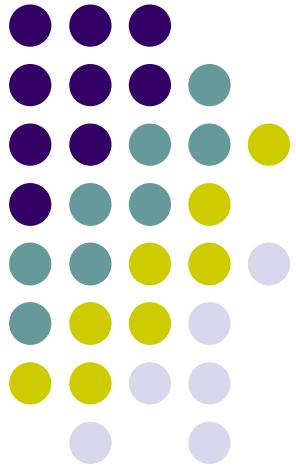
# Распределенные системы

Практическая часть 3

## Технологии параллельного программирования

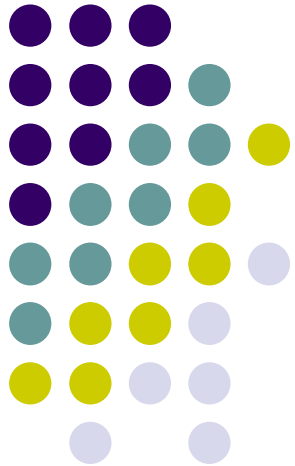
к.т.н. Приходько Т.А.



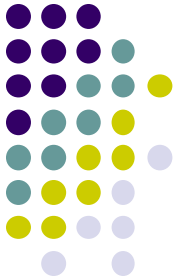
# Библиотека MPI

Message Passing Interface

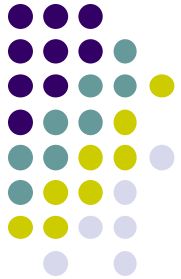
Коллективные обмены MPI



# План



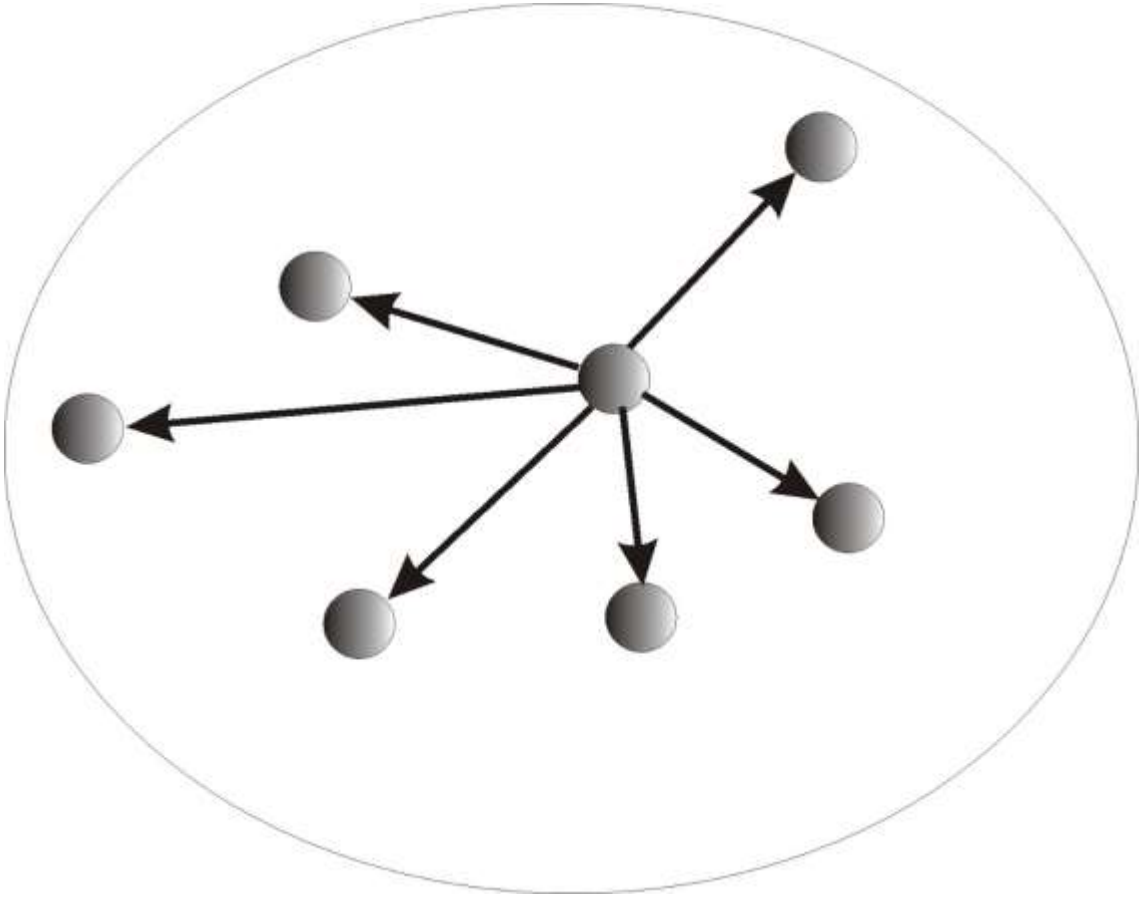
- Особенности коллективных обменов.
- Широковещательная рассылка.
- Операции распределения и сбора данных.
- Операции приведения.
- Синхронизация.



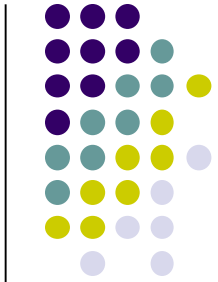
# Особенности коллективных обменов MPI

# Коллективные обмены

В операцию коллективного обмена вовлечены не два, а большее число процессов.



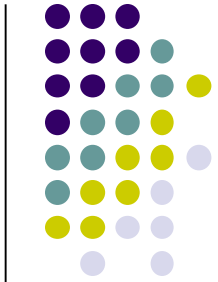
# Коллективные обмены



Общая характеристика коллективных обменов:

- коллективные обмены **не** могут взаимодействовать с двухточечными. Коллективная передача не может быть перехвачена двухточечной подпрограммой приема;
- коллективные обмены могут выполняться как с синхронизацией, так и без нее;
- теги сообщений в коллективных обменах назначаются системой.

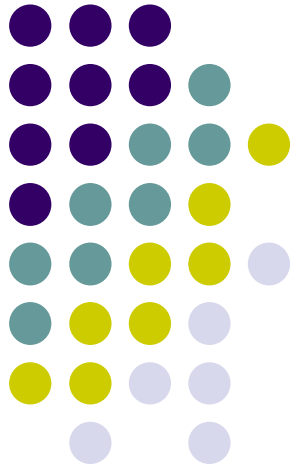
# Коллективные обмены



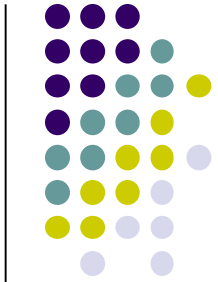
## Виды коллективных обменов:

- ***широковещательная передача*** - выполняется от одного процесса ко всем, варианты:
  - распределение данных;
  - сбор данных;
- ***синхронизация с барьером*** - это форма синхронизации работы процессов, когда выполнение программы продолжается только после того, как ***к соответствующей процедуре обратилось определенное число процессов***;
- ***операции приведения*** - входными являются данные нескольких процессов, а результат - одно значение, которое становится доступным всем процессам, участвующим в обмене;
- ***операции сканирования*** – операции частичного приведения.

# Виды коллективных обменов



# Коллективные обмены



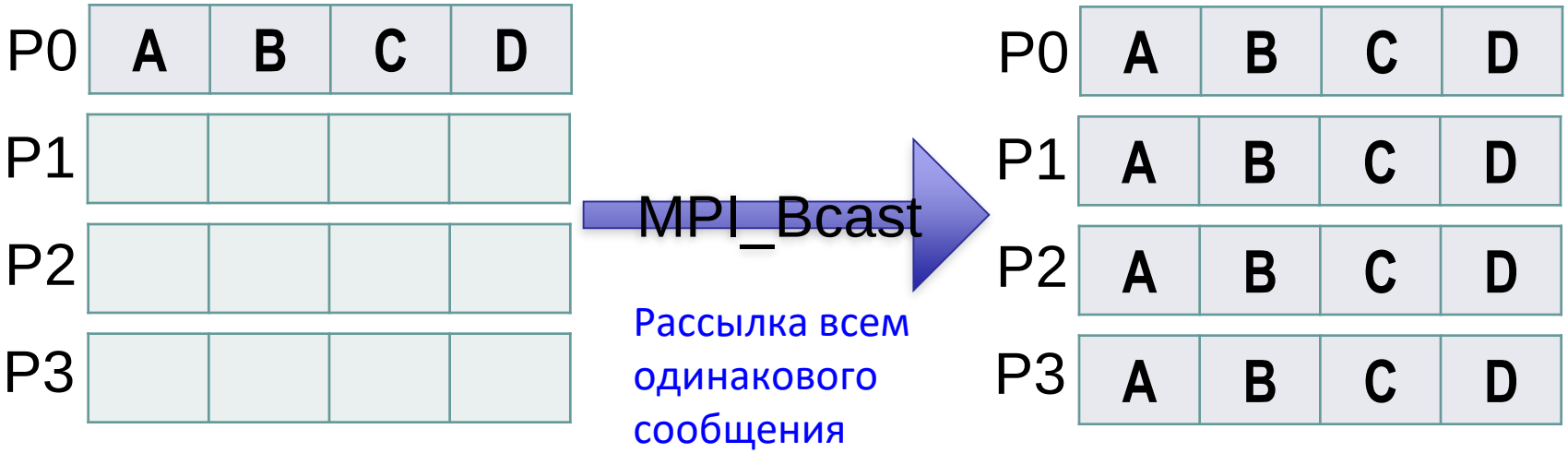
## Виды коллективных обменов:

- **Трансляционный обмен (One-to-all)**
  - MPI_Bcast
  - MPI_Scatter
  - MPI_Scatterv
- **Коллекторный обмен (All-to-one)**
  - MPI_Gather
  - MPI_Gatherv
  - MPI_Reduce
- **Трансляционно-циклический обмен (All-to-all)**
  - MPI_Allgather
  - MPI_Allgatherv
  - MPI_Alltoall
  - MPI_Alltoallv
  - MPI_Allreduce
  - MPI_Reduce_scatter

# Коллективные обмены

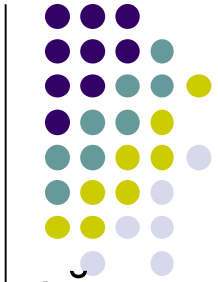
## Широковещательная рассылка


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




**MPI_Bcast** – рассылка всем процессам сообщения buf
Если номер процесса совпадает с root, то он отправитель, иначе – приемник

# Коллективные обмены



Широковещательная рассылка выполняется подпрограммой:

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

MPI_Bcast(buffer, count, datatype, root, comm, ierr)

- Параметры этой процедуры одновременно являются входными и выходными:
- **buffer** - адрес буфера;
- **count** - количество элементов данных в сообщении;
- **datatype** - тип данных MPI;
- **root** - ранг главного процесса, выполняющего широковещательную рассылку;
- **comm** - коммуникатор.

# Коллективные обмены

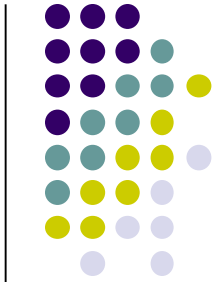
***JAVA*** Широковещательная рассылка

void MPI.COMM_WORLD.Bcast(Object buffer, int offset, int count, Datatype datatype, int root)

- **buf** - buffer array
- **offset** - initial offset in buffer
- **count** - number of items in buffer
- **datatype** - datatype of each item in buffer
- **root -** rank of broadcast root

Broadcast a message from the process with rank root to all processes of the group. Java binding of the MPI operation MPI_ BCAST.

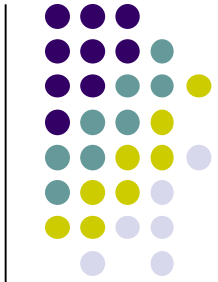
# Коллективные обмены



**Пример 1** использования широковещательной рассылки

```
 public static void main(String[] args) {
char data[]=new char[24];
 int size,myrank;
 Status status;
  MPI.Init(args);
  myrank = MPI.COMM_WORLD.Rank();
  size = MPI.COMM_WORLD.Size();
     if (myrank == 0)
   {
              data="Hi, Parallel Programmer!".toCharArray();
              MPI.COMM_WORLD.Bcast(data, 0, data.length, MPI.CHAR, 0);
             System.out.println("send: "+ Arrays.toString(data));
   }
else
             {MPI.COMM_WORLD.Bcast(data, 0, data.length, MPI.CHAR, 0);
              System.out.println("received: "+ Arrays.toString(data)+"by myrank "+myrank);
             }
   MPI.Finalize();
}
}
```

# Коллективные обмены



## Пример 1 модифицированный

```
public static void main(String[] args) {
 char data[]=new char[24];
 int size, myrank;

  MPI.Init(args);
  myrank = MPI.COMM_WORLD.Rank();
  size = MPI.COMM_WORLD.Size();

   data="Hi, Parallel Programmer!".toCharArray();
   MPI.COMM_WORLD.Bcast(data, 0, data.length, MPI.CHAR, 0);
    if (myrank == 0)
          System.out.println("send: "+ Arrays.toString(data));
    else
          System.out.println("received: "+ Arrays.toString(data)+"by myrank
"+myrank);

 MPI.Finalize();
   }
```

# Коллективные обмены

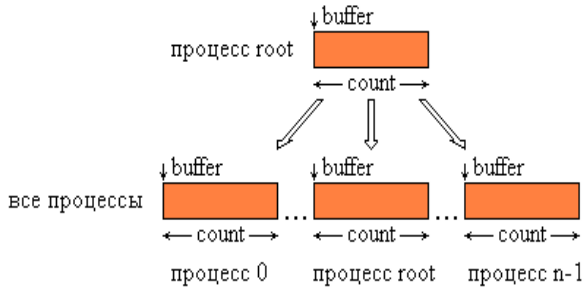
**Пример 2** использования широковещательной пересылки

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
 int myrank;
 int root = 0;
 int count = 1;
 float a, b;
 int n;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

# Коллективные обмены

## Пример 2 (продолжение)




```
 if (myrank == 0){
    printf("Enter a, b, n\n");
    scanf("%f %f %i", &a, &b, &n);
    MPI_Bcast(&a, count, MPI_FLOAT, root, MPI_COMM_WORLD);
    MPI_Bcast(&b, count, MPI_FLOAT, root, MPI_COMM_WORLD);
    MPI_Bcast(&n, count, MPI_INT, root, MPI_COMM_WORLD);
  }
else
{
    MPI_Bcast(&a, count, MPI_FLOAT, root, MPI_COMM_WORLD);
    MPI_Bcast(&b, count, MPI_FLOAT, root, MPI_COMM_WORLD);
    MPI_Bcast(&n, count, MPI_INT, root, MPI_COMM_WORLD);
    printf("%i Process got %f %f %i\n", myrank, a, b, n);
}
    MPI_Finalize();
    return 0;
}
```

# Коллективные обмены

## Распределение данных C

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype,
int root, MPI_Comm comm)
```

```
MPI_Scatter(sendbuf, sendcount, sendtype, rcvbuf, rcvcount,
rcvtype, root, comm, ierr)
```

Входные параметры:

- **sendbuf** - адрес буфера передачи;
- **sendcount** - количество элементов, пересылаемых каждому процессу (не суммарное количество пересылаемых элементов!);
- **sendtype** - тип передаваемых данных;
- **rcvcount** - количество элементов в буфере приема;
- **rcvtype** - тип принимаемых данных;
- **root** - ранг передающего процесса;
- **comm** - коммуникатор.

Выходной параметр: `rcvbuf` - адрес буфера приема.

# Коллективные обмены

## Распределение данных JAVA

```
MPI_Scatter(sendbuf, sendcount, sendtype, rcvbuf, rcvcount,
rcvtype, root, comm, ierr)(C)
void MPI.COMM_WORLD.Scatter(Object sendbuf, int sendoffset,
int sendcount, Datatype sendtype, Object recvbuf, int
recvoffset,int recvcount, Datatype recvtype, int root)
```

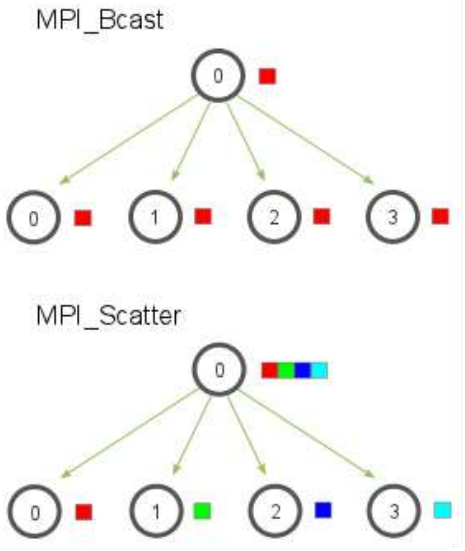
Входные параметры:

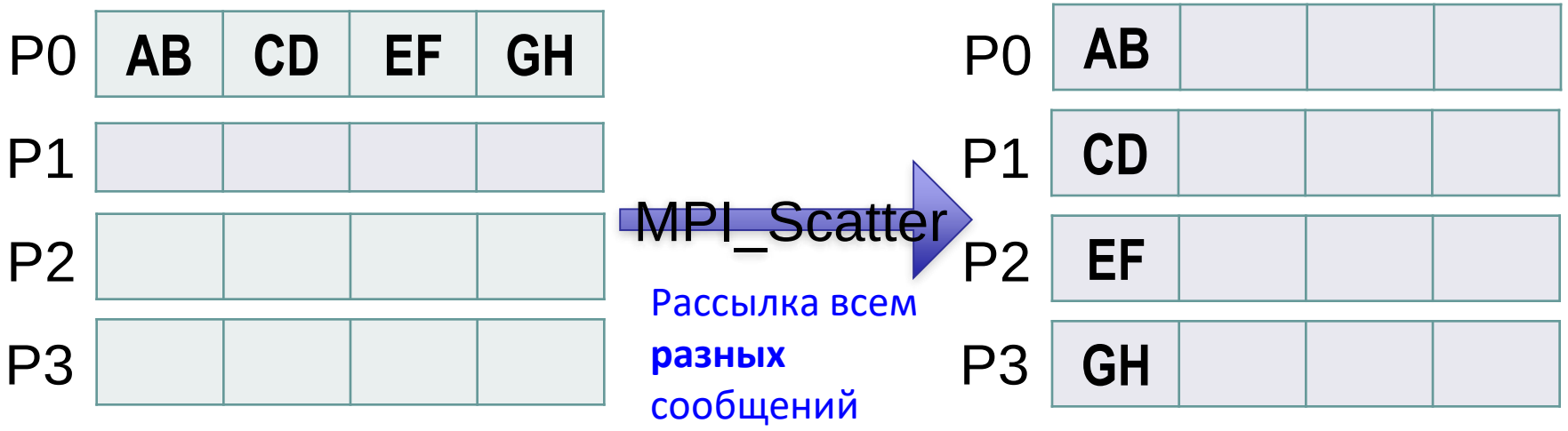
- **sendbuf** – send buffer array
- **sendoffset** – initial offset in send buffer
- **sendcount** – number of items sent to each process
- **sendtype** – datatype of send buffer items
- **recvbuf** – receive buffer array
- **recvoffset** – initial offset in receive buffer
- **recvcount** – number of items in receive buffer
- **recvtype** – datatype of receive buffer items
- **root** – rank of sending process

Java binding of the MPI operation MPI_SCATTER. Inverse of the operation Gather.

# Коллективные обмены

Процесс с рангом **root** распределяет содержимое буфера передачи **sendbuf** среди всех процессов, учитывая себя. Содержимое буфера передачи разбивается на несколько фрагментов, каждый из которых содержит **sendcount** элементов. Первый фрагмент передается процессу 0, второй процессу 1 и т. д. Аргументы **send** имеют значение только на стороне распределяющего процесса **root**.




Размер **sendbuf** = sizeof(sendtype) * sendcount * commsize

Размер **recvbuf** = sizeof(sendtype) * recvcount

# Коллективные обмены

## Распределение данных JAVA

**scatter**

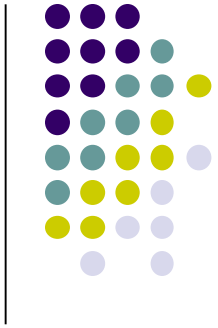
```
int unitSize=5,root=0;
        int sendbuf[]=null;

        sendbuf= new int[unitSize*size];
        for (int i=0; i<sendbuf.length;i++)
        { sendbuf[i]=i;}
        int [] recvbuf=new int[unitSize];

MPI.COMM_WORLD.Scatter(sendbuf,0,unitSize,MPI.INT,recvbuf,0,
unitSize,MPI.INT,root);
        if (myrank == 0) {
      System.out.println("send: " + Arrays.toString(sendbuf));
        }
System.out.println(myrank+" received: " + Arrays.toString(recvbuf));
```

```
5 received: [25, 26, 27, 28, 29]
3 received: [15, 16, 17, 18, 19]
4 received: [20, 21, 22, 23, 24]
2 received: [10, 11, 12, 13, 14]
send: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
0 received: [0, 1, 2, 3, 4]
1 received: [5, 6, 7, 8, 9]
```

# Коллективные обмены



## Распределение данных JAVA: Задача

**scatter**

```
int unitSize=5,root=0;
        int sendbuf[]=null;

        sendbuf= new int[unitSize*size];
        for (int i=0; i<sendbuf.length;i++)
        { sendbuf[i]=i;}
        int [] recvbuf=new int[unitSize];

MPI.COMM_WORLD.Scatter(sendbuf,0,unitSize,MPI.INT,recvbuf,0,
unitSize,MPI.INT,root);
        if (myrank == 0) {
      System.out.println("send: " + Arrays.toString(sendbuf));
        }
System.out.println(myrank+" received: " + Arrays.toString(recvbuf));
```

**Задача:** поэкспериментировать с размерами обоих буферов, сделать выводы о правилах использования операции **scatter**

# Коллективные обмены

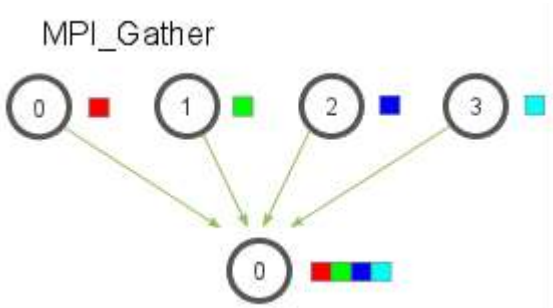
## Сбор данных

*Сбор данных* от всех процессов в буфер главной задачи выполняется подпрограммой:

```
int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *rcvbuf, int rcvcount,
MPI_Datatype rcvtype, int root, MPI_Comm comm)
```

**MPI_Gather(sendbuf, sendcount, sendtype, rcvbuf, rcvcount, rcvtype, root, comm, ierr)**

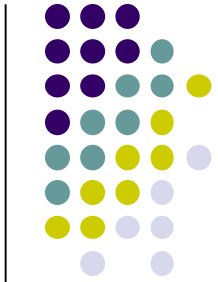
Каждый процесс в коммуникаторе **comm** пересылает содержимое буфера передачи **sendbuf** процессу с рангом **root**. Процесс **root** «склеивает» полученные данные в буфере приема.




Еще примеры: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

# Коллективные обмены

## Сбор данных JAVA



```
MPI_Gather(sendbuf, sendcount, sendtype, rcvbuf,
rcvcount, rcvtype, root, comm, ierr) (C)
```

**void MPI.COMM_WORLD.Gather(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)**

Входные параметры:

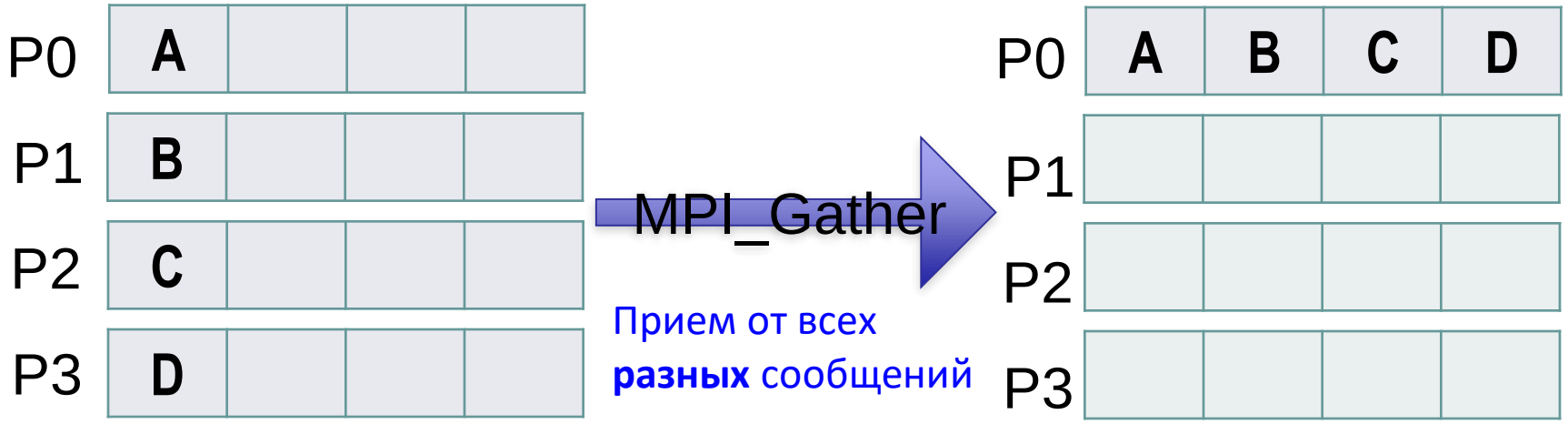
- **sendbuf** – send buffer array
- **sendoffset** – initial offset in send buffer
- **sendcount** – number of items sent to each process
- **sendtype** – datatype of send buffer items
- **recvbuf** – receive buffer array
- **recvoffset** – initial offset in receive buffer
- **recvcount** – number of items in receive buffer
- **recvtype** – datatype of receive buffer items
- **root** – rank of sending process

Each process sends the contents of its send buffer to the root process.

# Коллективные обмены

## Сбор данных

- Порядок склейки определяется рангами процессов, то есть в результирующем наборе после данных от процесса 0 следуют данные от процесса 1, затем данные от процесса 2 и т. д.
- Аргументы `rcvbuf`, `rcvcount` и `rcvtype` играют роль только на стороне главного процесса. Аргумент `rcvcount` указывает количество элементов данных, полученных от каждого процесса (но не суммарное их количество). При вызове подпрограмм `MPI_Scatter` и `MPI_Gather` из разных процессов следует использовать общий главный процесс.




Размер **sendbuf**: sizeof(sendtype) * sendcount
Размер **recvbuf**: sizeof(sendtype) * sendcount * commsize

# Коллективные обмены

## Векторная операция распределения данных

Векторная подпрограмма распределения данных:

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int
*displs, MPI_Datatype sendtype, void *rcvbuf, int rcvcount,
MPI_Datatype rcvtype, int root, MPI_Comm comm)
```

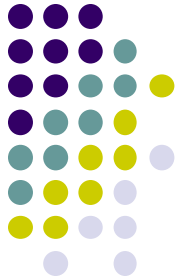
```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, rcvbuf,
rcvcount, rcvtype, root, comm, ierr)
```

Входные параметры:

- **sendbuf** - адрес буфера передачи;
- **sendcounts** - целочисленный одномерный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата). Его длина равна количеству процессов в коммуникаторе;

# Коллективные обмены

## Векторная операция распределения данных



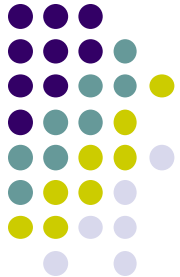
Входные параметры:

- **displs** - целочисленный массив, длина которого равна количеству процессов в коммуникаторе. Элемент с индексом i задает смещение относительно начала буфера передачи. Ранг адресата равен значению индекса i;
- **sendtype** - тип данных в буфере передачи;
- **rcvcount** - количество элементов в буфере приема;
- **rcvtype** - тип данных в буфере приема;
- **root** - ранг передающего процесса;
- **comm** - коммуникатор.

Выходной параметр: **rcvbuf** - адрес буфера приема.

# Коллективные обмены

## Векторная операция сбора данных



Сбор данных от всех процессов в заданном коммуникаторе и запись их в буфер приема с указанным смещением выполняется подпрограммой *векторного* сбора данных:

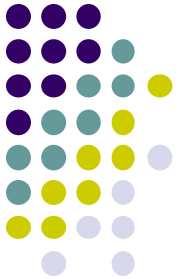
```
int MPI_Gatherv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf,
recvcounts, displs, recvtype, root, comm, ierr)
```

Список параметров у этой подпрограммы похож на список параметров подпрограммы `MPI_Scatterv`.
В обменах, выполняемых подпрограммами `MPI_Allgather` и `MPI_Alltoall`, нет главного процесса. Детали отправки и приема важны для всех процессов, участвующих в обмене.

# Коллективные обмены

## Пересылка данных по схеме «каждый - всем»



```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype,
MPI_Comm comm)
```

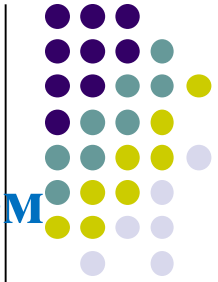
```
MPI_Alltoall(sendbuf, sendcount, sendtype, rcvbuf, rcvcount,
rcvtype, comm, ierr)
```

Входные параметры:

- **sendbuf** - начальный адрес буфера передачи;
- **sendcount** - количество элементов данных, пересылаемых каждому процессу;
- **sendtype** - тип данных в буфере передачи;
- **rcvcount** - количество элементов данных, принимаемых от каждого процесса;
- **rcvtype** - тип принимаемых данных;
- **comm** - коммуникатор.

Выходной параметр: **rcvbuf** - адрес буфера приема.

# Коллективные обмены



**Сбор данных от всех процессов и распределение их всем процессам:**

```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *rcvbuf, int rcvcount,
MPI_Datatype rcvtype, MPI_Comm comm)
```

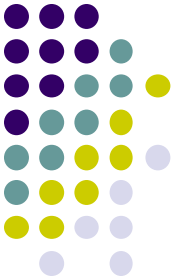
**MPI_Allgather(sendbuf, sendcount, sendtype, rcvbuf, rcvcount, rcvtype, comm, ierr)**

Входные параметры:

- **sendbuf** - начальный адрес буфера передачи;
- **sendcount** - количество элементов в буфере передачи;
- **sendtype** - тип передаваемых данных;
- **rcvcount** - количество элементов, полученных от каждого процесса;
- **rcvtype** - тип данных в буфере приема;
- **comm** - коммуникатор.

Выходной параметр: `rcvbuf` - адрес буфера приема.

# Коллективные обмены



## Сбор данных от всех процессов и распределение их всем процессам JAVA:

```
MPI_Allgather(sendbuf, sendcount, sendtype,
rcvbuf, rcvcount, rcvtype, comm, ierr)(C)
void MPI.COMM_WORLD.Allgather(Object sendbuf, int
sendoffset, int sendcount, Datatype sendtype,
Object recvbuf, int recvoffset, int recvcount,
Datatype recvtype)
```

Входные параметры:

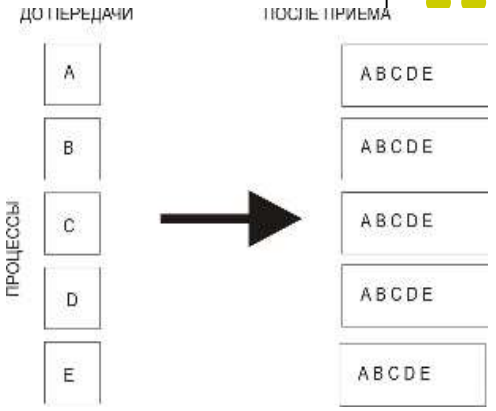
- **sendbuf** – send buffer array
- **sendoffset** – initial offset in send buffer
- **sendcount** – number of items sent to each process
- **sendtype** – datatype of send buffer items
- **recvbuf** – receive buffer array
- **recvoffset** – initial offset in receive buffer
- **recvcount** – number of items in receive buffer
- **recvtype** – datatype of receive buffer items

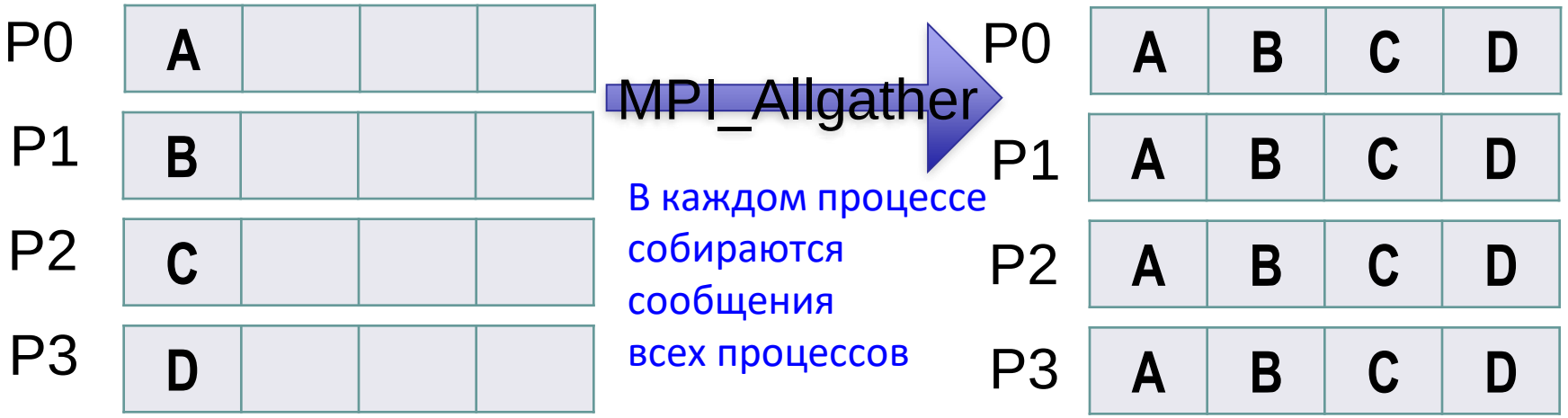
Similar to Gather, but all processes receive the result. Java binding of the MPI operation MPI ALLGATHER .

# Коллективные обмены

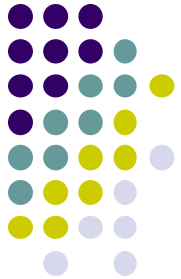
**собрать ото всех**

Блок данных, переданный от *j*-го процесса, принимается каждым процессом и размещается в *j*-м блоке буфера приема **`recvbuf`**.



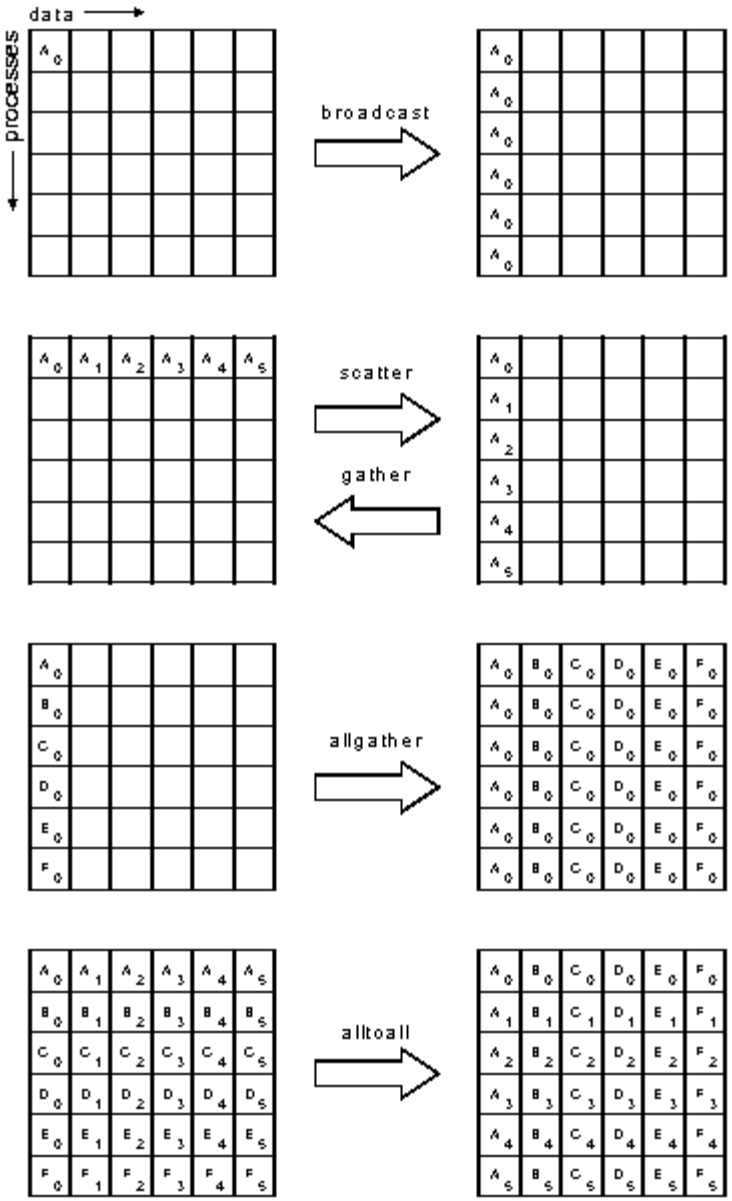



# Коллективные обмены



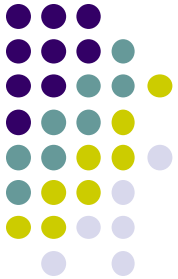
Векторными версиями `MPI_Allgather` и `MPI_Alltoall` являются подпрограммы `MPI_Allgatherv` и `MPI_Alltoallv`. Векторные операции позволяют детализировать процесс коллективного обмена.

# Коллективные обмены



# Коллективные обмены

## Операция приведения



Операция *приведения*, результат которой передается одному процессу

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

```
MPI_Reduce(buf, result, count, datatype, op, root,
comm, ierr)
```

Входные параметры:

- **buf** - адрес буфера передачи;
- **count** - количество элементов в буфере передачи;
- **datatype** - тип данных в буфере передачи;
- **op** - операция приведения;
- **root** - ранг главного процесса;
- **comm** - коммуникатор.

# Коллективные обмены

## Операция приведения JAVA

```
MPI_Reduce(buf, result, count, datatype, op, root,
comm, ierr)
void MPI.COMM_WORLD.Reduce(Object sendbuf, int
sendoffset, Object recvbuf, int recvoffset, int count,
Datatype datatype, Op op, int root)
```

Входные параметры:

- **sendbuf** – send buffer array
- **sendoffset** – initial offset in send buffer
- **recvbuf** – receive buffer array
- **recvoffset** – initial offset in receive buffer
- **count** – number of items in send buffer

**datatype** – datatype of send buffer items

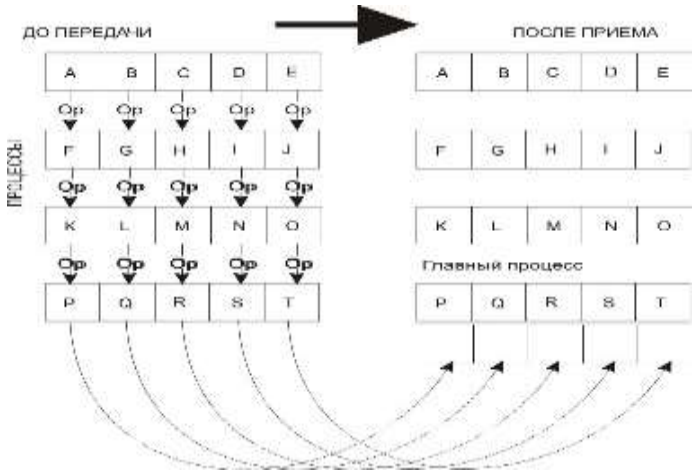
- **op** - операция приведения;
- **root** - ранг главного процесса.

Combine elements in input buffer of each process using the reduce operation, and return the combined value in the output buffer of the root process. Java binding of the MPI operation MPI_REDUCE.

# Коллективные обмены

## Операция приведения

- **MPI_Reduce** применяет операцию приведения к операндам из **buf**, а результат каждой операции помещается в буфер результата **result**
- **MPI_Reduce** должна вызываться всеми процессами в коммуникаторе **comm**, а аргументы **count**, **datatype** и **op** в этих вызовах должны совпадать.




| 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 & location |
| MPI_MINLOC | Minimum & location |

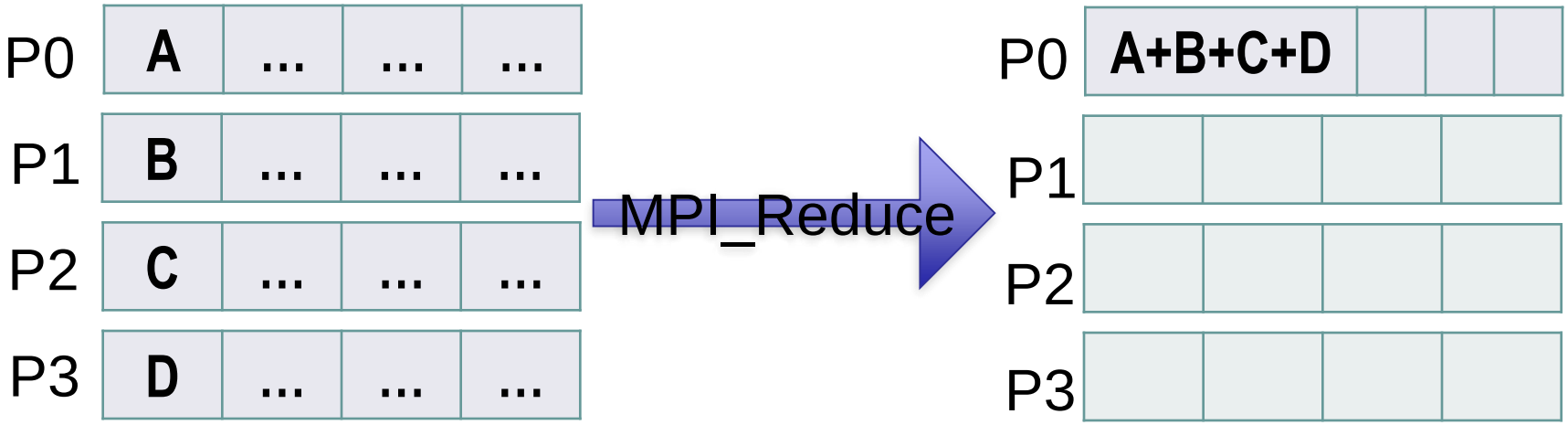
# Коллективные обмены

## Предопределенные операции приведения

| Операция | Описание |
|---|---|
| MPI_MAX | Определение максимальных значений элементов одномерных массивов целого или вещественного типа |
| MPI_MIN | Определение минимальных значений элементов одномерных массивов целого или вещественного типа |
| MPI_SUM | Вычисление суммы элементов одномерных массивов целого, вещественного или комплексного типа |
| MPI_PROD | Вычисление поэлементного произведения одномерных массивов целого, вещественного или комплексного типа |
| MPI_LAND | Логическое "И" |
| MPI_BAND | Битовое "И" |
| MPI_LOR | Логическое "ИЛИ" |
| MPI_BOR | Битовое "ИЛИ" |
| MPI_LXOR | Логическое исключающее "ИЛИ" |
| MPI_BXOR | Битовое исключающее "ИЛИ" |
| MPI_MAXLOC | Максимальные значения элементов одномерных массивов и их индексы |
| MPI_MINLOC | Минимальные значения элементов одномерных массивов и их индексы |

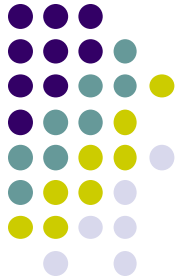
# Коллективные обмены

## Операция приведения


```
Размер sendbuf: sizeof(datatype) * count
Размер recvbuf: sizeof(datatype) * count
```

# Коллективные обмены



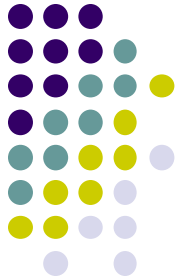
## Пример 1 использования операции редукции

- В качестве примера рассчитаем экспоненту (e). Один из вариантов ее нахождения — ряд Тейлора:

  ***e^x=∑((x^n)/n!)***,

  где суммирование происходит от ***n=0*** до бесконечности.
- Данная формула легко поддается распараллеливанию, так как искомое число является суммой отдельных слагаемых и благодаря этому каждый отдельный процессор может заняться вычислением отдельных слагаемых.
- Количество слагаемых, которое будет рассчитываться в каждом отдельно взятом процессоре, зависит как от длины интервала ***n***, так и от имеющегося количества процессов ***k***, которые будут участвовать в вычислениях. Так, например, если длина интервала ***n=4***, а в вычислениях участвуют пять процессов (***k=5***), то с первого по четвертый процессы получат по одному слагаемому, а пятый будет не задействован. В случае же если ***n=10***, а ***k=5***, каждому процессору достанется по два слагаемых для вычисления.

# Коллективные обмены

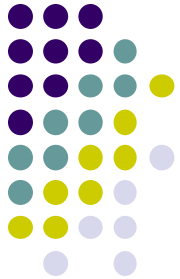


## Пример 1 Алгоритм

1. В программу передается значение числа ***n***, которое затем с помощью функции широковещательной рассылки отправляется по всем процессорам.
2. При инициализации главного процесса, запускается таймер.
3. Каждый процесс выполняет цикл, где значением приращения является количество процессов в системе. В каждой итерации цикла вычисляется слагаемое и сумма таких слагаемых сохраняется в переменную ***drobSum***.
4. После завершения цикла каждый процесс суммирует свое значение ***drobSum*** с переменной ***Result***, используя для этого функцию приведения ***MPI_Reduce.***
5. После завершения расчетов на всех процессах, главный процесс останавливает таймер и отправляет в поток вывода получившееся значение переменной ***Result***.
6. В поток вывода отправляется также и отмеренное таймером значение времени в милисекундах.

# Коллективные обмены

## Пример 1 (программа на C++)



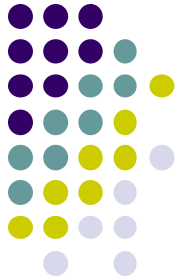
```
#include "mpi.h"
#include <iostream>
#include <windows.h>
using namespace std;
  double Fact(int n)
   {
      if (n==0)   return 1;
       else   return n*Fact(n-1);
       }
int main(int argc, char *argv[])
{
  SetConsoleOutputCP(1251);
  int n;
  int myrank;
  int numprocs;
  int i;
  long double drob,drobSum=0, Result, sum;
  double startwtime = 0.0;
  double endwtime;
```

# Коллективные обмены

## Пример 1 (окончание)

```
n = atoi(argv[1]); //аргумент для расчетов
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  if (myrank == 0)
            {  startwtime = MPI_Wtime();  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
          for (i = myid; i <= n; i += numprocs)
             {  drob = 1/Fact(i);
                drobSum += drob;
              }
  MPI_Reduce(&drobSum, &Result, 1, MPI_LONG_DOUBLE, MPI_SUM, 0,
  MPI_COMM_WORLD);
  cout.precision(20);
    if (myrank == 0)
    {
      cout << Result << endl;
      endwtime = MPI_Wtime();
      cout << (endwtime-startwtime)/1000 << endl;
    }
  MPI_Finalize();
  return 0;}
```

# Коллективные обмены



## Операция приведения MPI_Reduce
### (создание пользовательской операции)

Можно создать свою операцию помимо указанных:

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

MPI_Op_create(function, commute, op, ierr)
```

Входные параметры:

- **function** - пользовательская функция;
- **commute** - флаг, которому присваивается значение «истина», если операция коммутативна (результат не зависит от порядка операндов).

▪ Операция пользователя должна быть ассоциативной

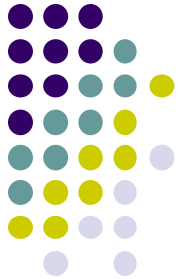
A * (B * C) = (A * B) * C

▪ Если commute = 1, то операция коммутативна:

A * B = B * A

# Коллективные обмены

## Операция приведения MPI_Reduce
### (создание пользовательской операции JAVA)



```
Op.Op(User_function function, boolean commute)
```

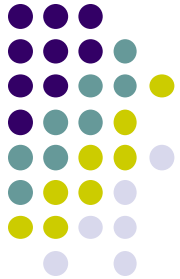
Входные параметры:

- **function** - пользовательская функция;
- **commute** - флаг, которому присваивается значение «истина», если операция коммутативна (результат не зависит от порядка операндов).

Bind a user-defined global reduction operation to an Op object. Java binding of the MPI operation MPI_OP_CREATE.
The abstract base class User function is defined by

```
class User_function {
        public abstract void Call(Object invec, int inoffset,
        Object inoutvec, int inoutoffset, int count,
        Datatype datatype);
}
```

# Коллективные обмены



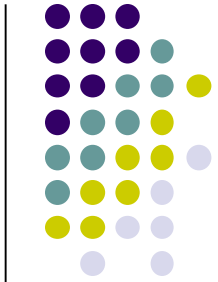
## Операция приведения MPI_Reduce
### (создание пользовательской операции)

Стандартный сценарий определения и использования производных типов включает следующие шаги:

- 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.
- Новый производный тип регистрируется вызовом функции MPI_Type_commit. Только после регистрации новый производный тип можно использовать в коммуникационных подпрограммах и при конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.
- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free.

# Коллективные обмены

## Операция приведения MPI_Reduce
### (создание пользовательской операции)



После завершения операций приведения пользовательская функция должна быть удалена.

Удаление пользовательской функции выполняется подпрограммой:

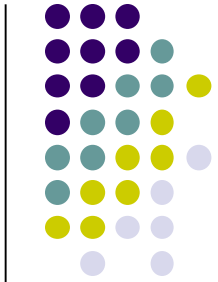
```
int MPI_Op_free(MPI_Op *op)
```

```
MPI_Op_free(op, ierr)  (C)
Void Op.finalize()     (JAVA)
```

После завершения вызова `op` присваивается значение `MPI_OP_NULL`.

# Коллективные обмены

## Операция приведения MPI_Reduce
### (создание производного типа JAVA)



```
static Datatype Datatype.Contiguous(int
count, Datatype oldtype)
```

**count** - replication count
**oldtype -** olddatatype

возвращает: новый тип дпнных, предтавляющий репликацию старого типа данных в новую локацию.
Java binding of the MPI operation MPI_TYPE_CONTIGUOUS.
Базовый тип нового типа данных тот же, что и базовый тип старого типа.

```
void Datatype.Commit()
```

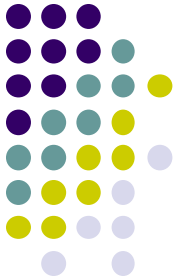
Commit a derived datatype. Java binding of the MPI operation MPI_TYPE_COMMIT.

```
void Datatype.finalize()
```

Destructor. Java binding of the MPI operation MPI TYPE FREE.

# Коллективные обмены

## Операция приведения MPI_Reduce
## (создание пользовательской операции)



число элементов базового типа, занимающих смежные области памяти.

Базовый тип, соответствующего datatype

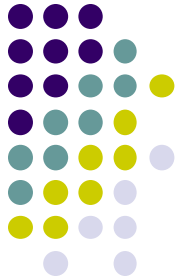
Название нового типа

### Пример 2

```
typedef struct {
double real, imag;
} Complex;
Complex sbuf[100], rbuf[100];

    MPI_Op complexmulop;
    MPI_Datatype ctype;
MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);//регистрация нового типа
// Умножение комплексных чисел
     MPI_Op_create(complex_mul, 1, &complexmulop);
     MPI_Reduce(sbuf, rbuf, 100, ctype, complexmulop,
        root, comm);
    MPI_Op_free(&complexmulop);
    MPI_Type_free(&ctype);
```

# Коллективные обмены



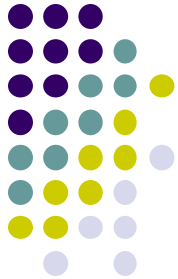
## Операция приведения MPI_Reduce
### (создание пользовательской операции)

**Пример 2 (окончание)**

```
// Умножение массивов комплексных чисел
void complex_mul(void *inv, void *inoutv, int *len,
MPI_Datatype *datatype)
{
 int i;
    Complex c;
    Complex *in = (Complex *) inv;
    *inout = (Complex *) inoutv;
for (i = 0; i < *len; i++) {
    c.real = inout->real * in->real-inout->imag * in->imag;
    c.imag = inout->real * in->imag +inout->imag * in->real;
    *inout = c;
    in++;
    inout++;
    }}
```

# Коллективные обмены

## Типы данных MPI

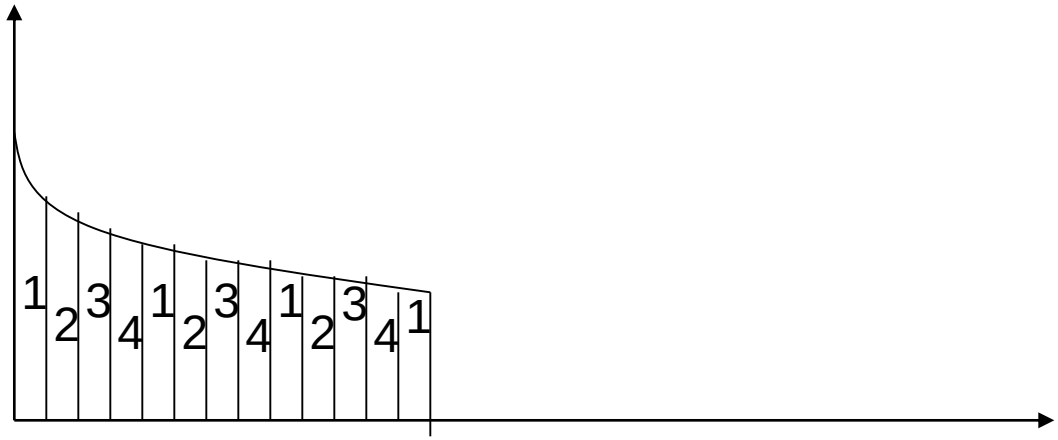


| Тип данных MPI | Тип данных C |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

# Коллективные обмены

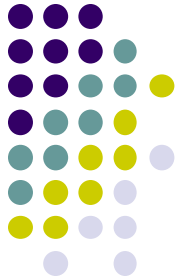
## Пример 3 (Вычисление числа $\pi$ )

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



# Коллективные обмены

## Пример 3 (Вычисление числа *π – продолжение* )



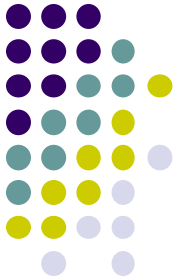
```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int n, myrank, numprocs, i;
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi, h, sum, x;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
```

# Коллективные обмены

## Пример 3 (Вычисление числа *π – продолжение* )

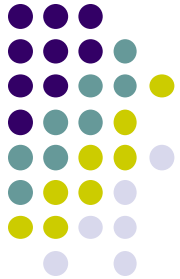


```
while (1)
{
  if (myrank == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
      }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n == 0) break;

  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myrank + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
  mypi = h * sum;
```

# Коллективные обмены

## Пример 3 (Вычисление числа *π – окончание* )

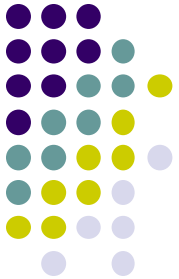


```
 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);

    if (myrank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```

# Коллективные обмены

## Операция сканирования



Операции *сканирования* (частичной редукции) выполняются следующей подпрограммой:

```
int MPI_Scan(void *sendbuf, void *rcvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_Scan(sendbuf, rcvbuf, count, datatype, op, comm,
ierr)
```

Входные параметры:

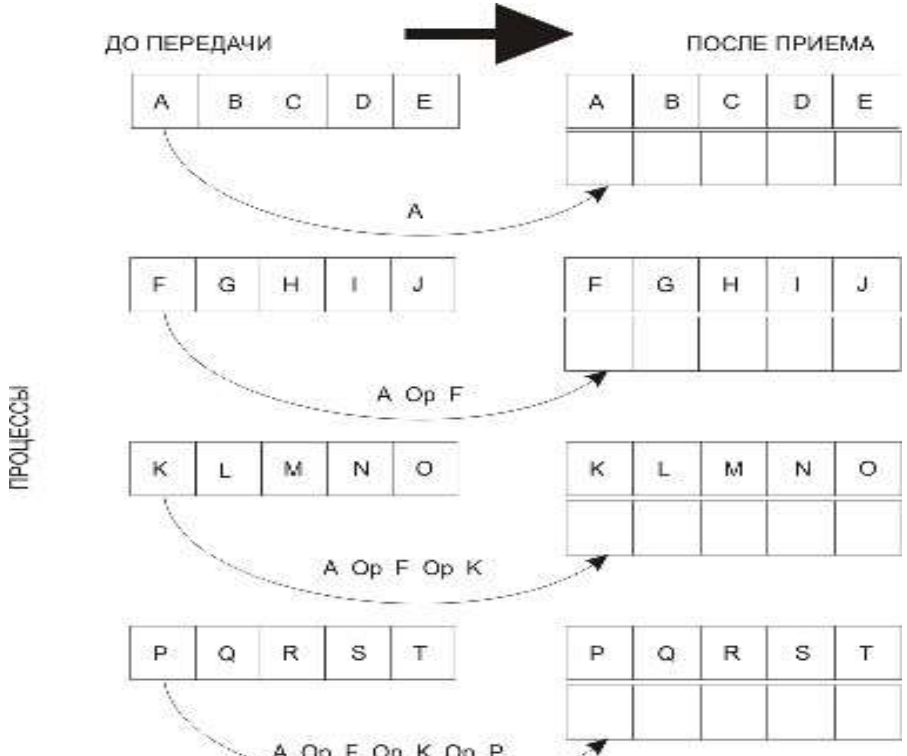
- **sendbuf** - начальный адрес буфера передачи;
- **count** - количество элементов во входном буфере;
- **datatype** - тип данных во входном буфере;
- **op** - операция;
- **comm** - коммуникатор.

Выходной параметр: **rcvbuf** - стартовый адрес буфера приема.

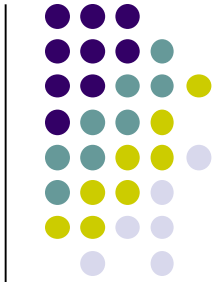
# Коллективные обмены

## Операция сканирования

При выполнении операции сканирования в буфере приёма процесса с рангом `i` будут содержаться результаты приведения значений в буферах передачи процессов с рангами `0, …, i`. В остальном эта операция аналогична операции `MPI_Reduce`.



# Синхронизация



- В ряде ситуаций независимо выполняемые в процессах вычисления необходимо синхронизировать. Так, например, для измерения времени начала работы параллельной программы необходимо, чтобы для всех процессов одновременно были завершены все подготовительные действия, перед окончанием работы программы все процессы должны завершить свои вычисления и т.п.
- Синхронизация процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

**int MPI_Barrier(MPI_Comm comm)**

**JAVA: MPI.COMM_WORLD.Barrier();**

Функция MPI_Barrier определяет коллективную операцию, и, тем самым, при использовании она должна вызываться всеми процессами используемого коммуникатора. При вызове функции MPI_Barrier выполнение процесса блокируется, продолжение вычислений процесса произойдет только после вызова функции MPI_Barrier всеми процессами коммуникатора.

# Коллективные обмены

## Синхронизация

Синхронизация с помощью «барьера» выполняется с помощью подпрограммы:

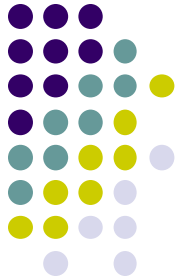
int MPI_Barrier(MPI_Comm comm)

**MPI_Barrier(comm, ierr)**

- Синхронизация с помощью «барьера» - простейшая форма синхронизации коллективных обменов. Она не требует пересылки данных.
- Обращение к подпрограмме MPI_Barrier блокирует выполнение каждого процесса из коммуникатора comm до тех пор, пока все процессы не вызовут эту подпрограмму, таким образом, «толщина барьера» здесь максимальная – она равна числу процессов в указанном коммуникаторе.
- Барьерная синхронизация относится к числу коллективных операций потому что выполнить соответствующий вызов должны все процессы.

# Коллективные обмены

## Синхронизация



Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован. Гарантирует, что к выполнению следующей за MPI_Barrier инструкции каждая задача приступит одновременно с остальными.

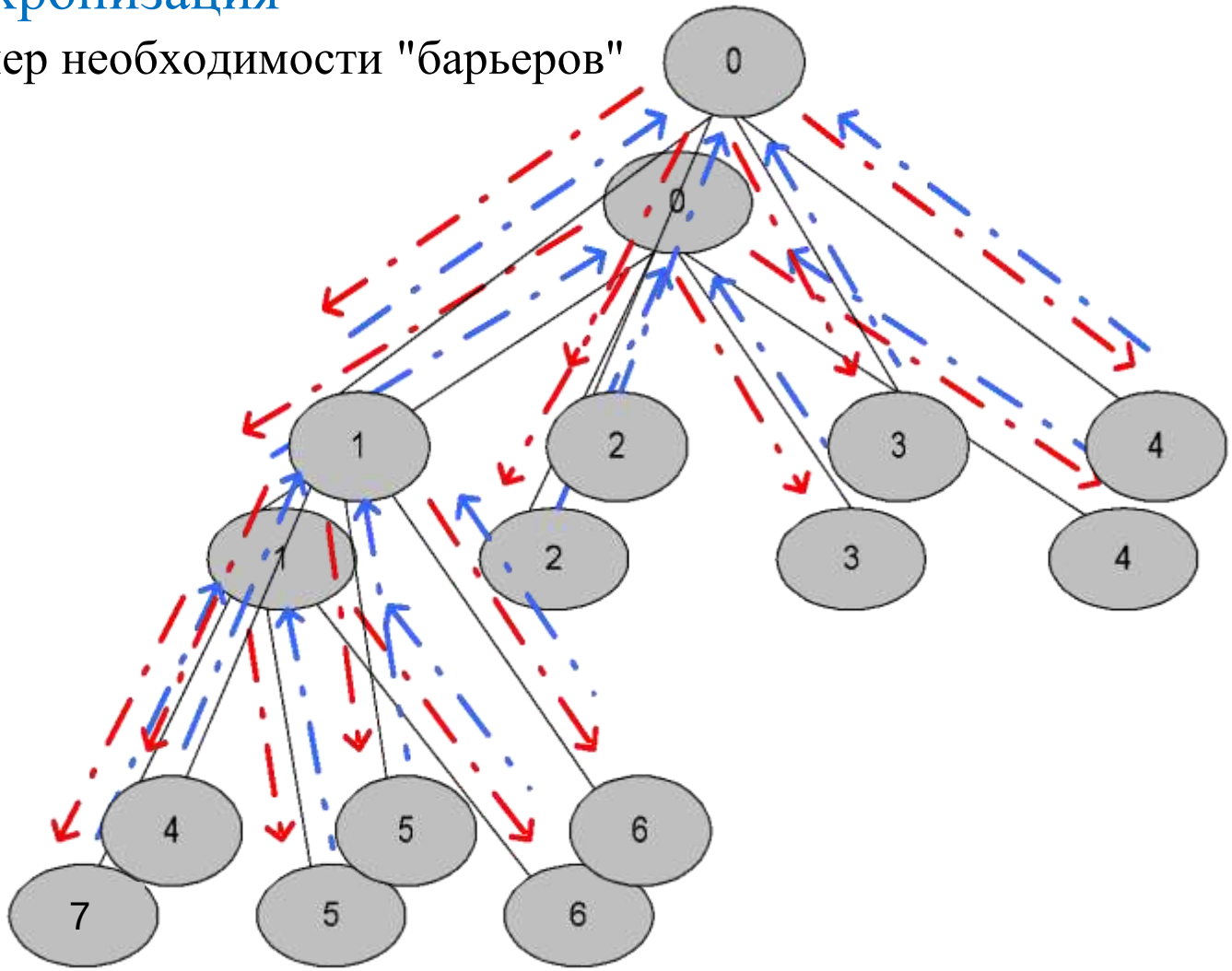
***Пример:***

```
int main(int argc,char* argv[])
{
/* Code */
MPI_Barrier(MPI_COMM_WORLD);
/* Code */
MPI_Finalize();
return 0;
}
```

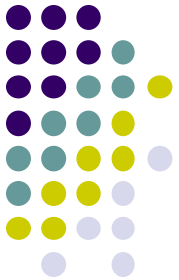
# Коллективные обмены

## Синхронизация

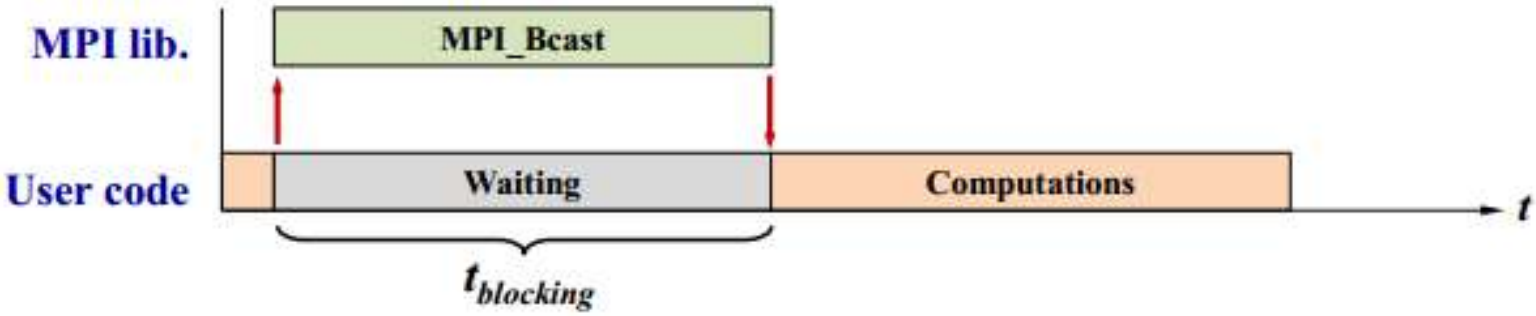
Пример необходимости "барьеров"



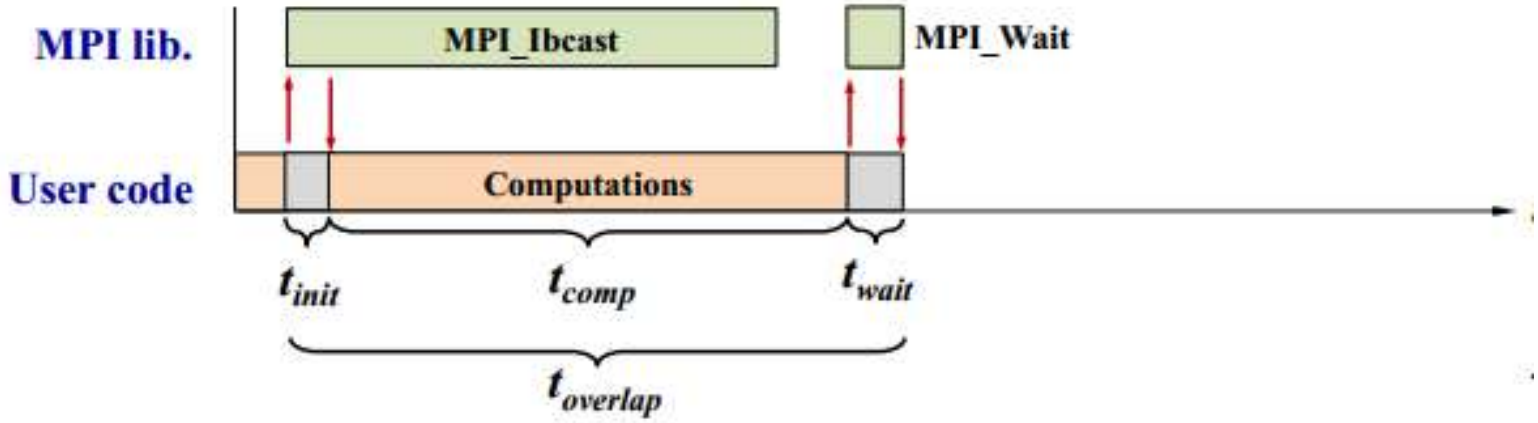
# Блокирующие и неблокирующие коллективные операции



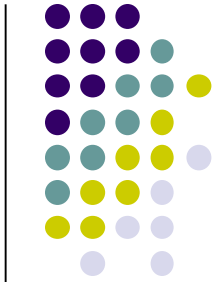
## Блокирующие коллективные операции




## Неблокирующие коллективные операции



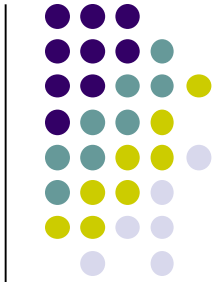
# Неблокирующие коллективные операции



- При вызове неблокирующей коллективной операции создается расписание выполнения обменов (collective schedule)
- **Progress engine** – механизм, который в фоновом режиме реализует обмены по созданному расписанию – как правило обмены выполняются при вызове MPI_Test (в противном случае необходим дополнительный поток)

```
MPI_Request req;
MPI_Ibcast(buf, count, MPI_INT, 0, MPI_COMM_WORLD, &req);
while (!flag) {
// Вычисления...
// Проверяем состояние и продвигаем обмены по расписанию
MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
}
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

# Заключение



В этом занятии мы рассмотрели:

- особенности и свойства коллективных обменов;
- различные операции коллективного обмена – широковещательную рассылку, сбор и распределение данных, приведение и сканирование и т. д.;
- синхронизацию при организации коллективных обменов.

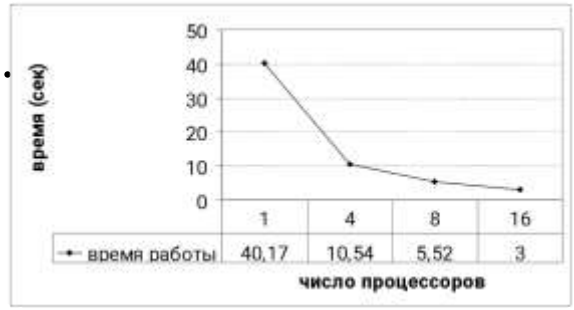
# Задания на лабораторную

4.1. Два вектора a и b размерности N представлены двумя одномерными массивами, содержащими каждый по N элементов. Напишите параллельную MPI-программу вычисления скалярного произведения этих векторов используя два любых известных способа двухточечного обмена сообщениями. Программа должна быть организована по схеме master-slave, причем master-процесс должен пересылать подчиненным процессам одинаковые(или почти одинаковые) по количеству элементов фрагменты векторов.

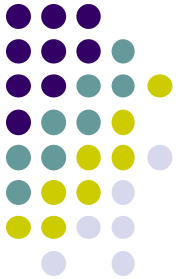
4.2. Решить задачу о нахождении скалярного произведения векторов A и B с учетом знания принципов коллективных обменов

- с помощью функции Broadcast/Reduse;
- с помощью функций Scatter(v) / Gather(v).

Все результаты сравнить, используя засечение времени *(MPI.Wtime())* и оформить в виде графиков:



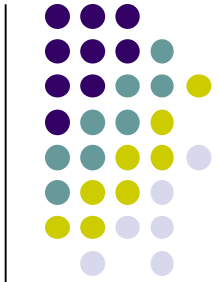
# Задания на лабораторную



5. Задачи с графами.
Задание выполняется по вариантам: Вариант определяет преподаватель. Каждый студент должен выполнить по 1 задаче из таблицы. Входные данные – матрица смежности.
*Исходные данные сформировать так, чтоб их легко было проверить (возможно Вам понадобятся несколько вариантов исходных данных ).*
***Выполнить подсчет временных затрат с разным кол-вом потоков, привести графики. ОТЧЕТ ОБЯЗАТЕЛЕН.***

| № | Задание |
|---|---|
| 0 | Разработать алгоритм вычисления диаметра произвольного неориентированного графа. |
| 1 | Разработать алгоритм вычисления максимальной из степеней вершин в графе. |
| 2 | Разработать алгоритм вычисления количества ребер в графе. |
| 3 | Разработать алгоритм вычисления центра графа |
| 4 | Разработать алгоритм определения того, является ли граф деревом. |
| 5 | Разработать алгоритм определения того, является ли граф тором. |
| 6 | Разработать алгоритм определения того, является ли граф гиперкубом. |
| 7 | Разработать алгоритм определения того, является ли граф регулярным. |

# Обзор технологий параллельного программирования
## (коллективные обмены)



***Вопросы к экзамену:***

1. Виды двухточечных обменов в MPI. Их схемы, достоинства и недостатки.
2. Принципы работы с коммуникторами.

Доп. Информация:
http://rsusu1.rnd.runnet.ru/tutor/method/m2/content.html