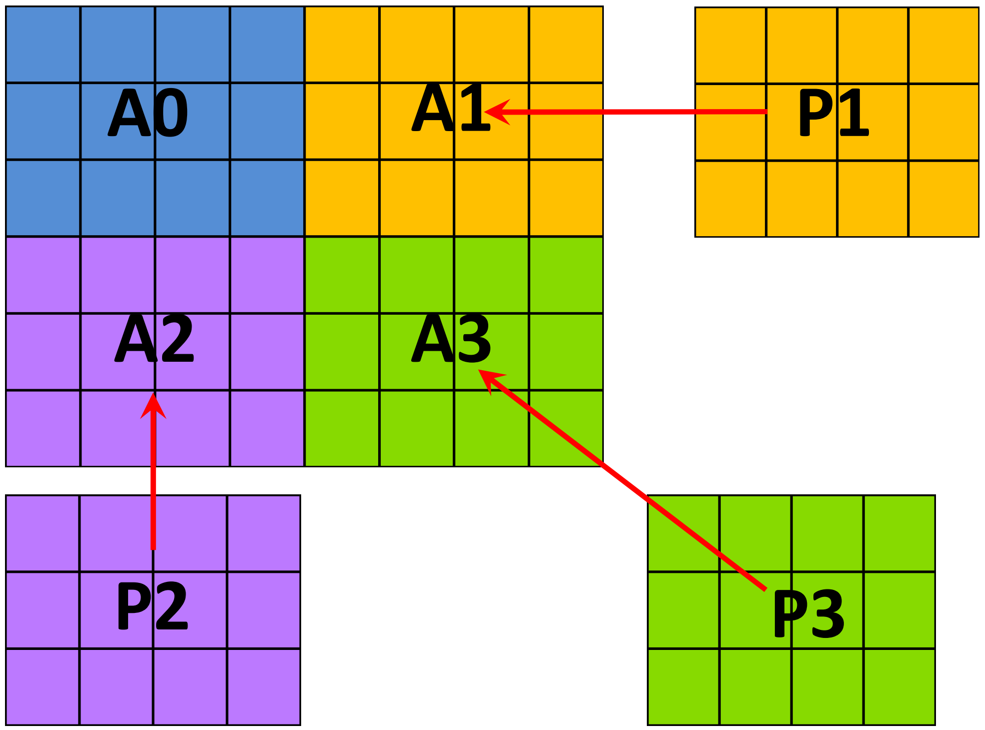


Exercise 4b: Laplace Solver version 2

- **Goal: Replace the part in version 1 that finds the global maximum convergence and distributes it to all processes with appropriate collective operation(s)**

Scatter/Gather 2D Matrix

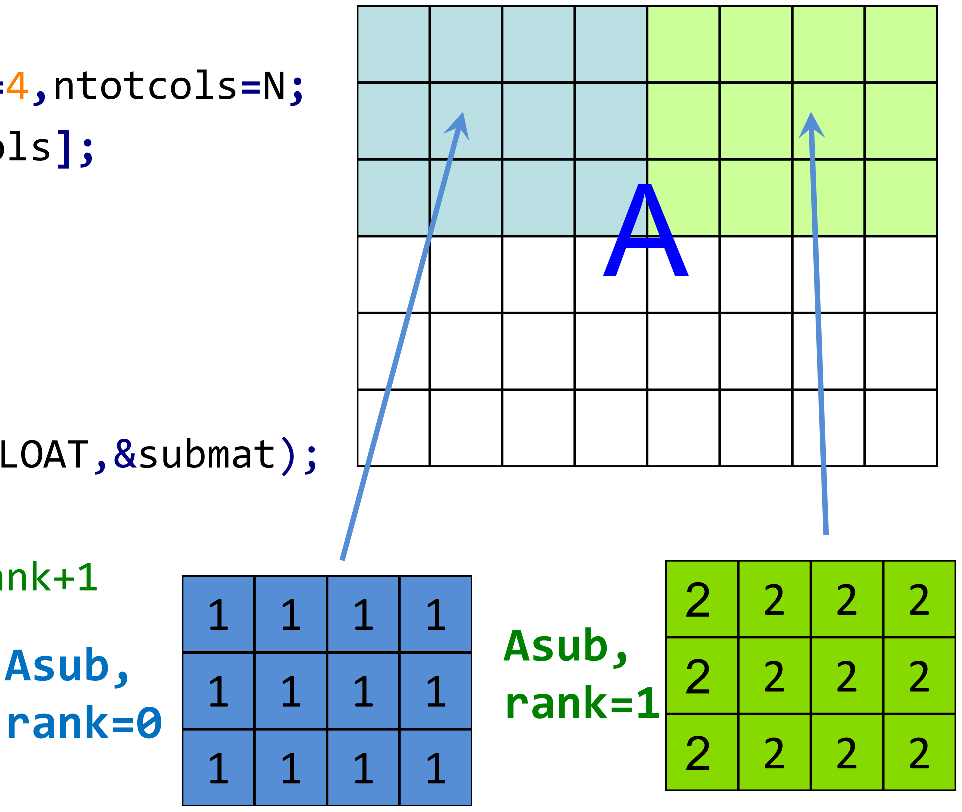
- It's relatively easy to interpret what will happen for 1D arrays' Scatter/Gather
- For 2D matrix assembly using collective operations need more preparation
- The data layout from the sender's point of view is different from the data layout from the receivers.



Attempt using MPI_Gather

- Consider the following code segments using Gather from 2 MPI processes, what will be the output if we print A?

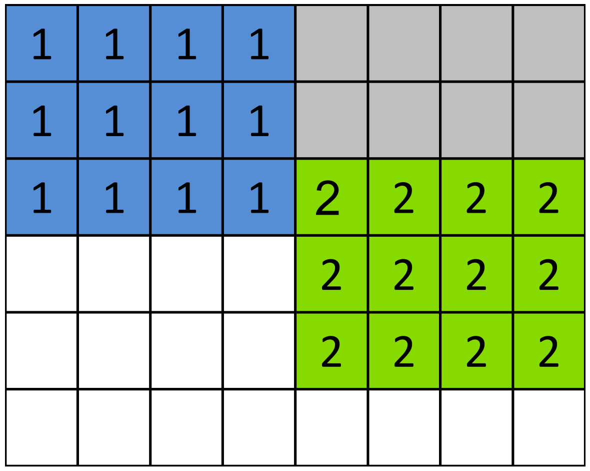
```
/* mpi_gather_2d.c */
int M=6,N=8,i,j,nrows=3,ncols=4,ntotcols=N;
float A[M][N], Asub[nrows][ncols];
/*assign each A to 0*/
/*define a submatrix type*/
MPI_Datatype submat;
MPI_Type_vector(nrows,ncols,
                ntotcols,MPI_FLOAT,&submat);
MPI_Type_commit(&submat);
//assign each Asub with the rank+1
for (i=0;i<nrows;i++)
    for (j=0;j<ncols;j++)
        Asub[i][j]=rank+1;

MPI_Gather(&(Asub[0][0]),nrows*ncols,MPI_FLOAT,
           &(A[0][0]),   1,submat   ,root     ,MPI_COMM_WORLD);
```



Results using MPI_Gather

- ❑ **Results can be explained using the memory layout figure (next slide).**

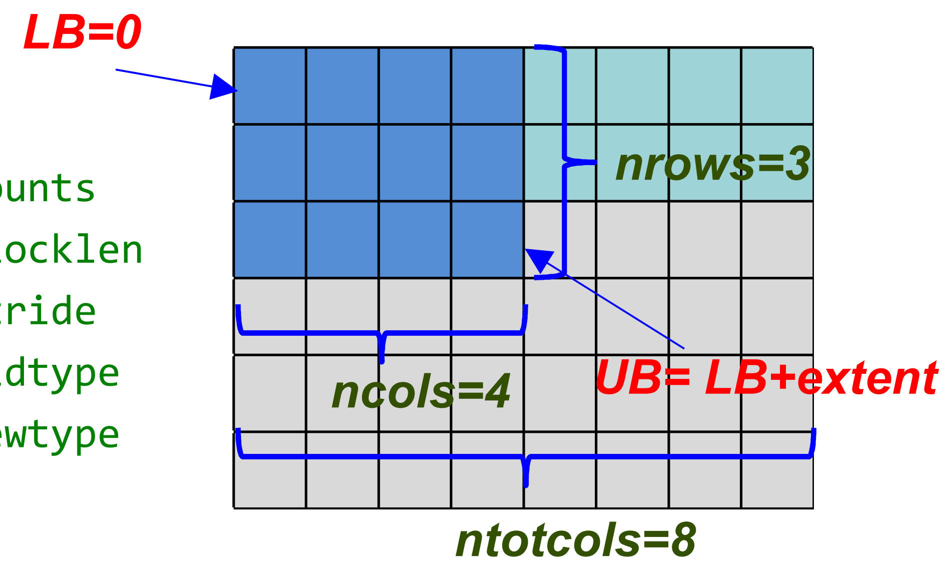
```
[fchen14@shelob001 mpitutorial]$ mpirun -np 2 ./a.out
1   1   1   1   0   0   0   0
1   1   1   1   0   0   0   0
1   1   1   1   2   2   2   2
0   0   0   0   2   2   2   2
0   0   0   0   2   2   2   2
0   0   0   0   0   0   0   0
```



Memory Layout of a MPI_Type_vector

- Consider the sub_matrix data type created by the following code segments (in C/C++)

```
MPI_Datatype sub_matrix;
MPI_Type_vector(nrows,        //counts
                ncols,        //blocklen
                ntotcols,     //stride
                MPI_FLOAT,    //oldtype
                &sub_matrix); //newtype
MPI_Type_commit(&sub_matrix);
```



- The above code segments actually defines a memory layout with a start (lower bound, *LB*) and end mark (upper bound, *UB*). They determine where the next instance could start.
- In the above example:
  - LB=0
  - UB=0+extent=0+((nrows-1)*ntotcols+ncols)*sizeof(MPI_FLOAT)
    =(2*8+4)*4=80

Steps to Assemble 2D array Using Collective Operations

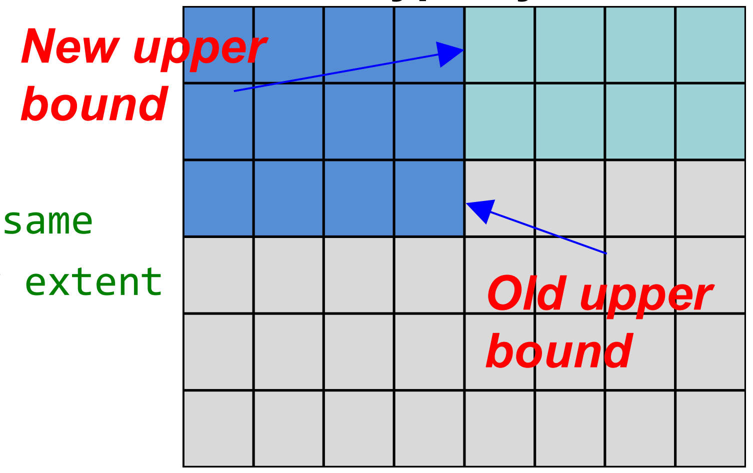
- **In order to gather data using collective operations, we need to**
  1. Change the upper bound location of the user defined type
     - `MPI_Type_create_resized`
  2. Manually specify the location of data from each processor
     - `Gatherv`

1. Create resized data type

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

- **Create a datatype with a new lower bound and extent from an existing datatype**
  - `oldtype:` input datatype (handle)
  - `lb:` new lower bound of datatype (address integer)
  - `extent:` new extent of datatype (address integer)
- **Use the following to change the extent of the user defined type by creating a resized datatype**

```
MPI_Datatype rs_submat;
MPI_Type_create_resized(submat,
        0,                    //lower bound, same
        ncols*sizeof(float),  //change to new extent
        &rs_submat);          //new type name
MPI_Type_commit(&rs_submat);
```



2. Collective Operations: Gatherv

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, const int recvcounts[], const int displs[],
    MPI_Datatype recvtype, int root, MPI_Comm comm) //C
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
        DISPLS, RECVTYPE, ROOT, COMM, IERROR)
    <type>    SENDBUF(*), RECVBUF(*)
    INTEGER    SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*)
    INTEGER    RECVTYPE, ROOT, COMM, IERROR
```

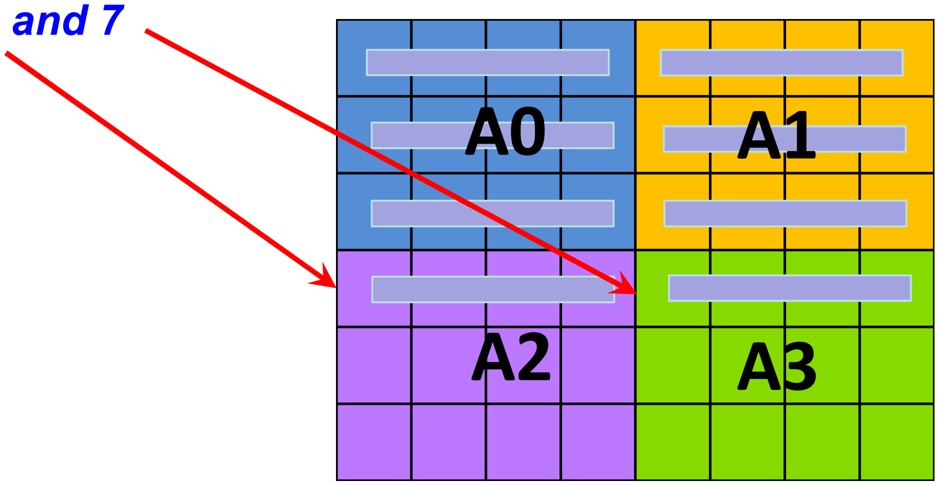
- **Gatherv gathers varying amounts of data from all processes to the root process**
- **Additional MPI Scatterv Parameters compared to MPI Gather:**
  - `recvcounts[]`: Integer array (of length group size) containing the number of elements that are received from each process (significant only at root).
  - `displs[]`: Integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root).

2. Use Gatherv to manually specify location

- **Using the following code to specify the counts/displacements for each sub-matrix:**

```
int recv_counts[]={1,1}; /* mpi_gatherv_2d.c*/
int recv_displs[]={0,1};
MPI_Gatherv(&(Asub[0][0]),nrows*ncols,MPI_FLOAT,
        &(A[0][0]),recv_counts,recv_displs,rs_submat,root,MPI_COMM_WORLD);
```

- **For complete assemble of the 2D matrix, need to calculate the displacement of each sub-matrix according to their ranks and expected locations in the global matrix.** `/* mpi_gatherv_2d4p.c*/`
  - What should be the displacements of sub-matrix A2 and A3?
    - Answer: ***6 and 7***



Results using Gatherv/resized type

```
[fchen14@shelob001 mpitutorial]$ mpirun -np 2 ./a.out

Using MPI_Gather:
1   1   1   1   0   0   0   0
1   1   1   1   0   0   0   0
1   1   1   1   2   2   2   2
0   0   0   0   2   2   2   2
0   0   0   0   2   2   2   2
0   0   0   0   0   0   0   0

Using MPI_Gatherv:
1   1   1   1   2   2   2   2
1   1   1   1   2   2   2   2
1   1   1   1   2   2   2   2
0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0
```

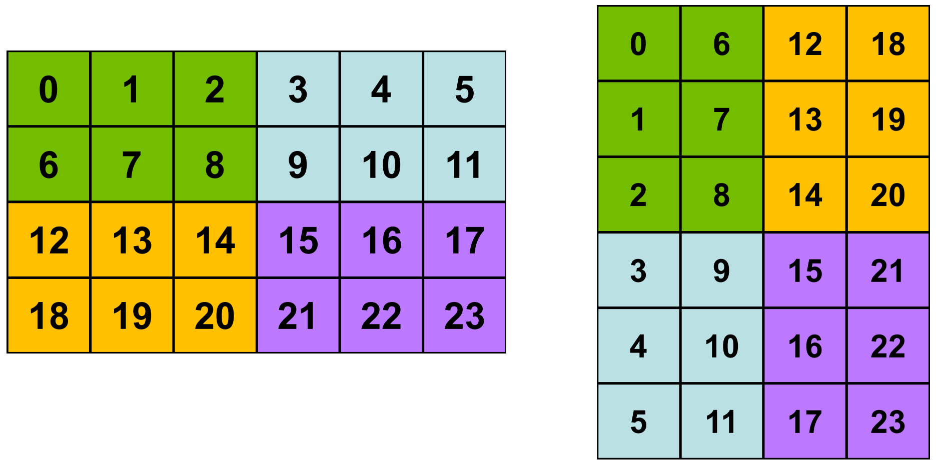
MPI_Scatterv is the reverse of MPI_Gatherv

```
int MPI_Scatterv(const void *sendbuf, const int sendcounts[], const int displs[],
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm) //C/C++
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
        RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
    <type>     SENDBUF(*), RECVBUF(*)
    INTEGER     SENDCOUNTS(*), DISPLS(*), SENDTYPE
    INTEGER     RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR ! Fortran
```

- **Scatterv scatters a buffer in parts to all processes in a communicator according to designated locations**
- **Additional MPI Scatterv Parameters compared to MPI Scatter:**
  - `sendcounts[]`: integer array (of length group size) specifying the number of elements to send to each processorsendtype: source datatype
  - `displs[]`: Integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i.

Exercise 4c: Matrix Transposition

- **Goal: write a MPI program that transposes a matrix in parallel**
  - Scatter/Scatterv a global matrix to each process on a 1D/2D process grid
  - Transpose each sub-matrix and then use Gather/Gatherv to root process
    - Two possible solutions:
      - Each process transpose its sub-matrix locally and then assemble
      - Directly send sub-matrix to root process by changing the ordering of the elements using user defined type
    - Use Gather/Gatherv to assemble the global transposed matrix to root.

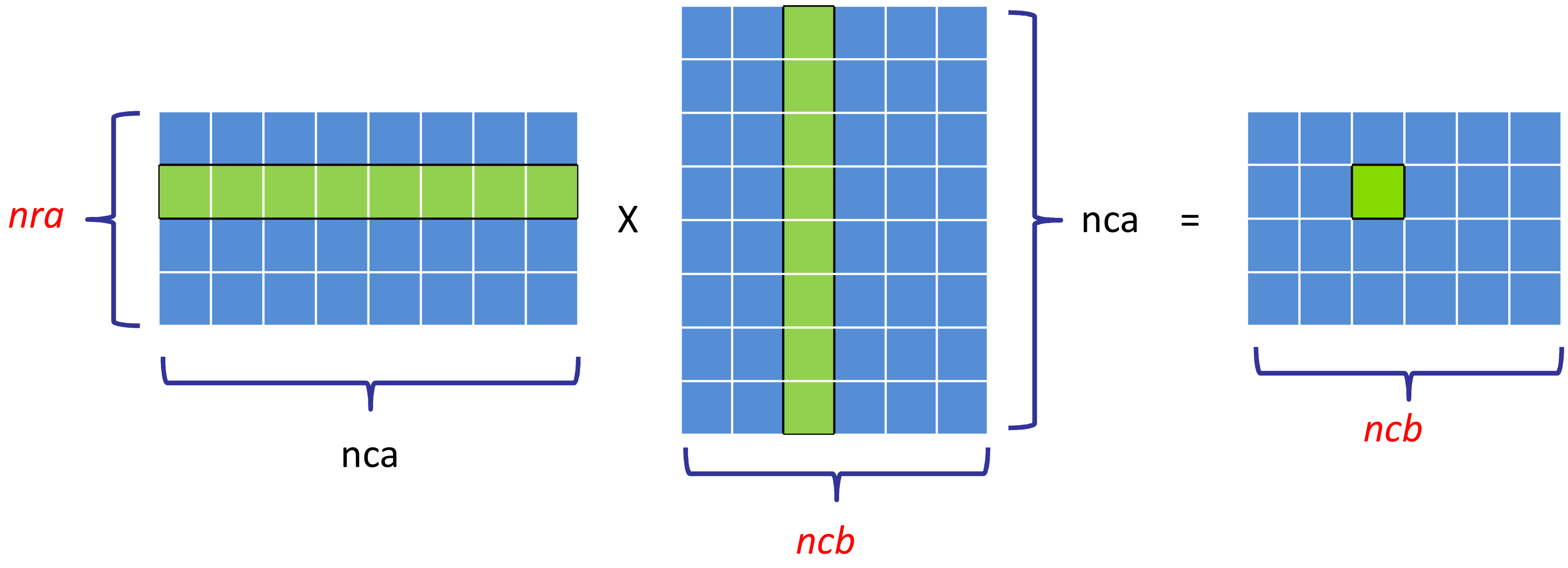


0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

0	6	12	18
1	7	13	19
2	8	14	20
3	9	15	21
4	10	16	22
5	11	17	23

Exercise 4d: Matrix Multiplication v1

- **Goal: Replace the part in version 2 that sends the result to the root process with appropriate collective operation(s)**



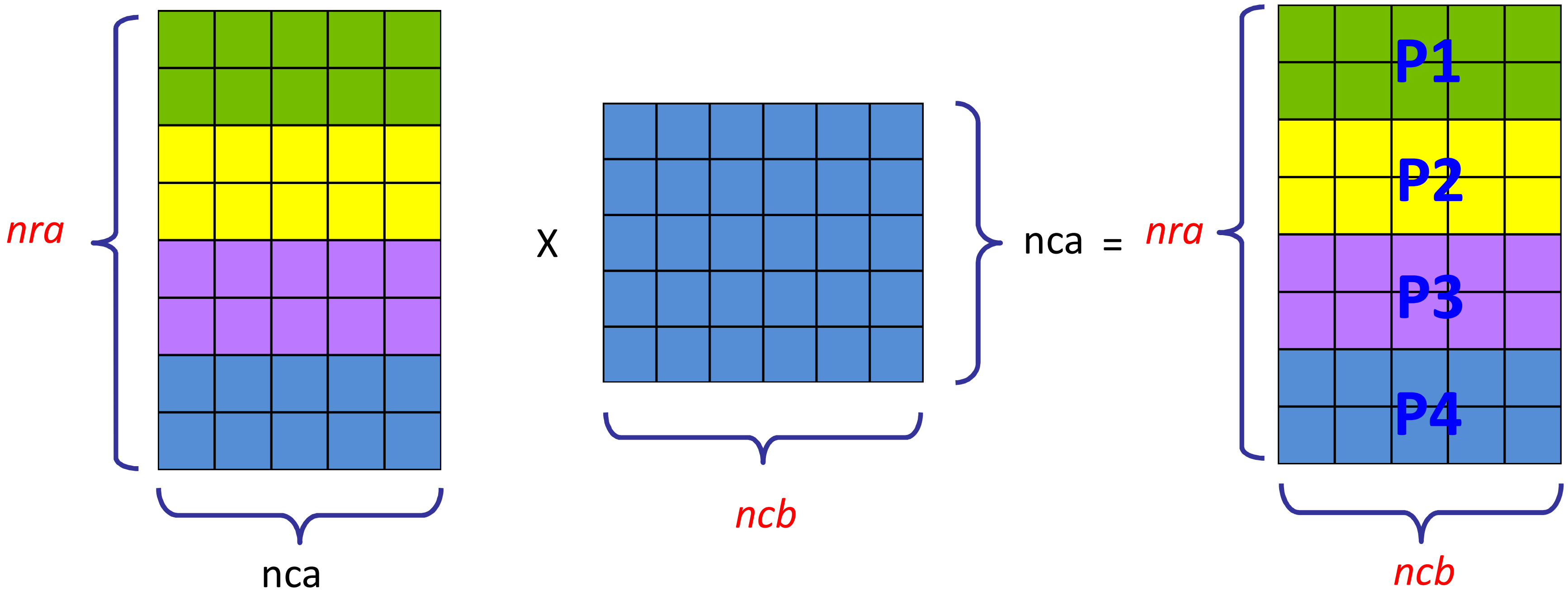
$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \cdot b_{k,j}$$

Pseudo Serial Version of Matrix Multiplication

```
//Read and validate command line arguments
Define dimensions of A, B

//Initialize the arrays
For all elements of A and B
    initial value = function( i , j )

// Matrix multiplication
For each C [i][j]
// Take the inner product of row i of A and column j of B
    For each A[i][k] and B[k][j]
        C[i][j] = C[i][j] + A[i][k] * B[k][j]
//Validate the result
Print out an element of C and validate results
```

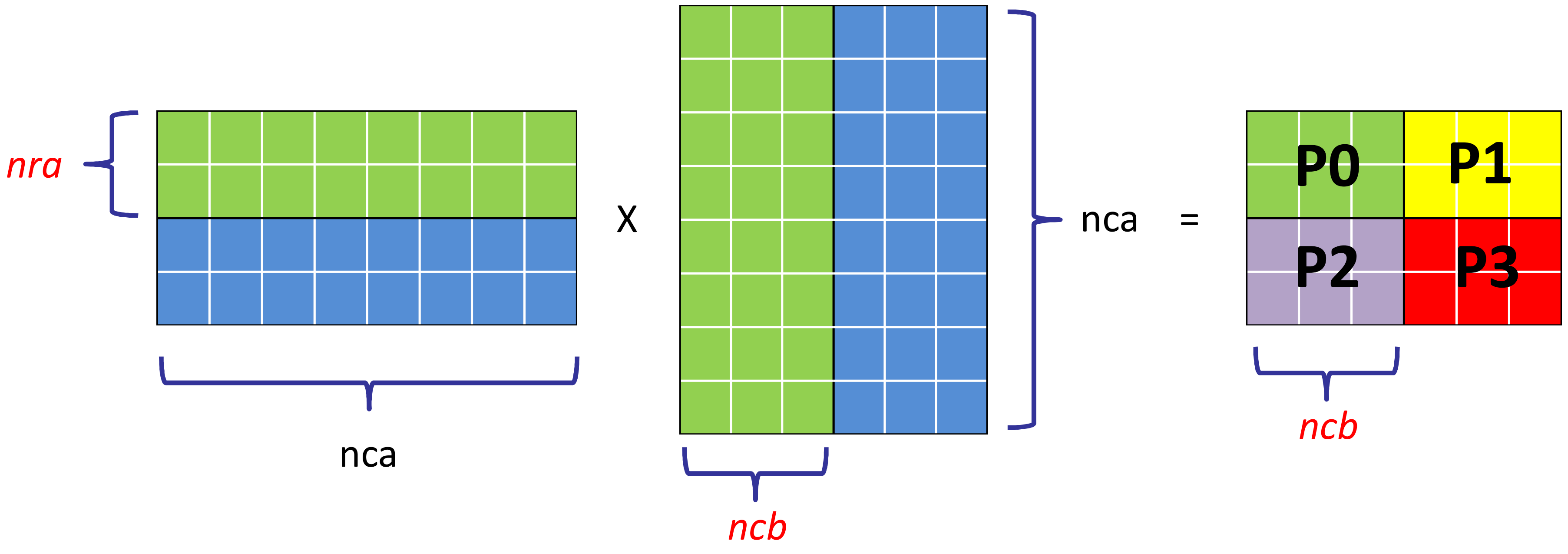
Exercise 4d: Matrix Multiplication v2

- **1D decomposition**
  - Each process owns the entire matrices, but only performs calculation on a part of them.
  - Assemble the matrix C using collective operations



Exercise 4d: Matrix Multiplication v3

- **2D decomposition**
  - Scatter the global matrix from root process, each process only owns a sub-matrix of A, B and C
  - Assemble the matrix C at the root process using the partial result from each process



$a_{nra,nca}$ x $b_{nca,ncb}$ $=$ $c_{nra,ncb}$

Thank you for your attention!

Any questions?