LA-UR-17-24660
Approved for public release; distribution is unlimited.

Title: An MPI Tutorial: Data Types and Threading

Author(s): Garrett, Charles Kristopher

Intended for: Presentation for the 2017 LANL Parallel Computing summer school.

Issued: 2017-06-10

An MPI Tutorial

Data Types and Threading

Kris Garrett

June 2017

Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

Data Types

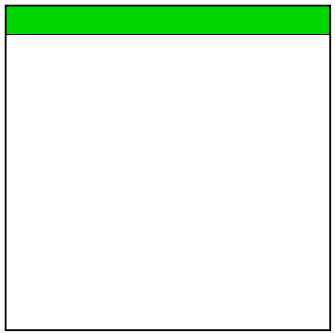
Why Data Types?

- **Basic messaging involves an array of a built-in type**
  - `MPI_Send(const void *buf, int count, MPI_Datatype datatype, ...`
- **If cluster is heterogeneous, basic data types may differ**
  - Example: `MPI_INT` is little endian on node 0 and big endian on node 1
  - Example: `MPI_CHAR` is 8 bytes on node 0 and 16 bytes on node 1
- **MPI is guaranteed to handle this for you**
  - But this is usually not a problem
  - Most clusters use the same architecture for every node

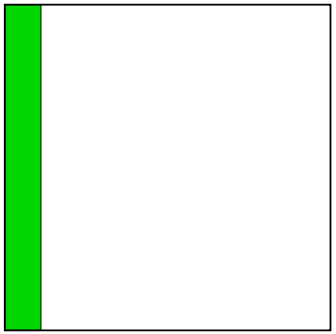
Why Create Data Types?

- **One reason to *create* a data type: handle striding in multi-D array**
- **Consider a 2D array held in memory row-wise**
  - First row data entries (contiguous): data[0], data[1], data[2], …
  - First column data entries (strided): data[0], data[N], data[2N], ...

Contiguous data



Strided Data



Strided Data

- **Options to send strided data**
  - Copy data into contiguous buffer
    - `data[0] -> buffer[0]  data[N] -> buffer[1]  data[2N] -> buffer[2]`
  - Create a subarray data type

- **Depending on implementation: creating a new data type can involve less memory movement**
  - Can be useful for both communications and IO

Subarray Data Type

```
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)
```

ndims	number of array dimensions
array_of_sizes	num elements of oldtype in each dimension of the full array
array_of_subsizes	num elements of type oldtype in each dimension of the subarray
array_of_starts	starting coordinates of the subarray in each dimension
order	array storage order flag
oldtype	array element datatype
newtype	new datatype

Subarray Data Type (Left Column)

```
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)
```

```
ndims              2
array_of_sizes     {N, N}
array_of_subsizes  {N, 1}
array_of_starts    {0, 0}
order              MPI_ORDER_C
oldtype            MPI_DOUBLE
newtype            leftColumnType
```

Using a Data Type

- **Order of calls:**
  - Create new data type
  - Call `MPI_Type_commit`
  - Use new data type as many times as you want
  - Call `MPI_Type_free`

Example: Subarray

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

int main(int argc, char **argv)
{
    int rank;
    double array[9];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (int i = 0; i < 9; i++) {
        array[i] = rank;
    }
```

Set all array values to rank

Example: Subarray

```
int sizes[2] = {3,3};
int subsizes[2] = {3,1};
int starts[2] = {0,0};
MPI_Datatype leftColumnType;

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_DOUBLE, &leftColumnType);

MPI_Type_commit(&leftColumnType);

if (rank == 0) {
    MPI_Send(array, 1, leftColumnType, 1, 0, MPI_COMM_WORLD);
}
```

Create left column data type

Commit the data type

Send the left column if rank 0

Example: Subarray

```
    if (rank == 1) {
        MPI_Recv(array, 1, leftColumnType, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                printf("%f ", array[i*3+j]);
            }
            printf("\n");
        }
    }

    MPI_Type_free(&leftColumnType);
    MPI_Finalize();
    return 0;
}
```

Receive left column if rank 1

Print array for rank 1

Output
0 1 1
0 1 1
0 1 1

Free data type

Creating Data Types

- **You can create data types to mimic structures**
  - Useful for data types made of different basic types
  - Example: Array of struct with struct made of an int and double

- **MPI Calls (Specific to General)**
  - `MPI_TYPE_CONTIGUOUS`
  - `MPI_TYPE_VECTOR, MPI_TYPE_CREATE_HVECTOR`
  - `MPI_TYPE_INDEXED, MPI_TYPE_CREATE_HINDEXED`
  - `MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_HINDEXED_BLOCK`
  - `MPI_TYPE_CREATE_STRUCT`

MPI + Threads

MPI + Threads

- **Only different call**
  - `MPI_Init_thread`
    - Replaces `MPI_Initialize`
    - Extra parameters to determine threading support

- **Thread specific calls**
  - `MPI_Query_thread`
    - returns thread support level
  - `MPI_Is_thread_main`
    - returns true if calling thread is main thread

MPI + Threads

- **Four levels of threading support**
  - `MPI_THREAD_SINGLE`
    - No threading support
  - `MPI_THREAD_FUNNELED`
    - Only master thread can call MPI functions
  - `MPI_THREAD_SERIALIZED`
    - Any thread can call MPI functions, but only one at a time
  - `MPI_THREAD_MULTIPLE`
    - Any thread can call MPI functions, even concurrently

- **`MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE`**

Anatomy of MPI_Init_thread

```
int MPI_Init_thread(int *argc, char ***argv, int required,
                    int *provided)
```

argc	Pointer to the number of arguments	(C/C++ only)
argv	Argument vector	(C/C++ only)
required	Desired level of thread support	
provided	Available level of thread support	

Threading Correctness

- **Certain functions are thread safe, regardless of library thread compliance**
  - `MPI_Initialized, MPI_Finalized, MPI_Query_thread, MPI_Is_thread_main, MPI_Get_version, MPI_Get_library_version`
- **Threads are not separately addressable. Communication is process to process only**
- **Two requirements for thread-compliant implementation**
  - All MPI calls are thread safe
  - Blocking MPI calls block only the calling thread
    - Example: Send from thread 0, Recv on thread 1 (same rank)

The End