MPI derived data types

Create your own datatypes!

Overview

- Creating data types to describe your data
  - Contiguous
  - Vectors
  - Structs
- Advanced topic
  - Define your own reduction operations

Derived data types

- MPI comes with a number of predefined data types
- Can create your own data types
  - Strided data (e.g. non-contiguous array boundaries)
  - Derived data types (C structs, Fortran type, ...)
- Typically a two stage process
  - Creating the data type
  - Committing the data type
- Performance strongly depends on architecture

Combining data: MPI_Type_contiguous

- Sending several of the same datatype, e.g.: array of real
- In C

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

- In Fortran 90

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, &
 IERROR)
 INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

`count:` Number of old data to be combined

`oldtype:` Data type of the original data

`newtype:` New data type (output)

- Remark: You still need to commit new type to use it

MPI_TYPE_COMMIT

- Created data types need committing **before** usage
- In C

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

- In Fortran

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
  INTEGER DATATYPE, IERROR
```

- `Datatype`: created datatype to be committed

MPI_TYPE_FREE

- You can also remove a data type to free up memory
- In C

```
int MPI_Type_free(MPI_Datatype *datatype)
```

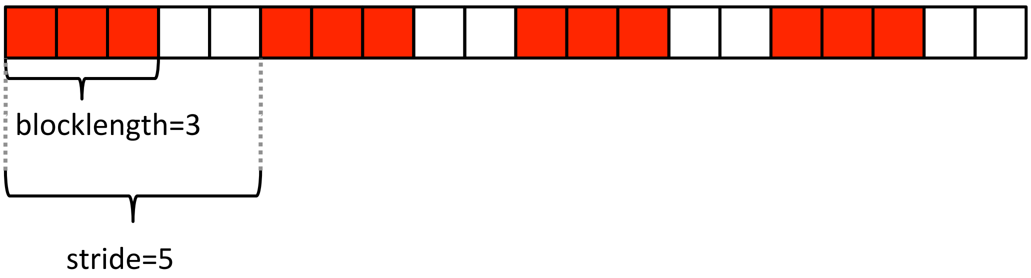
- In Fortran 90

```
MPI_TYPE_FREE(DATATYPE, IERROR)
  INTEGER DATATYPE, IERROR
```

- **`Datatype`**: commited datatype to be removed

Remark: Avoid frequent committing and freeing of datatypes (performance)

Describing vectors



- Sending regular patterns of data (e.g. from an array)

- Example, sending the red boxes:
  - Describe pattern with **`blocklength`** and **`stride`**
  - Give the number of patterns as **`count`**, here 4

MPI_TYPE_VECTOR

- In C

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

- In Fortran

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, & OLDTYPE,
  NEWTYPE, IERROR)
  INTEGER COUNT, BLOCKLENGTH, STRIDE, & OLDTYPE, NEWTYPE,
  IERROR
```

`count:` Number of blocks

`blocklength:` Number of elements to take

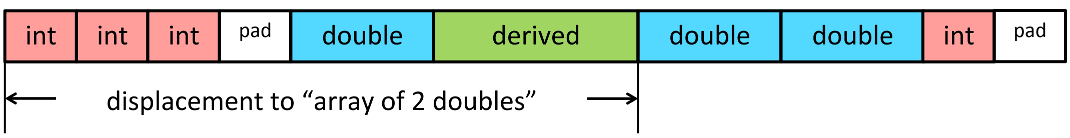
`stride:` Number of elements until the next block

`oldtype:` Old data type

`newtype:` New data type (output, needs commit)

Communicating Derived data types

Describing derived data



- Example of derived data type, with **`count`** 5
  - Array of 3 int, 1 double, 1 derived, array of 2 double, 1 int
    - Compiler introduced "pad" after the 3 int (not to be sent)
  - There might be a "pad" at the end
    - take care of when e.g. sending arrays of derived data
  - We assume mpi derived type for the derived block has already been created
- Need to supply array of dimension 5 for:
  - **`blocklength`**, **`displacement`** and **`oldtypes`**
  - **`displacement`** in bytes – don't use C to figure these

Portable displacements: MPI_Get_address

- To figure the displacements:
  - Create (a sample of) the object
  - Measure position of <u>each</u> member: **MPI_Get_address**

In C:

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

In Fortran 90:

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
 <type> LOCATION
 INTEGER IERROR
 INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
```

`location:` location in caller memory (element to be tested)

`address:` absolute address with respect to **MPI_BOTTOM**

Example: Displacement array in C

```
struct Pair { float f;
              double d; };
struct Pair myPair;
MPI_Aint displ[2], base;

MPI_Get_address(&(myPair.f), &base);
MPI_Get_address(&(myPair.d), &displ[1]);

displ[0]  = 0;
displ[1] -= base;
```

Example: Displacement array in Fortran

```
type pair
     real(kind(1.0))   :: f
     real(kind(1.0d0)):: d
end type pair

type(pair) :: myPair
Integer(kind=mpi_address_kind) :: displ(2), base

Call MPI_Get_address(myPair%f, base, ierror)
Call MPI_Get_address(myPair%d, displ(2), ierror)
displ(1) = 0
displ(2) = displ(2) - base
```

MPI_Type_create_struct in C

```
int MPI_Type_create_struct(int count,
 int array_of_blocklengths[],
 MPI_Aint array_of_displacements[],
 MPI_Datatype array_of_types[],
 MPI_Datatype *newtype)
```

`count:`	Number of blocks
`array_of_blocklengths:`	Number of elements per block
`array_of_displacements:`	byte displacement of each blk.
`array_of_types:`	data type of elements in block
`newtype:`	handle for the new datatype

MPI_TYPE_CREATE_STRUCT in Fortran 90

```
MPI_TYPE_CREATE_STRUCT(COUNT, &
 ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, &
 ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), &
 ARRAY_OF_TYPES(*), NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) &
 ARRAY_OF_DISPLACEMENTS(*)
```

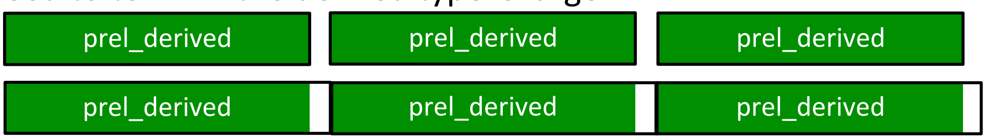
`count:`	Number of blocks
`array_of_blocklengths:`	Number of elements per block
`array_of_displacements:`	byte displacement of each block
`array_of_types:`	data type of elements in block
`newtype:`	handle for the new datatype

Dealing with a “pad” at the end

- Important for sending arrays of derived data type

- Created derived MPI type for 1 instance
- Using `count>1` of those leads to problem:

- Need to tell MPI the derived type is larger



MPI_TYPE_CREATE_RESIZED in C

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

`oldtype:` input data type (input, the one which is to small)
`lb:` new lower bound
`extent:` size of resized data type
`newtype:` resized data type (output)

Rem: `newtype` still needs committing

MPI_TYPE_CREATE_RESIZED in Fortran 90

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT,
NEWTYPE, IERROR)
INTEGER OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

`oldtype:` input data type (input, the one which is to small)
`lb:` new lower bound
`extent:` size of resized data type
`newtype:` resized data type (output)

Rem: `newtype` still needs committing

Example: Resize in Fortran 90

```
type ri_pair
     real(kind(1.0d0)) :: realpart
     integer           :: intpart
end type ri_pair
type(ri_pair), dimension(3)    :: my_values
Integer(kind=mpi_address_kind) :: start_address, secnd_address, &
    lowerb, extent

Call MPI_Get_address(my_values(1), start_address, merror)
Call MPI_Get_address(my_values(2), secnd_address, merror)
extent = secnd_address - start_address
lowerb = 0                        ! need a zero of right kind
Call MPI_Type_create_resized(ri_mpitype, lowerb, extent,
ri_mpitype_pad, merror)
```

Creating you own reduction operators

Creating your own operators for reductions (Advanced topic)

- You can supply your own function for reduction operations
- Particularly useful for derived data types
- User functions work on vectors
  - Don’t forget the loop inside

MPI_OP_CREATE in C

- To register the function:

```
int MPI_Op_create(MPI_User_function *function,
   int commute, MPI_Op *op)
```

`function:`	function pointer, see next slide
`commute:`	logical, is the operation commutative
`op:`	Handle for operation

Prototype for user supplied function in C

```
typedef void MPI_User_function( void *invec, void
 *inoutvec, int *len, MPI_Datatype *datatype);
```

`invec:`	vector of length **`len`**, with data from one rank
`inoutvec:`	vector of length **`len`**, with data from other rank, also returns result to MPI
`len:`	length of vectors – number of operations
`datatype:`	handle for MPI data type – use for overloading

MPI_OP_CREATE in Fortran 90

- To register the function:

```
MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
EXTERNAL FUNCTION, LOGICAL COMMUTE, INTEGER OP,
IERROR)
```

`function:`	function pointer
`commute:`	logical, is the operation commutative
`op:`	Handle for operation

Prototype for user supplied function in Fortran

```
SUBROUTINE USER_FUNCTION( INVEC(*), INOUTVEC(*),
 LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE
```

`invec:` vector of length **`len`**, with data from one rank

`inoutvec:` vector of length **`len`**, with data from other rank, also returns result to MPI

`len:` length of vectors – number of operations

`type:` handle for MPI data type – use for overloading

Example: function in Fortran

```
Module realint_pair
  type ri_pair
     real(kind(1.0d0)) :: rp
     integer           :: intp
  end type ri_pair
contains
  subroutine addmax_ri(first, second, len, type)
    type(ri_pair), dimension(len):: first, second
    integer, intent(in) :: len, type
    integer :: i
    do i = 1, len
     second(i)%rp   = first(i)%rp + second(i)%rp
     second(i)%intp = max(first(i)%intp, second(i)%intp)
    enddo
  end subroutine addmax_ri
End Module realint_pair
```

Example: function in C

```
typedef struct {
  double rpart;
  int    ipart;
} ri_pair;
void addmax_ri(void* pfrt, void* pscd, int* plen, MPI_Datatype* ptype)
{
  ri_pair* pf = (ri_pair*)pfrt;
  ri_pair* ps = (ri_pair*)pscd;
  for (int i=0; i < *plen; i++)
    {
      ps[i].rpart += pf[i].rpart;
      ps[i].ipart = fmax(pf[i].ipart, ps[i].ipart);
    }
  return;
}
```

Summary

- Selection of MPI derived data types
- Sending C, C++ and Fortran derived data types
- Showed how to create your own reduction operators

Exercise

Copy your code to send messages around a ring

- Create a C struct or Fortran type:
  - 1 double/double precision, 1 int/integer
- Create an MPI derived data type to send the above
- Replace message buffers with the above derived type
  - double: 0.1x rank number resp. 0.001x rank number
  - integer: rank number resp. 1000x rank number
- Modify MPI calls to transfer derived data
- Confirm results are correct

Additional exercises

- Send arrays of your derived data type
  - Resize you datatype to take care of padding
- Implement a reduction operator for your MPI data type
  - adds the double
  - maximum of first integer
  - sum of second integer
  - sum of thirds integer
- Use this for
  - the rows of your Cartesian communicator (if you have done that part in the previous exercise)
  - the entire communicator