COSC 4397
Parallel Computation

Introduction to MPI IV - Derived Data Types

Edgar Gabriel
Spring 2010

Derived Datatypes

- Basic idea: interface to describe memory layout of user data structures
e.g. a structure in C
typedef struct {
    char a;
    int b;
    double c;
} mystruct;

Memory layout
Derived Datatype examples

- E.g. describing a column or a row of a matrix

- Memory layout in C

- Memory layout in Fortran

How to describe non-contiguous data structures

typedef struct {
    char a;
    int b;
    double c;
} mystruct;

- using a list-I/O interface, e.g. <address, size>

    <baseaddr, sizeof(char)>
    <address1, sizeof(int)>
    <address2, sizeof(double)>

- or

    <baseaddr, sizeof(char)>
    <baseaddr+gap1, sizeof(int)>
    <baseaddr+gap2, sizeof(double)>
...or in MPI terminology...

- a list of `<address, count, datatype>` sequences
  
  `<baseaddr, 1, MPI_CHAR>`
  `<baseaddr+gap1, 1, MPI_INT>`
  `<baseaddr+gap2, 1, MPI_DOUBLE>`

- ...leading to the following interface...

  ```c
  MPI_Type_create_struct (int count, int blocklength[],
                          MPI_Aint displacements[], MPI_Datatype datatypes[],
                          MPI_Datatype *newtype);
  ```

  ```c
  MPI_Type_create_struct (int count, int blocklength[],
                          MPI_Aint displacements[], MPI_Datatype datatypes[],
                          MPI_Datatype *newtype);
  ```

**MPI_Type_struct/MPI_Type_create struct**

- **MPI_Aint:**
  
  - Is an MPI Address integer
  - An integer being able to store a memory address

- Displacements are considered to be relative offsets
  
  ⇒ `displacement[0] = 0` in most cases!
  
  ⇒ Displacements are not required to be positive, distinct or in increasing order

- How to determine the address of an element
  
  ```c
  MPI_Address (void *element, MPI_Aint *address);
  ```
  
  ```c
  MPI_Get_address (void *element, MPI_Aint *address);
  ```
Addresses in MPI

• Why not use the & operator in C?
  - ANSI C does NOT require that the value of the pointer returned by & is the absolute address of the object!
  - Might lead to problems in segmented memory space
  - Usually not a problem

• In Fortran: all data elements passed to a single MPI_Type_struct call have to be in the same common block

Type map vs. Type signature

• Type signature is the sequence of basic datatypes used in a derived datatype, e.g.
  typesig(mystruct) = {char, int, double}

• Type map is sequence of basic datatypes + sequence of displacements
  typemap(mystruct) = {(char,0),(int,8),(double,16)}

• Type matching rule of MPI: type signature of sender and receiver has to match
  - Including the count argument in Send and Recv operation (e.g. unroll the description)
  - Receiver must not define overlapping datatypes
  - The message need not fill the whole receive buffer
Committing and freeing a datatype

- If you want to use a datatype for communication or in an MPI-I/O operation, you have to commit it first

  ```c
  MPI_Type_commit (MPI_Datatype *datatype);
  ```

- Need not commit a datatype, if just used to create more complex derived datatypes

  ```c
  MPI_Type_free (MPI_Datatype *datatype);
  ```

- It is illegal to free any predefined datatypes

Our previous example looks like follows:

```
mystruct mydata;

MPI_Address ( &mydata, &baseaddr);
MPI_Address ( &mydata.b, &addr1);
MPI_Address ( &mydata.c, &addr2);

disp1[0] = 0;
disp1[1] = addr1 - baseaddr;
disp1[2] = addr2 - baseaddr;

dtype[0] = MPI_CHAR;
blength[0] = 1;
dtype[1] = MPI_INT;
blength[1] = 1;
dtype[2] = MPI_DOUBLE;
blength[2] = 1;

MPI_Type_struct ( 3, blength, disp1, dtype, &newtype );
MPI_Type_commit ( &newtype );
```
Basically we are done...

- With MPI_Type_struct we can describe any pattern in the memory
- Why other MPI datatype constructors?
  - Because description of some datatypes can become rather complex
  - For convenience

MPI_Type_contiguous

MPI_Type_contiguous (int count, MPI_Datatype datatype, MPI_Datatype *newtype);

- count elements of the same datatype forming a contiguous chunk in the memory

int myvec[4];
MPI_Type_contiguous (4, MPI_INT, &mybrandnewdatatype);
MPI_Type_commit (&mybrandnewdatatype);
MPI_Send (myvec, 1, mybrandnewdatatype, ...);

- Input datatype can be a derived datatype
  - End of one element of the derived datatype has to be exactly at the beginning of the next element of the derived datatype
MPI_Type_vector

MPI_Type_vector( int count, int blocklength, int stride,
               MPI_Datatype datatype, MPI_Datatype *newtype );

- count blocks of blocklength elements of the same datatype
- Between the start of each block there are stride elements of the same datatype

Example using MPI_Type_vector

- Describe a column of a 2-D matrix in C
  
dtype = MPI_DOUBLE;
stride = 8;
bleNGTH = 1;
count = 8;

MPI_Type_vector (count,blength,stride,dtype,&newtype);
MPI_Type_commit (&newtype);

- Which column you are really sending depends on the pointer which you pass to the according MPI_Send routine!
MPI_Type_hvector

MPI_Type_hvector( int count, int blocklength,
  MPI_Aint stride, MPI_Datatype datatype,
  MPI_Datatype *newtype );

MPI_Type_create_hvector( int count, int blocklength,
  MPI_Aint stride, MPI_Datatype datatype,
  MPI_Datatype *newtype );

- Identical to MPI_Type_vector, except that the stride is given in bytes rather than in number of elements

MPI_Type_indexed

MPI_Type_indexed( int count, int blocklengths[],
  int displacements[], MPI_Datatype datatype,
  MPI_Datatype *newtype );

- The number of elements per block do not have to be identical
- displacements gives the distance from the ‘base’ to the beginning of the block in multiples of the used datatype

<table>
<thead>
<tr>
<th>count</th>
<th>blocklengths[0]</th>
<th>displacements[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
MPI_Type_hindexed

```c
MPI_Type_hindexed( int count, int blocklengths[],
                  MPI_Aint displacements[], MPI_Datatype datatype,
                  MPI_Datatype *newtype );
```

- Identical to MPI_Type_indexed, except that the displacements are given in bytes and not in multiples of the datatypes

Duplicating a datatype

```c
MPI_Type_dup(MPI_Datatype datatype, MPI_Datatype *newtype);
```

- Mainly useful for library developers, e.g. datatype ownership
- The new datatype has the same ‘committed’ state as the previous datatype
  - If datatype has already been committed, newtype is committed as well
MPI_Type_create_subarray

MPI_Type_create_subarray (int ndims, int sizes[],
int subsizes[], int starts[], int order,
MPI_Datatype datatype, MPI_Datatype *newtype);

- Define sub-matrices of n-dimensional data
- sizes[]: dimension of the entire matrix
- subsizes[]: dimensions of the submatrix described by the derived data type
- starts[]: array describing the beginning of the submatrices
- **Order:** MPI_ORDER_C for row-major order or
  MPI_ORDER_FORTRAN for column-major data

Example

```
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 29 30 31 32
33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64
```

Dimension 0

Dimension 1

```
dims = 2;
sizes[0] = 8; sizes[1] = 8;
subsizes[0] = 4; subsizes[1] = 2
starts[0] = 2; starts[1] = 4;
MPI_Type_create_subarray (ndims, sizes, subsizes,
starts, MPI_ORDER_C, MPI_DOUBLE,
&newtype);
```
More datatype constructors

- MPI_Type_create_darray(int size, int rank, int ndims, int *gsizes[], int *distribrs[], int *dargs[], int *psizes[], int order, MPI_Datatype datatype, MPI_Datatype *newtype);
- Describe HPF-like data distributions
- MPI_Type_create_indexed_block(int count, int blocklength, int *disps[], MPI_Datatype datatype, MPI_Datatype *newtype);
- Further simplification of MPI_Type_indexed

Portable vs. non-portable datatypes

- Any data type constructors using byte-offsets are considered non-portable
  - Might rely on data alignment rules given on various platforms
- Non-portable datatype constructors:
  - MPI_Type_struct
  - MPI_Type_hvector/MPI_Type_create_hvector
  - MPI_Type_hindexed/MPI_Type_create_hindexed
- Non-portable datatypes are not allowed to be used in
  - one-sided operations
  - parallel File I/O operations
A problem with the specification up to now

typedef struct {
    char    a;
    int     b;
    double  c;
    float   d;
} mystruct;

mystruct mydata[5];

• ...but just want to send b and c of the structure, however multiple elements of mystruct

...simple description...

MPI_Address ( &(mydata[0], &baseaddr);
MPI_Address ( &(mydata[0].b, &addr1);
MPI_Address ( &(mydata[0].c, &addr2);

disp1[0] = addr1 - baseaddr;
disp1[1] = addr2 - baseaddr;

dtype[0] = MPI_INT;        blength[0] = 1;
dtype[1] = MPI_DOUBLE;     blength[1] = 1;

MPI_Type_struct ( 2, blength, disp1, dtype, &newtype );
MPI_Type_commit ( &newtype );
If we use this datatype....

- it is ok if we send one element
  MPI_Send ( mydata, 1, newtype,...);
- If we send more elements, all data at the receiver will be wrong, except for the first element
  MPI_Send ( mydata, 5, newtype, ...);
- Memory layout

  [Diagram of memory layout]

- What we send is

  [Diagram of what is sent]

- What we wanted to do is

  [Diagram of what was expected]

...so what we missed ...

- ...was to tell MPI where the next element of the structure starts
  - or in other words: we did not tell MPI where the begin and the end of the structure is

- Two ‘marker’ datatypes introduced in MPI
  - MPI_LB: lower bound of a structure
  - MPI_UB: upper bound of a structure
Correct description of the structure would be

```c
MPI_Address ( &(mydata[0]), &addr2);
MPI_Address ( &(mydata[0].b), &addr1);
MPI_Address ( &(mydata[0].c), &addr2);
MPI_Address ( &(mydata[1]), &addr3);

disp[0] = 0;
disp[1] = addr1 - baseaddr;
disp[2] = addr2 - baseaddr;
disp[3] = addr3 - baseaddr;

dtype[0] = MPI_LB;    blength[0] = 1;

MPI_Type_struct ( 4, blength, displ, dtype, &newtype );
```

Determining upper- and lower bound

- Two functions to extract the upper and the lower bound of a datatype

```c
MPI_Type_ub ( MPI_Datatype dat, MPI_Aint *ub );
MPI_Type_lb ( MPI_Datatype dat, MPI_Aint *lb );
```
extent vs. size of a datatype

MPI_Type_extent ( MPI_Datatype dat, MPI_Aint *ext);
MPI_Type_size ( MPI_Datatype dat, int *size );

extent := upper bound - lower bound;
size = amount of bytes really transferred

The MPI-2 view of the same problem

• Problem with the way MPI-1 treats this problem:
  upper and lower bound can become messy, if you have
  derived datatype consisting of derived datatype
  consisting of derived datatype consisting of... and each
  of them has MPI_UB and MPI_LB set

• No way to erase upper and lower bound markers once
  they are set

• MPI-2 solution: reset the extent of the datatype

  MPI_Type_create_resized ( MPI_Datatype datatype,
    MPI_Aint lb, MPI_Aint extent, MPI_Datatype
    *newtype );
  - Erases all previous lb und ub markers
MPI-2 view of the same problem (II)

MPI_Type_get_true_extent ( MPI_Datatype dat,
            MPI_Aint *lb, MPI_Aint *extent );

The true extent
- Extent of the datatype ignoring UB and LB markers: all gaps in the middle are still considered, gaps at the beginning and at the end are removed
- E.g. required for intermediate buffering

extent

true extent