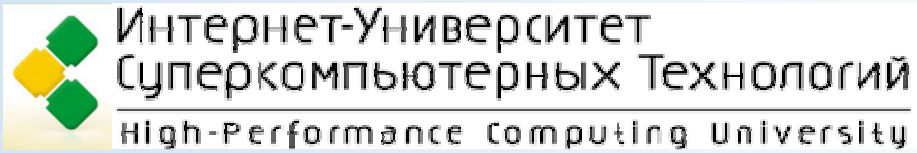
Основы параллельного программирования с использованием MPI

Лекция 6

Немнюгин Сергей Андреевич
Санкт-Петербургский государственный университет
кафедра вычислительной физики

snemnyugin@mail.ru



Лекция 6

Аннотация

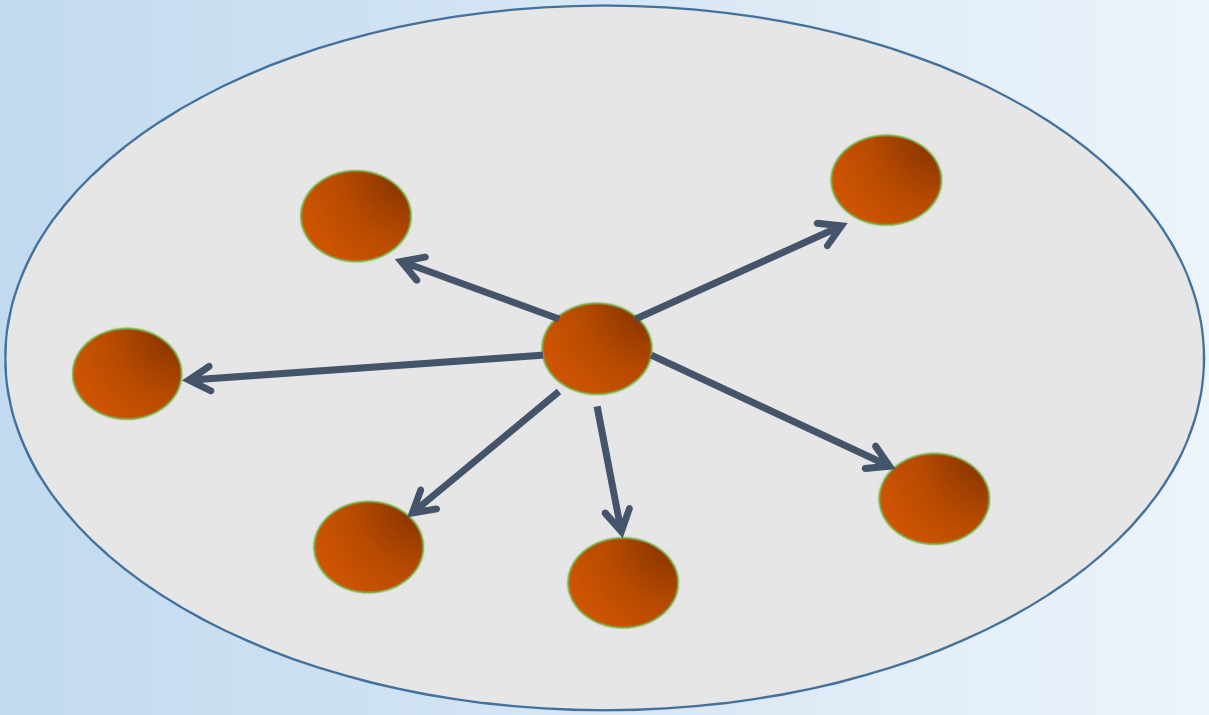
В лекции рассматриваются коллективные обмены. Обсуждаются такие операции, как широковещательная рассылка, распределение и сбор данных, а также операция приведения (редукции). Внимание уделяется роли синхронизации в параллельном программировании и средствам синхронизации в MPI.

План лекции

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

Коллективные обмены

В операции коллективного обмена вовлечены не два, а большее число процессов.
Пример – *широковещательная рассылка*:



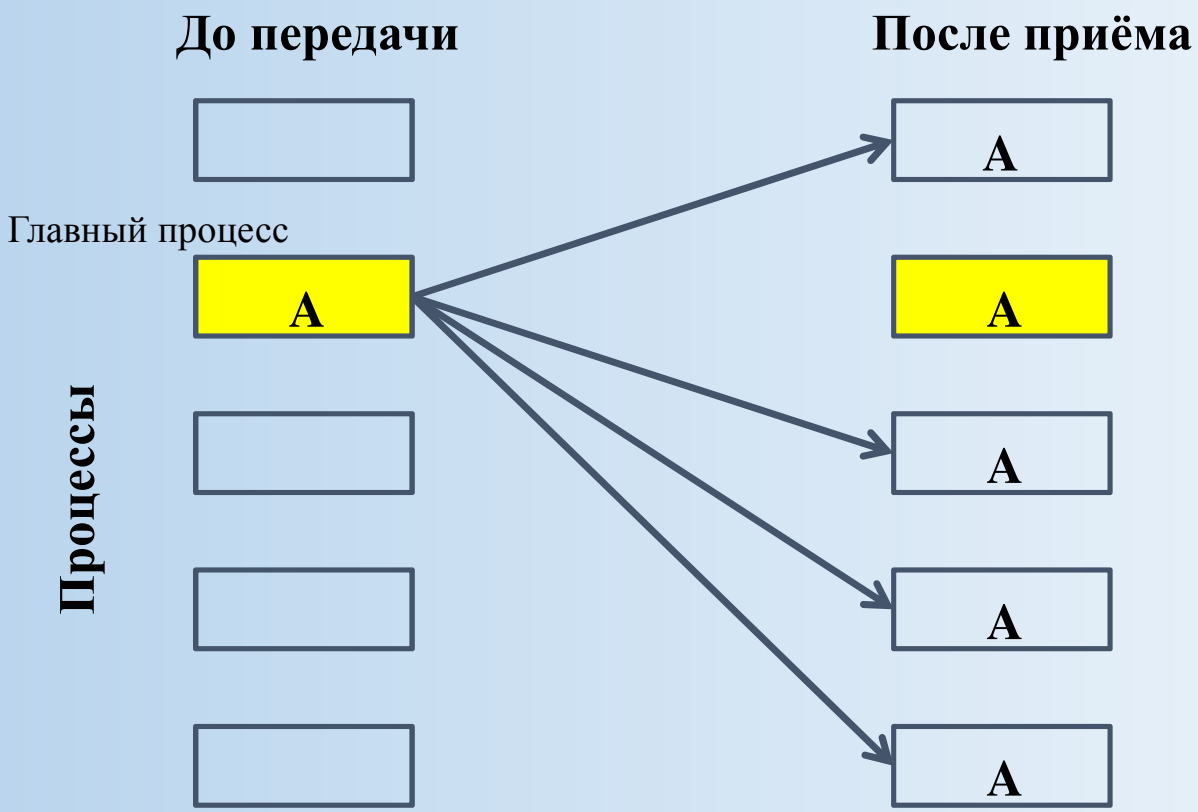
Общая характеристика коллективных обменов:

- 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;
- все коллективные обмены являются блокирующими для инициировавшего их обмена;
- теги сообщений в коллективных обменах назначаются системой.

Виды коллективных обменов:

- *широковещательная передача* - выполняется от одного процесса ко всем;
- *распределение* данных;
- *сбор* данных;
- *обмен с барьером* - это форма синхронизации работы процессов, когда обмен сообщениями происходит только после того, как к соответствующей процедуре обратилось определенное число процессов;
- *операции сканирования*;
- *операции приведения*. Входными являются данные нескольких процессов, а результат одно значение, которое становится доступным всем процессам, участвующим в обмене.

Операция широковещательной рассылки



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.

Пример. Использование широковещательной рассылки

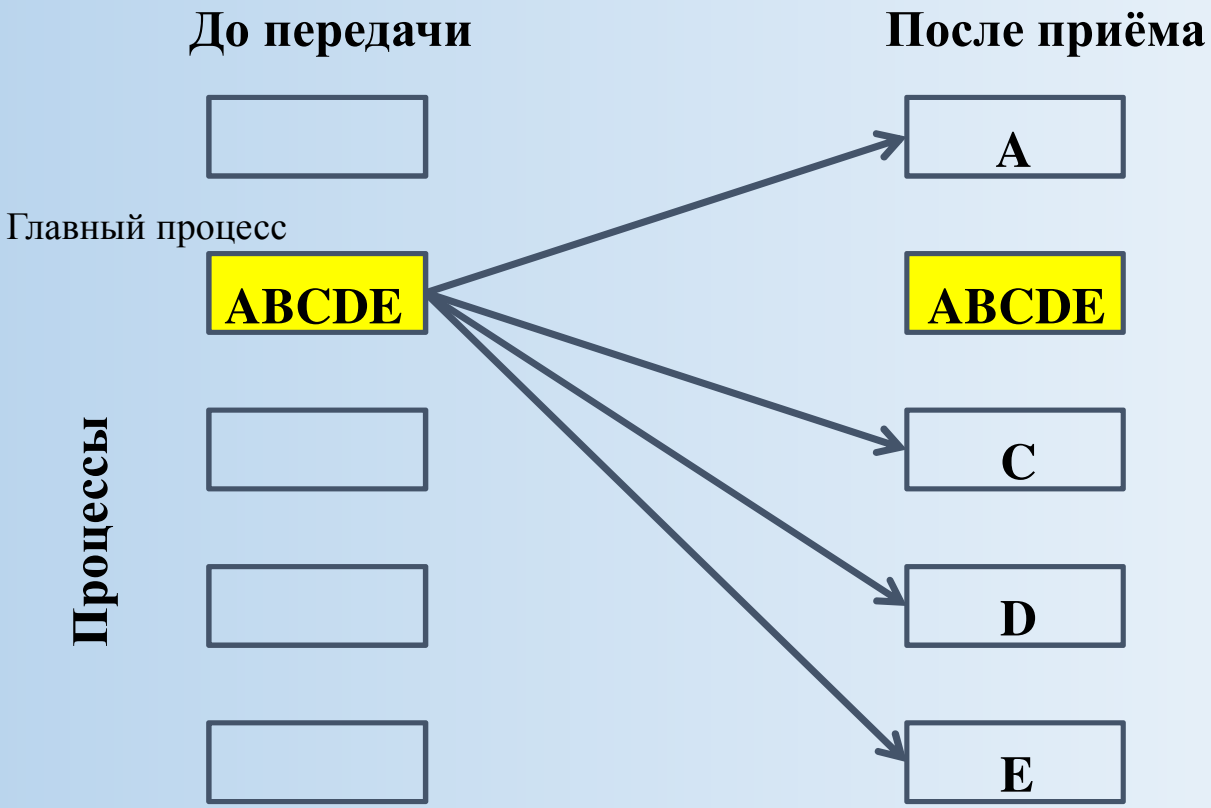
```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 char data[24];
 int myrank, count = 25;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if (myrank == 0)
{
 strcpy(data, "Hi, Parallel Programmer!");
 MPI_Bcast(&data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
 printf("send: %s\n", data);
}
Else
 MPI_Bcast(&data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
 printf("received: %s\n", data);
}
 MPI_Finalize();
 return 0;
}
```

Пример. Использование 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);
 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;
}
```

Операция распределения данных

Процесс с рангом `root` распределяет содержимое буфера передачи `sendbuf` среди всех процессов. 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`.



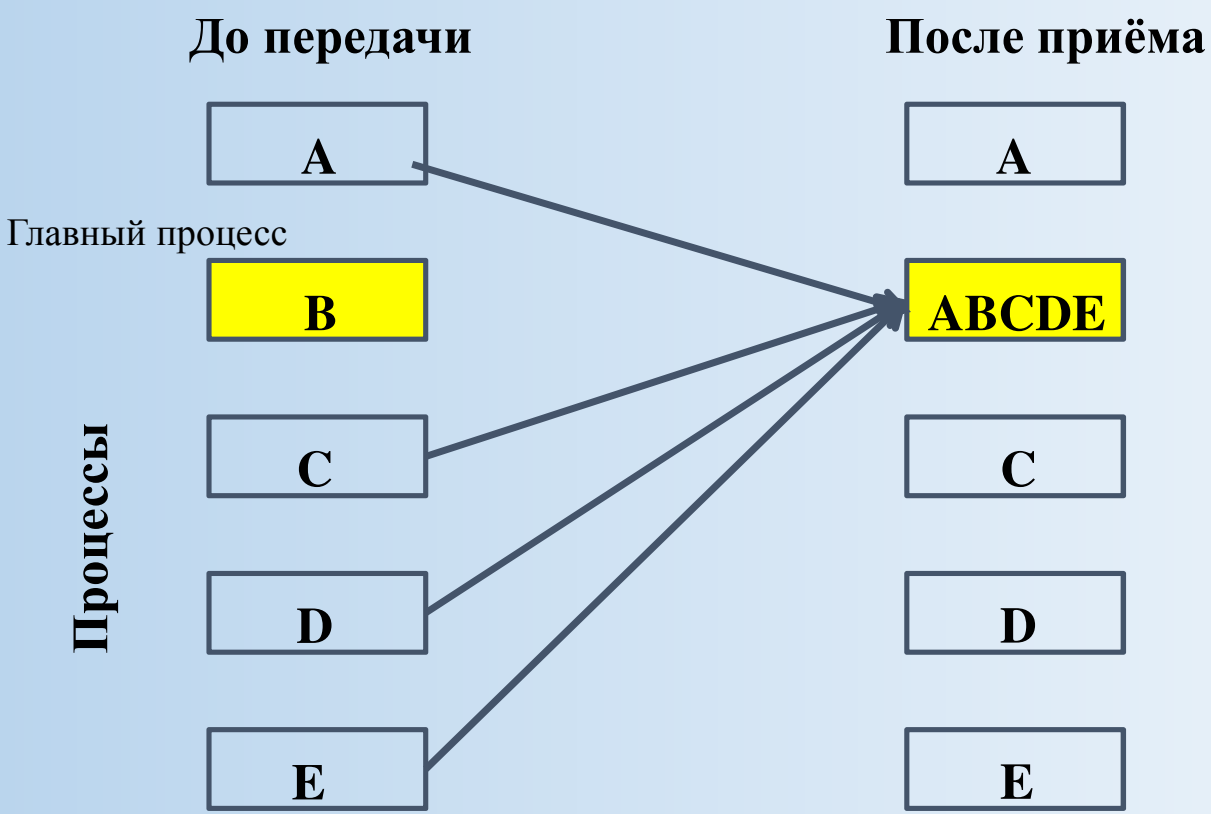
```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm)

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.

Операция сбора данных



Сбор данных от остальных 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.

Порядок склейки определяется рангами процессов:

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.

Сбор данных от всех процессов и распределение их всем процессам:

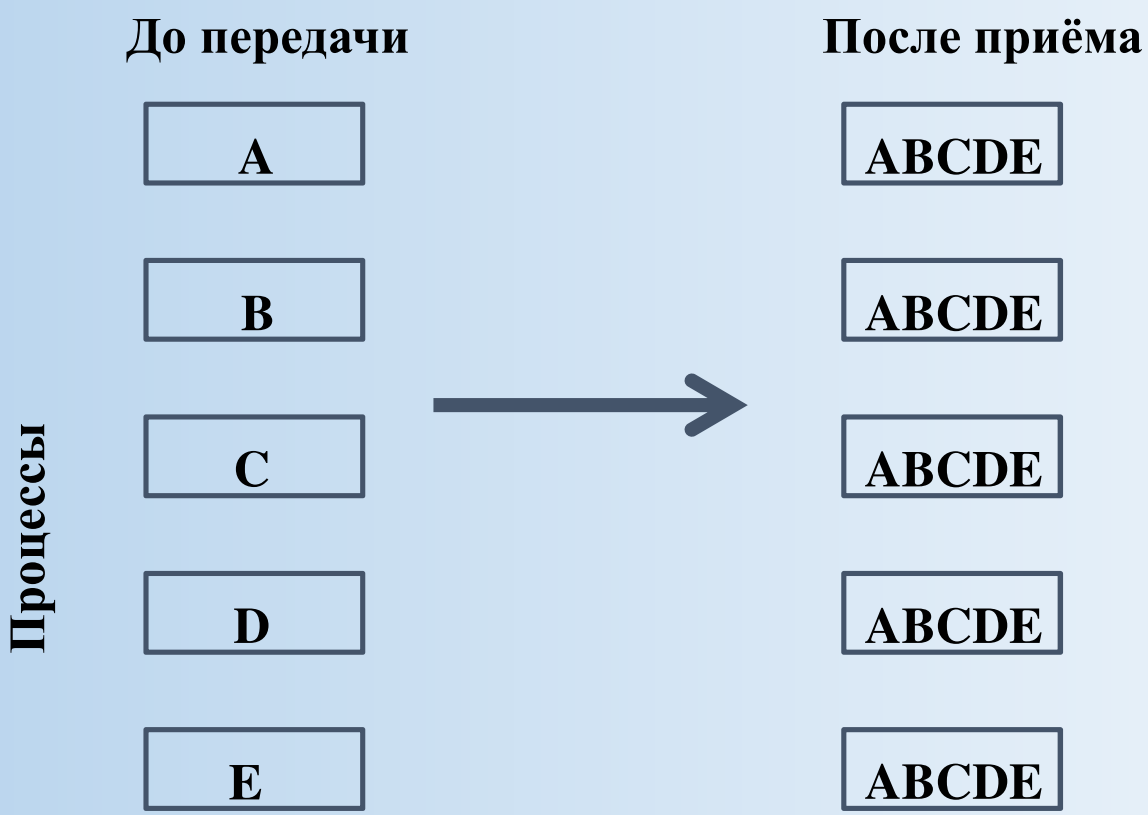
```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, MPI_Comm comm)

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.

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



Операция приведения (редукции)

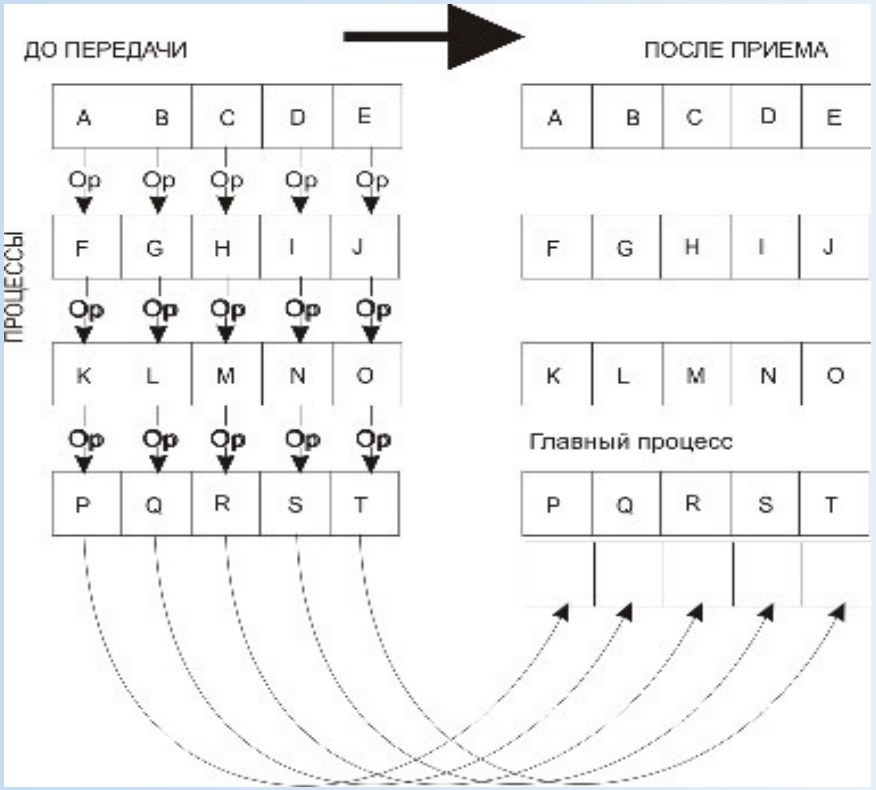
Операция приведения, результат которой передается одному процессу

```
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.

`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.



Пример. Использование операции редукции

В этой программе сначала создается подгруппа, состоящая из процессов с рангами 1, 3, 5 и 7 (запускать ее на выполнение надо не менее чем в восьми процессах), и соответствующий ей коммуникатор. Редукция выполняется только процессами из этой группы. В конце программы все созданные в процессе ее работы описатели должны быть удалены. Работе с группами и коммуникаторами посвящена следующая, 6 лекция.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 int myrank, i;
 int count = 5, root = 1;
 MPI_Group MPI_GROUP_WORLD, subgroup;
 int ranks[4] = {1, 3, 5, 7};
 MPI_Comm subcomm;
 int sendbuf[5] = {1, 2, 3, 4, 5};
 int recvbuf[5];

 MPI_Init(&argc, &argv);
 MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
 MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup);
 MPI_Group_rank(subgroup, &myrank);
 MPI_Comm_create(MPI_COMM_WORLD, subgroup, &subcomm);
```

```
if(myrank != MPI_UNDEFINED)
{
MPI_Reduce(&sendbuf, &recvbuf, count, MPI_INT, MPI_SUM, root, subcomm);

if(myrank == root) {
printf("Reduced values");
for(i = 0; i < count; i++){
printf(" %i ", recvbuf[i]);}
}
printf("\n");

MPI_Comm_free(&subcomm);
MPI_Group_free(&MPI_GROUP_WORLD);
MPI_Group_free(&subgroup);
}
 MPI_Finalize();
 return 0;
}
```

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

Допускается определение собственных операций приведения. Для этого используется подпрограмма:

```
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).

Описание типа пользовательской функции выглядит следующим образом:

```
typedef void (MPI_User_function)(void *a, void *b, int *len, MPI_Datatype
*dtype)
```

Здесь операция определяется так:

```
b[I] = a[I] op b[I]
```

для `I = 0, …, len – 1`.

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)
```

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

Пример. Использование операции приведения: вычисление числа π методом Монте-Карло

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
double dboard (int trials);

#define trials 5000
#define ROUNDS 10
#define MASTER 0

main(int argc, char **argv)
{
double homepi, pisum, pi, avepi;
int mytid, nproc, rcode, ;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
printf ("MPI task ID = %d\n", mytid);
```

```
srandom (mytid);

avepi = 0;
for (i = 0; i < ROUNDS; i++) {
     homepi = dboard(trials);

      rcode = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM, MASTER, MPI_COMM_WORLD);
     if (rcode != 0)
              printf("%d: failure on MPI_Reduce\n", mytid);

     if (mytid == MASTER) {
          pi = pisum/nproc;
          avepi = ((avepi * i) + pi)/(i + 1);
          printf("   After %3d throws, average value of pi = %10.8f\n",
                 (trials * (i + 1)),avepi);
          }
    }
    MPI_Finalize();
}
```

```
#include <stdlib.h>
#define sqr(x)  ((x)*(x))
long random(void);

double dboard(int trials)
     {
     double x_coord, y_coord, pi, r;
     int score, n;
     unsigned long cconst;
     cconst = 2 << (31 - 1);
     score = 0;

     for (n = 1; n <= trials; n++)  {
          r = (double)random()/cconst;
          x_coord = (2.0 * r) - 1.0;
          r = (double)random()/cconst;
          y_coord = (2.0 * r) - 1.0;

          if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
               score++;
          }
pi = 4.0 * (double)score/(double)trials;
     return(pi);
     }
```

Операция сканирования

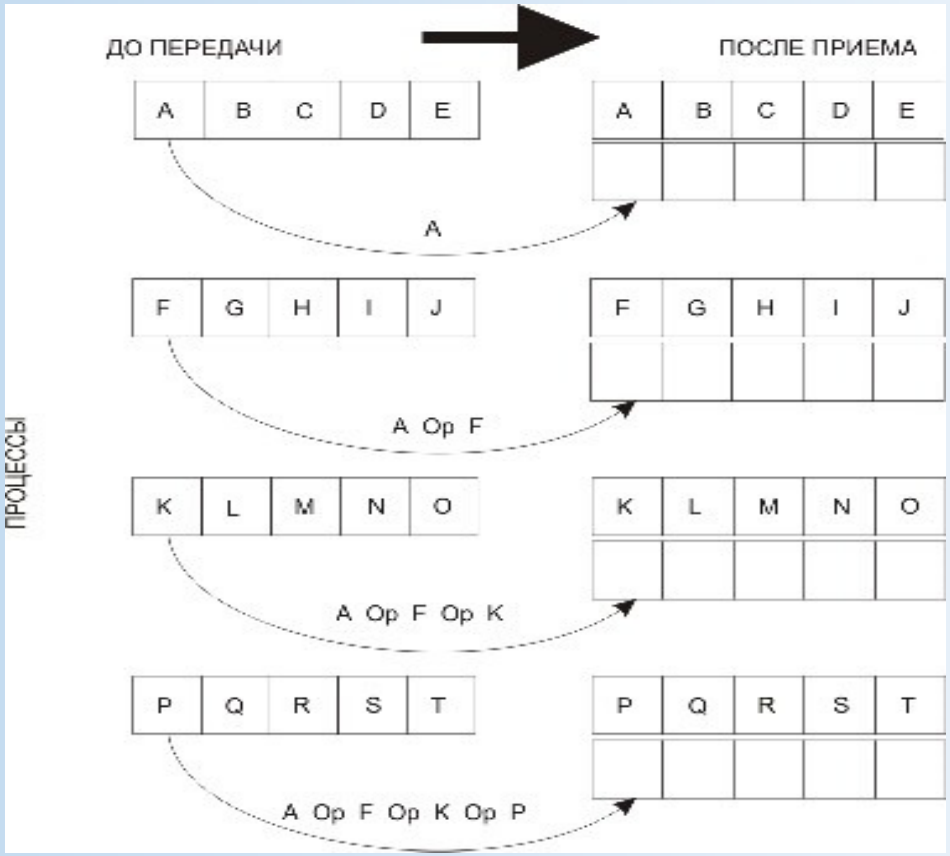
Операции сканирования (частичной редукции) выполняются следующей подпрограммой:

```
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.

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

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.

В общем случае барьер характеризуется ***«толщиной»***. «Толщина» – количество процессов, которые должны обратиться к подпрограмме барьерной синхронизации для того, чтобы барьер был преодолён.

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.

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**.**

Векторная операция сбора данных

Сбор данных от всех процессов в заданном коммуникаторе и запись их в буфер приема с указанным смещением:

```
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.

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.

Векторными версиями `MPI_Allgather` и `MPI_Alltoall` являются подпрограммы `MPI_Allgatherv` и `MPI_Alltoallv`.

Группы процессов и коммуникаторы

В MPI имеются средства создания и преобразования коммуникаторов, которые дают возможность программисту в дополнение к стандартным предопределенным объектам создавать свои собственные. Это позволяет использовать разнообразные схемы взаимодействия процессов.

Используя средства MPI, можно создать новый коммуникатор, содержащий, например, те же процессы, что и исходный, но с новым контекстом (новыми свойствами).

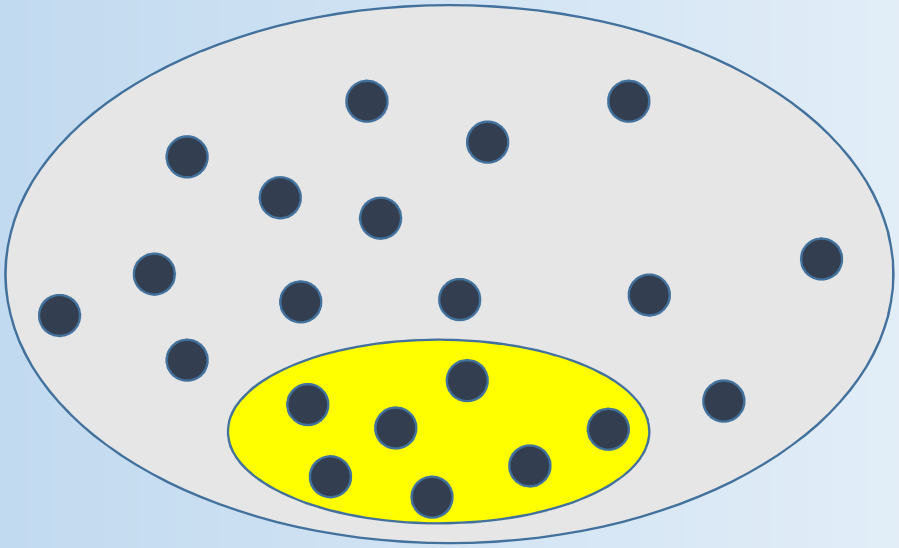
Обмен возможен только в рамках одного контекста, обмены в разных коммуникаторах происходят независимо.

Группы процессов

Группой называют упорядоченное множество процессов.

Каждому процессу в группе сопоставлен свой ранг. Операции с группами могут выполняться отдельно от операций с коммуникаторами, но в операциях обмена используются **только** коммуникаторы.

В MPI имеется специальная предопределенная *пустая* группа `MPI_GROUP_EMPTY`.



Коммуникаторы

Коммуникаторы бывают двух типов:

1. *интракоммуникаторы* — для операций внутри одной группы процессов. Интракоммуникатором является `MPI_COMM_WORLD`;
2. *интеркоммуникаторы* — для двухточечного обмена между двумя группами процессов.

В MPI-программах чаще используются интракоммуникаторы.

Интракоммуникатор включает экземпляр группы, контекст обмена для всех его видов, а также, возможно, виртуальную топологию и другие атрибуты. Контекст обеспечивает возможность создания изолированных друг от друга, а потому безопасных областей взаимодействия. Система сама управляет их разделением. Контекст играет роль дополнительного тега, который дифференцирует сообщения.

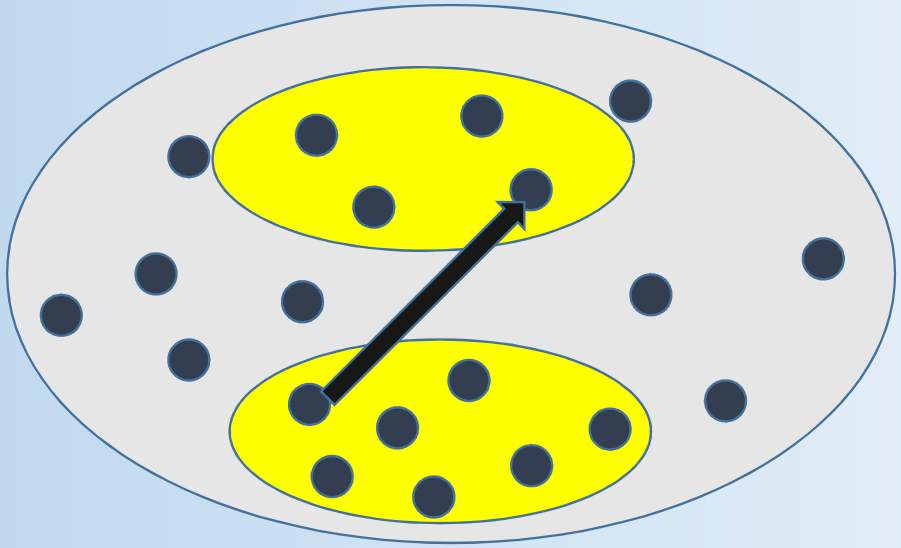
Можно организовать обмены между двумя непересекающимися группами процессов.

Если параллельная программа состоит из нескольких параллельных модулей, удобно разрешить одному модулю обмениваться сообщениями с другим, используя для адресации локальные по отношению ко второму модулю ранги. Такой подход удобен, например, при программировании параллельных клиент-серверных приложений.

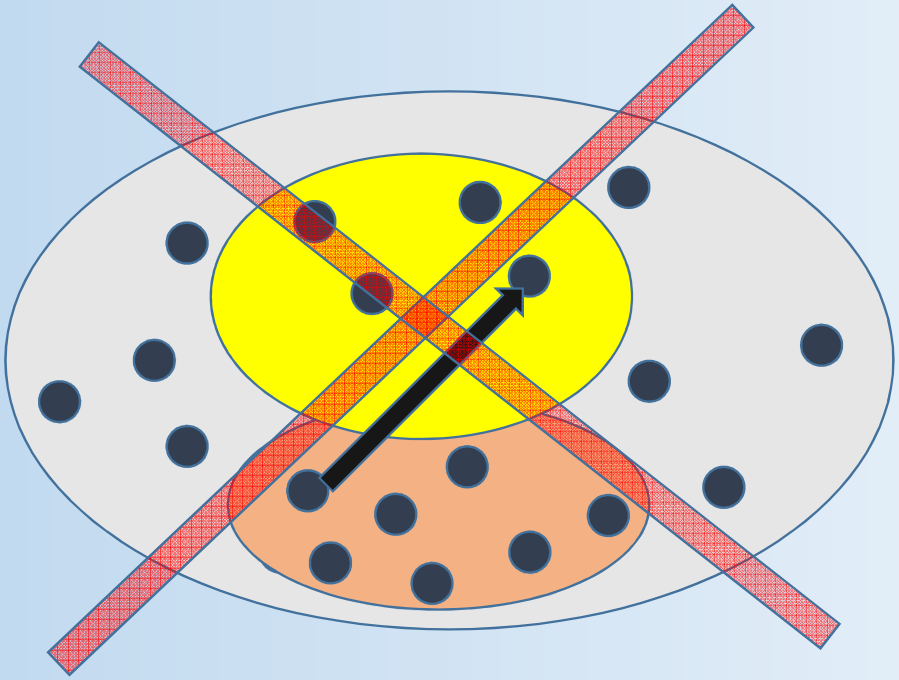
Интеробмены реализуются с помощью ***интеркоммуникаторов***, которые объединяют две группы процессов общим контекстом.

В контексте, связанном с интеркоммуникатором, передача сообщения в локальной группе всегда сопровождается его приемом в удаленной группе — это двухточечная операция. Группа, содержащая процесс, который инициирует операцию интеробмена, называется ***локальной*** группой, а группа, содержащая процесс-адресат, называется ***удаленной*** группой.

Интеробмен задается парой: **коммуникатор — ранг**, при этом ранг задается относительно удаленной группы.



Конструкторы интеркоммуникаторов являются блокирующими операциями, поэтому во избежание тупиковых ситуаций локальная и удаленная группа не должны пересекаться (они не должны содержать одинаковые процессы).

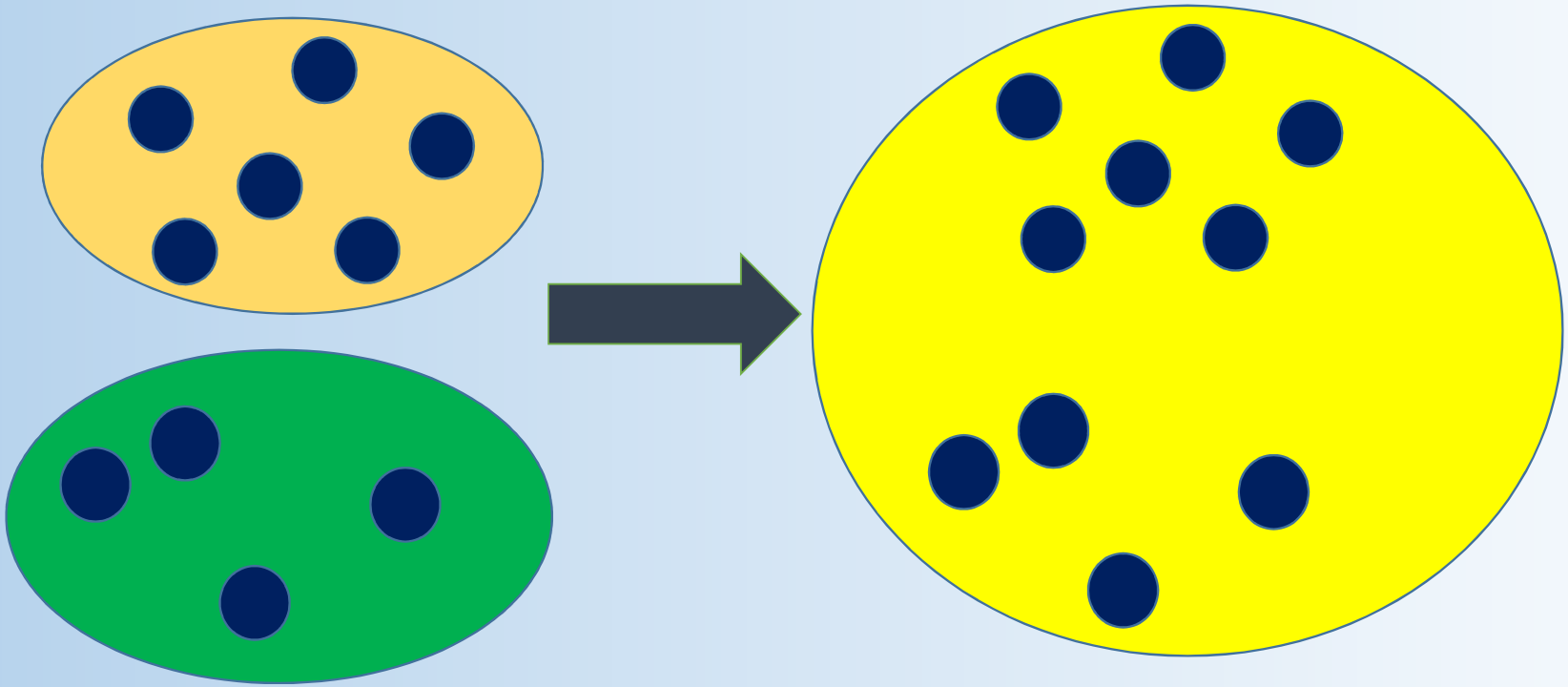


Создание групп процессов

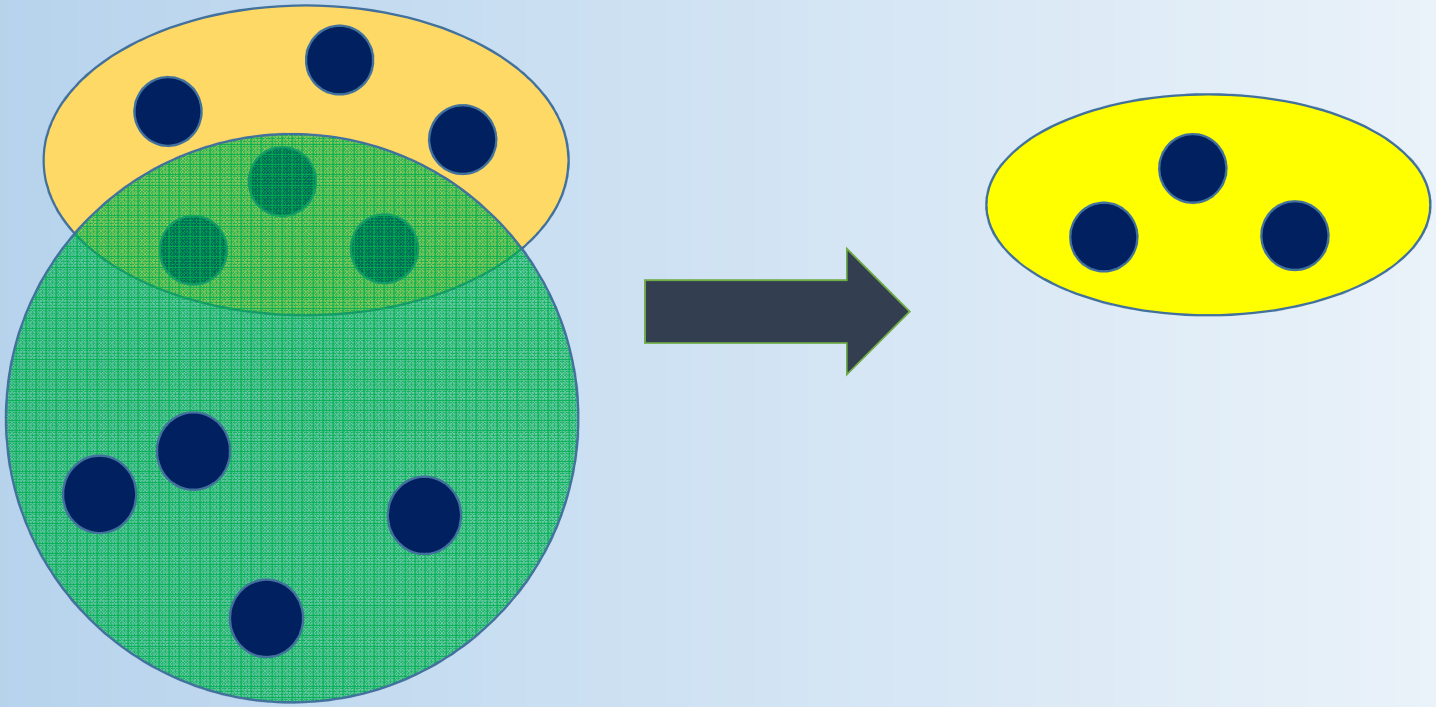
Создание групп процессов

Созданию нового коммуникатора предшествует создание соответствующей группы процессов. Операции создания групп аналогичны математическим операциям над множествами:

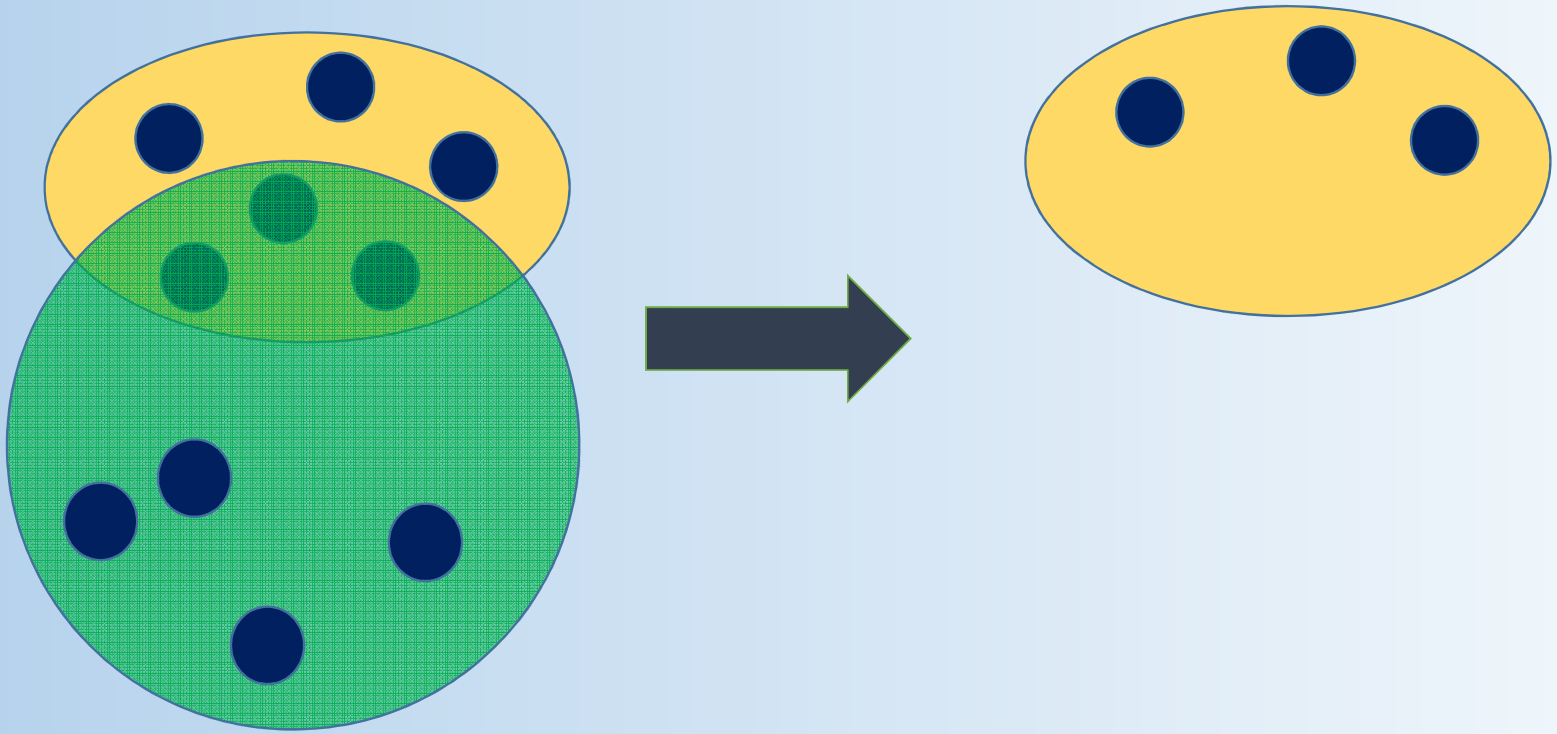
объединение — к процессам первой группы (`group1`) добавляются процессы второй группы (`group2`), не принадлежащие первой;



пересечение — в новую группу включаются все процессы, принадлежащие двум группам одновременно. Ранги им назначаются как в первой группе;



разность — в новую группу включаются все процессы первой группы, не входящие во вторую группу. Ранги назначаются как в первой группе.



Новую группу можно создать только из уже существующих групп. Базовая группа, из которой формируются все другие группы, связана с коммуникатором `MPI_COMM_WORLD`.

В подпрограммах создания групп, как правило, нельзя использовать пустой коммуникатор `MPI_COMM_NULL`.

Доступ к группе `group`, связанной с коммуникатором `comm` можно получить, обратившись к подпрограмме `MPI_Comm_group`:

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_Comm_group(comm, group, ierr)
```

Ее выходным параметром является группа.

Для выполнения операций с группой к ней сначала необходимо получить доступ.

Подпрограмма `MPI_Group_incl` создает новую группу `newgroup` из n процессов, входящих в группу `oldgroup`. Ранги этих процессов содержатся в массиве `ranks`:

```
int MPI_Group_incl(MPI_Group oldgroup, int n, int *ranks,
MPI_Group *newgroup)

MPI_Group_incl(oldgroup, n, ranks, newgroup, ierr)
```

В новую группу войдут процессы с рангами `ranks[0], …, ranks[n – 1]`, причем рангу i в новой группе соответствует ранг `ranks[i]` в старой группе. При n = 0 создается пустая группа `MPI_GROUP_EMPTY`.

С помощью данной подпрограммы можно не только создать новую группу, но и изменить порядок нумерации (назначения рангов) процессов в старой группе.

Подпрограмма `MPI_Group_excl` создает группу `newgroup`, исключая из исходной группы (`group`) процессы с рангами `ranks[0], …, ranks[n – 1]`:

```
int MPI_Group_excl(MPI_Group oldgroup, int n, int *ranks, MPI_Group
*newgroup)

MPI_Group_excl(oldgroup, n, ranks, newgroup, ierr)
```

При n = 0 новая группа тождественна старой.

Подпрограмма `MPI_Group_range_incl` создает группу `newgroup` из группы `group`, добавляя в нее `n` процессов, ранг которых указан в массиве `ranks`:

```
int MPI_Group_range_incl(MPI_Group oldgroup, int n, int ranks[][3],
MPI_Group *newgroup)

MPI_Group_range_incl(oldgroup, n, ranks, newgroup, ierr)
```

Массив `ranks` состоит из целочисленных триплетов вида (`первый_1`, `последний_1`, `шаг_1`), …, (`первый_n`, `последний_n`, `шаг_n`). В новую группу войдут процессы с рангами (по первой группе) первый_1, первый_1 + шаг_1, ….

Подпрограмма `MPI_Group_range_excl` создает группу `newgroup` из группы `group`, исключая из нее n процессов, ранг которых указан в массиве `ranks`:

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranks[][3],
MPI_Group *newgroup)

MPI_Group_range_excl(group, n, ranks, newgroup, ierr)
```

Массив `ranks` устроен так же, как аналогичный массив в подпрограмме `MPI_Group_range_incl`.

Подпрограмма `MPI_Group_difference` создает новую группу (`newgroup`) из разности двух групп (`group1`) и (`group2`):

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)

MPI_Group_difference(group1, group2, newgroup, ierr)
```

Подпрограмма `MPI_Group_intersection` создает новую группу (`newgroup`) из пересечения групп `group1` и `group2`:

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)

MPI_Group_intersection(group1, group2, newgroup, ierr)
```

Подпрограмма `MPI_Group_union` создает группу (`newgroup`), объединяя группы `group1` и `group2`:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group
*newgroup)

MPI_Group_union(group1, group2, newgroup, ierr)
```

Имеются и другие подпрограммы-конструкторы новых групп.

Деструктор группы

Вызов подпрограммы `MPI_Group_free` уничтожает группу `group`:

```
int MPI_Group_free(MPI_Group *group)
```

```
MPI_Group_free(group, ierr)
```

Удаление группы не приводит к завершению процессов, входящих в эту группу. Удаляется лишь формальный объект – объединение процессов логикой выполнения программы.

Получение информации о группе

Для определения количества процессов (`size`) в группе (`group`) используется подпрограмма `MPI_Group_size`:

```
int MPI_Group_size(MPI_Group group, int *size)

MPI_Group_size(group, size, ierr)
```

Подпрограмма `MPI_Group_rank` возвращает ранг (`rank`) процесса в группе `group`:

```
int MPI_Group_rank(MPI_Group group, int *rank)

MPI_Group_rank(group, rank, ierr)
```

Если процесс не входит в указанную группу, возвращается значение `MPI_UNDEFINED`.

Процесс может входить в несколько групп.

Подпрограмма `MPI_Group_translate_ranks` преобразует ранг процесса в одной группе в его ранг в другой группе:

```
int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1,
MPI_Group group2, int *ranks2)

MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierr)
```

Эта функция используется для определения относительной нумерации одних и тех же процессов в двух разных группах.

Подпрограмма `MPI_Group_compare` используется для сравнения групп `group1` и `group2`:

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int
*result)

MPI_Group_compare(group1, group2, result, ierr)
```

Если группы полностью совпадают, возвращается значение `MPI_IDENT`. Если члены обеих групп одинаковы, но их ранги отличаются, результатом будет значение `MPI_SIMILAR`.

Если группы различны, результатом будет `MPI_UNEQUAL`.

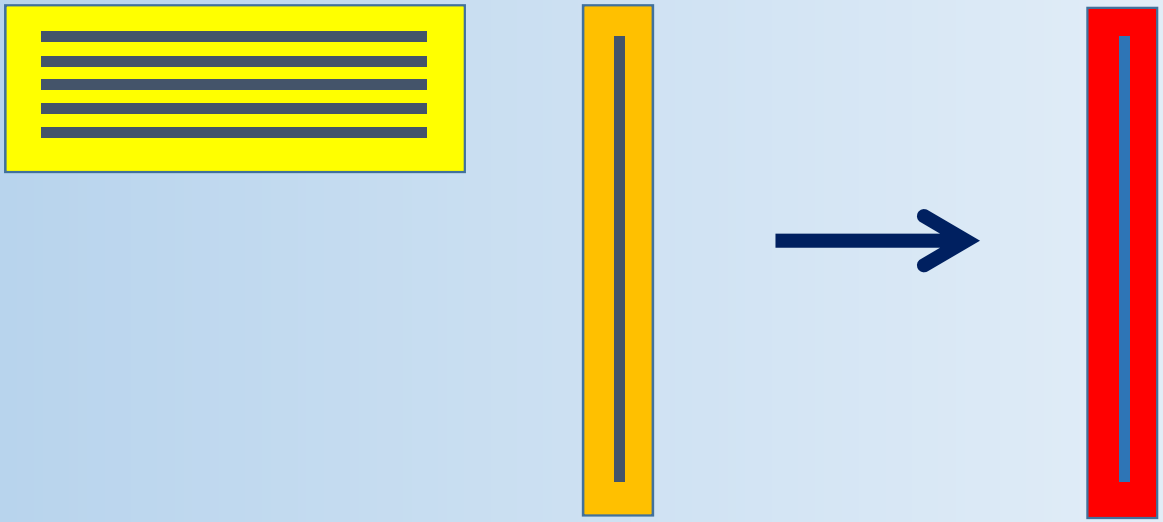
Заключение

В этой лекции мы рассмотрели:

- особенности и свойства коллективных обменов;
- различные операции коллективного обмена;
- синхронизацию при организации коллективных обменов;
- концепцию групп и коммуникаторов.

Задание для самостоятельной работы

Напишите параллельную программу вычисления произведения матрицы на вектор.
Используйте операции коллективного обмена.



Задание для самостоятельной работы

Напишите параллельную программу вычисления произведения матрицы на матрицу.
Используйте ленточную декомпозицию и операции коллективного обмена.