24th Summer School on PARALLEL COMPUTING

MPI Derived Data Types

Andrew Emerson – a.emerson@cineca.it
SuperComputing Applications and Innovation Department

CINECA

Derived Data Types

- What are they?
  - Data types built from the basic MPI datatypes. Formally, 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
  - An easy way to represent such an object is as a sequence of pairs of basic datatypes and displacements. MPI calls this sequence a typemap.

typemap = {(type 0, displ 0), … (type n-1, displ n-1)}

- But for most situations you do not need to worry about the typemap.

Derived Data Types

- Why use them?
  - Sometimes more convenient and efficient. For example, 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.
- As well as improving program readability and portability they may improve performance.

How to use

1. Construct the datatype using a template or *constructor*.
2. Allocate the datatype.
3. Use the datatype.
4. Deallocate the datatype.

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

Datatype constructors

- MPI_Type_contiguous
  - Simplest constructor. Makes count copies of an existing datatype
- MPI_Type_vector, MPI_Type_hvector
  - Like contiguous, but allows for regular gaps (stride) in the displacements. For MPI_Type_hvector the stride is specified in bytes.
- MPI_Type_indexed, MPI_Type_hindexed
  - An array of displacements of the input data type is provided as the map for the new data type.MPI_Type_hindexed is identical to MPI_Type_indexed except that offsets are specified in byte
- MPI_Type_struct
  - The most general of all derived datatypes. The new data type is formed according to completely defined map of the component data types

Allocating/deallocating and using datatypes

Allocate and deallocate

- C
  - `int MPI_Type_commit (MPI_datatype *datatype)`
  - `int MPI_Type_free (MPI_datatype *datatype)`

- FORTRAN
  - `INTEGER DATATYPE, MPIERROR`
  - `MPI_TYPE_COMMIT(DATATYPE, MPIERROR)`
  - `MPI_TYPE_FREE(DATATYPE, MPIERROR)`

- C

```
MPI_Type_vector(count, blocklength, stride, oldtype, &newtype);
MPI_Type_commit (&newtype);
MPI_Send(buffer, 1, newtype, dest, tag, comm);
```

MPI_TYPE_CONTIGUOUS

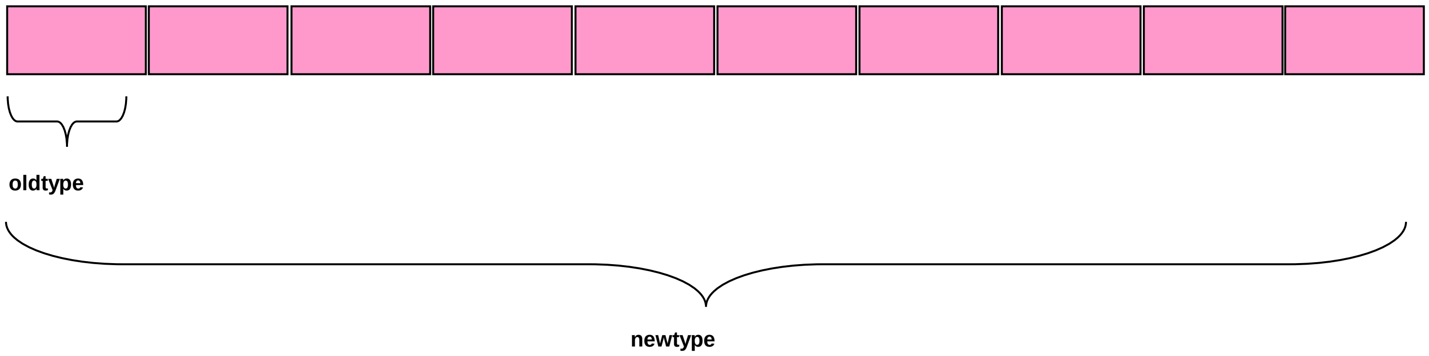
MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)

IN count: replication count (non-negative integer)

IN oldtype: old datatype (handle)

OUT newtype: new datatype (handle)

- MPI_TYPE_CONTIGUOUS constructs a typemap consisting of the **replication** of a **datatype** into contiguous locations.
- newtype is the datatype obtained by concatenating count copies of oldtype.

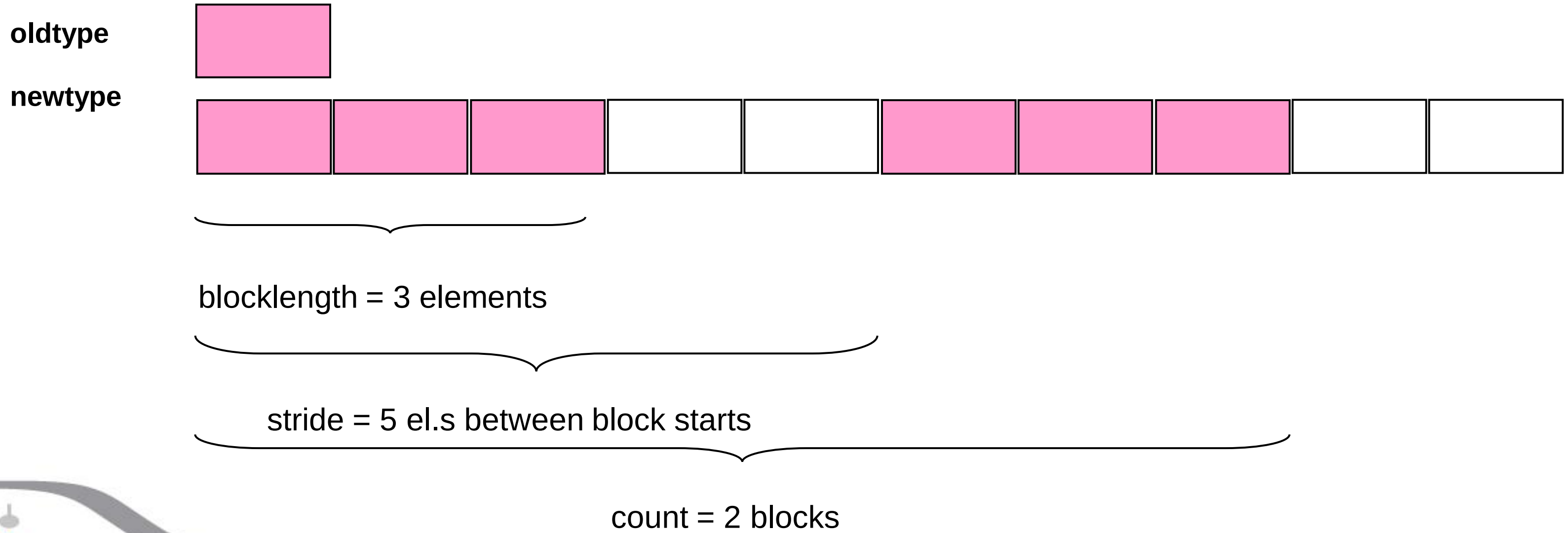


MPI_TYPE_VECTOR

MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)

- IN count: Number of blocks (non-negative integer)
- IN blocklen: Number of elements in each block (non-negative integer)
- IN stride: Number of elements (NOT bytes) between start of each block (integer)
- IN oldtype: Old datatype (handle)
- OUT newtype: New datatype (handle)

- Consists of a number of elements of the same datatype repeated with a certain stride



Example 1 - A rowtype

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

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_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

1 element of rowtype

9.0	10.0	11.0	12.0

Example 2 - columntype

```
count = 4;   blocklength = 1;   stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
                &columntype);
```

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

a[4][4]

```
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```

2.0	6.0	10.0	14.0

1 element of columntype

Other tools

- MPI_GET_COUNT,MPI_GET_ELEMENTS
  - Routines which return the number of "copies" of type datatype and the number of basic elements (often used after a MPI_RECV).

```
int MPI_Get_count( const MPI_Status *status, MPI_Datatype datatype, int *count )
int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,  int *count)
```

- MPI_TYPE_GET_EXTENT (*Advanced*)
  - Returns the lower bound and extent of a datatype (i.e. upper bound + padding to align the datatype). Useful for creating new datatypes with **MPI_TYPE_CREATE_RESIZED,** for example.

Derived Datatype Summary

- Provide a portable and elegant way of communicating non-contiguous or mixed types in a message.
- By optimising how data is stored, should improve efficiency during MPI send and receive (perhaps avoiding buffering).
- Derived datatypes are built from basic MPI datatypes, according to a template. Can be used for many variables of the same form.
- Remember to commit the datatypes before using them.