

Системы и средства параллельного программирование

2016 – 2017 уч. г.

Лектор: доцент Н.Н.Попова,

Лекция 4

Тема: «**МРІ. Базовые операции передачи
данных**»

10 октября 2016 г.

Пример : скалярное произведение.

Реализация

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
void dataRead(double *a, double *b, int n);
double localSum(double *v,int n);
/*****/
int main(int argc, char *argv[]){
    int nProc, myRank;
    int root =0;
    int tag =0;
    int i, N, N_local;
    double partialSum, totalSum=0.0;
    double *a_local, *b_local;
    double timeStart,timeFinish;

    MPI_Status status;
```

Пример : скалярное произведение.

Реализация. Master.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &nProc);
if (myRank == root){
    timeStart = -MPI_Wtime();
    N = (int)(atof(argv[1]));
    N_local= N/nProc;
    for (i=1;i<nProc;i++)
        MPI_Send(&N_local, 1, MPI_INT, i, tag, MPI_COMM_WORLD);

    for (i=1;i<nProc;i++){
        MPI_Recv(&partialSum, 1, MPI_DOUBLE, i, MPI_ANY_SOURCE,
tag,MPI_COMM_WORLD,&status);
totalSum+=partialSum;
    }
    timeFinish = MPI_Wtime();
    printf(" Sum of vector =%e\n",totalSum);
    printf(" Program Run time = %d\n", timeStart+timeFinish);
}
```

Пример : скалярное произведение.

Реализация

```
else {  
    MPI_Recv(&N_local, 1, MPI_INT, root, tag, MPI_COMM_WORLD,  
    &status);  
    a_local=(double *) malloc(N_local*sizeof(double));  
    b_local=(double *) malloc(N_local*sizeof(double));  
    dataRead(a_local, N_local);  
    dataRead(b_local, N_local);  
    partialSum=localSum(a_local,b_local,N_local);  
  
    MPI_Send(&partialSum,1,MPI_DOUBLE,root,tag,MPI_COMM_WORLD)  
    ;  
    free(a_local); free(b_local);  
}  
MPI_Finalize();  
return 0;  
}
```

Пример : скалярное произведение.

Реализация

```

/*****
void dataRead(double *v, int n){
    int i;
    for (i=0; i<n; i++)
        v[i]= (double)i;
}  /*dataRead*/

/*****
double localSum(double *a, double *b, int n){
    int i;
    double sum=0.0;
    for (i=0; i<n; i++)
        sum+=a[i]*b[i];
    return sum;
}/*localSum*/

```

Замер времени MPI_Wtime

- Время измеряется в секундах
- Выделяется интервал в программе

```
double MPI_Wtime(void);
```

Пример.

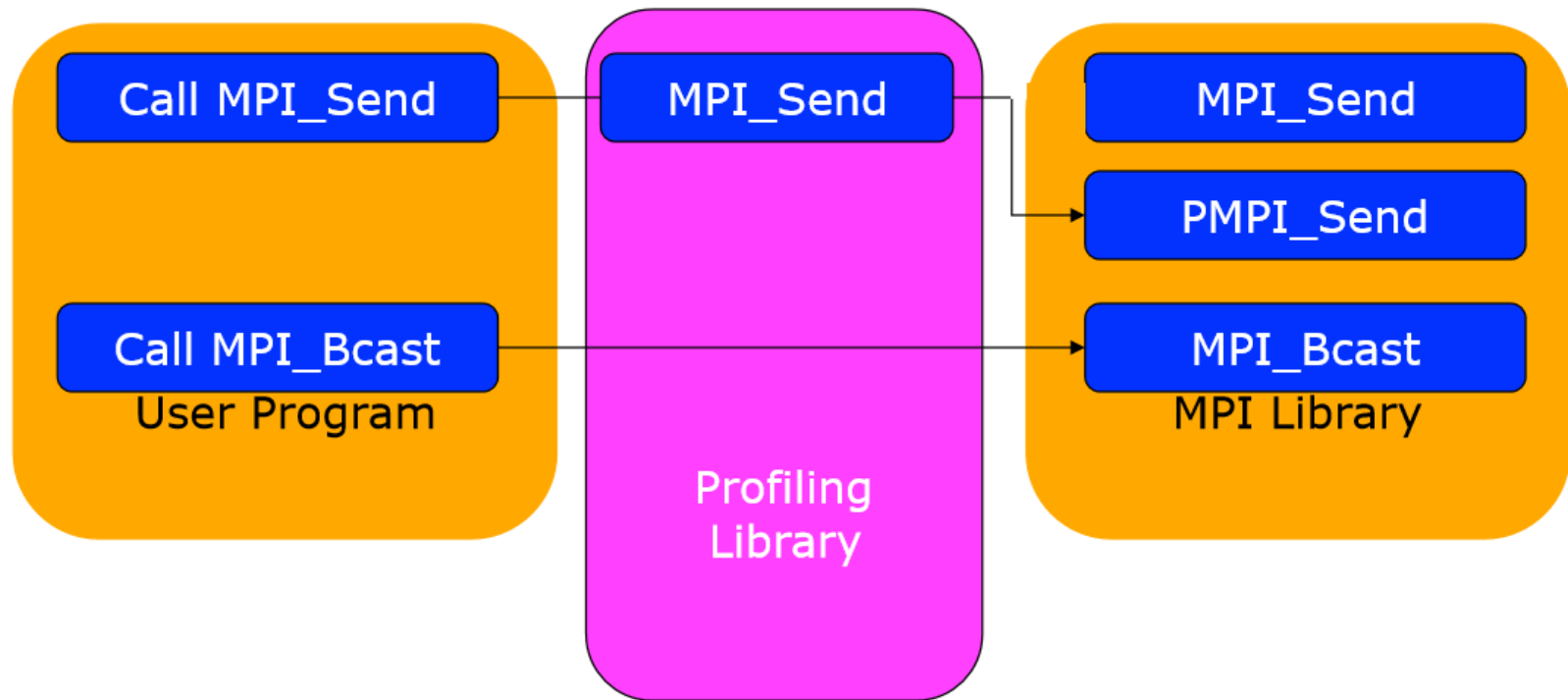
```
double start, finish, elapsed, time ;  
start=-MPI_Wtime;  
MPI_Send(...);  
finish = MPI_Wtime();  
time= start+finish;
```

Функции MPI передачи «Точка-точка»

Point-to-Point Communication Routines		
<u>MPI Bsend</u>	<u>MPI Bsend_init</u>	<u>MPI Buffer_attach</u>
<u>MPI Buffer_detach</u>	<u>MPI Cancel</u>	<u>MPI Get_count</u>
<u>MPI Get_elements</u>	<u>MPI Ibsend</u>	<u>MPI Iprobe</u>
<u>MPI Irecv</u>	<u>MPI Irsend</u>	<u>MPI Isend</u>
<u>MPI Issend</u>	<u>MPI Probe</u>	<u>MPI Recv</u>
<u>MPI Recv_init</u>	<u>MPI Request_free</u>	<u>MPI Rsend</u>
<u>MPI Rsend_init</u>	<u>MPI Send</u>	<u>MPI Send_init</u>
<u>MPI Sendrecv</u>	<u>MPI Sendrecv_replace</u>	<u>MPI Ssend</u>
<u>MPI Ssend_init</u>	<u>MPI Start</u>	<u>MPI Startall</u>
<u>MPI Test</u>	<u>MPI Test_cancelled</u>	<u>MPI Testall</u>
<u>MPI Testany</u>	<u>MPI Testsome</u>	<u>MPI Wait</u>
<u>MPI Waitall</u>	<u>MPI Waitany</u>	<u>MPI Waitsome</u>

Конец лекции №4, 10 октября

Профилировочный интерфейс



Пример использования профилировочного интерфейса

```
int MPI_Send(start, count, datatype, dest, tag, comm) {  
    double start, send_time;  
    start = -MPI_Wtime();  
    PMPI_Send(start, count, datatype, dest, tag, comm);  
    send_time= start + MPI_Wtime();  
    printf ('MPI_Send from %d to %d, count=%d, time= %f \n",  
           MPI_Comm_rank(...), dest, count, send_time);  
}
```

Режимы (моды) операций передачи сообщений

- Режимы MPI-коммуникаций определяют, при каких условиях операции передачи завершаются
- Режимы могут быть блокирующими или неблокирующими
 - Блокирующие: возврат из функций передачи сообщений только по завершению коммуникаций
 - Неблокирующие (асинхронные): немедленный возврат из функций, пользователь должен контролировать завершение передач

Standard send (MPI_Send)

- Критерий завершения: Не предопределен
- Завершается, когда сообщение отослано
- Можно предполагать, что сообщение достигло адресата
- Зависит от реализации

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.

Пути решения «unsafe» 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

Deadlock

```
/* simple deadlock */  
#include <stdio.h>  
#include <mpi.h>  
void main (int argc, char **argv) {  
int myrank;  
MPI_Status status;  
double a[100], b[100];  
MPI_Init(&argc, &argv); /* Initialize MPI */  
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */  
if( myrank == 0 ) {  
/* Receive, then send a message */  
MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );  
MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );  
}else if( myrank == 1 ) {  
/* Receive, then send a message */  
MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );  
MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );  
} MPI_Finalize(); /* Terminate MPI */}
```

Без Deadlock

```
/* safe exchange */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int myrank;
    MPI_Status status;
    double a[100], b[100];
    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    } else if( myrank == 1 ) { /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
    } MPI_Finalize(); /* Terminate MPI */ }
```


Без Deadlock, но зависит от реализации

```
/* depends on buffering */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int myrank;
    MPI_Status status;
    double a[100], b[100];
    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );}
    else if( myrank == 1 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
    }
    MPI_Finalize(); /* Terminate MPI */
}
```

Совмещение отправки и приема сообщений

Для обмена сообщениями MPI обеспечивает функции:

```
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)
```

Блокирующие и неблокирующие передачи

- **Блокирующие:** возврат из функций передачи сообщений только по завершению коммуникаций
- **Неблокирующие (асинхронные):** немедленный возврат из функций, пользователь должен контролировать завершение передач

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 завершения передач выполняются специальными функциями.

Параметры неблокирующих операций

Datatype	Тип MPI_Datatype
Communicator	Аналогично блокирующим (тип MPI_Comm)
Request	Тип MPI_Request

- Параметр request задается при инициации неблокирующей операции
- Используется для проверки завершения операции

Совмещение блокирующих и неблокирующих операций

- Send и receive могут блокирующими и неблокирующими
- Блокирующий send может соответствовать неблокирующему receive, и наоборот, например,
MPI_Isend  *MPI_Recv*
- Неблокирующий send может быть любого типа – synchronous, buffered, standard, ready

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_Request

MPI_REQUEST_NULL – обнуление

MPI_Wait() ожидание завершения.

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

Неблокирующий send

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


Неблокирующий receive

```
int MPI_Irecv (void *buf,  
               int count,  
               MPI_Datatype datatype,  
               int dest,  
               int tag,  
               MPI_Comm comm,  
               MPI_Request *request)
```

Wait/Test функции

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

Блокирует процесс до завершения передачи по request. В случае успешного завершения request устанавливается в MPI_REQUEST_NULL.

Информация по завершившейся передаче – в status.

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

flag – true, если передача завершилась.

Множественные проверки

- Test или wait для завершения одной (и только одной) передачи:
 - int `MPI_Waitany` (...)
 - int `MPI_Testany` (...)
- Test или wait завершения всех передач:
 - int `MPI_Waitall` (...)
 - int `MPI_Testall` (...)
- Test или wait завершения всех возможных к данному моменту:
 - int `MPI_Waitsome`(...)
 - int `MPI_Testsome`(...)

Свойства

- Сохраняется упорядоченность передач, задаваемая порядком вызовов асинхронных функций
- Гарантируется завершение соответствующей асинхронной передачи



Пример использования асинхронных передач

```
MPI_Request req1;
MPI_Status status;

.....
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend (&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, &req1);
    compute();
    MPI_Wait (&req1, &status);
} else if (myrank == 1) {
    int x;
    MPI_Recv (&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, &status);
}
```

Коллективные передачи

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

Функции коллективных передач

Collective Communication Routines		
<u>MPI Allgather</u>	<u>MPI Allgatherv</u>	<u>MPI Allreduce</u>
<u>MPI Alltoall</u>	<u>MPI Alltoallv</u>	<u>MPI Barrier</u>
<u>MPI Bcast</u>	<u>MPI Gather</u>	<u>MPI Gatherv</u>
<u>MPI Op create</u>	<u>MPI Op free</u>	<u>MPI Reduce</u>
<u>MPI Reduce scatter</u>	<u>MPI Scan</u>	<u>MPI Scatter</u>
<u>MPI Scatterv</u>		

Характеристики коллективных передач

- Коллективные операции не являются помехой операциям типа «точка-точка» и наоборот
- Все процессы коммутатора должны вызывать коллективную операцию
- Синхронизация не гарантируется (за исключением барьера)
- Нет неблокирующих коллективных операций
- Нет тэгов
- Принимающий буфер должен точно соответствовать размеру отсылаемого буфера

Барьерная синхронизация

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

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

Широковещательная рассылка

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

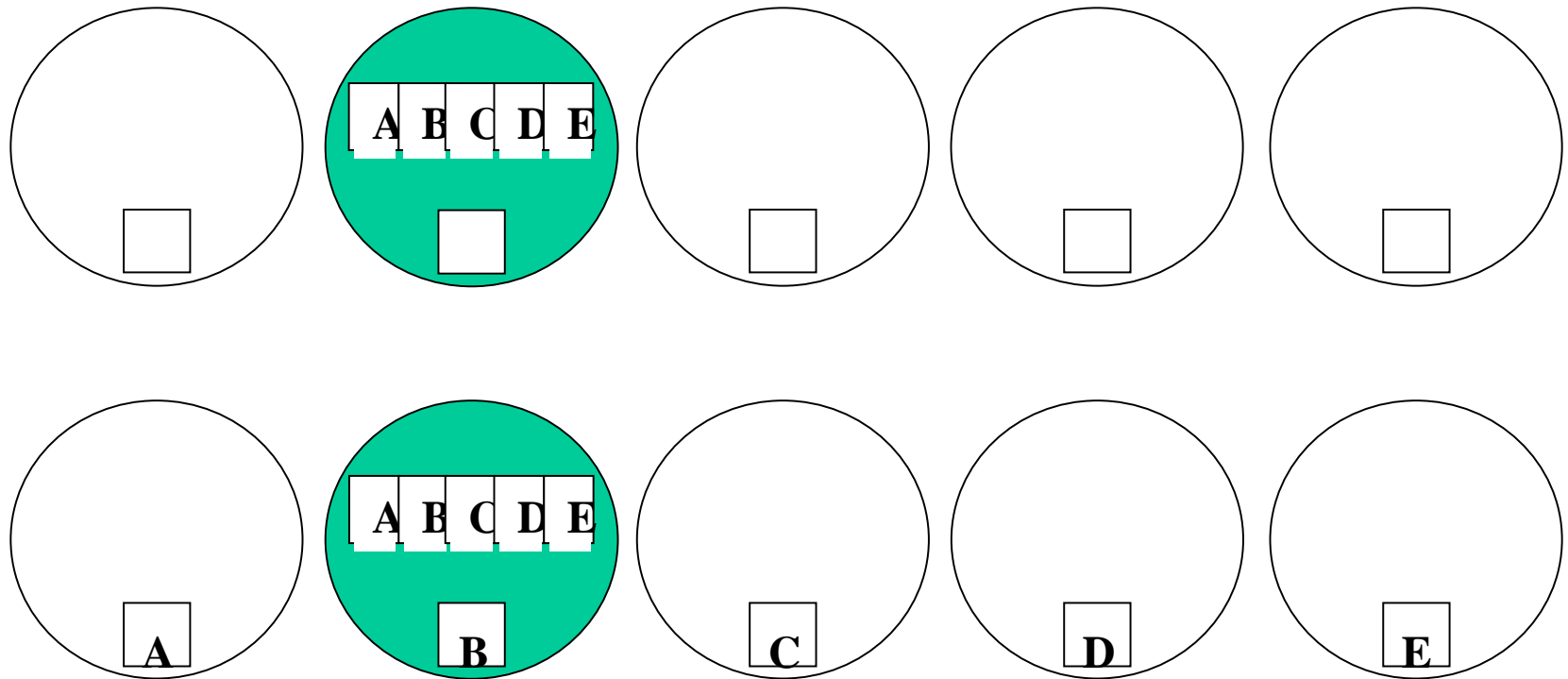
Scatter

- One-to-all communication: различные данные из одного процесса рассылаются всем процессам коммутатора (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 имеют смысл только для процесса root

Scatter – графическая иллюстрация

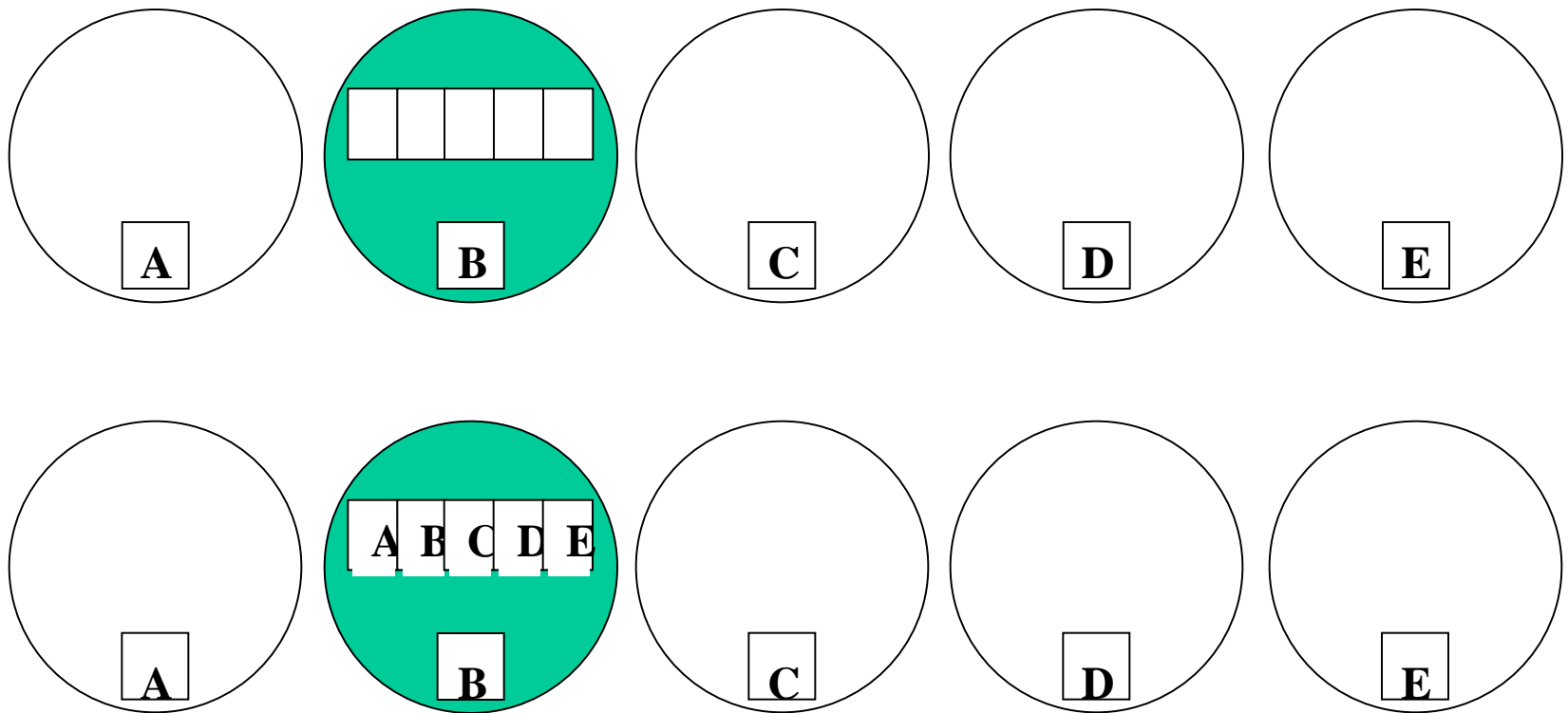


Gather

- All-to-one передачи: различные данные 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 иллюстрация



Глобальные операции редукции

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

Общая форма

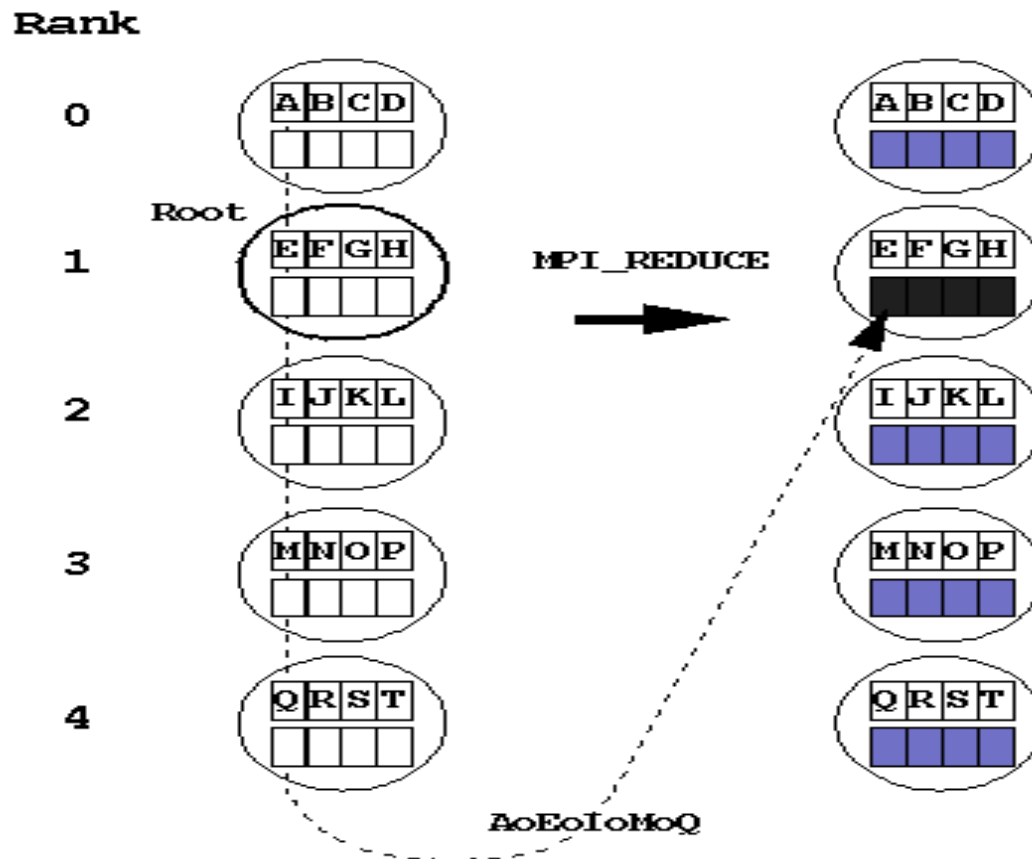
```
int MPI_Reduce(void* sendbuf, void* recvbuf,  
int count, MPI_Datatype datatype, MPI_Op op,  
int root, MPI_Comm comm)
```

- **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

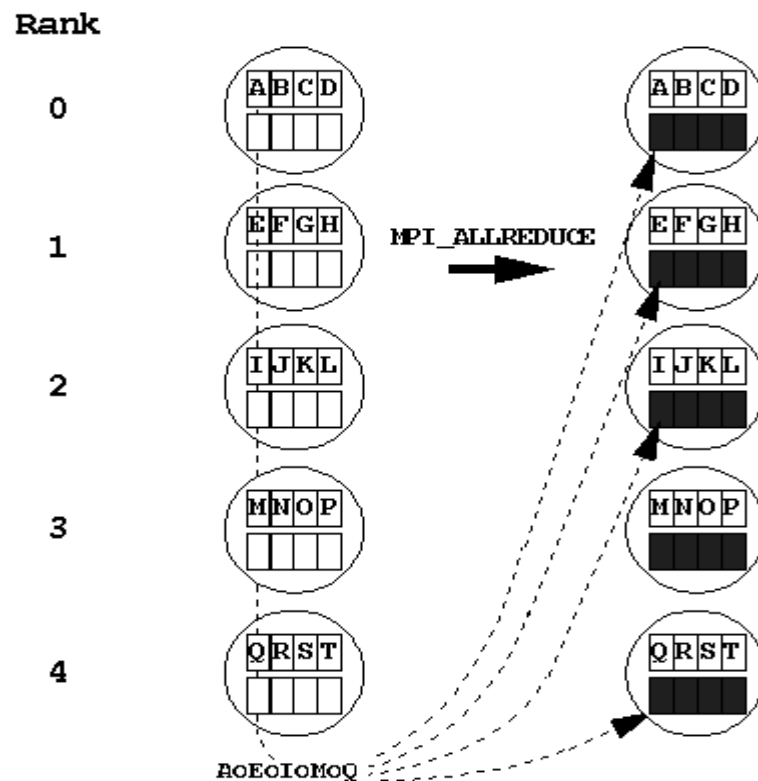
Предопределенные операции редукции

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

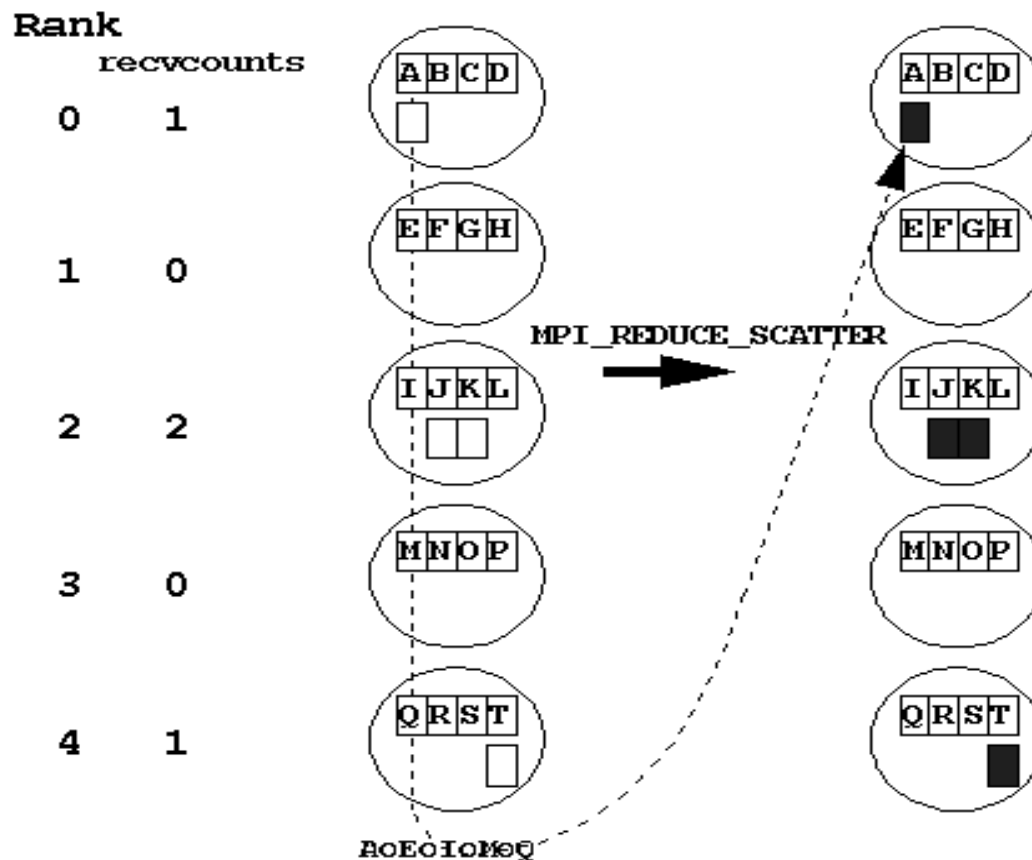
MPI Reduce



MPI_ALLREDUCE



MPI_REDUCE_SCATTER



MPI_SCAN

Rank

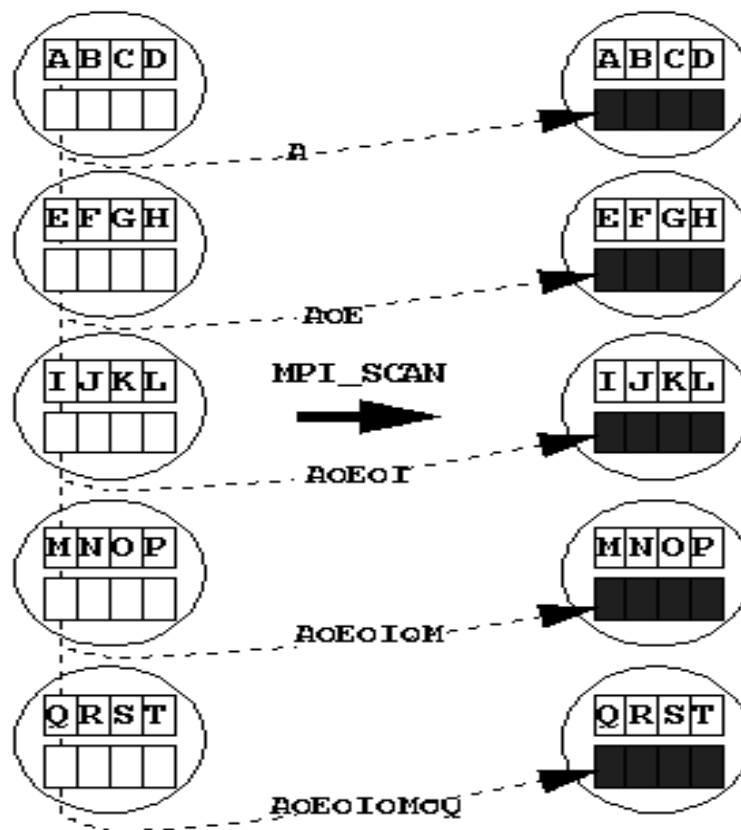
0

1

2

3

4



MPI_ALLTOALL

```
int MPI_Alltoall( void* sendbuf,
                  int sendcount,      /* in */
                  MPI_Datatype sendtype, /* in */
                  void* recvbuf,      /* out */
                  int recvcnt,        /* in */
                  MPI_Datatype recvtype, /* in */
                  MPI_Comm comm);
```

Описание:

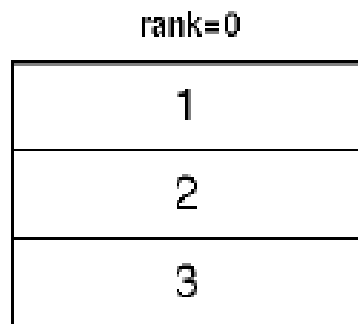
- Рассылка сообщений от каждого процесса каждому
- j-ый блок данных из процесса i принимается j-ым процессом и размещается в i-ом блоке буфера recvbuf

MPI ALLTOALL

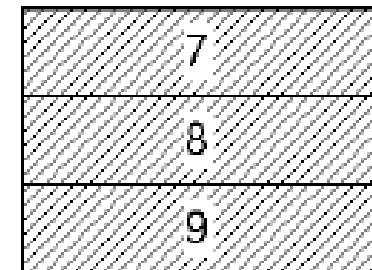
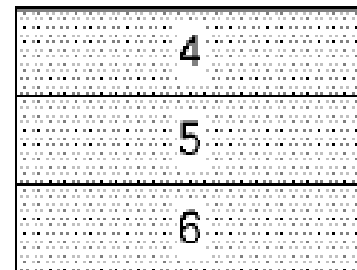
comm

sendcount

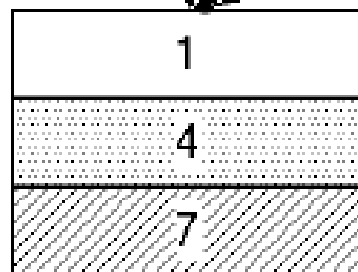
recvcount



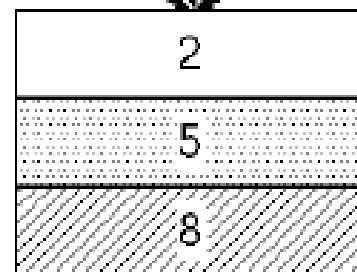
sendbuf



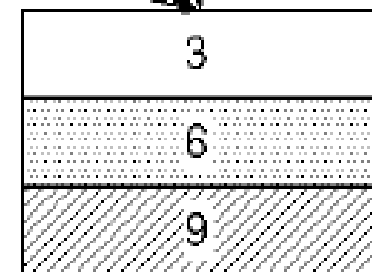
sendbuf



recvbuf



recvbuf



recvbuf

Пример: вычисление PI (1)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);}
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
```


Пример: PI (2)

```
h = 1.0 / (double) n;  
sum = 0.0;  
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, Error is %.16f\n",  
           pi, fabs(pi - PI25DT));  
}  
MPI_Finalize();  
return 0;}
```