

# Технология программирования MPI (3)

Антонов Александр Сергеевич,  
к.ф.-м.н., вед.н.с. лаборатории Параллельных  
информационных технологий НИВЦ МГУ

Суперкомпьютерная академия МГУ  
Москва, 2020

**МРІ**

**Коллективные взаимодействия  
процессов**

# МРІ

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

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

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 коммуникатора, ни даже того, что она ими начата (если это возможно по смыслу операции).

# МРІ

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

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

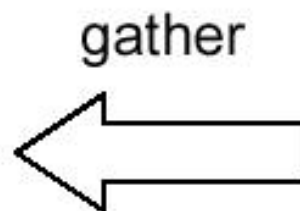
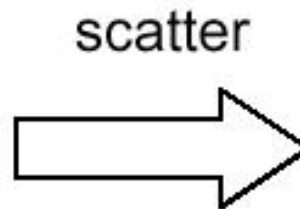
Если для отправки и приема данных должен использоваться один и тот же буфер, то на месте аргумента **sbuf** процесса **root** можно указать значение **MPI\_IN\_PLACE**. В этом случае аргументы **scount** и **stype** игнорируются, и предполагается, что порция данных процесса **root** уже расположена в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

# MPI

данные

процессы

A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>



A <sub>0</sub>					
A <sub>1</sub>					
A <sub>2</sub>					
A <sub>3</sub>					
A <sub>4</sub>					
A <sub>5</sub>					

# 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 коммутаторе).

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



# 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

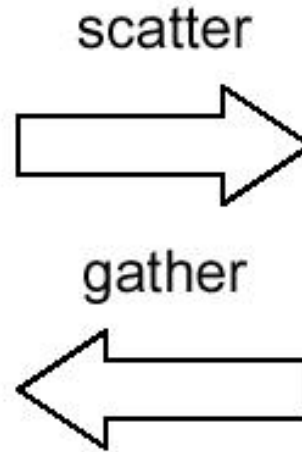
Если для посылки и приема данных должен использоваться один буфер, то на месте аргумента **rbuf** процесса **root** можно указать значение **MPI\_IN\_PLACE**. В этом случае аргументы **rcount** и **rtype** игнорируются, и предполагается, что порция данных процесса **root** не пересылается.

# MPI

данные

процессы

A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>



A <sub>0</sub>					
A <sub>1</sub>					
A <sub>2</sub>					
A <sub>3</sub>					
A <sub>4</sub>					
A <sub>5</sub>					

# 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 коммутаторе).

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

# 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 посылки 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

```
int MPI_Allgather(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

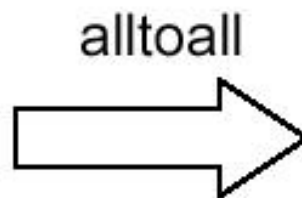
Если для посылки и приема должен использоваться один буфер, то на месте аргумента **sbuf** всех процессов можно указать значение **MPI\_IN\_PLACE**. В этом случае аргументы **scount** и **stype** игнорируются, и предполагается, что порции исходных данных всех процессов уже расположены в соответствующих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

# MPI

данные

процессы

A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>
B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>
C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>
D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>
E <sub>0</sub>	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>
F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>



A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>
A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>	E <sub>1</sub>	F <sub>1</sub>
A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>	E <sub>2</sub>	F <sub>2</sub>
A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>	E <sub>3</sub>	F <sub>3</sub>
A <sub>4</sub>	B <sub>4</sub>	C <sub>4</sub>	D <sub>4</sub>	E <sub>4</sub>	F <sub>4</sub>
A <sub>5</sub>	B <sub>5</sub>	C <sub>5</sub>	D <sub>5</sub>	E <sub>5</sub>	F <sub>5</sub>

# 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\_LAND**, **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  $n * rcount$  ( $n$  – 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 коммутатора ***comm***.



# MPI

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

Выполнение  $\sum_i \mathbf{rcounts}(i)$  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...*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

Задание 6: Напишите программу на MPI, в которой операция глобального суммирования элементов вектора моделируется схемой сдваивания (каскадная схема) с использованием пересылок данных типа точка-точка. Сравнить эффективность такого моделирования с использованием процедуры **MPI\_Reduce** для разного числа процессов и разной длины векторов.