Архитектура компьютеров

ст. преп. кафедры МСС А.С. Гусейнова

Функция MPI_Wtime, вычисление производительности вычислений

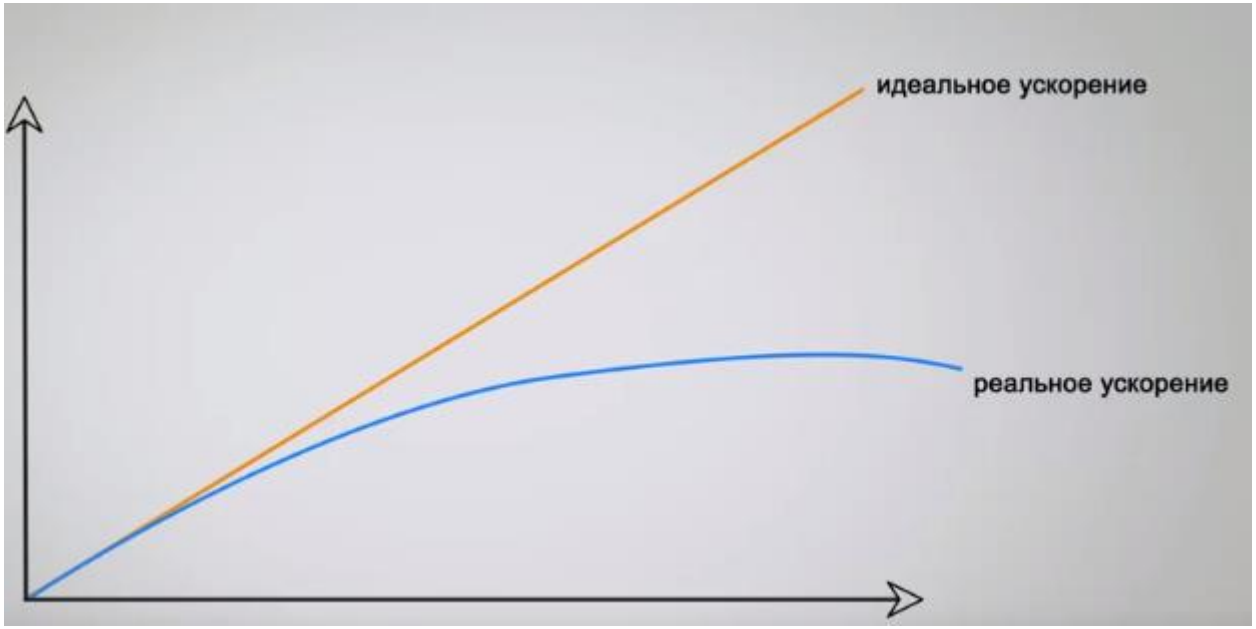
```
int main(int argc, char **argv)
{
        double time_work = MPI_Wtime();

        //вычисления

        time_work = MPI_Wtime() - time_work;
        cout << "Time     = " << time_work << endl;

    MPI_Finalize();
    return 0;
}
```

Ускорение программы

$$T_p \equiv \alpha T_1 + \frac{(1-\alpha)T_1}{p}$$

$$S = \frac{T_1}{T_p}.$$

Ускорение



Закон Амдала

$$S = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} \leq \frac{1}{\alpha}$$

Передача данных содержит следующие три фазы:

1. Данные выталкиваются из буфера процесса отправителя, и части сообщения объединяются
2. Сообщение передается от отправителя к получателю
3. Данные выделяются из буфера сообщения и отправляются к получателю

Отличительные особенности коллективных операций:

- Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
- Коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Количество получаемых данных должно быть равно количеству посланных данных.
- Типы элементов посылаемых и получаемых сообщений должны совпадать.
- Сообщения не имеют идентификаторов.

Набор коллективных операций включает:

синхронизацию всех процессов с помощью барьеров (MPI_Barrier)

int MPI_Barrier(MPI_Comm comm)

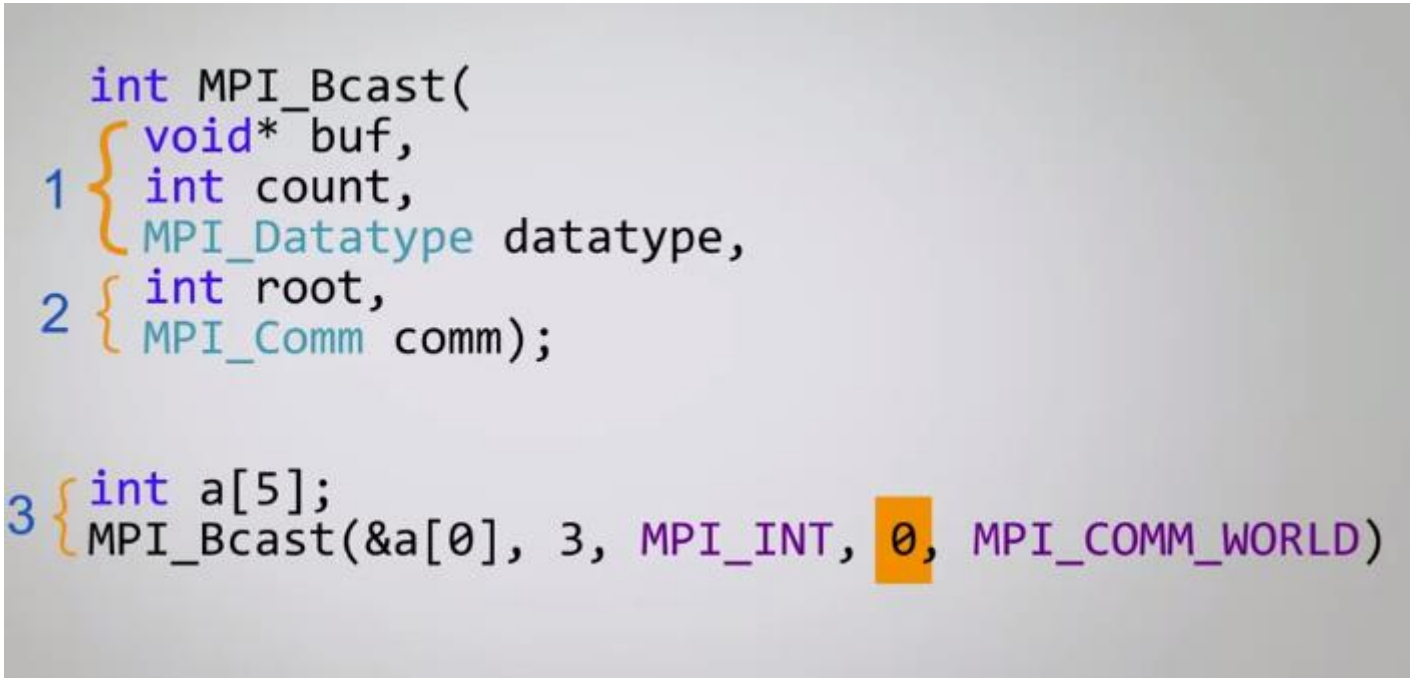
Коллективные коммуникационные операции:

- MPI_Bcast
- MPI_Gather, MPI_Gatherv
- MPI_Allgather, MPI_Allgatherv
- MPI_Scatter, MPI_Scatterv
- MPI_Alltoall, MPI_Alltoallv

Операции редукции:

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter
- MPI_Scan

Широковещательная рассылка



Напишем функцию my_bcast

```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
    MPI_Comm communicator) {
    if (world_rank == root) {
        // If we are the root process, send our data to everyone
        int i;
        for (i = 0; i < world_size; i++) {
            if (i != world_rank) {
                MPI_Send(data, count, datatype, i, 0, communicator);
            }
        }
    }
    else {
        // If we are a receiver process, receive the data from the root
        MPI_Recv(data, count, datatype, root, 0, communicator,
            MPI_STATUS_IGNORE);
    }
}
```

Результат работы программы

```
Process 0 broadcasting data 100
Process 2 received data 100 from root process
Process 3 received data 100 from root process
Process 1 received data 100 from root process
```

Сравним my_bcast и MPI_Bcast

```
for (i = 0; i < num_trials; i++) {
    // Time my_bcast
    // Synchronize before starting timing
    MPI_Barrier(MPI_COMM_WORLD);
    total_my_bcast_time -= MPI_Wtime();
    my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    // Synchronize again before obtaining final time
    MPI_Barrier(MPI_COMM_WORLD);
    total_my_bcast_time += MPI_Wtime();

    // Time MPI_Bcast
    MPI_Barrier(MPI_COMM_WORLD);
    total_mpi_bcast_time -= MPI_Wtime();
    MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    total_mpi_bcast_time += MPI_Wtime();
}
```

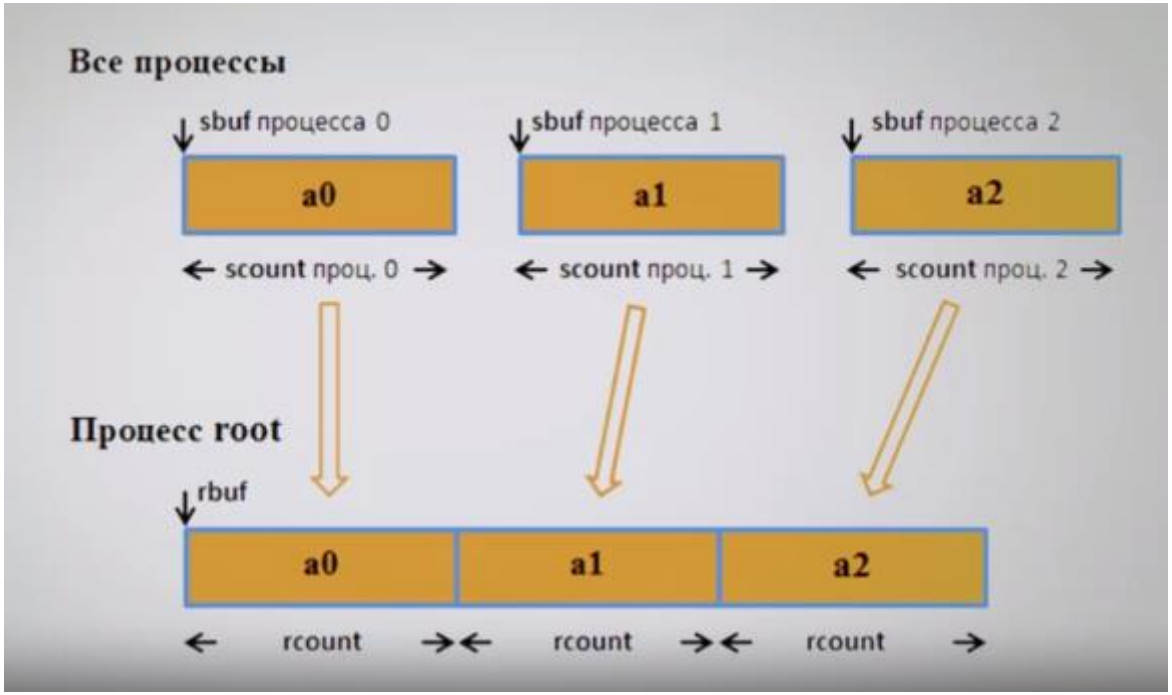
Результат

```
Data size = 400000, Trials = 10
Avg my_bcast time = 0.510873
Avg MPI_Bcast time = 0.126835
```

Сбор данных от всех процессов

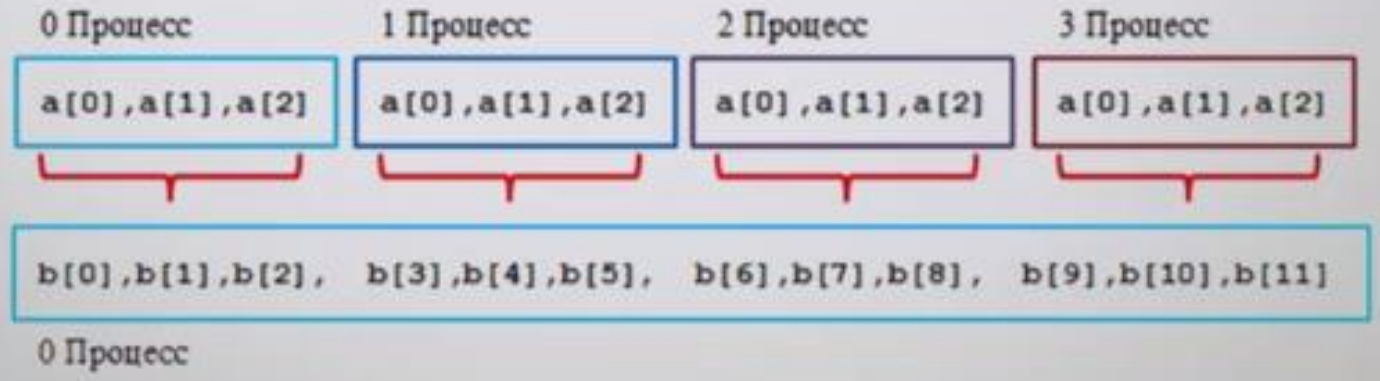
```
int MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

MPI_Gather

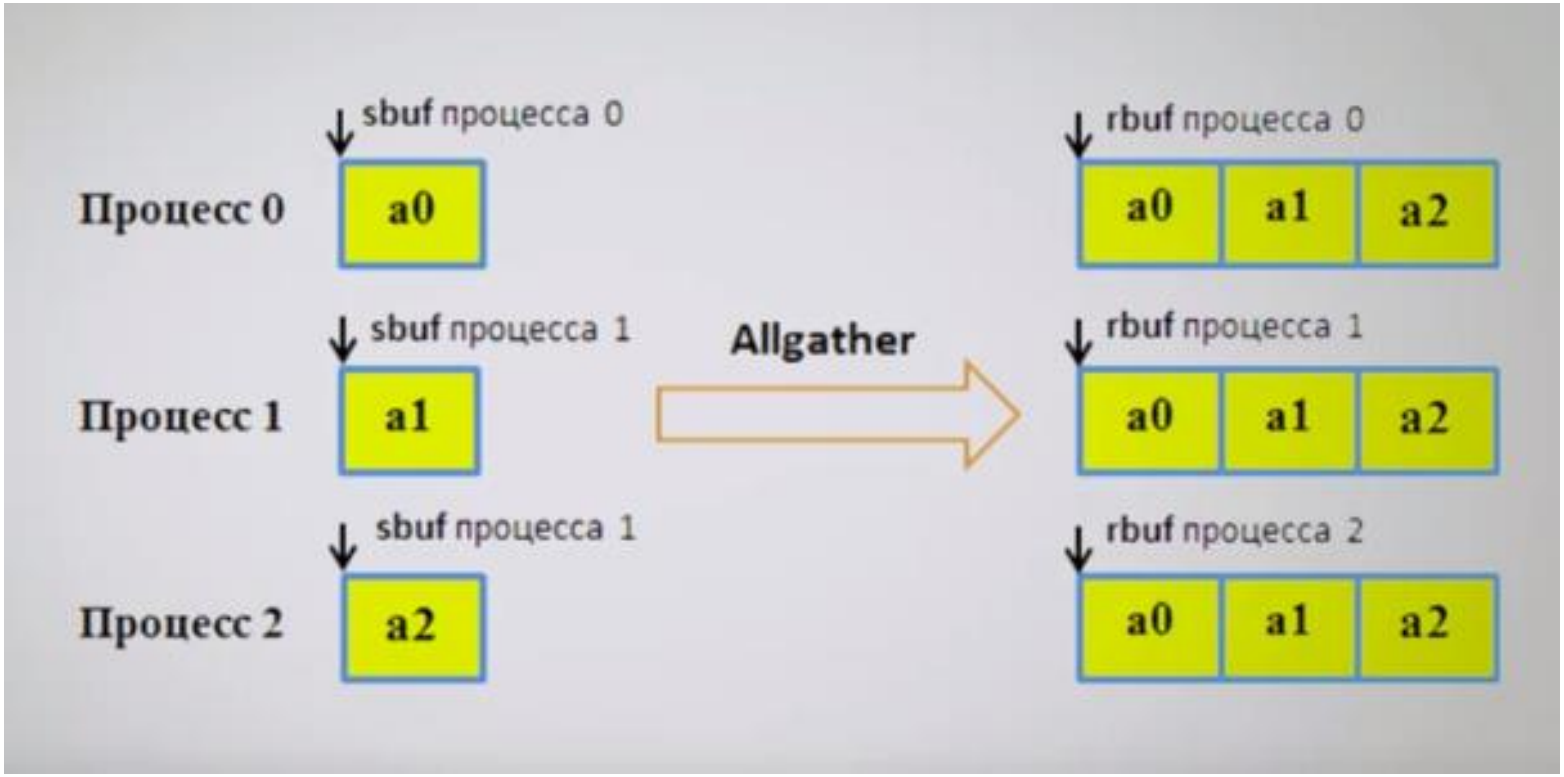

Все процессы
sbuf процесса 0
sbuf процесса 1
sbuf процесса 2
a0
a1
a2
scount проц. 0
scount проц. 1
scount проц. 2
Процесс root
rbuf
a0
a1
a2
rcount
rcount
rcount

Пример.

```
int a[3], b[20];
MPI_Gather (
    a[0], 3, MPI_INT,
    b[0], 3, MPI_INT,
    0, MPI_COMM_WORLD);
```



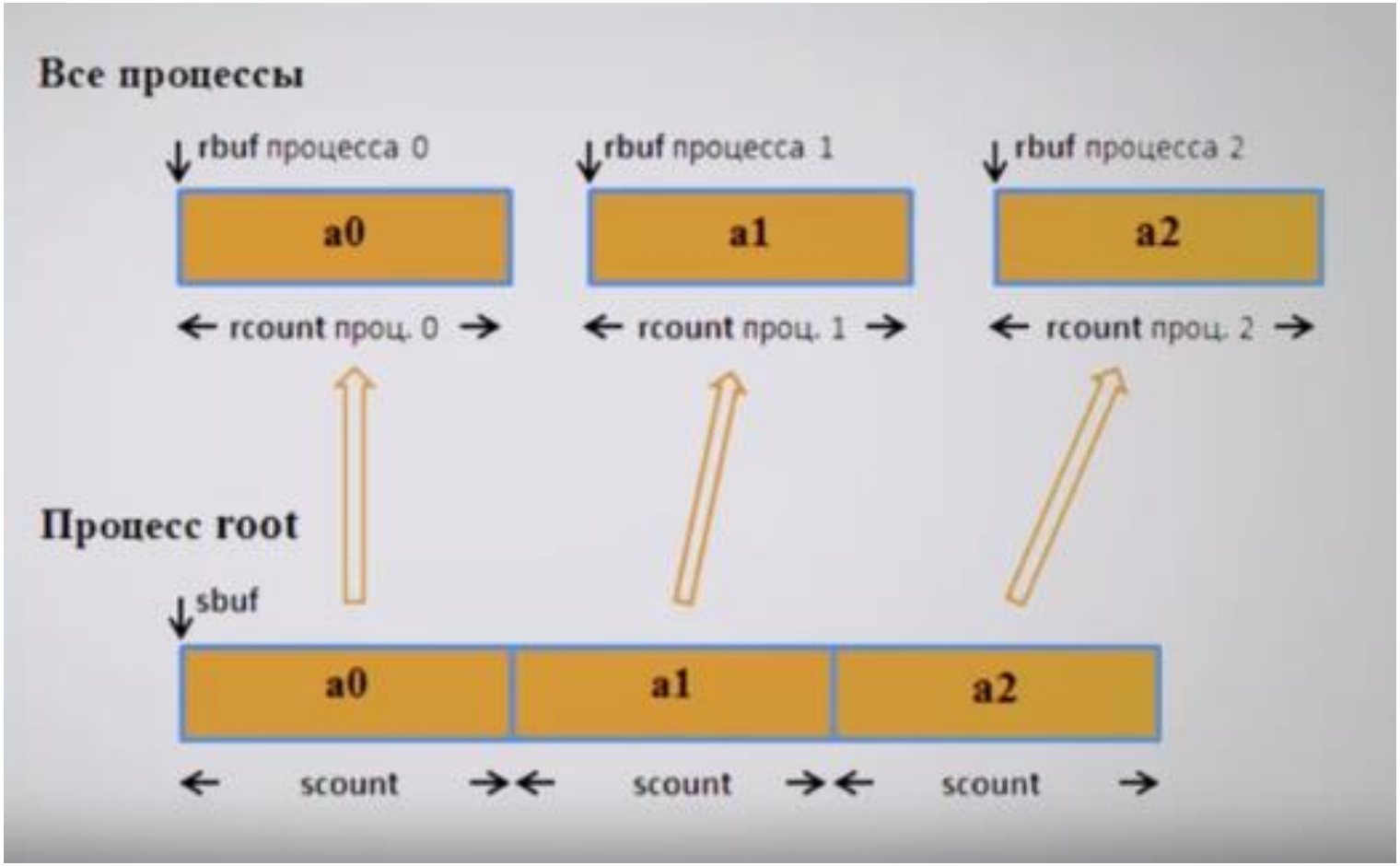
MPI_Allgather



Функции распределения данных

```
int MPI_Scatter(
    void* sbuf,
    int   scount,
    MPI_Datatype stype,
    void* rbuf,
    int   rcount,
    MPI_Datatype rtype,
    int root,
    MPI_Comm comm)
```

MPI_Scatter


Все процессы
rbuf процесса 0
rbuf процесса 1
rbuf процесса 2
a0
a1
a2
rcount проц. 0
rcount проц. 1
rcount проц. 2
Процесс root
sbuf
a0
a1
a2
scount
scount
scount

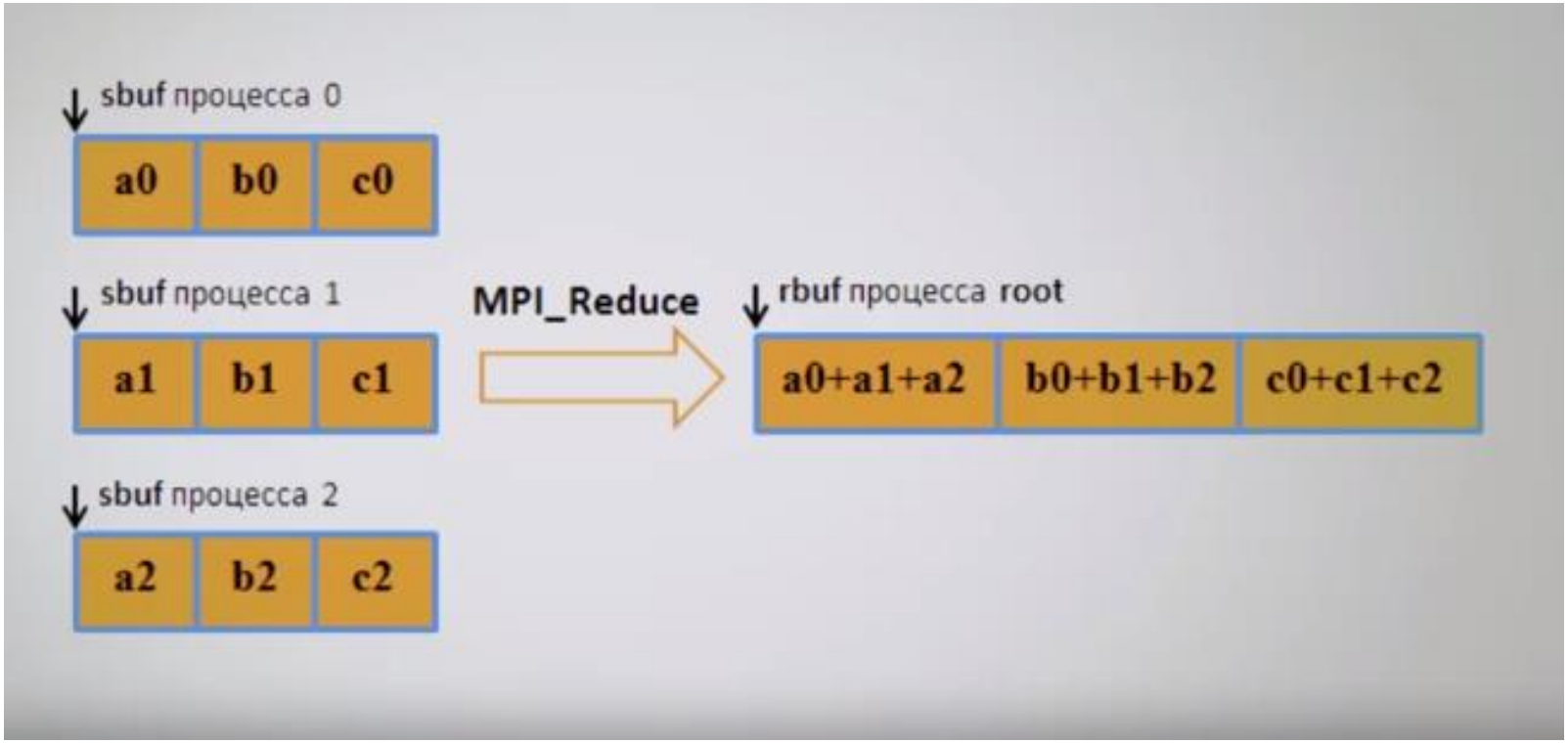
Пример распределения и сбора данных в задаче сложение векторов

```
#include "mpi.h"
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int const n = 10;
    int n1 = n / size;
    int a[n],  b[n],  c[n];
    int a1[n], b1[n], c1[n];

    // Заполнение массивов
    if (rank == 0)
        for (int i = 0; i < n; i++)
        {
            a[i] = rand() % 10;
            b[i] = rand() % 10;
        }
```

Сложение векторов

```
// Распределение данных с нулевого процесса
MPI_Scatter(&a[0], n1, MPI_INT, &a1[0], n1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter(&b[0], n1, MPI_INT, &b1[0], n1, MPI_INT, 0, MPI_COMM_WORLD);

// Вычисления
for (int i = 0; i <= n1; i++)
    c1[i] = a1[i] + b1[i];

// Сбор данных на нулевом процессе
MPI_Gather(&c1[0], n1, MPI_INT, &c[0], n1, MPI_INT, 0, MPI_COMM_WORLD);

//Выдача результатов
if (rank == 0) {
    cout << "        a[i]  b[i]  c[i]" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << " i= " << i << "     " << a[i] << "     " << b[i] << "     " << c[i] << endl;
    }
}

MPI_Finalize();
return 0;
```

Редукция


sbuf процесса 0
a0
b0
c0
sbuf процесса 1
a1
b1
c1
sbuf процесса 2
a2
b2
c2
MPI_Reduce
rbuf процесса root
a0+a1+a2
b0+b1+b2
c0+c1+c2

MPI_Reduce

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

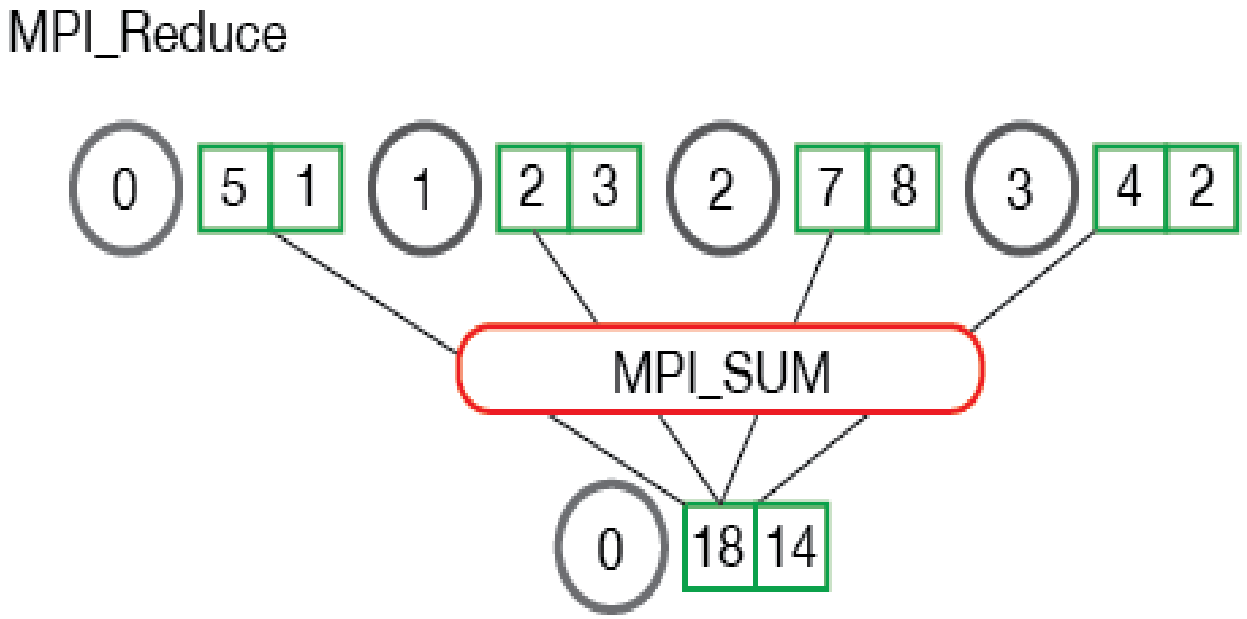
Параметры

- *sendbuf* [in]
  Дескриптор буфера, который содержит данные, отправляемые корневому процессу.
- *recvbuf* [out, optional]
  Дескриптор буфера для получения результата операции сокращения. Этот параметр имеет значение только в корневом процессе.
- *count*
  Количество элементов, отправляемых из этого процесса.
- *datatype*
  Тип данных каждого элемента в буфере. Этот параметр должен быть совместим с операцией, как указано в параметре *op* .
- *Op*
  Выполняемая операция глобального сокращения. Дескриптор может указывать на встроенную или определяемую приложением операцию.
- *root*
  Ранг принимающего процесса в указанном коммуникаторе.
- *Comm*
  Дескриптор **коммуникатора MPI_Comm** .

Список операций:

- typedef enum _MPI_Op {
- MPI_OP_NULL = 0x18000000,
- MPI_MAX = 0x58000001,
- MPI_MIN = 0x58000003,
- MPI_SUM = 0x58000003,
- MPI_PROD = 0x58000004,
- MPI_LAND = 0x58000005,
- MPI_BAND = 0x58000006,
- MPI_LOR = 0x58000007,
- MPI_BOR = 0x58000008,
- MPI_LXOR = 0x58000009,
- MPI_BXOR = 0x5800000a,
- MPI_MINLOC = 0x5800000b,
- MPI_MAXLOC = 0x5800000c,
- MPI_REPLACE = 0x5800000d
- } MPI_Op;

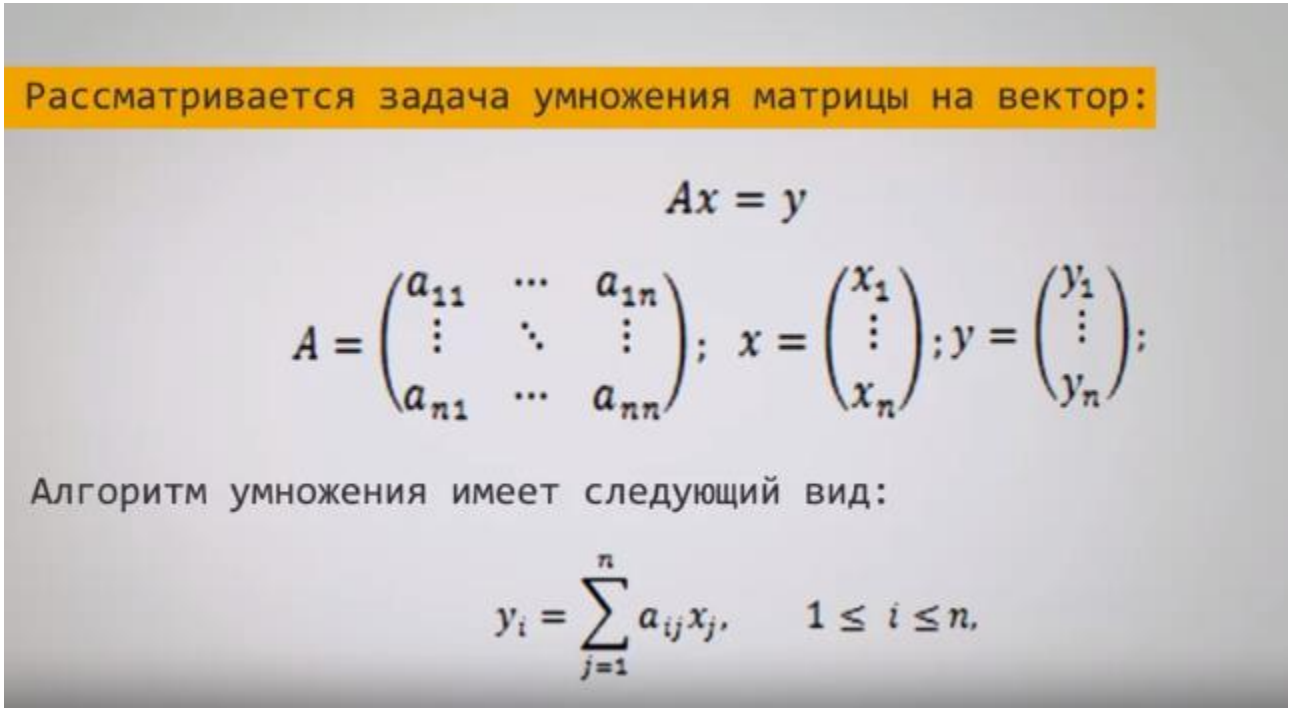
Редукция нескольких данных в каждом процессе



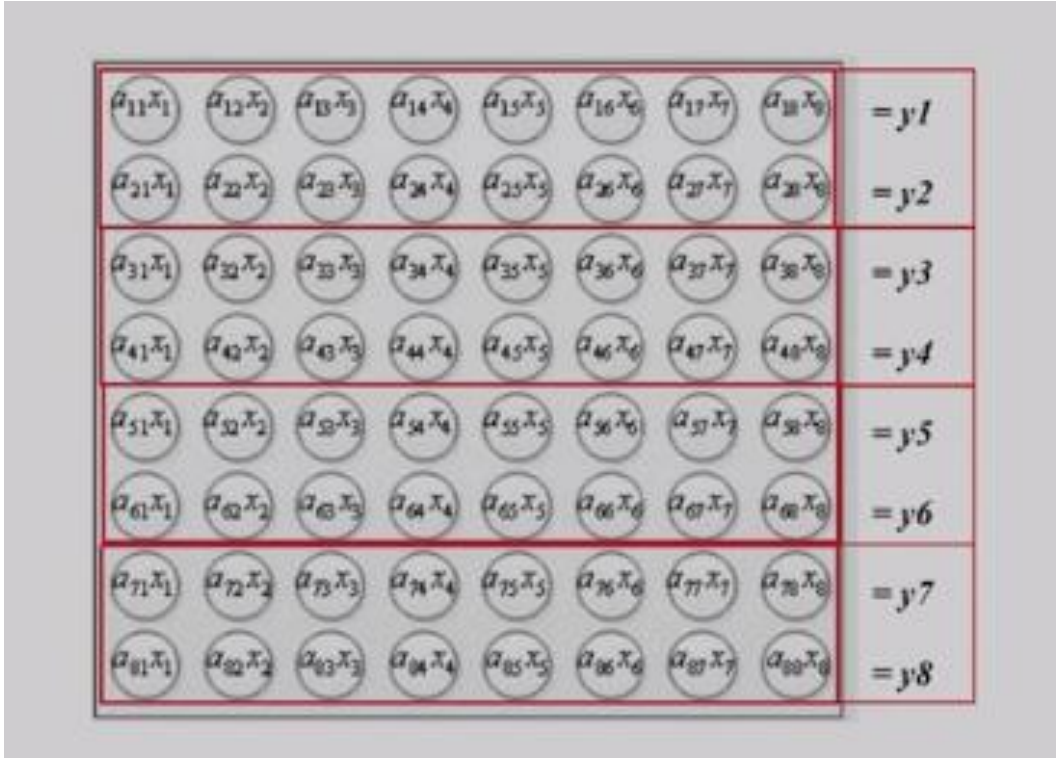
Пример:

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

                if (myid == 0)
                {
                        printf("Enter the number of intervals: (0 quits) ");
                        scanf_s("%d", &n);
                }
                MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
                        h = 1.0 / (double)n;
                        sum = 0.0;
                        for (i = myid + 1; i <= n; i += numprocs)
                        {
                                x = h * ((double)i - 0.5);
                                sum += (4.0 / (1.0 + x * x));
                        }
                        mypi = h * sum;
                        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                                MPI_COMM_WORLD);
                        if (myid == 0)
                        {
        printf("pi is approximately %.16f. Error is %.16f\n", pi, fabs(pi - PI25DT));
                                                        }
```

Умножение матрицы на вектор



Распараллеливание на 4 процесса можно представить:



```cpp
// Инициализации библиотеки
MPI_Init(&argc, &argv);

// Получаем размер группы и индекс текущего процесса
int current_process_rank;
int processes_number;
MPI_Comm_rank(MPI_COMM_WORLD, &current_process_rank);
MPI_Comm_size(MPI_COMM_WORLD, &processes_number);

int size;
std::vector<int> matrix;
std::vector<int> vector;
// Процесс с индексом 0 выполянет считывание данных
if (current_process_rank == 0) {
  size = ReadInput("../big_input.txt", &matrix, &vector);
}
```

Отправляем размер матрицы и вектор

```
// Пересылаем размер матрицы и вектор чисел
MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
vector.resize(size);
MPI_Bcast(vector.data(), size, MPI_INT, 0, MPI_COMM_WORLD);

// Вычисляем сколько строк матрицы достанется каждому процессу
int lines_per_process = (size + processes_number - 1) / processes_number;
```

Распределяем исходную матрицу по процессам

```
// Процесс с индексом 0 выполяет пересылку соответствующего куска данных в соответствующий процесс
std::vector<int> lines;
lines.resize(lines_per_process * size);
MPI_Scatter(
    matrix.data(),
    lines_per_process * size,
    MPI_INT,
    lines.data(),
    lines_per_process * size,
    MPI_INT,
    0,
    MPI_COMM_WORLD);
```

Выполняем умножение

```
std::vector<int64_t> result;
result.reserve(lines_per_process);
for (int i = 0; i < lines_per_process; ++i) {
  int64_t sum = 0;
  for (int j = 0; j < size; ++j) {
    sum += lines[i * size + j] * vector[j];
  }
  result.push_back(sum);
}
```

Сбор результирующего вектора

```
// Сбор результата главным процессом
std::vector<int64_t> joined_result;
joined_result.resize(size * 2);
MPI_Gather(
    result.data(),
    lines_per_process,
    MPI_INT64_T,
    joined_result.data() + lines_per_process * current_process_rank,
    lines_per_process,
    MPI_INT64_T,
    0,
    MPI_COMM_WORLD);
```

функция MPI_Scatterv

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

- ***sendbuf* [in]**
  Указатель на буфер, содержащий данные, отправляемые корневым процессом.

Параметр sendbuf игнорируется для всех процессов, не являющихся корневыми.

- **sendcounts [in]**
  Количество элементов, отправляемых в каждый процесс.
  Если *значение sendcount[i]* равно нулю, часть данных сообщения для этого процесса пуста.

Параметр sendcount игнорируется для всех процессов, не являющихся корневыми.

- ***displs* [in]**
  Расположения данных для отправки в каждый процесс communicator. Каждое расположение в массиве относительно соответствующего элемента массива *sendbuf* .

Этот параметр важен только в корневом процессе.

- ***sendtype***
  Тип данных MPI каждого элемента в буфере.

Параметр sendcount игнорируется для всех процессов, не являющихся корневыми.

- ***recvbuf* [out]**
  Указатель на буфер, содержащий данные, полученные при каждом процессе. Число и тип данных элементов в буфере указываются в параметрах *recvcount* и *recvtype* .
- ***recvcount***
  Количество элементов в буфере получения. Если число равно нулю, часть данных сообщения пуста.
- ***recvtype***
  Тип данных элементов в буфере получения.
- ***root***
  Ранг в процессе отправки в указанном коммуникаторе.
- ***Comm***
  Дескриптор **MPI_Comm** .

```
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
        int Size, int RowNum) {
        int* pSendNum;
        int* pSendInd;
        int RestRows = Size;
        MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        pSendInd = new int[process_amount];
        pSendNum = new int[process_amount];
        RowNum = (Size / process_amount);
        pSendNum[0] = RowNum * Size;
        pSendInd[0] = 0;
        for (int i = 1; i < process_amount; i++) {
                RestRows -= RowNum;
                RowNum = RestRows / (process_amount - i);
                pSendNum[i] = RowNum * Size;
                pSendInd[i] = pSendInd[i - 1] + pSendNum[i - 1];
        }
        MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
                pSendNum[process_rank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
        delete[] pSendNum;
        delete[] pSendInd;
}
```