Пересылка локализованных разнородных данных.

Если требуется переслать структуру типа:

```
typedef struct{
 int n;
 char s[20];
 double v;
}USERTYPE;
```

Возможно использовать три решения:
1) Один раз создать новый `MPI` тип, соответсвующий C типу `USERTYPE`, использовать его в функциях обмена.
2) Запаковывать объект типа `USERTYPE` в один объект типа `MPI_PACKED` и пересылать его процессу-получателю, которы его распаковывает.
3) Рассматривать структуру как набор байтов типа `MPI_BYTE`, длины `sizeof(USERTYPE)` (решение работает только на однородных вычислительных установках).

Создание нового типа.

При создании нового типа MPI необходима информация о размещении данных. Её можно передать в виде:

```
n, {(c0, d0, t0),  (c1, d1, t1), … , (cn-1,dn-1, tn-1)}
```

где
n(int) - количество элементов базовых типов в новом типе,
ci(int) - количество элементов типа ti, в i-м базовом элементе нового типа,
di(тип MPI_Aint «address int»), смещение i-го базового элемента от начала объекта,
ti(MPI_Datatype) - тип i-го базового элемента.

Для типа USERTYPE спецификация MPI-типа на 32-битной вычислительной установке будет иметь вид:

```
typedef struct{
 int n;
 char s[20];
 double v;
}USERTYPE;

{(1,0,MPI_INT),(20,4,MPI_CHAR),
(1,24,MPI_DOUBLE)}
```

Использование типа `MPI_Aint` (равного `int` или `long int`) позволяет не зависеть от выбранного языке Си размера типа `int` на 64-битных компьютерах. Получить адрес любого объекта в терминах типа `MPI_Aint` можно с помощью функции `MPI_Address`. Функция `MPI_Address` не связанна с межпроцессорным взаимодействием и вызывается локально:

```
int MPI_Address( void *location, MPI_Aint
*address);
```

Входные параметры:

`location` – адрес объекта в терминах Си

Выходной параметр:

`address` – адрес объекта как `MPI_Aint`

```
(MPI_Address was removed in MPI-3.0.  Use
MPI_Get_address instead.)
```

Построением нового типа занимается функция MPI_Type_struct. Она не связанна с межпроцессорным взаимодействием и вызывается локально:

```
int MPI_Type_struct (int count, int blocklens[],
MPI_Aint displacement[], MPI_Datatype oldtypes[],
MPI_Datatype *newtype);
```

Входные данные:

`count -` кол-во элементов в новом типе `(n)`

`blocklens -` массив длины count, содержащий количество элементов базового типа `{c0, c1,…}`

`displacement -` массив длины `count`, смещения элементов базовых типов `{d0, d1, …}`

`oldtypes -` массив длины count, содержит типы базовых элементов нового типа `{t0, t1, …}`

Выходные параметры:

`newtype -` идентификатор нового типа

(MPI_Type_struct was removed in MPI-3.0. Use MPI_Type_create_struct instead.)

Созданный тип нельзя сразу же использовать для пересылки. Перед этим необходимо вызвать функцию `MPI_Type_commit,` которая создает для указанного типа все структуры данных для эффективной пересылки:

```
int MPI_Type_commit(MPI_Datatype *newtype);
```

Эту функцию необходимо вызвать всеми процессами, которые будут обмениваться типами `newtype.`

Когда построенный тип больше не нужен для дальнейшей работы программы, то необходимо освободить все выделенные в момент его построения ресурсы с помощью функции MPI_Type_free:

```
int MPI_Type_free( MPI_Datatype *newtype);
```

Где единственный входной аргумент это построенный тип даных, после вызова функции тип становится равным константе MPI_DATATYPE_NULL. Если в качестве аргумента этой функции передать стандартный MPI тип, то это приведет к ошибке.

```
int main(void){

    MPI_Init(NULL, NULL);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    USERTYPE t = {1, {'h','e','l','l','o'}, 2.0};
    int t_c[3] = {1, 20, 1};
    MPI_Aint t_d[3];
    MPI_Datatype t_t[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE};
    MPI_Datatype MPI_USERTYPE;
    MPI_Aint start_address, address;
    MPI_Get_address(&t, &start_address);//адрес смещения начала
    MPI_Get_address(&(t.n), &address);
    t_d[0] = address - start_address;
    MPI_Get_address(&(t.s), &address);
    t_d[1] = address - start_address;
    MPI_Get_address(&(t.v), &address);
    t_d[2] = address - start_address;
    MPI_Type_create_struct(3, t_c, t_d, t_t, &MPI_USERTYPE);
    MPI_Type_commit(&MPI_USERTYPE);

    if(rank==0){
        t.n = 777;
        t.s[5] = '\0';
        t.v = 123.456;
        MPI_Send(&t, 1, MPI_USERTYPE, 1, 0, MPI_COMM_WORLD);
    }
    else{
        MPI_Recv(&t, 1, MPI_USERTYPE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Recv %d n=%d %s %lf\n", rank, t.n, t.s, t.v );
    }

    MPI_Finalize();
}
```

Процедура создания нового типа длительна и может быть оправдана в том случае, если этот тип будет использоваться многократно.

Запаковка и распаковка данных.

Запаковка и распаковка данных может применяться к любым данным, необязательно локализованным в одном блоке памяти. При запаковке данные записываются подряд в указанном пользователем буфере, а затем пересылаются как один объект типа MPI_PACKED. При распаковке данные последовательно извлекаются из буфера.

Запаковать данные можно процедурой MPI_Pack.

```
int MPI_Pack( void *pack_data, int pack_count,
MPI_Datatype pack_type, void *buffer, int
buffer_size, int *position, MPI_Comm comm);
```

Входные параметры:

`pack_data` – адрес пакуемых данных,

`pack_count` – кол-во пакуемых данных,

`pack_type` – тип пакуемых `pack_count` данных,

`buffer` – буфер, куда будет складываться результат,

`buffer_size` – размер буфера,

`position` – адрес целого числа, задающего смещение в байтах первой позиции в `buffer.`

Выходные параметры:

`buffer` – буфер с готовым результатом,

`position` – указывает уже на новое значение первой свободной позиции в буфере.

Узнать размер буфера, необходимого для запаковки запаковки данных, можно с помощью функции `MPI_Pack_size:`

```
int MPI_Pack_size( int pack_count,
MPI_Datatype pack_type, MPI_Comm comm, int
*size);
```

Входные параметры:

`pack_count -` кол-во пакуем элементов

`pack_type -` тип каждого из пакуемых `pack_count` данных,

Выходные параметры:

`size -` размер буфера, достаточных для запаковки `pack_count` элементов типа `pack_type.`

Запакованные данные можно переслать и получить любой функцией передачи сообщений, указав в качестве типа `MPI_PACKED`, а в качестве размера `buffer_size`.

Распакован данные можно с помощью процедуры `MPI_Unpack:`

```
int MPI_Unpack( void *buffer, int buffer_size, int
*position, void *unpack_data, int unpack_count,
MPI_Datatype unpack_type, MPI_Comm comm);
```

Входные данные:

`buffer –` адрес буфера с запакованными данными

`buffer_size –` размер буфера

`position –` адрес целого числа, задающего смещение в байтах в позиции `buffer`, с которого нужно начать распаковку

`unpack_data –` адрес буфера, где следует расположить распакованные данные

`unpack_count –` кол-во распакованных элементов

`unpack_type –` тип каждого из `unpack_count` распаковываемых данных

Выходные данные:

unpack_data - распакованные данные

position - указывает уже на новое значение позиции в буфере

```
#define BUF_SIZE 128
char buffer[BUF_SIZE];
int position;
USERTYPE t;

int main(void){

    int rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank == 0){

        strcpy(t.s, "Hello");
        t.n = 777;
        t.v = 123.456;

        position = 0;
        MPI_Pack(&t.n, 1, MPI_INT, buffer, BUF_SIZE, &position, MPI_COMM_WORLD);
        MPI_Pack(&t.s, 20, MPI_CHAR, buffer, BUF_SIZE, &position, MPI_COMM_WORLD);
        MPI_Pack(&t.v, 1, MPI_DOUBLE, buffer, BUF_SIZE, &position, MPI_COMM_WORLD);

        MPI_Send(&buffer, BUF_SIZE, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
    }
    else{
        position = 0;

        MPI_Recv(&buffer, BUF_SIZE, MPI_PACKED, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        MPI_Unpack(buffer, BUF_SIZE, &position, &t.n, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, BUF_SIZE, &position, &t.s, 20, MPI_CHAR, MPI_COMM_WORLD);
        MPI_Unpack(buffer, BUF_SIZE, &position, &t.v, 1, MPI_DOUBLE, MPI_COMM_WORLD);

        printf("Recv %d n=%d %s %lf\n", rank, t.n, t.s, t.v );
    }

    MPI_Finalize();
}
```