Суперкомпьютерное моделирование и технологии

Лекция

Технология параллельного программирования MPI.
Часть 2.

.

Попова Нина Николаевна

доцент кафедры СКИ

popova@cs.msu.su

14 октября 2022 г.

MPI стандарт для построения параллельных программ для вычислительных систем с распределенной памятью

6 основных функций MPI

- **Как стартовать/завершить параллельное выполнение**
  - MPI_Init
  - MPI_Finalize
- **Кто я (и другие процессы), сколько нас**
  - MPI_Comm_rank
  - MPI_Comm_size
- **Как передать сообщение коллеге (другому процессу)**
  - MPI_Send
  - MPI_Recv

Deadlocks

- Процесс 0 посылает большое сообщение процессу 1
  - Если в принимающем процессе недостаточно места в системном буфере, процесс 0 должен ждать пока процесс 1 не предоставит необходимый буфер.
  - Что произойдет:

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- Называется “unsafe” потому, что зависит от системного буфера.

Deadlocks

Пример. Пересылка по кольцу

Пусть каждый i-ый процесс посылает сообщение i+1 процессу (по модулю = число процессов) и получает сообщение от i-1 процесса:

```
int a[10], b[10], size, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%size, 1,
       MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+size)%size, 1,
       MPI_COMM_WORLD, &status);
...
```

Deadlock

Пути решения «ubsafe» передач

- Упорядочить передачи:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Использовать неблокирующие передачи:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Неблокирующие коммуникации

Цель – уменьшение времени работы параллельной программы за счет совмещения вычислений и обменов.

Неблокирующие операции завершаются, не дожидаясь окончания передачи данных. В отличие от аналогичных блокирующих функций изменен критерий завершения операций – немедленное завершение.

Проверка состояния передач и ожидание завершение передач выполняются специальными функциями.

Форматы неблокирующих функций

```
MPI_Isend(buf,count,datatype,dest,tag,comm,request)

MPI_Irecv(buf,count,datatype,source,tag,comm,
request)
```

request – “квитанция» о завершении передачи.
Тип: MPI_Requset
MPI_REQUEST_NULL – обнуление

MPI_Wait() ожидание завершения.

MPI_Test() проверка завершения. Возвращается флаг, указывающий на результат завершения.

Асинхронные передачи

*int MPI_Isend(void *buf,int count, MPI_Datatype datatype,int dest, int tag, MPI_Comm comm, MPI_Requset *request)*

*int MPI_Irecv(void *buf,int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Status *status, MPI_Requset *request)*

*int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);*

*int MPI_Wait(MPI_Request *request, MPI_Status *status)*

*int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)*

flag : true, если передача завершена

После выполнения MPI_Wait или MPI_Test значение status устанавливается в MPI_REQUEST_NULL

```
MPI_Request request;
MPI_Status status;
 int request_complete = 0; // Rank 0 sends, rank 1 receives
 if (rank == 0)
 { MPI_Isend(buffer, buffer_count, MPI_INT, 1, 0, MPI_COMM_WORLD,
     &request);
 // Here we do some work while waiting for process 1 to be ready
 while (has_work) {
 do_work();
 // We only test if the request is not already fulfilled
 if (!request_complete)
 MPI_Test(&request, &request_complete, &status); }
 // No more work, we wait for the request to be complete if it's not the case
 if (!request_complete) MPI_Wait(&request, &status);
 }
else
{
MPI_Irecv(buffer, buffer_count, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
// Here we just wait for the message to come
   MPI_Wait(&request, &status); }
```

Ожидание завершения асинхронных передач

int **MPI_Waitall (int count, MPI_Request array_of_requests[],**
MPI_Status array_of_statuses[])
// waits for all given communications to finish and fills in the statuses
Int **MPI_Waitany (int count, MPI_Request array_of_requests[], int *index,**
MPI_Status *status)
// waits for one of the given communications to finish, sets the index to indicate
// which one and fills in the status
int **MPI_Waitsome (int incount, MPI_Request array_of_requests[],**
int *outcount, int array_of_indices[], MPI_Status array_of_statuses[])
// waits for at least one of the given communications to finish, sets the number
// of communication requests that have finished, their indices and status

Проверка состояния передач

int **MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)**

// tests if the communication is finished. Sets flag to 1 and fills in the status if
// finished or sets the flag to 0 if not finished.

int **MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,**
MPI_Status array_of_statuses[])

// test whether all given communications are finished. Sts flag to 1 and fills in
// the status aray if all are finished or sets the flag to 0 if not all are finished.

int **MPI_Testany(int count, MPI_Request array_of_requests[], int *index,**
int *flag, MPI_Status *status)

// test whether one of the given communications is finished. Sets flag to 1 and fills
// in the index and status if one finished or sets the flag to 0 if none is finished.

int **MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount,**
int array_of_indices[], MPI_Status array_of_statuses[])

// tests whether some of the given communications is finished, sets the number
// of communication requests that have finished, their indices and statuses.

int **MPI_Cancel(MPI_Request *request)**

Пример. Пересылка по кольцу с использованием асинхронных передач (1)

```
/******************************************************************************/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Пример. Пересылка по кольцу с использованием асинхронных передач (2)

```
**********************************************************************/
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);
printf("Task %d communicated with tasks %d & %d\n",rank,prev,next);
MPI_Finalize();
}
```

Совмещение посылки и приема сообщений

```
int MPI_Sendrecv(
void *sendbuf, int sendcount,MPI_Datatype senddatatype, int dest, int sendtag,
void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag,
            MPI_Comm comm, MPI_Status *status)
```

Если использовать один буфер:

```
int MPI_Sendrecv_replace(
            void *buf, int count,MPI_Datatype datatype,
             int dest, int sendtag,
             int source, int recvtag,
            MPI_Comm comm, MPI_Status *status)
```

Пример использования MPI_Sendrecv

Пересылка по кольцу. Решение .

```
int a[10], b[10], size, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Sendrecv(a, 10, MPI_INT, (myrank+1)%size , 1,
                    b, 10, MPI_INT, (myrank-1+size)%size, 1,
                    MPI_COMM_WORLD, &status)

...
```

Коллективные передачи

- Передача сообщений между группой процессов
- Вызываются ВСЕМИ процессами в коммуникаторе
- Примеры:
  - Broadcast, scatter, gather (рассылка данных)
  - Global sum, global maximum, и т.д. (Коллективные операции)
  - Барьерная синхронизация

Характеристики коллективных передач

- Коллективные операции не являются помехой операциям типа «точка-точка» и наоборот
- Все процессы коммуникатора должны вызывать коллективную операцию
- Синхронизация не гарантируется (за исключением барьера).
  Завершение операции – локально в процессе
- Нет тэгов
- Принимающий буфер должен точно соответствовать размеру отсылаемого буфера
- Асинхронные коллективные передачи - в MPI-3

Функции коллективных передач

- MPI_Bcast() – широковещательная передача (one to all)
- MPI_Ibcast() – асинхронная широковещательная передача
- MPI_Reduce() – Рекдукция (all to one)
- MPI_Allreduce() – Редукция (all to all)
- MPI_Scatter() – Распределение данных (one to all)
- MPI_Gather() – Сборка данных (all to one)
- MPI_Alltoall() – Распределение данных (all to all)
- MPI_Allgather() – Сборка данных (all to all)

Барьерная синхронизация

- Приостановка процессов до выхода ВСЕХ процессов коммуникатора в заданную точку синхронизации

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

Пример – упорядоченный вывод:

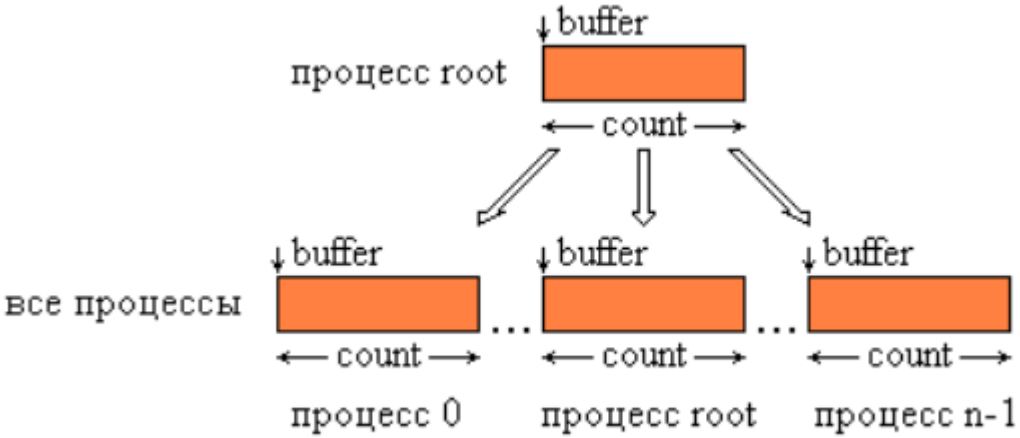
```
int size,rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
for (i=0;i<size;i++)
{ MPI_Barrier(MPI_COMM_WORLD);
if (rank==i) printf("%d\n", rank);
}
```

Широковещательная рассылка

- One-to-all передача: один и тот же буфер отсылается от процесса root всем остальным процессам в коммуникаторе

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

- Все процессы должны указать один тот же root и communicator



Широковещательная рассылка

- Асинхронная широковещательная рассылка (MPI-3)

```
int MPI_Ibcast (void *buffer, int count,
    MPI_Datatype datatype,int root, MPI_Comm comm,
  MPI_Request *request)
```

Пример использования асинхронной широковещательной рассылки (1)

```
#include <stdio.h>
 #include <string.h>
 #include "mpi.h“
 int main(int argc, char **argv) {
 char message[20];
int i, rank, size;
 MPI_Status status; MPI_Request request;
 int root = 0;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 if (rank == root) { strcpy(message, "Hello, world"); }
 MPI_Ibcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD,
    MPI_Request *request);
printf( "Message from process %d : %s\n", rank, message);// ERROR
 MPI_Finalize(); }
```

Пример использования асинхронной широковещательной рассылки (2)

```
#include <stdio.h>
 #include <string.h>
 #include "mpi.h“
 int main(int argc, char **argv) {
 char message[20];
int i, rank, size;
 MPI_Status status; MPI_Request request;
 int root = 0;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 if (rank == root) { strcpy(message, "Hello, world"); }
 MPI_Ibcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD,
     &request); MPI_wait (&request, &status);
printf( "Message from process %d : %s\n", rank, message);
 MPI_Finalize(); }
```

Глобальные операции редукции

- Операции выполняются над данными, распределенными по процессам коммуникатора
- Примеры:
  - Глобальная сумма или произведение
  - Глобальный максимум (минимум)
  - Глобальная операция, определенная пользователем

Общая форма

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

- **count** число операций “*op*” выполняемых над последовательными элементами буфера **sendbuf**
- (также размер **recvbuf**)

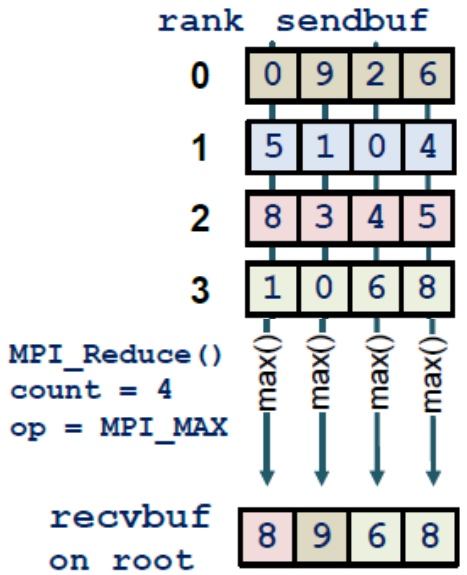
- **op** является ассоциативной операцией, которая выполняется над парой операндов типа **datatype** и возвращает результат того же типа

Предопределенные операции редукции

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Пример MPI_Reduce

MPI_Reduce (sendbuf, recvbuf, 4, MPI_INT,
MPI_MAX, 0, MPI_COMM_WORLD)



Варианты реализации MPI_Reduce

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

int MPI_Allreduce(const void* sbuf, void* rbuf, int count MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter_block(const void* sbuf, void* rbuf, int rcount, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter(const void* sbuf, void* rbuf, const int[] rcount, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

int MPI_Ireduce(const void* sbuf, void* rbuf, int count, MPI_Datatype stype, MPI_Op op, int root, MPI_Comm comm, MPI_Request *request)

In Place

Позволяет избегать локальное копирование, например из **send** буфер в **receive** буфер.

В качестве одного из буферов (всегда меньшего, если они различаются по размеру) можно использовать специальное значение MPI_IN_PLACE .Тип данных и количество передаваемых элементов этого буфера игнорируются.

Использование In Place в MPI_Reduce

Неверно:

```
double result;
MPI_Reduce(&result,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM
    _WORLD);
```

ОШИБКА

Верно:

```
if (rank==0)
    MPI_Reduce(&result,MPI_IN_PLACE,1,MPI_DOUBLE,MPI_SUM,0,
    MPI_COMM_WORLD);
else
    MPI_Reduce(NULL,&result,1,MPI_DOUBLE,MPI_SUM,0,
    MPI_COMM_WORLD);
```

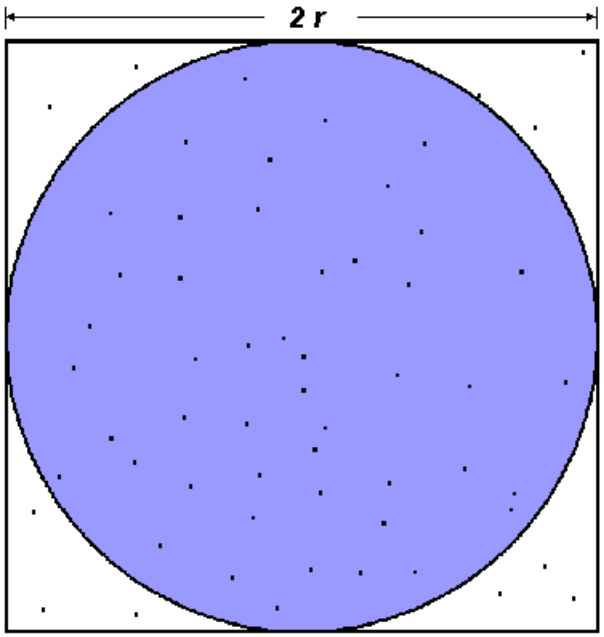
Вычисление числа π с использованием MPI. Вариант 2.

```
#include "mpi.h"
#include <stdio.h>
int main (int argc, char *argv[])
{
    int n =100000, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    h   = 1.0 / (double) n;
    sum = 0.0;
```

Вычисление числа π с использованием MPI (2)

```
for (i = myid + 1; i <= n; i += numprocs)
  {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
   }
   mypi = h * sum;
   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
              0, MPI_COMM_WORLD);
   if (myid == 0) printf("pi is approximately
                          %.16f", pi);
   MPI_Finalize();
   return 0;
}
```

Вычисление числа Pi методом Монте-Карло



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

АЛГОРИТМ

```
npoints = 10000
circle_count = 0
do j = 1,npoints
   generate 2 random numbers between 0 and 1:
   xcoordinate = random1
    ycoordinate = random2
    if (xcoordinate, ycoordinate) inside circle
    then circle_count = circle_count + 1
end do
```

PI = 4.0*circle_count/npoints

Однородная схема параллельного алгоритма вычисления Pi

Все процессы.

Пока точность не достигнута :

- генерируют последовательность случайных чисел – координат точек
- определяют, куда они попадают (круг, вне круга)
- вычисляют редукционную сумму .

Master-Worker схема параллельного алгоритма вычисления Pi

Процесс **Master** (только он генерирует последовательность случайных чисел):

Пока точность не достигнута :

- получает запрос на генерацию чисел
- генерирует последовательность случайных чисел
- отсылает процессу Worker.

Процессы **Worker**:

Пока точность не достигнута:

- получает серию случайных чисел – координат точек от Мастера
- определяет, куда они попадают (круг, вне круга)
- отсылает текущее значение Мастеру
- отсылает запрос на получение новой порции

Задание 2. Численное интегрирование многомерных функций методом Монте-Карло

2. Математическая постановка задачи

Функция $f(x, y, z)$ — непрерывна в ограниченой замкнутой области $G \subset \mathbb{R}^3$. Требуется вычислить определённый интеграл:

$$I = \iiint\limits_G f(x, y, z)\, dxdydz$$

Преобразуем искомый интеграл:

$$I = \iiint\limits_G f(x, y, z)\, dxdydz = \iiint\limits_\Pi F(x, y, z)\, dxdydz$$

Пусть $p_1(x_1, y_1, z_1), p_2(x_2, y_2, z_2), \ldots$ — случайные точки, равномерно распределённые в Π. Возьмём n таких случайных точек. В качестве приближённого значения интеграла предлагается использовать выражение:

$$I \approx |\Pi| \cdot \frac{1}{n} \sum_{i=1}^{n} F(p_i) \qquad (1)$$

где $|\Pi|$ — объём параллелепипеда Π. $|\Pi| = (b_1 - a_1)(b_2 - a_2)(b_3 - a_3)$

Задание 2. Численное интегрирование многомерных функций методом Монте-Карло

Срок сдачи задания:

30 октября 2022 г.

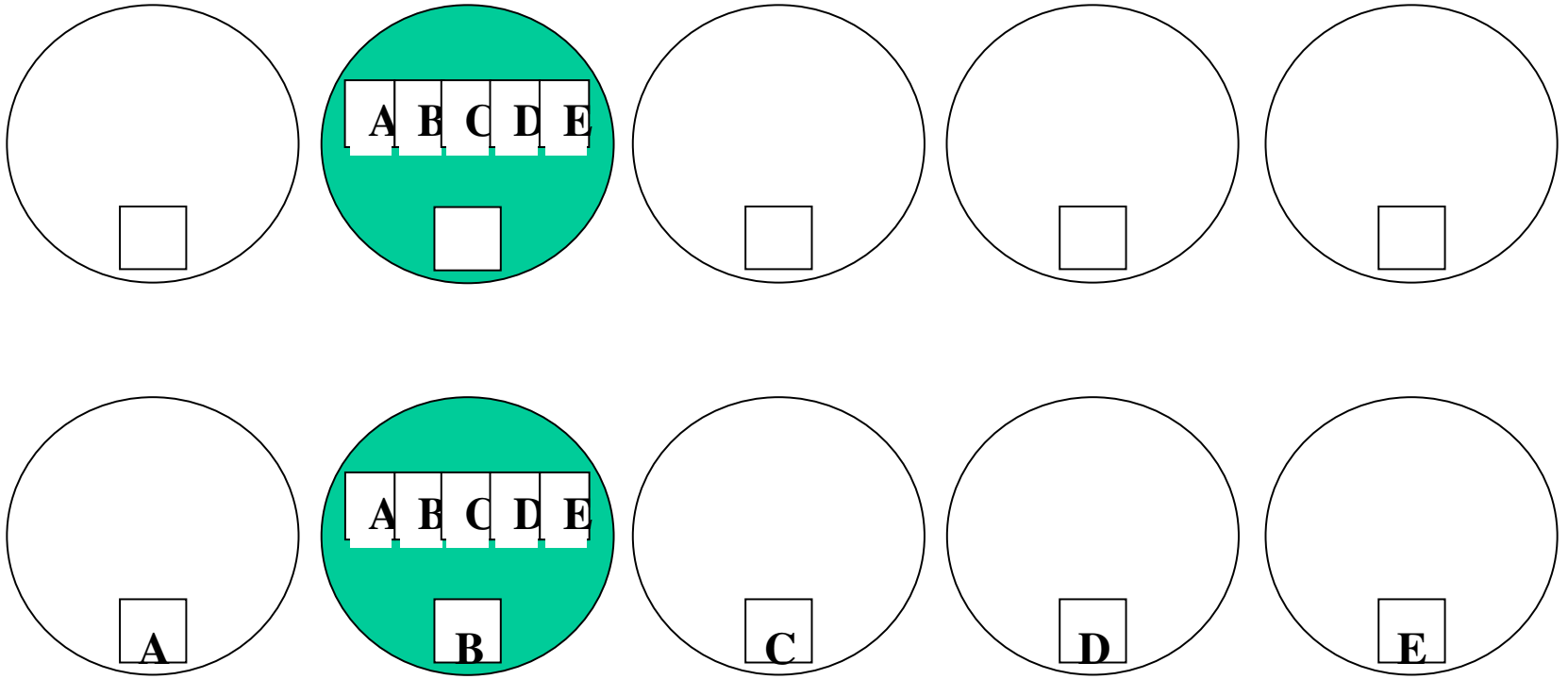
Функция Scatter рассылки блоков данных

- One-to-all communication: блоки данных одного размера из одного процесса рассылаются всем процессам коммуникатора (в порядке их номеров)

```
int MPI_Scatter(void* sendbuf, int sendcount,
    MPI_Datatype sendtype, void* recvbuf,
    int recvcount,MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

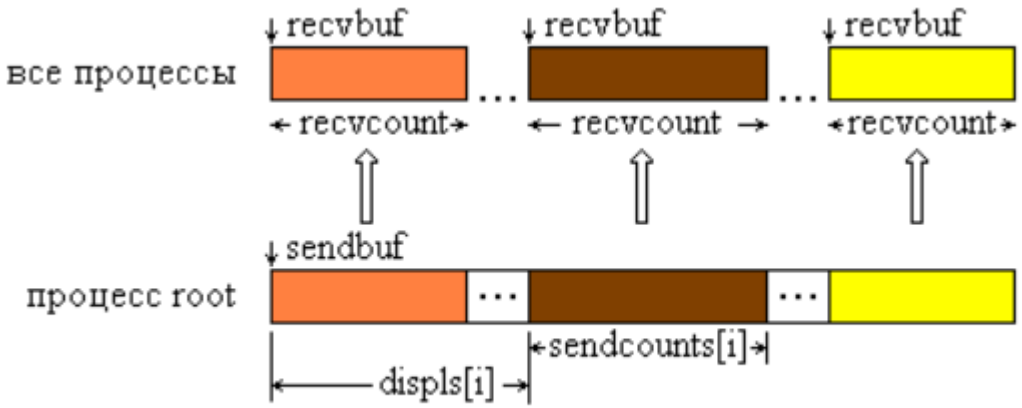
- *sendcount* – число элементов, посланных каждому процессу, **не общее число отосланных элементов;**
- send параметры (sendbuf, sendcount, sendtype) имеют смысл только для процесса root

Scatter – графическая иллюстрация

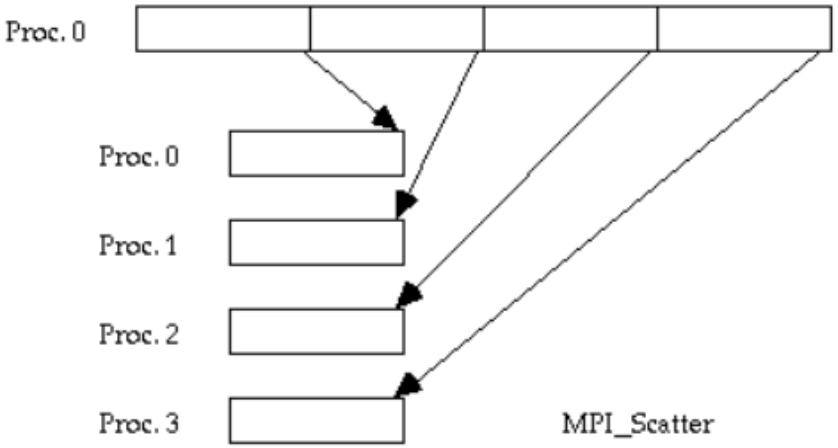


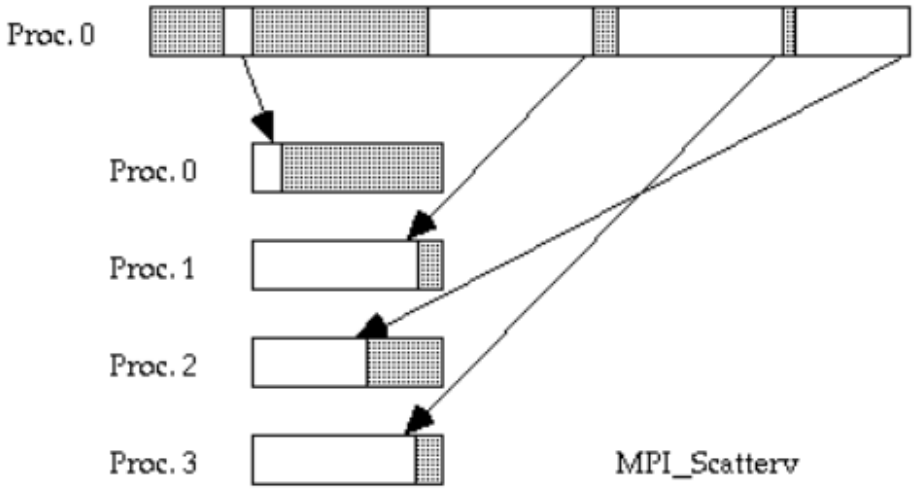
Функция Scatterv рассылки блоков разной длины

```
int MPI_Scatterv(void* sendbuf, int *sendcounts,
int *displs, MPI_Datatype sendtype, void* recvbuf,
int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



Сравнение: MPI_Scatter & MPI_Scatterv





Пример использования MPI_Scatterv

Рассылка массива в случае, если размер массива N НЕ ДЕЛИТСЯ нацело на число процессов size

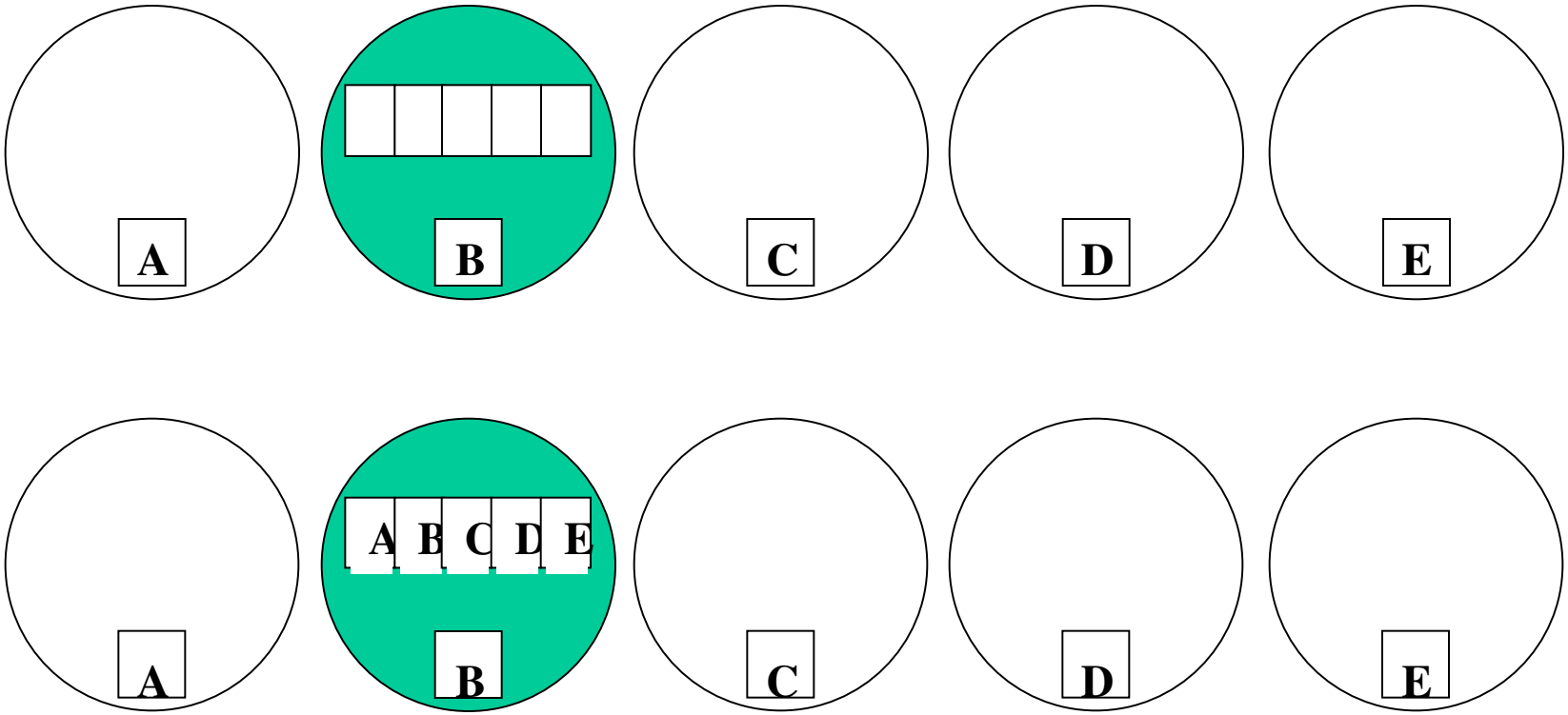
```
nmin = N/size;
 nextra = N%size;
 k = 0;
 for (i=0; i<size; i++) {
    if (i<nextra) sendcounts[i] = nmin+1;
    else sendcounts[i] = nmin; displs[i] = k; k = k+sendcounts[i]; }
  // need to set recvcount also ...
  MPI_Scatterv( sendbuf, sendcounts, displs, ...
```

Функция Gather сбора данных

- All-to-one передачи: блоки данных одинакового размера собираются процессом root
- Сбор данных выполняется в порядке номеров процессов
- Длина блоков предполагается одинаковой, т.е. данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

```
int MPI_Gather(void* sendbuf, int sendcount,
  MPI_Datatype sendtype,
  void* recvbuf, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

Gather – графическая илюстрация



Функция Gatherv сбора блоков данных разной длины

- Сбор данных разного размера в процессе root в порядке номеров процессов
- Длина блоков предполагается разной для процессов, т.е. данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Начало i-ой порции определяется смещением, указанным в массиве displs. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

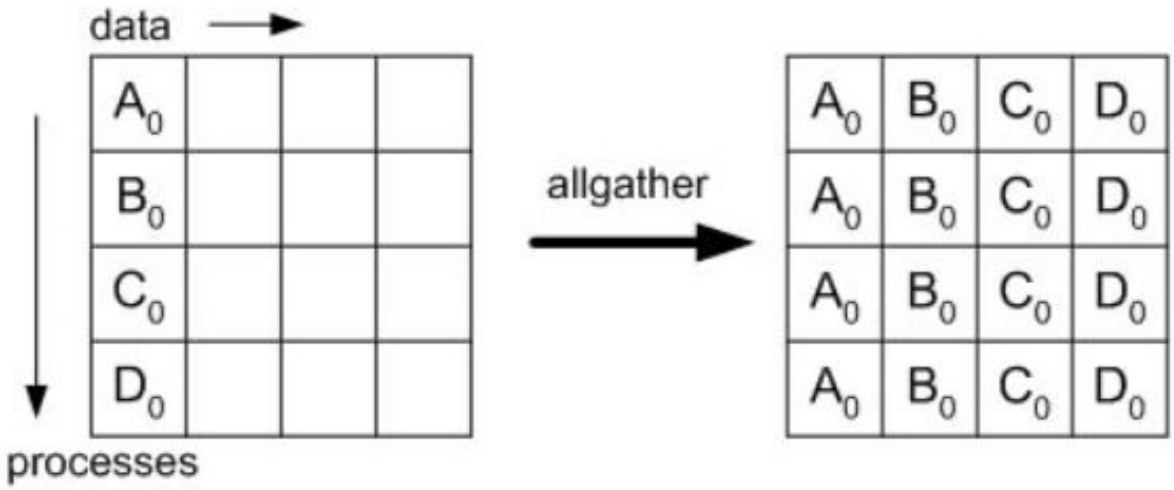
```
int MPI_Gatherv (void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

MPI_Allgather

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

int MPI_Allgatherv(const void* sbuf, int scounts, MPI_Datatype stype, void* rbuf, const int rcounts[], const int displs[], MPI_Datatype rtype, MPI_Comm comm)

Иллюстрация MPI_Allgather



Пример использования In Place

```
int value = …;
MPI_Gather(&value,              1,   MPI_INT,
                  recv_buf,           1,   MPI_INT,
                   root, comm);
intvalue = …;
if (rank == root) {
    recv_buf[root] = value;
    MPI_Gather(MPI_IN_PLACE, 1, MPI_INT,
                      recv_buf,           1, MPI_INT,
                       root, comm);}
 else {
   MPI_Gather(&value,              1,  MPI_INT,
                      recv_buf,           1, MPI_INT,
                       root, comm);
}
```

Требуется различать root и не-root

MPI_ALLTOALL

```
int MPI_Alltoall(
void* sendbuf, int sendcount, MPI_Datatype sendtype,
 void* recvbuf, int recvcount,  MPI_Datatype recvtype,
 MPI_Comm comm);
```

Описание:

- Рассылка сообщений от каждого процесса каждому
- j-ый блок данных из процесса i принимается j-ым процессом и размещается в i-ом блоке буфера recvbuf

MPI_ALLTOALL