Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

Гетерогенные системы (графические процессоры)

- CUDA

Лекция 14. MPI. Коллективные взаимодействия процессов

А. А. Букатов, В. Н. Дацюк, А. И. Жегуло. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. Издательство ООО «ЦВВР», 2003, 208 с.

Антонов А.С.
Параллельное программирование с использованием технологии MPI: Учебное пособие. – М.: Изд-во МГУ, 2004. – 71 с.

Лекции и семинары

- http://www.slideshare.net/Aleximos/mpi-9793227

Технологии

- http://parallel.ru/tech/tech_dev/mpi.html
- http://parallel.ru/tech/tech_dev/MPI/examples/ (примеры)

Примеры из учебника "Технологии параллельного программирования MPI и OpenMP"

- http://parallel.ru/tech/tech_dev/MPI%26OpenMP/examples/

Коллективные обмены (блокирующие)

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

В операциях коллективного взаимодействия процессов участвуют все процессы коммуникатора.
Соответствующая процедура должна быть вызвана каждым процессом, м.б. со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено.
Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

Коллективные операции

Синхронизация всех процессов с помощью барьеров (MPI_Barrier).

Коллективные коммуникационные операции:

- рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);
- сборка (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv);
- сборка (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv);
- разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
- совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами, в свой буфер приема (MPI_Alltoall, MPI_Alltoallv).

Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
- с рассылкой результата всем процессам (MPI_Allreduce);
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter);
- префиксная редукция (MPI_Scan).

Синхронизация процессов

int MPI_Barrier(MPI_Comm comm) *comm* – идентификатор группы.

Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы *comm* также не выполнят эту процедуру.
Все процессы должны вызвать барьер, реально вызовы могут располагаться в разных частях программы.

```
int main(int argc, char *argv[])
{
    /* Code */

    MPI_Barrier(MPI_COMM_WORLD);

    /* Code */

    MPI_Finalize();
    return 0;
}
```

Функциональность MPI_Barrier смоделирована при помощи отложенных запросов на взаимодействие.
Производится Ntime операций обмена – процессы посылают сообщение 0-му и получают ответный сигнал.
Время на моделирование сравнивается с временем на синхронизацию.

```
rank = 0 model time = 0.016768
rank = 2 model time = 0.016326
rank = 3 model time = 0.017564
rank = 1 model time = 0.016780
rank = 0 barrier time = 0.021987
rank = 2 barrier time = 0.021974
rank = 1 barrier time = 0.021990
rank = 3 barrier time = 0.021207
```

http://parallel.ru/sites/default/files/tech/tech_dev/MPI%26OpenMP/example16a.c

Коммуникационные процедуры

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант – все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- "векторный" вариант предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "**v**" в конце имени функции.

int MPI_Bcast(void *buf, int count,
MPI_Datatype datatype,
int source,
MPI_Comm comm)
(см. лекцию 12)

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

Сбор блоков данных от всех процессов

Каждая из указанных подпрограмм расширяет функциональные возможности предыдущих.

int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf,
int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)

sbuf – адрес начала буфера посылки
scount – число элементов в посылаемом сообщении
stype – тип элементов отсылаемого сообщения
OUT *rbuf* – адрес начала буфера сборки данных
rcount – число элементов в принимаемом сообщении
rtype – тип элементов принимаемого сообщения
dest – номер процесса, на котором происходит сборка данных

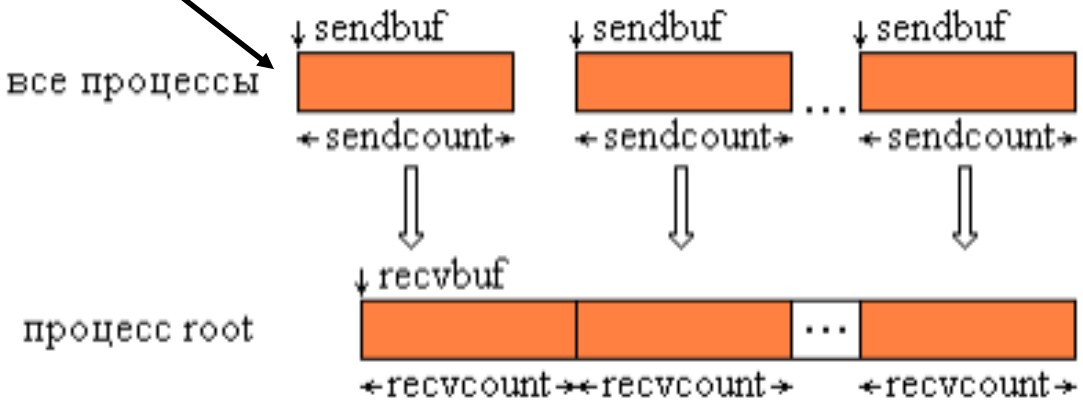
```
int MPI_Gather(void *sendbuf, int sendcnt,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Сборка данных со всех процессов в буфере *rbuf* процесса *dest*. Каждый процесс, включая *dest*, посылает содержимое своего буфера *sbuf* процессу *dest*. Собирающий процесс сохраняет данные в буфере *rbuf*, располагая их в порядке возрастания номеров процессов. Параметр *rbuf* имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров *count*, *datatype* и *dest* должны быть одинаковыми у всех процессов.

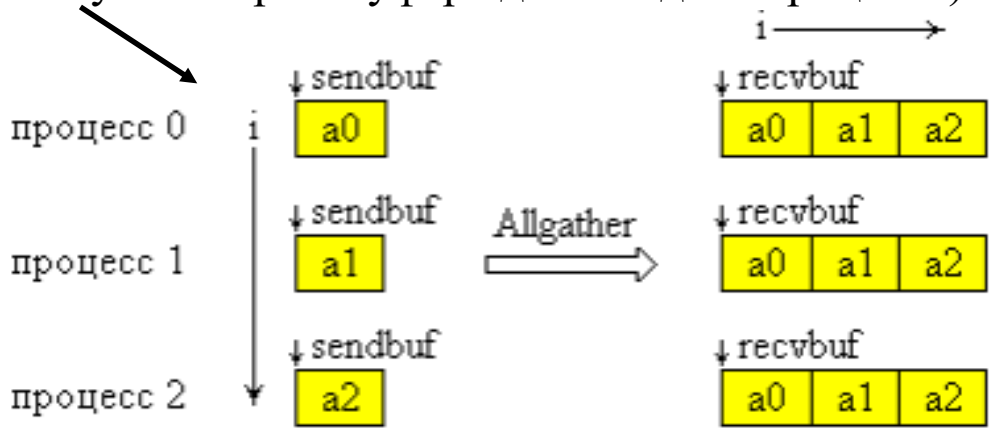
Графическая интерпретация

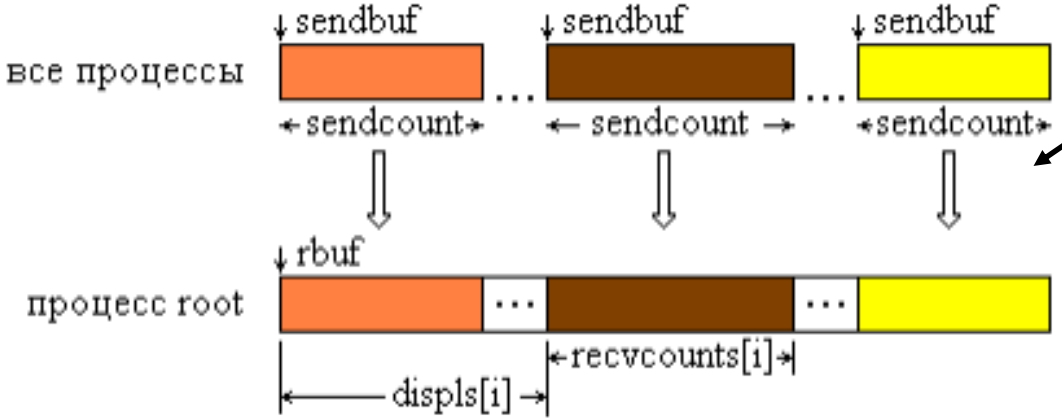
Семейство функций сбора блоков данных от всех процессов группы:
MPI_Gather, MPI_Allgather, MPI_Gatherv, MPI_Allgatherv.

MPI_Gather



MPI_Allgather, (сборка данных со всех процессов коммуникатора в буфере для каждого процесса)





MPI_Gatherv,

(сборка различного количества данных, передается смещение)

MPI_Allgatherv является аналогом функции MPI_Gatherv,
но сборка выполняется всеми процессами группы.
Поэтому в списке параметров отсутствует параметр root.

Пример сумма строк

```
#define N 4
int main(int argc, char **argv)
{
  int rank, size, i, j;          int a[N][N]; int s[N]; MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int sum = 0;        int b[N];
        for(j = 0; j < N; j++){ b[j] = rank + j*2;        sum += b[j];}
        MPI_Gather(b, N, MPI_INT, a, N, MPI_INT, 0, MPI_COMM_WORLD);
        if (rank == 0)
                for(i = 0; i < N; i++) { cout << i << " ! ";
                        for(j = 0; j<N; j++) cout << a[i][j] << " ";     cout << endl;
                }
        MPI_Gather(&sum, 1, MPI_INT, s, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (rank == 0) { cout << " S ";
                for(i = 0; i<N; i++) cout << s[i] << " ";        cout << endl; }
  MPI_Finalize();
}
```

```
0 !     0 2 4 6
1 !     1 3 5 7
2 !     2 4 6 8
3 !     3 5 7 9
 S 12 16 20 24
```

```
mpirun -n 4 ./mas
```

Функции распределения блоков данных по всем процессам группы

MPI_Scatter разбивает сообщение из буфера посылки процесса root на равные части размером sendcnt и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

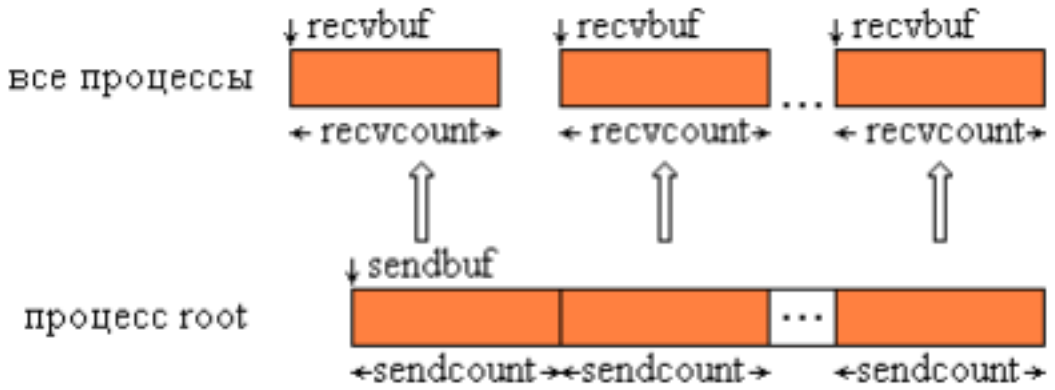
```
int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcnt,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

MPI_Scatter обратна MPI_Gather.

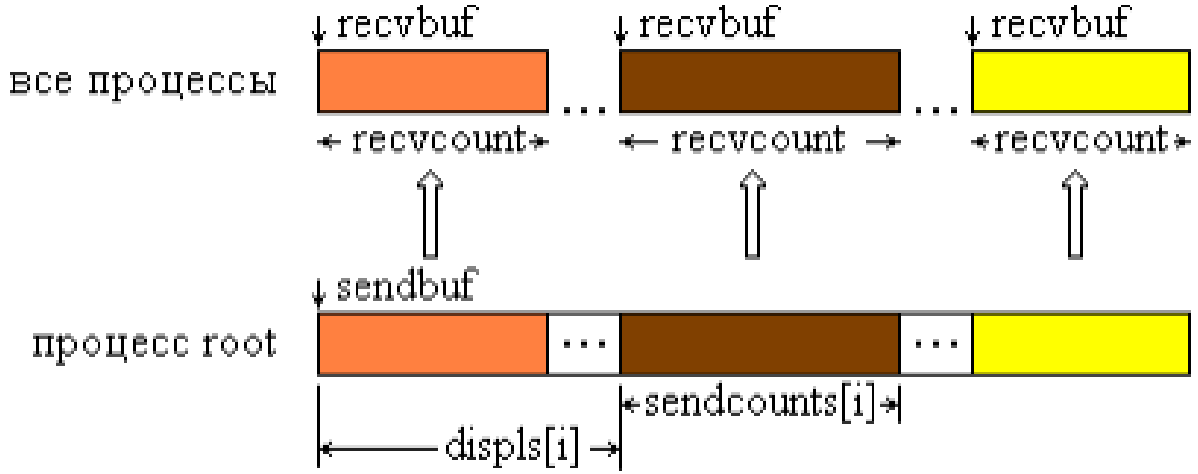
Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, несущественны.

Еще Scatter и Scatterv

Графическая интерпретация операции Scatter.



Графическая интерпретация операции Scatterv.



Функция MPI_Scatterv является векторным вариантом функции MPI_Scatter, позволяющим посылать каждому процессу различное количество элементов.

Пример Scatter

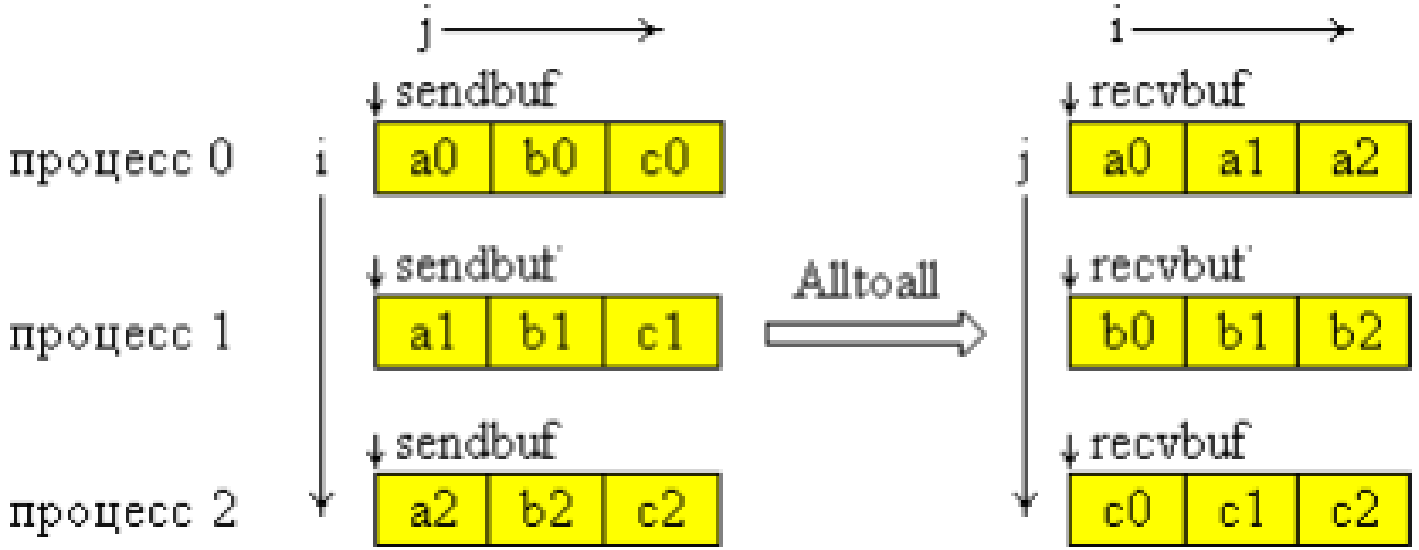
```
int main(int argc, char **argv) {
        MPI_Init(&argc, &argv);
        int rank, size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int a[size]; int b;
        if (rank == 0) {
                srand(time(NULL));
                cout << "A: ";
                for (int i = 0; i < size; i++) {
                        a[i] = rand(); cout << a[i] << " ";
                }
                cout << endl;
        }
        MPI_Scatter(a, 1, MPI_INT, &b, 1, MPI_INT, 0, MPI_COMM_WORLD);
        cout << " rank: " << rank <<  " b = " << b << endl;
        MPI_Finalize();
        return 0;
}
```

```
mpirun -n 5 ./08
```

```
A: 2041956780 541818132 562629060 91751649 555020432
 rank: 0 b = 2041956780
 rank: 1 b = 541818132
 rank: 2 b = 562629060
 rank: 3 b = 91751649
 rank: 4 b = 555020432
```

Совмещенные коллективные операции

MPI_Alltoall совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям. Процесс i посылает j-й блок своего буфера sendbuf процессу j, который помещает его в i-й блок своего буфера recvbuf.
Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов.



Функция MPI_Alltoallv реализует векторный вариант Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

Еще Alltoall

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

Глобальные вычислительные операции

Математические операции над блоками данных, распределенных по процессорам, называют глобальными операциями редукции.

Операцией редукции называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора.

Если компоненты вектора расположены в адресных пространствах процессов, выполняющихся на различных процессорах, то говорят о глобальной (параллельной) редукции.

Использование операций редукции является одним из основных средств организации распределенных вычислений.

MPI варианты глобальных операций редукции:

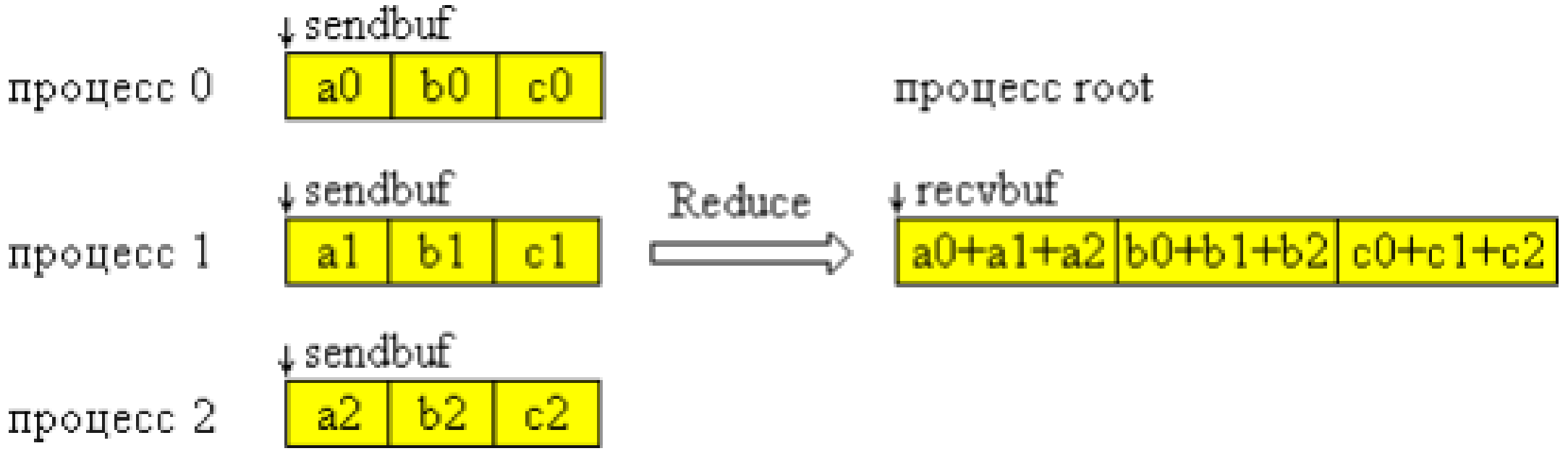
- с сохранением результата в адресном пространстве одного процесса (**MPI_Reduce**);
- с сохранением результата в адресном пространстве всех процессов (**MPI_Allreduce**);
- префиксная операция редукции, которая в качестве результата операции возвращает вектор, i-я компонента которого является результатом редукции первых i компонент распределенного вектора (**MPI_Scan**);
- совмещенная операция Reduce/Scatter (**MPI_Reduce_scatter**).

Еще Reduce

int MPI_Reduce(void *sbuf, void *rbuf, int count,

MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

(см. лекцию 12)



Allreduce

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

sbuf – адрес начала буфера для аргументов

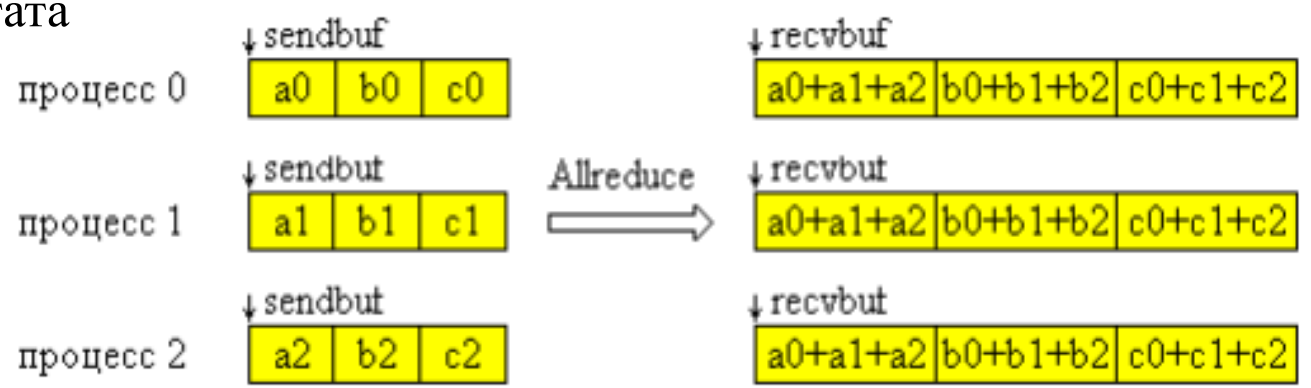
OUT *rbuf* – адрес начала буфера для результата

count – число аргументов каждого процесса

datatype – тип аргументов

op – идентификатор глобальной операции

comm – идентификатор группы

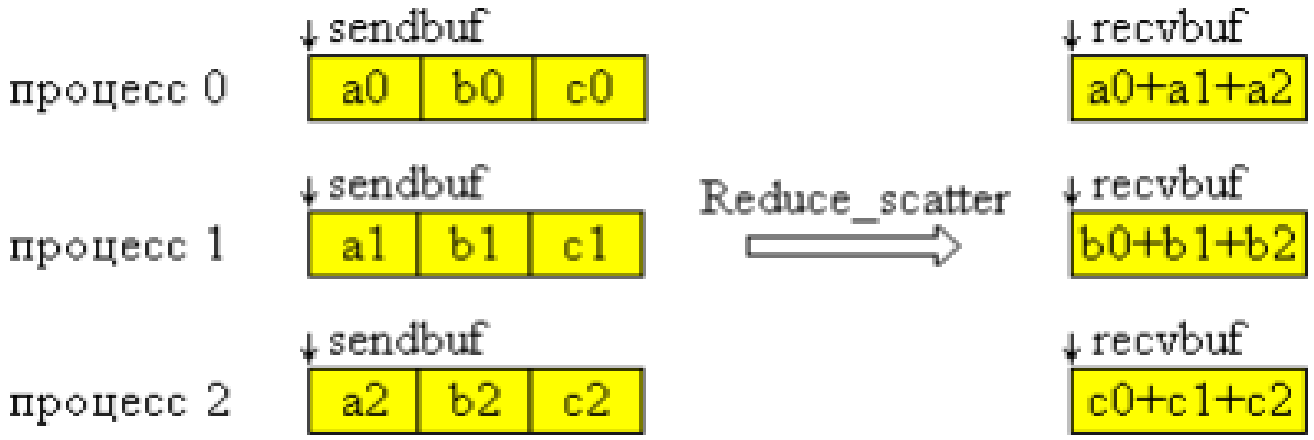


Выполнение *count* глобальных операций *op* с возвратом *count* результатов во всех процессах в буфере *rbuf*. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров *count* и *datatype* у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция *op* обладает свойствами ассоциативности и коммутативности.

Еще глобальные вычислительные операции

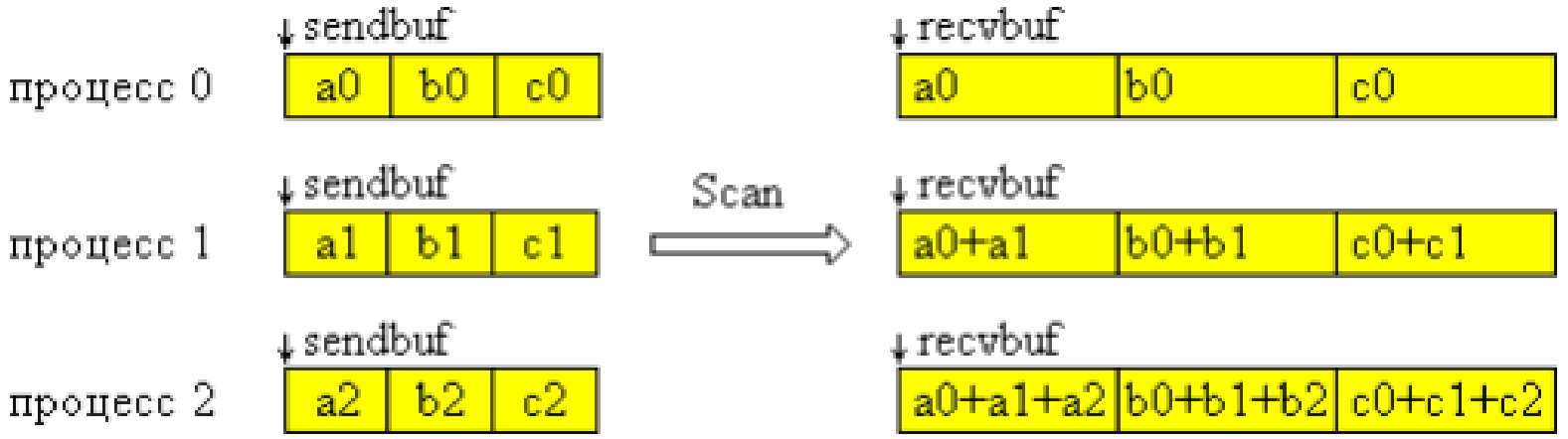
MPI_Reduce_scatter совмещает в себе операции редукции и распределения результата по процессам.

Графическая интерпретация операции Reduce_scatter



MPI_Scan выполняет префиксную редукцию. Параметры такие же, как в MPI_Allreduce, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересылает в буфер приема i-го процесса редукцию значений из входных буферов процессов с номерами 0, ... , i включительно.

Графическая интерпретация операции Scan



Производные типы данных и пересылка упакованных данных

MPI предоставляет два механизма эффективной пересылки данных разных типов (например, структуры) или данные, расположенные в несмежных областях памяти (части массивов, не образующих непрерывную последовательность элементов):

- путем создания производных типов для использования в коммуникационных операциях вместо предопределенных типов MPI;
- пересылку упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения).

Производные типы MPI не являются в полном смысле типами данных, как в языках программирования. Они используются только в коммуникационных операциях.

Производные типы MPI – описатели расположения в памяти элементов базовых типов.

Производный тип представляет собой скрытый (*opaque*) объект, который специфицирует две вещи: последовательность базовых типов и последовательность смещений.

Производные типы данных

Стандартный сценарий определения и использования производных типов включает 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**.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

- Протяженность типа определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как адрес последней ячейки данных – адрес первой ячейки данных + длина последней ячейки данных (опрашивается подпрограммой **MPI_Type_extent**).
- Размер типа определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа (опрашивается подпрограммой **MPI_Type_size**). Для простых типов протяженность и размер совпадают.

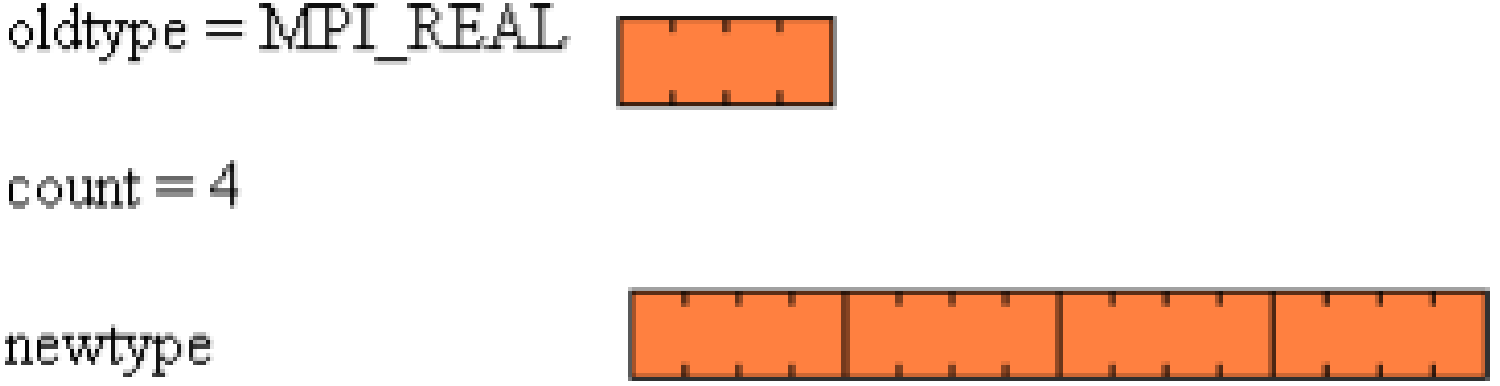
Один из типов MPI

Самый простой конструктор типа **MPI_Type_contiguous** создает новый тип, элементы которого состоят из указанного числа элементов базового типа, занимающих смежные области памяти.

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

count – число элементов базового типа;
oldtype – базовый тип данных;
OUT newtype – новый производный тип данных.

Графическая интерпретация работы конструктора MPI_Type_contiguous



Работа с группами процессов

Сравнение двух коммуникаторов – **MPI_Comm_compare.**
Дублирование коммуникатора – **MPI_Comm_dup.**
Создание коммуникатора – **MPI_Comm_create**.

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

comm – идентификатор группы
color – признак разделения на группы
key – параметр, определяющий нумерацию в новых группах
OUT *newcomm* – идентификатор новой группы

Процедура разбивает все множество процессов группы *comm*, на непересекающиеся подгруппы – одну на каждое значение параметра *color* (неотрицательное число). Если в качестве *color* указано значение *MPI_UNDEFINED*, то в *newcomm* будет возвращено значение *MPI_COMM_NULL*.

int MPI_Comm_free(MPI_Comm comm)

OUT *comm* – идентификатор группы

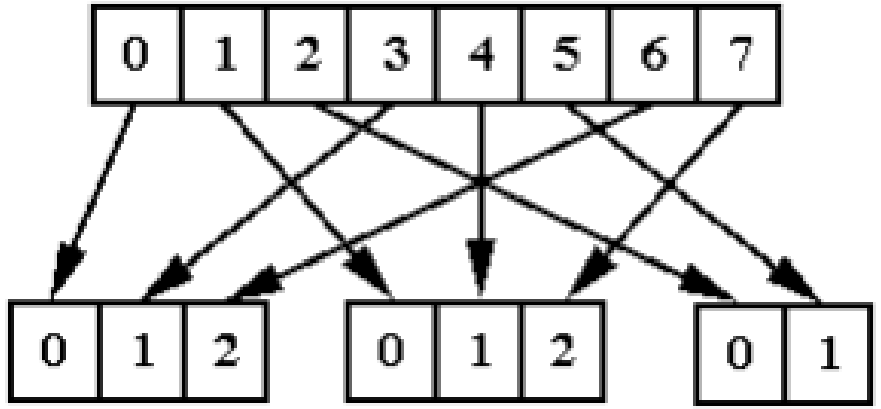
Уничтожает группу, ассоциированную с идентификатором *comm*, который после возвращения устанавливается в *MPI_COMM_NULL*.

Пример работа с группами

Разбиение группы из восьми процессов на три подгруппы

Первую подгруппу образовали процессы, номера которых делятся на 3 без остатка, вторую, для которых остаток равен 1, и третью, для которых остаток равен 2. После выполнения функции MPI_Comm_split значения коммуникатора newcomm в процессах разных подгрупп будут отличаться.

```
MPI_comm comm, newcomm;
int  myid, color;
……
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```



Создание двух непересекающихся групп процессов

```
call MPI_GROUP_INCL(group, size1, ranks, group1, ierr)
call MPI_GROUP_EXCL(group, size1, ranks, group2, ierr)
```

Пример разбиение коммуникатора

```
int main(int argc, char **argv)
{
        int rank, size, i, rbuf, ranks[128], new_rank;
        MPI_Group group, new_group; MPI_Comm new_comm;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_group(MPI_COMM_WORLD, &group);
        for(i = 0; i < size/2; i++) ranks[i] = i;
        if(rank < size/2)
                MPI_Group_incl(group, size/2, ranks, &new_group);
        else
                MPI_Group_excl(group, size/2, ranks, &new_group);
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
        MPI_Allreduce(&rank, &rbuf, 1, MPI_INT, MPI_SUM, new_comm);
        MPI_Group_rank(new_group, &new_rank);
        cout << "rank = " << rank << " newrank = " << new_rank << " rbuf = "  << rbuf << endl;
        MPI_Finalize();
}
```

Создаются две новые группы.
При нечетном числе процессов во вторую войдет на один процесс больше.
Для каждой новой группы создается коммуникатор new_comm.
Allreduce выполняется по отдельности для процессов, входящих в разные группы.

```
rank=2 newrank=0 rbuf=5
rank=0 newrank=0 rbuf=1
rank=3 newrank=1 rbuf=5
rank=1 newrank=1 rbuf=1
```