

*Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики*

*СУПЕРКОМПЬЮТЕРЫ И
ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ
(лекция 11)*

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2024

MPI

```
int MPI_Sendrecv(void *sbuf, int  
scount, MPI_Datatype stype, int  
dest, int stag, void *rbuf, int  
rcount, MPI_Datatype rtype, int  
source, int rtag, MPI_Comm comm,  
MPI_Status *status)
```

MPI

Совмещённые приём и передача сообщений с блокировкой. Буферы передачи и приёма не должны пересекаться. Тупиковой ситуации не возникает!

Сообщение, отправленное операцией **MPI_Sendrecv**, может быть принято обычным образом, и операция **MPI_Sendrecv** может принять сообщение, отправленное обычной операцией.

MPI

```
int MPI_Sendrecv_replace(void  
*buf, int count, MPI_Datatype  
datatype, int dest, int stag,  
int source, int rtag, MPI_comm  
comm, MPI_Status *status)
```

Совмещённые приём и передача сообщений с блокировкой через общий буфер **buf**.

MPI

Обмен по кольцевой топологии при помощи процедуры **MPI_Sendrecv**:

```
prev = rank - 1;  
next = rank + 1;  
if (rank == 0) prev = size - 1;  
if (rank == (size - 1)) next = 0;  
MPI_Sendrecv(&sbuf[0], 1, MPI_FLOAT, prev,  
tag2, &rbuf[0], 1, MPI_FLOAT, next, tag2,  
MPI_COMM_WORLD, &status1);  
MPI_Sendrecv(&sbuf[1], 1, MPI_FLOAT, next,  
tag1, &rbuf[1], 1, MPI_FLOAT, prev, tag1,  
MPI_COMM_WORLD, &status2);
```

MPI

Специальное значение **MPI_PROC_NULL** для несуществующего процесса. Операции с таким процессом завершаются немедленно с кодом завершения **MPI_SUCCESS**.

```
next = rank + 1;  
if(rank == (size - 1)) next=MPI_PROC_NULL;  
MPI_Send (&buf, 1, MPI_FLOAT, next, tag,  
MPI_COMM_WORLD);
```

МРІ

Коллективные взаимодействия процессов

MPI

В операциях коллективного взаимодействия процессов *участвуют все процессы коммуникатора!*

Как и для блокирующих процедур, возврат означает то, что разрешён свободный доступ к буферу приёма или отправки.

Сообщения, вызванные коллективными операциями, не пересекутся с другими сообщениями.

MPI

Нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций (кроме процедуры **`MPI_Barrier`**).

Если какой-то процесс завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммутатора, ни даже того, что она ими начата (если это возможно по смыслу операции).

MPI

В коллективных операциях не используются идентификаторы сообщений (теги). Таким образом, коллективные операции строго упорядочены согласно их появлению в тексте программы. Несоблюдение такого порядка может привести к возникновению тупиковых ситуаций.

MPI

```
int MPI_Barrier(MPI_Comm comm)
```

Работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора **comm** не выполнят эту процедуру. Все процессы должны вызвать **MPI_Barrier**, хотя реально исполненные различными процессами коммуникатора вызовы могут быть расположены в разных местах программы.

MPI

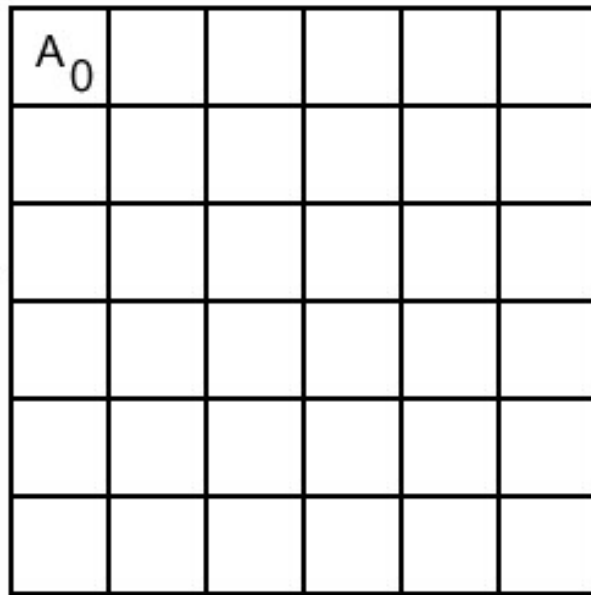
```
int MPI_Bcast(void *buf, int  
count, MPI_Datatype datatype,  
int root, MPI_Comm comm)
```

Рассылка **count** элементов данных типа **datatype** из массива **buf** от процесса **root** всем процессам данного коммуникатора **comm**, включая сам рассылающий процесс. Значения параметров **count**, **datatype**, **root** и **comm** должны быть одинаковыми у всех процессов.

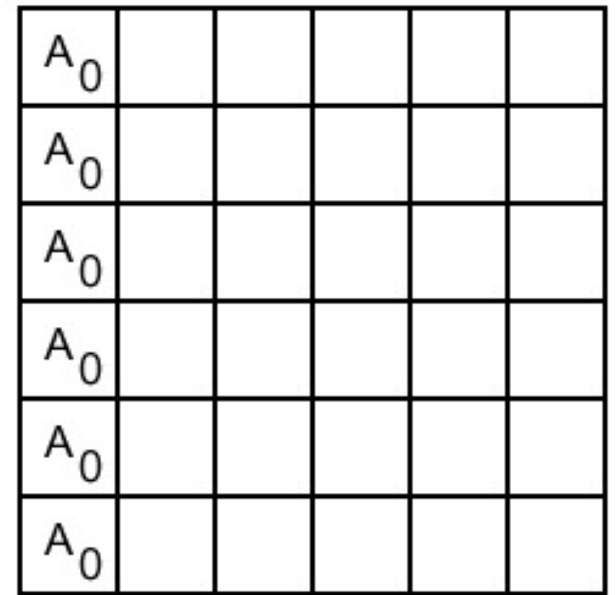
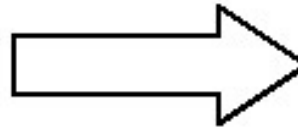
MPI

данные

процессы



broadcast



MPI

Например, для пересылки от процесса 2 всем остальным процессам приложения массива **buf** из 100 целочисленных элементов, нужно, чтобы во всех процессах встретился следующий вызов:

```
MPI_Bcast(buf, 100, MPI_INT, 2,  
MPI_COMM_WORLD);
```

MPI

```
int MPI_Gather(void *sbuf, int  
scount, MPI_Datatype stype, void  
*rbuf, int rcount, MPI_Datatype  
rtype, int root, MPI_Comm comm)
```

Сборка **scount** элементов данных типа **stype** из массивов **sbuf** со всех процессов коммуникатора **comm** в буфер **rbuf** процесса **root**. Данные сохраняются в порядке возрастания номеров процессов.

MPI

На процессе **root** существенными являются значения всех параметров, а на остальных процессах - только значения параметров **sbuf**, **scount**, **stype**, **root** и **comm**. Значения параметров **root** и **comm** должны быть одинаковыми у всех процессов. Параметр **rcount** у процесса **root** обозначает число элементов типа **rtype**, принимаемых от каждого процесса.

MPI

Если для отправки и приема данных должен использоваться один и тот же буфер, то на месте аргумента **sbuf** процесса **root** можно указать значение **MPI_IN_PLACE**. В этом случае аргументы **scount** и **stype** игнорируются, и предполагается, что порция данных процесса **root** уже расположена в соответствующем месте буфера приема **rbuf**.

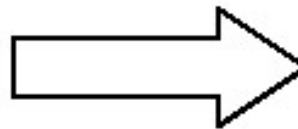
MPI

данные

процессы

A_0	A_1	A_2	A_3	A_4	A_5

scatter



gather



A_0					
A_1					
A_2					
A_3					
A_4					
A_5					

MPI

Например, чтобы процесс **2** собрал в массив **rbuf** по **10** целочисленных элементов массивов **buf** со всех процессов приложения, нужно, чтобы во всех процессах встретился следующий вызов:

```
MPI_Gather(buf, 10, MPI_INT,  
rbuf, 10, MPI_INT, 2,  
MPI_COMM_WORLD);
```

MPI

```
int MPI_Gatherv(void *sbuf, int  
scount, MPI_Datatype stype, void  
*rbuf, int *rcounts, int  
*displs, MPI_Datatype rtype, int  
root, MPI_Comm comm)
```

Сборка различного количества данных из массивов **sbuf**. Порядок расположения данных в результирующем буфере **rbuf** задаёт массив **displs**.

MPI

rcounts – целочисленный массив, содержащий количество элементов, передаваемых от каждого процесса (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

displs – целочисленный массив, содержащий смещения относительно начала массива **rbuf** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

MPI

```
int MPI_Scatter(void *sbuf, int  
scount, MPI_Datatype stype, void  
*rbuf, int rcount, MPI_Datatype  
rtype, int root, MPI_Comm comm)
```

Рассылка по **scount** элементов данных типа **stype** из массива **sbuf** процесса **root** в массивы **rbuf** всех процессов коммуникатора **comm**, включая сам процесс **root**. Данные рассылаются в порядке возрастания номеров процессов.

MPI

На процессе **root** существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров **rbuf**, **rcount**, **rtype**, **source** и **comm**. Значения параметров **source** и **comm** должны быть одинаковыми у всех процессов.

MPI

Если для отправки и приема данных должен использоваться один буфер, то на месте аргумента **rbuf** процесса **root** можно указать значение **MPI_IN_PLACE**. В этом случае аргументы **rcount** и **rtype** игнорируются, и предполагается, что порция данных процесса **root** не пересылается.

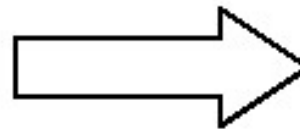
MPI

данные

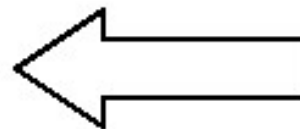
процессы

A_0	A_1	A_2	A_3	A_4	A_5

scatter



gather



A_0					
A_1					
A_2					
A_3					
A_4					
A_5					

MPI

```
float sbuf[SIZE][SIZE], rbuf[SIZE];
if(rank == 0)
    for(i=0; i<SIZE; i++)
        for (j=0; j<SIZE; j++)
            sbuf[i][j]=...;
if (numtasks == SIZE)
    MPI_Scatter(sbuf, SIZE, MPI_FLOAT, rbuf,
SIZE, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

MPI

```
int MPI_Scatterv(void *sbuf, int  
*scounts, int *displs,  
MPI_Datatype stype, void *rbuf,  
int rcount, MPI_Datatype rtype,  
int root, MPI_Comm comm)
```

Рассылка различного количества данных из массива **sbuf**. Начало рассылаемых порций задает массив **displs**.

MPI

counts – целочисленный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

displs – целочисленный массив, содержащий смещения относительно начала массива **sbuf** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

MPI

```
int MPI_Allgather(void *sbuf,  
int scount, MPI_Datatype stype,  
void *rbuf, int rcount,  
MPI_Datatype rtype, MPI_Comm  
comm)
```

Сборка данных из массивов **sbuf** со всех процессов коммуникатора **comm** в буфере **rbuf** каждого процесса. Данные сохраняются в порядке возрастания номеров процессов.

MPI

Если для отправки и приема данных должен использоваться один буфер, то на месте аргумента **sbuf** всех процессов можно указать значение **MPI_IN_PLACE**. В этом случае аргументы **scount** и **stype** игнорируются, и предполагается, что порции исходных данных всех процессов уже расположены в соответствующих местах буферов приема **rbuf**.

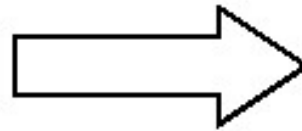
MPI

данные

процессы

A_0					
B_0					
C_0					
D_0					
E_0					
F_0					

allgather

[illegible]

MPI

```
int MPI_Allgatherv(void *sbuf,  
int scount, MPI_Datatype stype,  
void *rbuf, int *rcounts, int  
*displs, MPI_Datatype rtype,  
MPI_Comm comm)
```

Сборка на всех процессах различного количества данных из **sbuf**. Порядок расположения данных в массиве **rbuf** задаёт массив **displs**.

MPI

```
int MPI_Alltoall(void *sbuf, int  
scount, MPI_Datatype stype, void  
*rbuf, int rcount, MPI_Datatype  
rtype, MPI_Comm comm)
```

Рассылка каждым процессом коммуникатора **comm** различных порций данных всем другим процессам. **j**-й блок массива **sbuf** процесса **i** попадает в **i**-й блок массива **rbuf** процесса **j**.

MPI

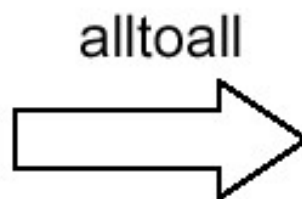
Если для отправки и приема должен использоваться один буфер, то на месте аргумента **sbuf** всех процессов можно указать значение **MPI_IN_PLACE**. В этом случае аргументы **scount** и **stype** игнорируются, и предполагается, что порции исходных данных всех процессов уже расположены в соответствующих местах буферов приема **rbuf**.

MPI

данные

процессы

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
B ₀	B ₁	B ₂	B ₃	B ₄	B ₅
C ₀	C ₁	C ₂	C ₃	C ₄	C ₅
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅
E ₀	E ₁	E ₂	E ₃	E ₄	E ₅
F ₀	F ₁	F ₂	F ₃	F ₄	F ₅



A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₁	B ₁	C ₁	D ₁	E ₁	F ₁
A ₂	B ₂	C ₂	D ₂	E ₂	F ₂
A ₃	B ₃	C ₃	D ₃	E ₃	F ₃
A ₄	B ₄	C ₄	D ₄	E ₄	F ₄
A ₅	B ₅	C ₅	D ₅	E ₅	F ₅

MPI

```
int MPI_Alltoallv(void* sbuf,  
int *scounts, int *sdispls,  
MPI_Datatype stype, void* rbuf,  
int *rcounts, int *rdispls,  
MPI_Datatype rtype, MPI_Comm  
comm)
```

Рассылка со всех процессов коммуникатора **comm** различного количества данных всем другим процессам. Размещение данных в буфере **sbuf** отсылающего процесса определяется массивом **sdispls**, а в буфере **rbuf** принимающего процесса – массивом **rdispls**.

MPI

```
int MPI_Alltoallw(void *sbuf,  
int scounts[], int sdispls[],  
MPI_Datatype stypes[], void  
*rbuf, int rcounts[], int  
rdispls[], MPI_Datatype  
rtypes[], MPI_Comm comm)
```

Процедура позволяет осуществить наиболее общий обмен данными с заданием для каждой порции своего размера, типа данных и размещения в буфере. Смещения в массивах **sdispls** и **rdispls** задаются в байтах.

MPI

```
int MPI_Reduce(void *sbuf, void  
*rbuf, int count, MPI_Datatype  
datatype, MPI_Op op, int root,  
MPI_Comm comm)
```

Выполнение **count** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Результат операции над **i**-ми элементами массивов **sbuf** получается в **i**-ом элементе массива **rbuf** процесса **root**.

MPI

Типы предопределённых глобальных операций:

MPI_MAX, MPI_MIN — максимальное и минимальное значения;

MPI_SUM, MPI_PROD — глобальная сумма и глобальное произведение;

MPI_BAND, MPI_LOR, MPI_LXOR — логические “И”, “ИЛИ”, искл. “ИЛИ”;

MPI_BAND, MPI_BOR, MPI_BXOR — побитовые “И”, “ИЛИ”, искл. “ИЛИ”.

MPI

Если для отправки и приема данных должен использоваться один буфер, то на месте аргумента **sbuf** процесса **root** можно указать значение **MPI_IN_PLACE**. В этом случае входные данные процесса **root** берутся из буфера результата **rbuf**.

MPI

```
int MPI_Allreduce(void *sbuf,  
void *rbuf, int count,  
MPI_Datatype datatype, MPI_Op  
op, MPI_Comm comm)
```

Выполнение **count** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Результат получается в массиве **rbuf** каждого процесса.

MPI

```
for(i=0; i<n; i++) s[i]=0.0;
for(i=0; i<n; i++)
    for(j=0; j<m; j++)
        s[i]=s[i]+a[i][j];
MPI_Allreduce(s, r, n, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Reduce_local(void*  
inbuf, void* inoutbuf, int  
count, MPI_Datatype datatype,  
MPI_Op op)
```

Процедура на вызвавшем процессе
поэлементно выполняет **op** операций над
count элементами массивов **inbuf** и
inoutbuf. Результат помещается в массив
inoutbuf.

MPI

```
int MPI_Reduce_scatter_block(  
void* sbuf, void* rbuf, int  
rcount, MPI_Datatype datatype,  
MPI_Op op, MPI_Comm comm)
```

Выполнение **$n \times rcount$** (**n** – число процессов) независимых глобальных операций **op** над соответствующими элементами массивов **$sbuf$** всех процессов.

MPI

Сначала выполняются глобальные операции, а затем результат рассылается по процессам.

При этом **i**-ый процесс получает **(i+1)**-ую порцию результатов из **rcount** элементов и помещает ее в массив **rbuf**. Значение **rcount** должно быть одинаковым на всех процессах коммуникатора **comm**.

MPI

```
int MPI_Reduce_scatter(void  
*sbuf, void *rbuf, int *rcounts,  
MPI_Datatype datatype, MPI_Op  
op, MPI_Comm comm)
```

Выполнение $\sum_i \text{rcounts}(i)$ независимых
глобальных операций **op** над
соответствующими элементами массивов
sbuf.

MPI

Сначала выполняются глобальные операции, затем результат рассылается по процессам.

i-ый процесс получает **rcounts(i)** значений результата и помещает в массив **rbuf**.

MPI

```
int MPI_Scan(void *sbuf, void  
*rbuf, int count, MPI_Datatype  
datatype, MPI_Op op, MPI_Comm  
comm)
```

Выполнение **count** независимых
частичных глобальных операций **op** над
соответствующими элементами массивов
sbuf.

MPI

i-ый процесс выполняет глобальную операцию над соответствующими элементами массива **sbuf** процессов $0 \dots i$ и помещает результат в массив **rbuf**.

Окончательный результат глобальной операции получается в массиве **rbuf** последнего процесса.

MPI

```
int MPI_Exscan(void *sbuf, void  
*rbuf, int count, MPI_Datatype  
datatype, MPI_Op op, MPI_Comm  
comm)
```

Выполнение **count** независимых частичных
глобальных операций **op** над
соответствующими элементами массивов
sbuf.

MPI

i-ый процесс (**i**>0) выполняет **count** глобальных операций над соответствующими элементами массива **sbuf** процессов с номерами от 0 до **i-1** включительно и помещает полученный результат в массив **rbuf**. На процессе 0 массив **rbuf** не задействуется.

MPI

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

Создание пользовательской глобальной операции **op**, которая будет вычисляться функцией **func**. Если **commute=1**, то операция должна быть коммутативной и ассоциативной. Иначе порядок фиксируется по увеличению номеров процессов.

MPI

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

Интерфейс пользовательской функции.

Первый аргумент **invec**, второй аргумент – **inoutvec**, результат – **inoutvec**. **len**

задает количество элементов входного и выходного массивов, а **type** – тип данных. В пользовательской функции не должны производиться никакие обмены.

MPI

```
int MPI_Op_free(MPI_Op *op)
```

Уничтожение пользовательской глобальной операции. После выполнения процедуры переменной **op** присваивается значение **MPI_OP_NULL**.

MPI

```
#include <stdio.h>
#include "mpi.h"
#define n 1000
void smod5(void *in, void *inout, int *l, MPI_Datatype
 *type) {
    int i;
    for(i=0; i<*l; i++) ((int*)inout)[i] = (((int*)in)[i]
+ ((int*)inout)[i])%5;
}
int main(int argc, char **argv)
{
    int rank, size, i;
    int a[n];
    int b[n];
```

MPI

```
MPI_Op op;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
for(i=0; i<n; i++) a[i] = i + rank +1;
printf("process %d a[0] = %d\n", rank, a[0]);
MPI_Op_create(&smod5, 1, &op);
MPI_Reduce(a, b, n, MPI_INT, op, 0, MPI_COMM_WORLD);
MPI_Op_free(&op);
if(rank==0) printf("b[0] = %d\n", b[0]);
MPI_Finalize();
}
```

MPI

Пересылка разнотипных данных

MPI

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Для пересылки разнотипных данных можно использовать:

- Производные типы данных
- Упаковку данных

MPI

Производные типы данных

MPI

Производные типы данных создаются во время выполнения программы с помощью подпрограмм-конструкторов.

Создание типа:

- Конструирование типа
- Регистрация типа

MPI

Производный тип данных характеризуется последовательностью базовых типов и набором значений смещения относительно начала буфера обмена.

Смещения могут быть как положительными, так и отрицательными, не требуется их упорядоченность.

MPI

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

Создание нового типа данных **newtype**, состоящего из **count** последовательно расположенных элементов базового типа данных **type**.

```
MPI_Type_contiguous(5, MPI_INT, &newtype);
```

MPI

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

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** элементов после начала предыдущего.

MPI

```
count=2;  
blocklen=3;  
stride=5;  
MPI_Type_vector(count, blocklen, stride,  
MPI_DOUBLE, &newtype);
```

Создание нового типа данных (тип элемента, количество элементов от начала буфера):

```
{(MPI_DOUBLE, 0), (MPI_DOUBLE, 1),  
(MPI_DOUBLE, 2),  
 (MPI_DOUBLE, 5), (MPI_DOUBLE, 6),  
(MPI_DOUBLE, 7)}
```

MPI

```
int MPI_Type_create_hvector(int  
count, int blocklen, MPI_Aint  
stride, MPI_Datatype type,  
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** байт после начала предыдущего.

MPI

```
int  
MPI_Type_create_indexed_block(  
    int count, int blocklen, int  
    displs[], MPI_Datatype type,  
    MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**.

Смещения блоков с начала буфера послыки в количестве элементов базового типа данных **type** задаются в массиве **displs**.

MPI

```
int MPI_Type_indexed(int count,  
int *blocklens, int *displs,  
MPI_Datatype type, MPI_Datatype  
*newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** элементов с начала буфера.

MPI

```
for(i=0; i<n; i++){  
    blocklens[i]=n-i;  
    displs[i]=(n+1)*i;  
}  
MPI_Type_indexed(n, blocklens, displs,  
MPI_DOUBLE, &newtype)
```

Создание нового типа данных для описания верхнетреугольной матрицы.

MPI

```
int MPI_Type_create_hindexed(int  
count, int *blocklens, MPI_Aint  
*displs, MPI_Datatype type,  
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

MPI

```
int MPI_Type_create_struct(int  
count, int *blocklens, MPI_Aint  
*displs, MPI_Datatype *types,  
MPI_Datatype *newtype)
```

Создание структурного типа данных из **count** блоков по **blocklens[i]** элементов типа **types[i]**. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

MPI

```
blocklens[0]=3;  
blocklens[1]=2;  
types[0]=MPI_DOUBLE;  
types[1]=MPI_CHAR;  
displs[0]=0;  
displs[1]=24;  
MPI_Type_create_struct(2, blocklens, displs,  
types, &newtype);
```

Создание нового типа данных (тип элемента, количество байт от начала буфера):

```
{(MPI_DOUBLE, 0), (MPI_DOUBLE, 8),  
(MPI_DOUBLE, 16),  
(MPI_CHAR, 24), (MPI_CHAR, 25)}
```

MPI

```
int MPI_Type_create_subarray(int  
ndims, int sizes[], int  
subsizes[], int starts[], int  
order, MPI_Datatype type,  
MPI_Datatype *newtype)
```

newtype задаёт **ndims**–мерный подмассив исходного **ndims**–мерного массива. **sizes** задает размеры по каждому измерению исходного массива, **subsizes** – размеры по каждому измерению выделяемого подмассива.

MPI

starts задаёт стартовые координаты каждого измерения выделяемого подмассива в исходном массиве. Все массивы индексируются с 0. Задаваемые значения не должны выводить подмассив за пределы исходного массива ни по одному из измерений. **order** задаёт порядок хранения элементов многомерного массива: **MPI_ORDER_C** (по строкам), **MPI_ORDER_FORTRAN** (по столбцам). **type** задаёт тип элементов массива.

MPI

```
MPI_Datatype newtype;  
int sizes[2], subsizes[2], starts[2];  
int rank;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
sizes[0] = 100;  
sizes[1] = 100;  
subsizes[0] = 100;  
subsizes[1] = 25;  
starts[0] = 0;  
starts[1] = rank*subsizes[1];  
MPI_Type_create_subarray(2, sizes, subsizes,  
starts, MPI_ORDER_C, MPI_DOUBLE, &newtype);
```

MPI

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

Регистрация созданного производного типа данных **datatype**. После регистрации этот тип данных можно использовать в операциях обмена. Предопределенные типы данных регистрировать не нужно.

MPI

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

Аннулирование производного типа данных **datatype**.

datatype устанавливается в значение **MPI_DATATYPE_NULL**.

Производные от **datatype** типы данных остаются. Предопределённые типы данных не могут быть аннулированы.

MPI

```
int MPI_Type_dup(MPI_Datatype  
type, MPI_Datatype *newtype)
```

Создает новый тип данных **newtype**, аналогичный типу данных **type**. При этом производится копирование всех ассоциированных с типом **type** атрибутов. Если исходный тип данных уже был зарегистрирован, то автоматически будет зарегистрирован и создаваемый тип данных.

MPI

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

Определение абсолютного байт-адреса **address** размещения массива **location** в оперативной памяти компьютера. Адрес отсчитывается от некоторого базового адреса, значение которого содержится в системной константе **MPI_BOTTOM**.

MPI

```
blocklens[0] = 1;
blocklens[1] = 1;
types[0] = MPI_DOUBLE;
types[1] = MPI_CHAR;
MPI_Get_address(dat1, &displs[0]);
MPI_Get_address(dat2, &displs[1]);
MPI_Type_create_struct(2, blocklens, displs,
types, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(MPI_BOTTOM, 1, newtype, dest, tag,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Type_size(MPI_Datatype  
datatype, int *size)
```

Определение размера типа **datatype** в байтах (объёма памяти, занимаемого одним элементом данного типа).

MPI

```
int MPI_Type_get_extent(  
MPI_Datatype datatype, MPI_Aint  
*lb, MPI_Aint *extent)
```

Для элемента типа данных **datatype** определяет смещение от начала буфера данных нижней границы **lb** и диапазон **extent** (разницу между верхней и нижней границами) в байтах.

MPI

Упаковка данных

MPI

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

Упаковка **incount** элементов типа **datatype** из массива **inbuf** в массив **outbuf** со сдвигом **position** байт. **outbuf** должен содержать хотя бы **outsize** байт.

MPI

Параметр **position** увеличивается на число байт, равное размеру записи.

Параметр **comm** указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение.

Для пересылки упакованных данных используется тип данных **MPI_PACKED**.

MPI

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

Распаковка из массива **inbuf** со сдвигом **position** байт в массив **outbuf** **outcount** элементов типа **datatype**.
Массив **inbuf** имеет размер не менее **insize** байт.

MPI

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

Определение необходимого объёма памяти (в байтах) для упаковки **incount** элементов типа **datatype**. Необходимый для упаковки размер может превышать сумму размеров пакуемых элементов данных.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int size, rank, position, i;
    float a[10];
    char b[10], buf[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for(i = 0; i<10; i++){
        a[i] = rank + 1.0;
        if(rank==0) b[i]='a';
        else b[i] = 'b';
    }
    position=0;
```

MPI

```
if(rank==0){
    MPI_Pack(a, 10, MPI_FLOAT, buf, 100, &position,
MPI_COMM_WORLD);
    MPI_Pack(b, 10, MPI_CHAR, buf, 100, &position,
MPI_COMM_WORLD);
    MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
} else{
    MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
    MPI_Unpack(buf, 100, &position, a, 10, MPI_FLOAT,
MPI_COMM_WORLD);
    MPI_Unpack(buf, 100, &position, b, 10, MPI_CHAR,
MPI_COMM_WORLD);
}

for(i = 0; i<10; i++) printf("process %d a=%f b=%c\n",
rank, a[i], b[i]);
MPI_Finalize();}
```

MPI

Производные типы данных:

- сложнее использовать;
- + не нужны дополнительные буфера;
- + нет дополнительных копирований.

Упаковка данных:

- + проще использовать;
- требуется дополнительный буфер;
- требуются дополнительные копирования.

*Московский государственный университет имени М.В.Ломоносова
Факультет Вычислительной математики и кибернетики*

*СУПЕРКОМПЬЮТЕРЫ И
ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ДАННЫХ
(лекция 11)*

А.С.Антонов

Вед. н.с. НИВЦ МГУ, к.ф.-м.н.

asa@parallel.ru

ВМК МГУ, 2024