消息传递编程接口 MPI

（三）数据类型

目录页

Contents

目录页

Contents

1 MPI 数据类型

- 为什么要自定义数据类型
- 数据类型图

为什么要自定义数据类型

MPI 消息传递通常只能处理连续存放的同一类型的数据

MPI 自定义数据类型

- 如果需要传递具有复杂结构的数据，可以使用自定义数据类型
- 使用自定义数据类型的好处：有效减少消息传递次数，增大通信粒度，同时可以避免或减少消息传递时数据在内存中的拷贝。

† 注意：MPI 的数据类型主要用于消息传递！

MPI 数据类型图

MPI 数据类型的组成

- 由两个相同长度的序列组成：类型序列和位移序列

$$\{t_1, t_2, t_3, \ldots, t_n\}$$

$$\{d_1, d_2, d_3, \ldots, d_n\}$$

其中 t_i 的取值为基本数据类型，d_i 代表位移，取值为整数，以字节为单位，新建的数据类型称为复合数据类型。

$\{(t_1,d_1),(t_2,d_2), \ldots, (t_n,d_n)\}$ ⇒ MPI 数据类型图

该类型图所代表的数据类型包含 n 个数据，其中第 i 个数据的数据类型为 t_i，该数据离首地址的距离为 d_i

数据类型举例

例：设数据类型 mytype 的数据类型图为

{(MPI_FLOAT,4),(MPI_FLOAT,12),(MPI_FLOAT,0)}

则下面的语句：

```
float A(100)
... ...
MPI_Send(A, 1, mytype, ... )
```

发送的数据为

```
A(2), A(4), A(1)
```

目录页

Contents

2 数据类型的大小和域

- 数据类型的大小
- 数据类型的上下界和域
- 数据类型查询函数

数据类型的大小

数据类型的大小：包含的数据长度，即字节数。

设一个数据的类型图为

$$\{(t_1,d_1),(t_2,d_2),(t_3,d_3),\ \ldots,\ (t_n,d_n)\}$$

则它的大小为

$$\mathrm{sizeof}(t_1) + \mathrm{sizeof}(t_2) + \ldots + \mathrm{sizeof}(t_n)$$

Example

例：如果 mytype 的数据类型图为：

{(MPI_FLOAT,4),(MPI_FLOAT,12),(MPI_FLOAT,0)}

则 mytype 的大小为 12

数据类型的域

$\{(t_1,d_1),(t_2,d_2),(t_3,d_3),\ \ldots,\ (t_n,d_n)\}$

- 数据类型的下界：类型图中的最小位移，即 $\min\limits_{1\le i\le n}\{d_i\}$
- 数据类型的上界： $\max\limits_{1\le i\le n}\{d_i+\mathrm{sizeof}(t_i)\}+\epsilon$

  其中 ϵ 为地址对界修正量

数据类型的长度（extent）= 上界 − 下界

数据类型的对界量

数据类型的对界量

- 原始数据类型的对界量：由编译系统决定
- 复合数据类型的对界量：其所包含的基本数据类型的对界量的最大值
- 地址对界要求：

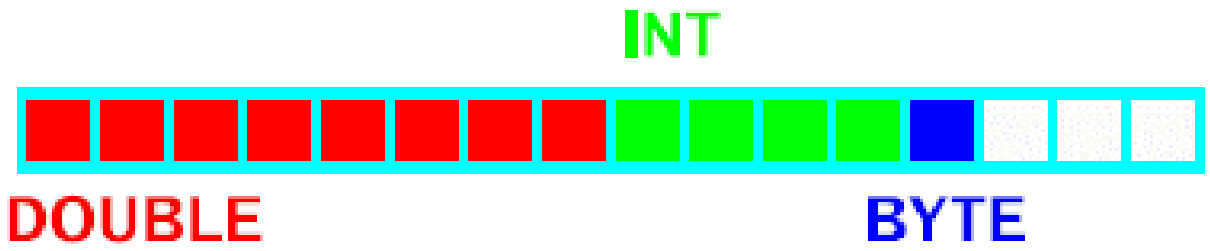
  一个数据类型在内存中所占的字节数必须是其对界量的整数倍
- 地址对界修正量：

  使得新建数据类型的长度（extent）能被其对界量整除的最小非负整数

对界修正量举例

例：假设 MPI_DOUBLE 和 MPI_INT 的对界量均为 4，MPI_BYTE 的对界量为 1，考虑下面的数据类型

{(MPI_DOUBLE,0), (MPI_BYTE,12), (MPI_INT,8)}



问：大小为____，对界量为 ____，上界为____，下界为 ____，长度为_____，地址对界修正量为 ____ 。

13, 4, 16, 0, 16, 3

两个概念上的数据类型

lb_marker、ub_marker

- 概念上数据类型（conceptual datatypes）

  大小为 0，它们的作用仅仅是用来人工指定新建数据类型的上下界，除此之外，对数据没有任何影响。

  - 若数据类型中含 lb_marker，则下界为 lb_marker 的位移的最小值；
  - 若数据类型中含 ub_marker，则上界为 ub_marker 的位移的最大值；

Example

例：下面的数据类型的下界为 -4

```
{(MPI_REAL,4), (lb_marker,12), (MPI_REAL,0), (lb_marker,-4)}
```

相关查询函数

MPI_TYPE_SIZE(datatype, size)

- 返回指定数据类型的大小（即字节数）

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

- 返回指定数据类型的下界和长度

MPI_TYPE_SIZE

MPI_TYPE_SIZE(datatype, size)

C	`int MPI_Type_size(MPI_Datatype datatype, int *size)`
F77	`MPI_TYPE_SIZE(DATATYPE, SIZE, IERR)` `INTEGER DATATYPE, SIZE, IERR`

- 返回指定数据类型的大小（即字节数）

MPI_TYPE_GET_EXTENT

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

C	`int MPI_Get_type_extent(MPI_Datatype datatype,` `  MPI_Aint *lb, MPI_Aint *extent)`

F77	`MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERR)`
	`INTEGER DATATYPE, IERR`
	`INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT`

- 这里 MPI_Aint 用来声明存放地址或位移的变量

目录页

Contents

3 新数据类型的创建

- 数据类型的创建函数
- 新数据类型的提交与释放

新数据类型的创建

- 新数据类型创建函数

MPI_TYPE_CONTIGUOUS

MPI_TYPE_VECTOR

MPI_TYPE_INDEXED

MPI_TYPE_CREATE_HVECTOR

MPI_TYPE_CREATE_HINDEXED

MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CONTIGUOUS

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

连续复制：

将原数据类型 **oldtype** 按顺序依次连续复制，得到一个新的数据类型

参数	IN count 复制个数 IN oldtype 原数据类型 OUT newtype 新数据类型
C	int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
F77	MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERR) INTEGER COUNT, OLDTYPE, NEWTYPE, IERR

- 注：**oldtype** 可以是原始数据类型，也可以是已创建的复合数据类型。

提交与释放

- 新数据类型的提交

  MPI_TYPE_COMMIT(newdatatype)

  - 使用新数据类型进行通信，则必须先提交
  - 过渡数据类型不用提交，用完后就可直接释放

- 新数据类型的释放

  MPI_TYPE_FREE(newdatatype)

举例

```
const int n=100;
MPI_Datatype newtype;
float a[n];
... ...
... ...
MPI_Type_contiguous(n, MPI_FLOAT, &newtype);
MPI_Type_commit(newtype);
MPI_Sendrecv_replace(a,1,newtype,dst,111,src,
                     111,MPI_COMM_WORLD,status);
... ...
```

上面的消息传递等价于

```
MPI_Sendrecv_replace(a,n,MPI_FLOAT,dst,111,src,
                     111,MPI_COMM_WORLD,status);
```

MPI_TYPE_VECTOR

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

创建向量数据类型：

先连续复制 **blocklen** 个 **oldtype** 类型的数据，形成一个数据块；再通过等间隔地复制 **count** 个该数据块而形成新的数据类型；相邻两个数据块的起始位置的位移相差为 **stride*extent(oldtype)** 个字节。

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

F77	`MPI_TYPE_VECTOR(COUNT, BLOCKLEN, STRIDE,`
	`  OLDTYPE, NEWTYPE, IERR)`
	`INTEGER  COUNT, BLOCKLEN, STRIDE,`
	`  OLDTYPE, NEWTYPE, IERR`

举例

设 oldtype 的类型图为 {(double,0),(char,8)}，则调用 MPI_Type_vector(2,3,4,oldtype,newtype) 后，newtype 的类型图为：

{(double,0),(char,8), (double,16),(char,24), (double,32),(char,40),
 (double,64),(char,72), (double,80),(char,88), (double,96),(char,104)}

设 oldtype 的类型图为 {(double,0),(char,8)}，则调用 MPI_Type_vector(3,1,-2,oldtype,newtype) 后，newtype 的类型图是什么？

{(double,0),(char,8), (double,-32),(char,-24), (double,-64),(char,-56)}

举例

```
const int n=100;
MPI_Datatype newtype;
float A[n][n];
... ...
MPI_Type_vector(n, 1, n, MPI_FLOAT, &newtype);
MPI_Type_commit(newtype);
MPI_Send(A,1,newtype,dst,...)  // 发送的是哪些数据?
... ...
```

```
MPI_Type_vector(n-2, n-2, n, MPI_FLOAT, &newtype);
MPI_Type_commit(newtype);
MPI_Send(&A(2,2),1,newtype1,dst,...); // 发送的是哪些数据?
```

思考：怎样发送矩阵的对角线?

MPI_TYPE_CREATE_HVECTOR

MPI_TYPE_CREATE_HVECTOR(count, blocklen, stride, oldtype, newtype)

- 功能同 **MPI_TYPE_VECTOR**
- 唯一区别为这里的 **stride** 以字节为单位

C

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

F77

```
MPI_TYPE_CREATE_VECTOR(COUNT, BLOCKLEN, STRIDE,
                       OLDTYPE, NEWTYPE, IERR)
INTEGER  COUNT, BLOCKLEN, OLDTYPE, NEWTYPE, IERR
INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
```

MPI_TYPE_INDEXED

MPI_TYPE_INDEXED(count, array_of_blocklens, array_of_disps, oldtype, newtype)

创建索引数据类型：

新数据类型由 **count** 个数据块构成，第 **i** 个数据块包含 **array_of_bloklens(i)** 个连续存放的 **oldtype**，第 **i** 个数据块与首地址的偏移量(字节数)为 **array_of_disps(i)*extent(oldtype)**

- 是 **MPI_TYPE_VECTOR** 的扩展，区别是每个数据块的长度可以不同，数据块之间的间隔也可以不同。

MPI_TYPE_INDEXED

MPI_TYPE_INDEXED(count, array_of_blocklens, array_of_disps, oldtype, newtype)

C

```
int MPI_Type_indexed(int count,
     int array_of_blocklens[], int array_of_disps[],
     MPI_Datatype oldtype, MPI_Datatype *newtype)
```

F77

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENS,
                 ARRAY_OF_DISPS, OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, ARRAY_OF_BLOCKLENS(*),
        ARRAY_OF_DISPS(*), OLDTYPE, NEWTYPE, IERR
```

举例

设 oldtype 的类型图为 {(double,0),(char,8)}，则数组 p=(3,1), q=(4,0)，则调用 MPI_Type_indexed(2,p,q,oldtype,newtype) 后，newtype 的类型图为：

```
{ (double,64),(char,72), (double,80),(char,88),
  (double,96),(char,104), (double,0),(char,8) }
```

MPI_TYPE_CREATE_HINDEXED

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklens, array_of_disps, oldtype, newtype)

- 功能同 **MPI_TYPE_INDEXED**
- 唯一区别为这里的 **array_of_disps** 以字节为单位

C

```
int MPI_Type_create_hindexed(int count,
      const int array_of_blocklens[],
      const MPI_Aint array_of_disps[],
      MPI_Datatype oldtype, MPI_Datatype *newtype)
```

F77

```
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENS,
                ARRAY_OF_DISPS, OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, ARRAY_OF_BLOCKLENS(*),
        OLDTYPE, NEWTYPE, IERR
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPS(*)
```

MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklens, array_of_disps, array_of_types, newtype)

创建结构数据类型：

与 **MPI_TYPE_HINDEXED** 的区别在于每个数据块的数据类型可以不同。

这里的 **array_of_disps** 以字节为单位

- 该函数是最一般的新数据类型的构造函数，也是使用最广泛的一个，正确使用此函数在实际应用中非常重要。

MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklens, array_of_disps, array_of_types, newtype)

C

```
int MPI_Type_create_struct(int count,
     const int array_of_blocklens[],
     const MPI_Aint array_of_disps[],
     const MPI_Datatype array_of_types[],
     MPI_Datatype *newtype)
```

F77

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENS,
      ARRAY_OF_DISPS, ARRAY_OF_TYPES, NEWTYPE, IERR)
INTEGER COUNT, ARRAY_OF_BLOCKLENS(*),
        ARRAY_OF_TYPES(*), NEWTYPE, IERR
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPS(*)
```

目录页

Contents

4 数据的打包与解包

- 获取数据的地址
- 数据的打包
- 数据的解包

地址函数

MPI_GET_ADDRESS(location, address)

返回指定变量在内存中的“绝对” 地址（相对于地址 **MPI_BOTTOM**）

C	`int MPI_Address(const void *location,` `  MPI_Aint *address)`

F77	`MPI_ADDRESS(LOCATION, ADDRESS, IERR)`
	`<type>  LOCATION(*)`
	`INTEGER  IERR`
	`INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS`

地址函数

MPI_AINT_DIFF(addr1, addr2)

返回两个地址之间的差（MPI 4.0）

C	`MPI_Aint MPI_Aint_diff(MPI_Aint *addr1,` `  MPI_Aint *addr2)`

F77	`INTEGER(KIND=MPI_ADDRESS_KIND)`
	`  MPI_AINT_DIFF(ADDR1, ADDR2)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2`

地址函数

MPI_AINT_ADD(base, disp)

在原有地址的基础上加一个位移（**MPI 4.0**）

C	`MPI_Aint MPI_Aint_add(MPI_Aint base,` `  MPI_Aint disp)`

F77	`INTEGER(KIND=MPI_ADDRESS_KIND)`
	`  MPI_AINT_ADD(BASE, DISP)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP`

举例

MPI_get_address01.c

```
const int n=100;
float A[n][n];
MPI_Aint i1, i2;
... ...
MPI_Get_address(&A[0][0], &i1);
MPI_Get_address(&A[10][10], &i2);
diff=MPI_Aint_diff(i2,i1);
... ...
```

数据的打包

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

- 将缓冲区 **inbuf** 中的 **incount** 个类型为 **datatype** 的数据进行打包，打包后的数据放在缓冲区 **outbuf** 中。**outsize** 给出的是 **outbuf** 的总长度（字节数），**comm** 是发送打包数据时将使用的通信器。
- **position** 是打包缓冲区中的位移，每次打包第一次调用 **MPI_PACK** 时用户应该将其置为 **0**，随后 **MPI_PACK** 将自动修改它，使得它总是指向打包缓冲区中尚未使用部分的起始位置。每次调用 **MPI_PACK** 后的 **position** 实际上就是已打包数据的总长度。通过连续几次对不同位置的数据进行打包，就可以将不连续的数据放到一个连续的空间中。

数据的解包

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

- 是 **MPI_PACK** 的逆操作：它从 **inbuf** 中拆包 **outcount** 个类型为 **datatype** 的数据到 **outbuf** 中。
- 函数的各项参数与 **MPI_PACK** 类似，只不过这里的 **inbuf** 和 **insize** 对应于 **MPI_PACK** 中的 **outbuf** 和 **outsize**，而 **outbuf** 和 **outcount** 则对应于 **MPI_PACK** 中的 **inbuf** 和 **incount**

谢 谢

THANK YOU