Introduction to Parallel Programming with MPI

Lecture #4: *MPI Derived Datatypes*

Andrea Mignone[1]

[1]Dipartimento di Fisica- Turin University, Torino (TO), Italy

Derived Datatypes

- MPI allows the construction of new, user-defined, datatypes.
- These are built from the basic MPI datatypes. The MPI Standard defines a general datatype as an object that specifies two things:
  - a sequence of basic datatypes
  - a sequence of integer (byte) displacements
- Derived datatypes are sometimes more convenient and efficient in those situations where you may need to send messages that contain
  1. non-contiguous data of a single type (e.g. a sub-block of a matrix)
  2. contiguous data of mixed types (e.g., an integer count, followed by a sequence of real numbers) 3.non-contiguous data of mixed types.
- In addition, derived datatypes improve program readability, portability as well as performance.

Constructing a Datatype

- The construction of datatypes consists in the following steps:
  1. Construct the datatype using a template or *constructor* → `MPI_Type_XXX()`. The new datatype has type `MPI_Datatype;`
  2. Allocate the datatype → `MPI_Type_commit()`
  3. Use the datatype.
  4. Deallocate the datatype → `MPI_Type_free()`

- You must <u>*construct*</u> and <u>*allocate*</u> a datatype before using it. You are not required to use it or deallocate it, but it is recommended (there may be a limit).

- A typical example is

```
MPI_Datatype new_type;              // Declare the datatype name
...
MPI_Type_XXX(..., &new_type);       // Construct the datatype
MPI_Type_commit (&new_type);        // Allocate

// ... Some work here  ...

MPI_Type_free(&new_type);
```

MPI Datatype Constructors

- These table summarized some of the most commonly used constructors

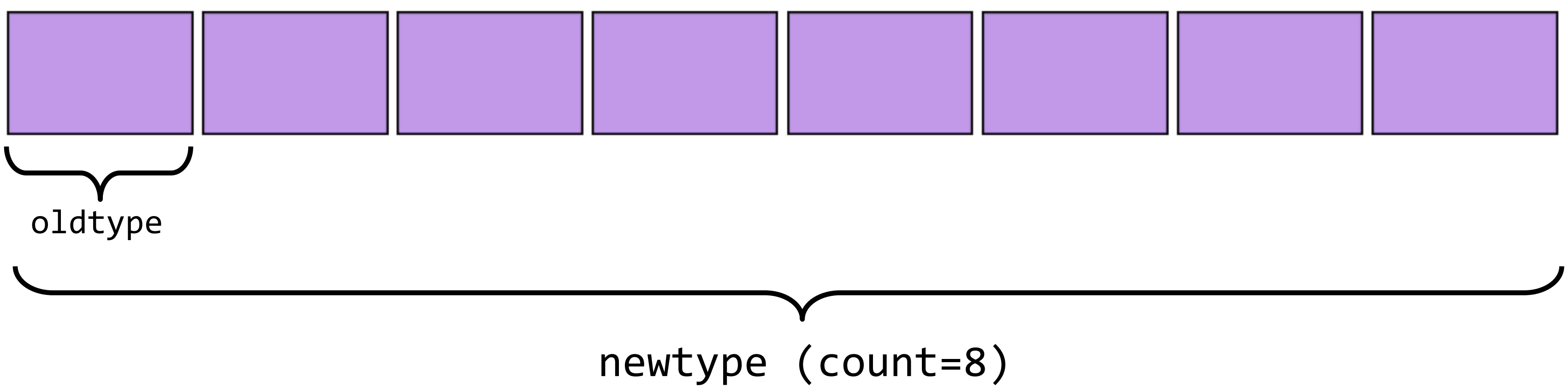
Constructor	Purpose
`MPI_Type_contiguous()`	Allows replication of a datatype into contiguous locations
`MPI_Type_vector()`	Replication of a datatype into contiguous locations with stride
`MPI_Type_hvector()`	Same as above (stride given in bytes)
`MPI_Type_indexed()`	Creates a new type from blocks comprising identical elements with varying size and displacement.
`MPI_Type_hindexed()`	Same as above (displ in bytes)
`MPI_Type_create_subarray()`	creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array
`MPI_Type_create_darray()`	Creates a datatype corresponding to a distributed, multidim. array.
`MPI_Type_create_struct()`	Creates an MPI datatype from a general set of datatypes, displacements, and block sizes

Contiguous Datatypes: `MPI_Type_contiguous()`

- `MPI_Type_contiguous()` constructs a typemap consisting of the replication of a datatype into contiguous locations.

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

- `newtype` is the datatype obtained by concatenating `count` copies of `oldtype`.

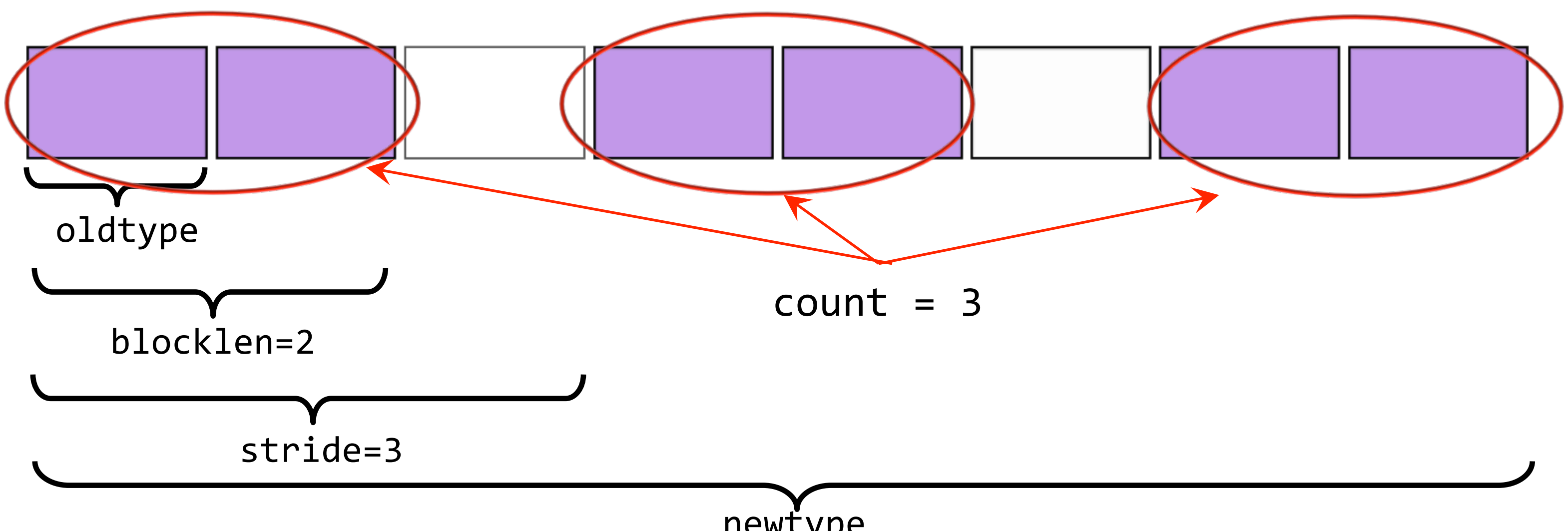


Strided Datatype: MPI_Type_vector()

- `MPI_Type_vector()` creates a derived datatype consisting of a number of elements of the same datatype repeated with a certain stride.

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

  - `count`: number of blocks;
  - `blocklen`: number of elements in each block;
  - `stride`: number of elements between start of each block;
  - `oldtype`: old datatype;
  - `newtype`: new datatype (handle).



Strided Datatype: `MPI_Create_hvector()`

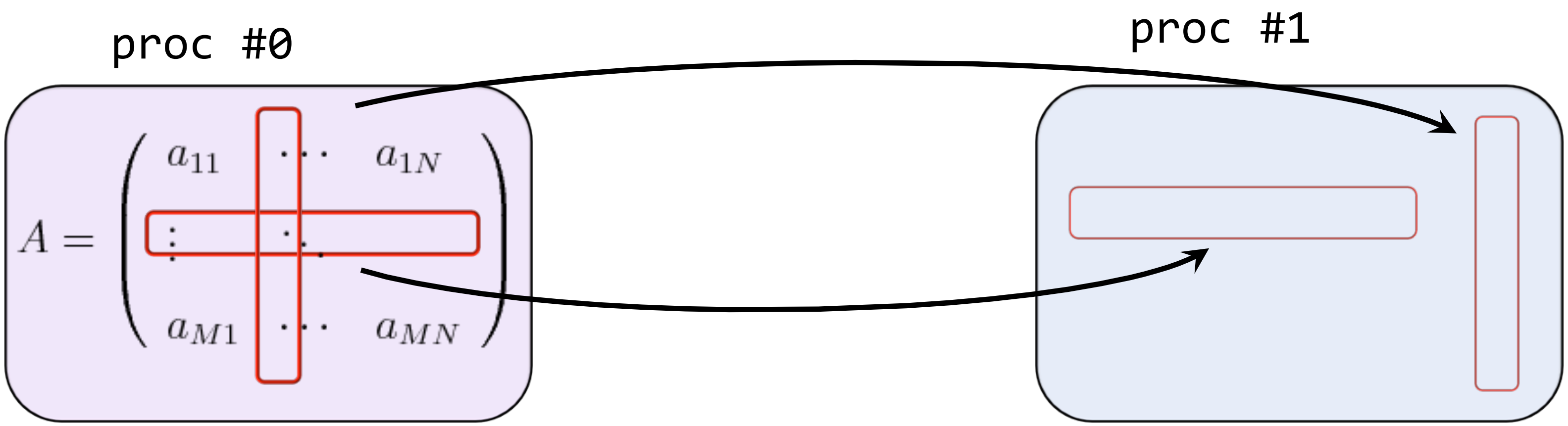
- This function is identical to `MPI_Type_vector()` with the only exception that the stride is specified in bytes rather than number of elements.

```
int MPI_Type_hvector(count, blocklen, stride, oldtype, newtype)
```

- Here "**h**" stands for **h**eterogeneous.

Example #1: Exchanging row and column

- Construct two new MPI datatypes to exchange a given row or column in a matrix.
- Process #0 owns the matrix while proc #1 has to receive the row and the column:



- Let the matrix have M row and N columns.
- Since C stores element in row-major order:
  - Row type is contiguous
  - Col type must be strided.

Example #1: Setting datatypes

- Step #1: we first declare and define our variables (e.g. the matrix `A[NROWS][NCOLS]`) as well as our new MPI datatypes "`row_type`" and "`col_type`":

```
...
#define NROWS  4
#define NCOLS  7

int main(int argc, char ** argv)
{
  int i, j, rank, size;
  int A[NROWS][NCOLS], row[NCOLS], col[NROWS];
  MPI_Datatype row_type, col_type;            // Declare new datatypes

  ... // Initialize MPI environment here

  MPI_Type_contiguous (NCOLS, MPI_INT, &row_type);           // Create row_type
  MPI_Type_vector     (NROWS, 1, NCOLS, MPI_INT, &col_type); // Create col_type
  MPI_Type_commit (&row_type);
  MPI_Type_commit (&col_type);

 if (rank == 0){                            // Only process #0 initializes the matrix.
    for (i = 0; i < NROWS; i++) {           // Matrix elements are initialized in column order.
      for (j = 0; j < NCOLS; j++) {
        A[i][j] = 1 + j + NCOLS*i;
      }
    }
  }
...
```

Example #1: Exchanging Data

- Step #2: process #0 send one specific row and column to process #1.
- Process #1 receives the data only.

```
...
  int irow = 2;   // Index of the row we want to send
  int jcol = 1;   // Index of the col we want to send
  if (rank == 0){
    printf ("Sending row = %d, col = %d\n",irow, jcol);
    MPI_Ssend (&(A[irow][0]), 1, row_type, 1, 10, MPI_COMM_WORLD);
    MPI_Ssend (&(A[0][jcol]), 1, col_type, 1, 11, MPI_COMM_WORLD);
  }else{
    MPI_Recv (row, NCOLS, MPI_INT, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv (col, NROWS, MPI_INT, 0, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Do some printing here
    printf ("Row    = ");
    for (j = 0; j < NCOLS; j++) printf ("%2d  ", row[j]);
    printf ("\n");
    printf ("Column = ");
    for (i = 0; i < NROWS; i++) printf ("%2d  ", col[i]);
    printf ("\n");
  }

  MPI_Type_free(&row_type);
  MPI_Type_free(&col_type);
...
```

Example #1: Program Output

- We send, for instance, the the 3rd row (`irow = 2`) and the 2nd column (`jcol=1`).
- The program output should look something like:

```
A[][] =
-------------------------------
 1   2   3   4   5   6   7
 8   9  10  11  12  13  14
15  16  17  18  19  20  21
22  23  24  25  26  27  28
-------------------------------
[Proc #0] Sending row = 2, col = 1
[Proc #1] Received row  = 15  16  17  18  19  20  21
[Proc #1] Received col  =  2   9  16  23
```

MPI Structure

- The structure type, created with `MPI_Type_create_struct()`, can contain multiple data types. C structure can be passed with this mechanism, using

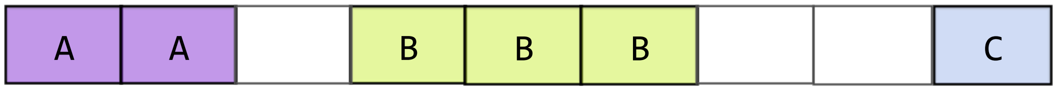
```
int MPI_Type_create_struct(int   nblocks,
                           const int blocklen,
                           const MPI_Aint displacements,
                           const MPI_Datatype oldtypes,
                           MPI_Datatype *newtype)
```

- Here each "block" is a collection of data of the same type.
  - `nblocks`: the number of blocks;
  - `blocklen`: an array of integers specifying the size of each block;
  - `displacements`: an array specifying the relative offset for each block*;
  - `oldtypes`: an array specifying the data type of the old array;
  - `newtype`: derived MPI data type (handle);

* Displacements must be expressed in bytes (since the type can change!).

MPI Structure: Example

- In the next example, `nblocks = 3`, `blocklengths={2,3,1}`, `displacements={0, 3*sizeof(A), 3*sizeof(A) + 5*sizeof(B)}`, `oldtypes = {A, B, C}`,



- Here `A, B` or `C` can be any basic MPI datatype (e.g. `A = MPI_INT, B = MPI_DOUBLE,` etc…).

- *!ATTENTION!*: In order to align the data in memory, the C compiler may insert one or more empty bytes (addresses) between memory addresses ("structure padding"):

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

→

```
struct mystruct {
    char a;
    char gap_0[3];
    int b;
    char c;
    char gap_1[3];
} x
```

- Specifying the displacements manually is not safe neither portable ! → use `MPI_Get_address()` instead.

Example #2: Creating a particle structure

- A typical example is that of a particle structure,

```
typedef struct Particle_{
  float  x;
  float  y;
  int type;
} Particle;
```

- Write a program that exchanges a particle structure between two processors.

```
  int nblocks = 2, blocklen[]   = {2, 1};
  oldtypes[]   = {MPI_FLOAT, MPI_INT};
  MPI_Aint displ[] = {0, 8};  // Manual setting (not very recommended)
  Particle p;

  ...  // Initialize MPI environment

  MPI_Get_address(&(p.x),    &displ[0]);
  MPI_Get_address(&(p.type), &displ[1]);
  displ[1] -= displ[0];
  displ[0] -= displ[0];

  MPI_Type_create_struct (nblocks, blocklen, displ, oldtypes, &MPI_Particle);
  MPI_Type_commit(&MPI_Particle);

  p.x    = ... // Initialize particle here
  int dst = 0, src = 1;
  if (rank == src) MPI_Send (&p, 1, MPI_Particle, dst, 10, MPI_COMM_WORLD);
  else      MPI_Recv (&p, 1, MPI_Particle, src, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Subarray: `MPI_Type_create_subarray()`

- The subarray type constructor creates an MPI datatype describing an n-dimensional subarray of an n-dimensional array.

- The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. The prototype of this function is:

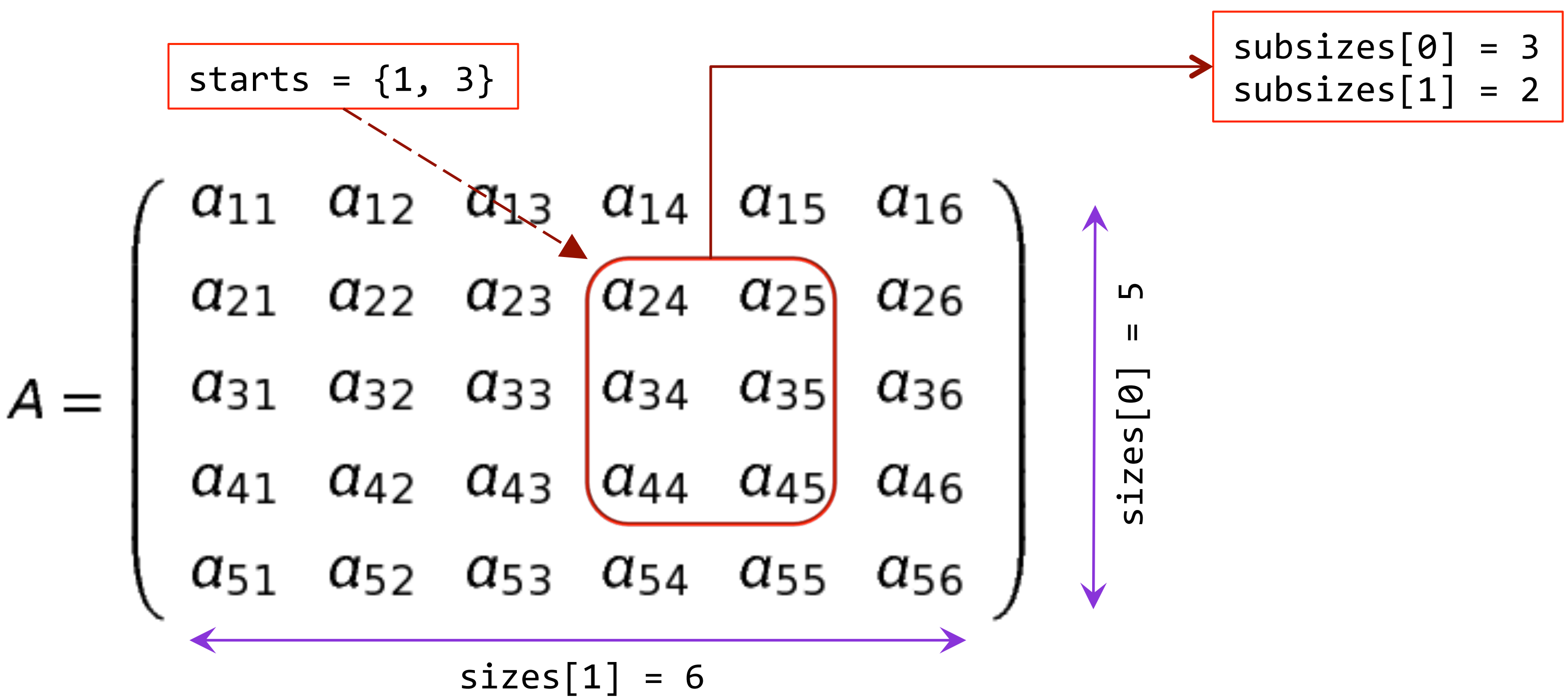
```
int MPI_Type_create_subarray(int ndims,
                             const int sizes[],
                             const int subsizes[],
                             const int starts[],
                             int order,
                             MPI_Datatype oldtype,
                             MPI_Datatype * newtype)
```

where

- `ndims`: number of array dimensions
- `sizes`: number of elements of type `oldtype` in each dimension of the full array
- `subsizes`: number of elements of type `oldtype` in each dimension of the subarray
- `starts`: starting coordinates of the subarray in each dimension
- `order`: array storage order flag (`MPI_ORDER_C / MPI_ORDER_FORTRAN`);
- `newtype`: new datatype (handle)

Subarray Example

- Suppose we wish to extract a `3x2` matrix out of a `5x6` matrix `A`.
- Remember that in C, matrices are stored in row-major format, e.g., matrix elements are stored sequentially as `{a_11, a_12, a_13, ...}` (as opposed to FORTRAN, where storage is column-major, i.e., `{a_11, a_21, a_31, ...}`).
- Then:



Example #3: Sending subarray

- Write a program that creates a general matrix `Abig[i][j]` with `0 ≤ i < NROWS`, `0 ≤ j < NCOLS` on proc #0. Array elements should be numbered sequentially by column index (row-major).
- Define a subarray type and send a buffer `Asub[][]` to proc #1.
- In order to allocate dynamic memory and print the matrices, you can use the functions `Allocate_2DintArray()` and `Show_2DintArray()` provided by the file `tools.c`. This file can be included just as another header file, e.g.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "tools.c”

int main(int argc, char **argv)
{
  int **Abig;
...
  Abig = Allocate_2DintArray(NROWS, NCOLS)
  Show_2DintArray(Abig, NROWS, NCOLS, “Abig = “)
...
}
```

Example #3: Sending subarray

- The code flowchart is

```
Initialize MPI environment, create subarray datatype
if (rank == 0){
  Allocate memory for Abig;
  Fill and print 2D array;
  Send to proc #1
}else{
  Allocate memory for Asub;
  Recv from proc #0
  print Asub
}
```

- Use `NROWS=5, NCOLS=6, subsizes={3,2}, starts={1,3}`. The output of the program should be something like

```
Big array (proc #0):
------------------------------
000  001  002  003  004  005
006  007  008  009  010  011
012  013  014  015  016  017
018  019  020  021  022  023
024  025  026  027  028  029
------------------------------

Received subarray (proc #1):
----------
009  010
015  016
021  022
----------
```