# *Системы и средства параллельного программирования*

3 курс кафедры СКИ
сентябрь – декабрь 2016 г.

Лектор доцент Н.Н.Попова

---

Лекция 8
07 ноября 2016 г.

# Тема

- Производные типы данных

# Производные типы данных MPI

Назначение:

- пересылка данных, расположенных в несмежных областях памяти в одном сообщении;
- пересылка разнотипных данных в одном сообщении;
- облегчение понимания программы;

# Производные типы данных

- не могут использоваться ни в каких операциях, кроме коммуникационных
- производные ТД следует понимать как описатели расположения в памяти элементов базовых типов
- производный ТД - скрытый объект
- отображение типа:

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

# СЦЕНАРИЙ РАБОТЫ С ПРОИЗВОДНЫМИ ТИПАМИ

- Создание типа с помощью конструктора.
- Регистрация.
- Использование.
- Освобождение памяти.

# Стандартный сценарий создания и использования производного ТД

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct *MPI_Type_subarray* *MPI_Type_darray*
- Новый производный тип регистрируется вызовом функции MPI_Type_commit
- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free

# Производные типы данных.

Typemap = {($type_0$, $disp_0$), ... , ($type_{n-1}$, $disp_{n-1}$)}

- отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена (состоит из n элементов)
- i-й элемент имеет адрес buf+$disp_i$ и базовый тип $type_i$

# Typemaps, продолжение

- Дополнительные определения
  - *lower_bound(Typemap)* = min $disp_j$ , j = 0, …, n-1
  - *upper_bound(Typemap)* = max($disp_j$ + sizeof($type_j$)) + ε
  - *extent(Typemap)* = *upper_bound(Typemap)* - *lower_bound(Typemap)*

# Характеристики ТД в MPI

- Протяженность (кол-во байт, которое переменная данного типа занимает в памяти)
  - MPI_Type_extent
- Размер (кол-во реально передаваемых байт в коммуникационных операциях)
  - MPI_Type_size

# MPI_Type_extent

```
int MPI_Type_extent(MPI_Datatype datatype,
    MPI_Aint *extent)
```

datatype - тип данных

extent - протяженность элемента заданного типа

# MPY_Type_size

int MPI_Type_size(MPI_Datatype datatype, int *size)

datatype - тип данных

size - размер элемента заданного типа

# Дополнительные функции

- *MPI_Type_get_extent* (MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)
  - datatype (datatype you are querying)
  - lb (lower bound of datatype)
  - extent (extent of datatype)
- Возвращает нижнюю границу и extent заданного типа.

# MPI_BOTTOM, MPI_Get_address

- **MPI_BOTTOM**

  - нулевой адрес

- **MPI_Get_address**

  - возвращает адрес относительно MPI_BOTTOM

  - обеспечивает переносимость (нельзя использовать “&” операцию C)

# MPI_Type_commit

- Каждый конструктор возвращает незарегистрированный (*uncommited)* тип. Процесс  commit можно представить себе как процесс компиляции типа во внутренне представление.
- Должен вызываться *MPI_Type_commit* (&datatype).
- После регистрации тип может повторно использоваться.
- Повторный вызов commit не имеет эффекта.

# MPI_Type_commit

- int MPI_Type_commit(MPI_Datatype *datatype)

**datatype -** новый производный тип данных

# MPI_Type_contiguous

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

**count -** число элементов базового типа

**oldtype -** базовый тип данных

**newtype -** новый производный тип данных

# MPI_Type_contiguous

oldtype = MPI_REAL

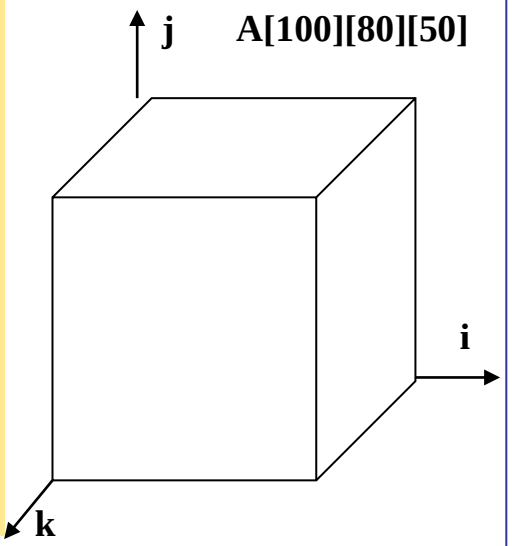
count = 4

newtype

# MPI_Type_contiguous. Пример 1

**Surface: A[0][:][:]**

```
MPI_Datatype face_jk;
MPI_Type_contiguous(80*50, MPI_DOUBLE,
&face_jk);
MPI_Type_commit(&face_jk);
MPI_Send(&A[0][0][0],1,face_jk,rank,tag,comm);
//
MPI_Send(&A[0][0][0],80*50,MPI_DOUBLE,rank,tag,
comm);
MPI_Send(&A[99][0][0],1,face_jk,rank,tag,comm);
...
MPI_Type_free(&face_jk);
```



# MPI_Type_contiguous – пример 2

```
/* create a type which describes a line of ghost cells */
/* buf[0..n-1] set to ghost cells */
int n;
MPI_Datatype ghosts;

MPI_Type_contiguous (n, MPI_DOUBLE, &ghosts);
MPI_Type_commit(&ghosts)
MPI_Send (buf, 1, ghosts, dest,  tag, MPI_COMM_WORLD);
..
..
MPI_Type_free(&ghosts);
```

# MPI_Type_vector

oldtype = MPI_REAL 

count = 3, blocklenght = 2, stride = 3

newtype

# MPI_Type_vector

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

***count*** **-** число блоков

***blocklength*** **-** число элементов базового типа в каждом блоке

***stride*** - шаг между началами соседних блоков, измеренный числом элементов базового типа

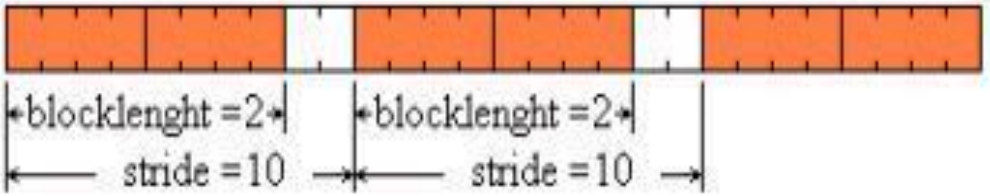
***oldtype*** **-** базовый тип данных

***newtype*** **-** новый производный тип данных**.**

# MPI_Type_hvector

oldtype = MPI_REAL 

count = 3, blocklenght = 2, stride = 10

newtype 

# MPI_Type_hvector

- int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

**count -** число блоков
**blocklength -** число элементов базового типа в каждом блоке
**stride -** шаг между началами соседних блоков в байтах
**oldtype -** базовый тип данных
**newtype -** новый производный тип данных.

**Отличие от MPI_Type_vector: stride определяется в байтах, не в элементах (‘h’ – heterogeneous)**

# Пример 1

```
double A[4][4];
MPI_Datatype column;

MPI_Type_hvector(4,1,4*sizeof(double),
                 MPI_DOUBLE, &column);
MPI_Type_commit(&column);
MPI_Send(&A[0][1],1,column,rank,tag,comm);
...
```

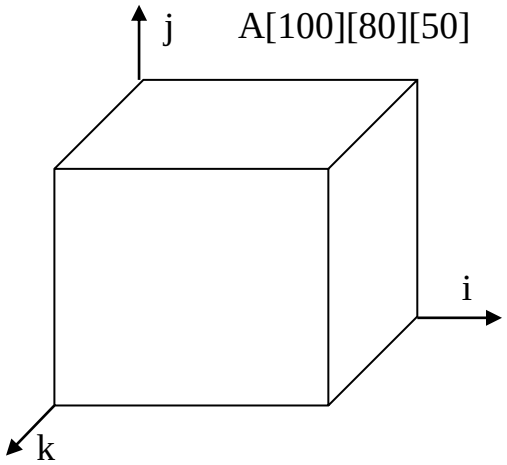
| | | | |
|---|---|---|---|
| 1.0 | 2.0 | 3.0 | 4.0 |
| 5.0 | 6.0 | 7.0 | 8.0 |
| 9.0 | 10.0 | 11.0 | 12.0 |
| 13.0 | 14.0 | 15.0 | 16.0 |

A[4][4]

Surface: A[:][:][49]

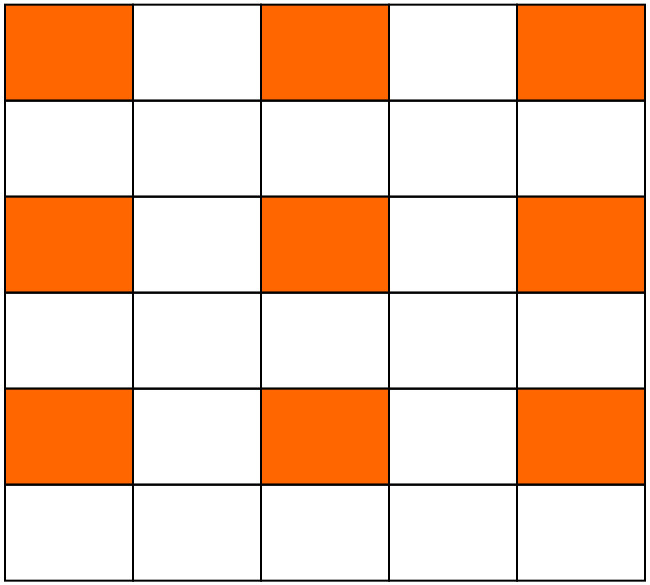
```
double A[100][80][50];
MPI_Datatype face_ij, line_j;

MPI_Type_vector(80,1,50,MPI_DOUBLE,&line_j);
MPI_Type_hvector(100,1,80*50*sizeof(double),
                 line_j, &face_ij);
MPI_Type_commit(&face_ij);
MPI_Send(&A[0][0][49],1,face_ij,rank,tag,comm);
...
```



# Пример 2 – рассылка клеточной структуры

Используется MPI_type_vector и MPI_Type_create_hvector вместе для рассылки отмеченных клеток:



# Пример 2, продолжение.

```
double a[6][5], e[3][3];
MPI_Datatype oneslice, twoslice
MPI_Aint lb, sz_dbl
int myrank, ierr

MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
MPI_Type_get_extent (MPI_DOUBLE, &lb, &sz_dbl);
MPI_Type_vector (3,1,2,MPI_DOUBLE, &oneslice);
MPI_Type_create_hvector (3,1,12*sz_dbl, oneslice, &twoslice);
MPI_Type_commit (&twoslice);
```

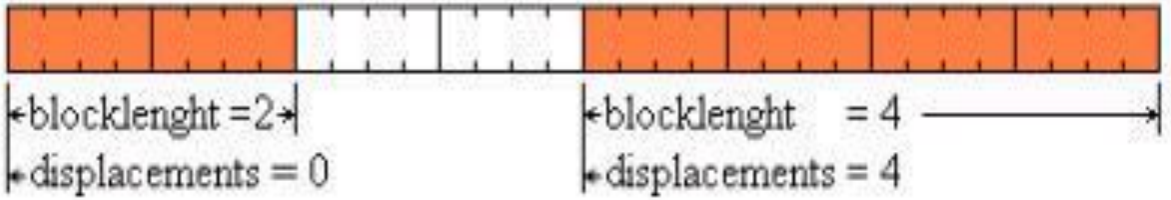
# Пример – транспонирование матрицы

```
double a[100][100], b[100][100];
int myrank;
MPI_Status *status;
MPI_Datatype row,matr
MPI_Aint  lb, sz_dbl
MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
MPI_Type_get_extent (MPI_DOUBLE, &lb, &sz_dbl);
MPI_Type_vector (100, 1, 100, MPI_DOUBLE, &row);
MPI_Type_create_hvector (100, 1, 100*sz_dbl, col, &matr);
MPI_Type_commit (&matr);
MPI_Sendrecv (&a[0][0], 1, matr, myrank, 0, &b[0][0], 100*100,
                    MPI_DOUBLE, myrank, 0, MPI_COMM_WORLD,
                    &status);
```

# MPI_Type_indexed

oldtype = MPI_REAL 

count = 2 blocklenght = (2, 4) displacements = (0,4)

newtype 

# MPI_Type_indexed

```
int MPI_Type_indexed(int count, int *array_of_blocklengths, int
    *array_of_displacements, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

***count* -** число блоков

***array_of_blocklengths* -** массив, содержащий число элементов базового типа в каждом блоке

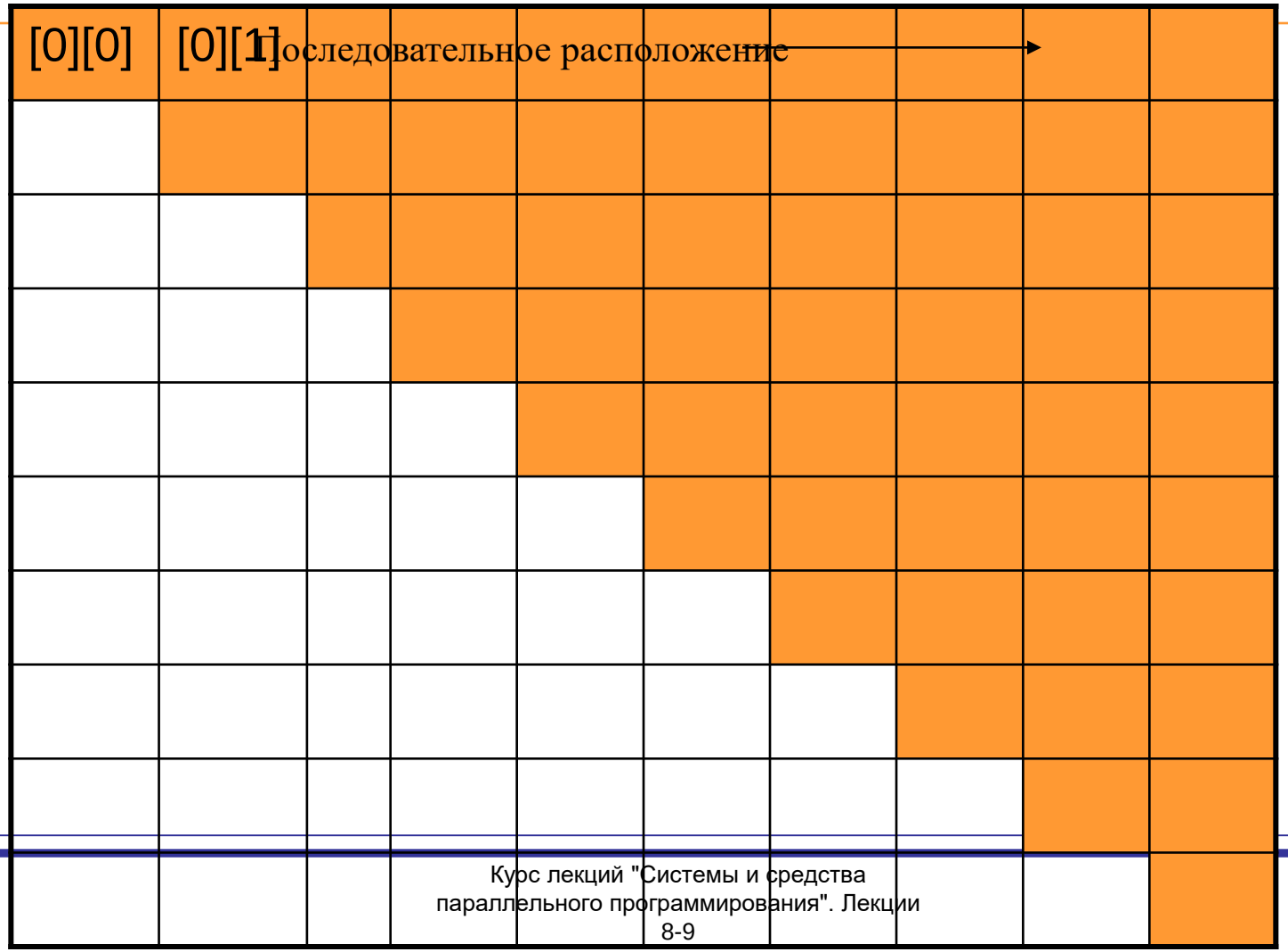
**array_of_displacements -** массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются числом элементов базового типа

**oldtype -** базовый тип данных

**newtype -** новый производный тип данных

- Смещения между последовательными блоками не обязательно должны совпадать. Это позволяет выполнять пересылку данных одним вызовом.

# Пример: верхнетреугольная матрица

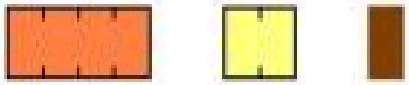


# Пересылка верхнетреугольной матрицы

```
double a[100][100];
Int disp[100], blocklen[100], i, dest, tag;
MPI_Datatype upper;
/* compute start and size of each row */
for (i = 0; i < 100; ++i){
  disp[i] = 100*i + i;
  blocklen[i] = 100 – i;
}
MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit(&upper);
MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
```

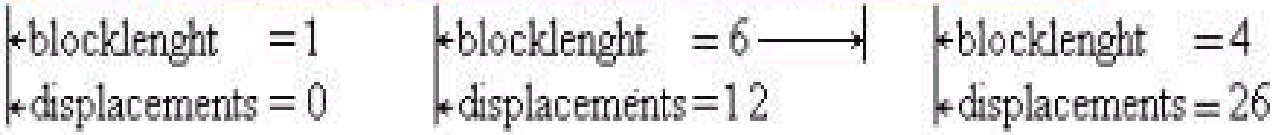
# MPI_Type_struct

oldtypes = (MPI_INT, MPI_SHORT, MPI_CHAR)



count = 3 blocklenght = (1, 6, 4) displacements = (0, 12, 26)

newtype



# MPI_Type_struct

int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

***count*** **-** число блоков;

***array_of_blocklength*** **-** массив, содержащий число элементов одного из базовых типов в каждом блоке;

***array_of_displacements*** **-** массив смещений каждого блока от начала размещения структуры, смещения измеряются в байтах;

***array_of_type*** **-** массив, содержащий тип элементов в каждом блоке;

***newtype*** **-** новый производный тип данных**.**

Наиболее общий конструктор.

# Пример

```
struct _tagStudent {
  int id;
  double grade;
  char note[100];
};
struct _tagStudent Students[25];
MPI_Datatype one_student, all_students;
int block_len[3];
MPI_Datatype types[3];
MPI_Aint disp[3];
block_len[0] = block_len[1] = 1;
block_len[2] = 100;
types[0] = MPI_INT;
types[1] = MPI_DOUBLE;
types[2] = MPI_CHAR;
MPI_Address(&Students[0].id, &disp[0]); // memory address
MPI_Address(&Students[0].grade, &disp[1]);
MPI_Address(&Students[0].note[0],&disp[2]);
disp[1] = disp[1]-disp[0];
disp[2] = disp[2]-disp[0];
disp[0] = 0;
MPI_Type_struct(3, block_len, disp, types, &one_student);
MPI_Type_contiguous(25, one_student, &all_students);
MPI_Type_commit(&all_students);
MPI_Send(Students, 1, all_students, rank, tag, comm);
// MPI_Type_commit(&one_student);
// MPI_Send(Students, 25, one_student, rank, tag, comm);
...
```

# Упаковка данных

- Упаковка / распаковка:
  - MPI_Pack
  - MPI_Unpack
- Определение размера буфера для упаковки:
  - MPI_Pack_size

# MPI_Pack

int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype,
void *outbuf, int outsize, int *position, MPI_Comm comm)

**IN inbuf - адрес начала области памяти с элементами, которые требуется упаковать;**

**IN incount - число упаковываемых элементов;**
**IN datatype - тип упаковываемых элементов;**
**OUT outbuf - адрес начала выходного буфера для упакованных данных;**
**IN outsize - размер выходного буфера в байтах;**
**INOUT position -текущая позиция в выходном буфере в байтах;**
**IN comm - коммуникатор.**

# MPI_Unpack

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

**IN inbuf - адрес начала входного буфера с упакованными данными;**

**IN insize - размер входного буфера в байтах;**

**INOUT position - текущая позиция во входном буфере в байтах;**

**OUT outbuf - адрес начала области памяти для размещения распакованных элементо**

**IN outcount - число извлекаемых элементов;**

**IN datatype - тип извлекаемых элементов;**

**IN comm - коммуникатор.**

# Пересылка элементов разного типа

- элементы нужно предварительно запаковать в один массив, последовательно обращаясь к функции упаковки MPI_Pack
- при первом вызове функции упаковки параметр position, как правило, устанавливается в 0, чтобы упакованное представление размещалось с начала буфера
- для непрерывного заполнения буфера необходимо в каждом последующем вызове использовать значение параметра position, полученное из предыдущего вызова
- распаковывать - аналогично

# MPI_Pack_size

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

incount - число элементов, подлежащих упаковке

datatype - тип элементов, подлежащих упаковке

comm - коммуникатор

size - размер сообщения в байтах после его упаковки

# Пример рассылки разнотипных данных из 0-го процесса с использованием функций MPI_Pack и MPI_Unpack

```
char buff[100];
double x, y;
int position, a[2];
{
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{  /* Упаковка данных*/
position = 0;
MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
MPI_Pack(a, 2, MPI_INT, buff, 100, &position, MPI_COMM_WORLD);
}
   /* Рассылка упакованного сообщения */
MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD);
   /* Распаковка сообщения во всех процессах */
if (myrank != 0)
position = 0;
MPI_Unpack(buff, 100, &position, &x, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buff, 100, &position, &y, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buff, 100, &position, a,  2, MPI_INT, MPI_COMM_WORLD);
}
```