Ковальский Евгений

Лабораторная работа №1

Пример 2

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "headers/mpi.h"

int **main**(int argc, char\* argv[])

{

double x[100], TotalSum, ProcSum = 0.0;

int ProcRank, ProcNum, N=100;

MPI\_Status Status;

*//* *инициализация*

MPI\_Init(&argc,*&argv*);

MPI\_Comm\_size(MPI\_COMM\_WORLD,*&ProcNum*);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,*&ProcRank*);

*//* *подготовка* *данных*

*if* ( ProcRank == 0 )

*for*(int i = 0; i < 100; i++){

x[i] = (i+42)\*(i+451);

}

*//* *рассылка* *данных* *на* *все* *процессы*

MPI\_Bcast(*x*, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

*//* *вычисление* *частичной* *суммы* *на* *каждом* *из* *процессов*

*//* *на* *каждом* *процессе* *суммируются* *элементы* *вектора* *x* *от* *i1* *до* *i2*

int k = N / ProcNum;

int i1 = k \* ProcRank;

int i2 = k \* (ProcRank + 1);

*if* ( ProcRank == ProcNum-1 )

i2 = N;

*for* ( int i = i1; i < i2; i++ )

ProcSum = ProcSum + x[i];

*//* *сборка* *частичных* *сумм* *на* *процессе* *с* *рангом* *0*

*if* ( ProcRank == 0 )

{

TotalSum = ProcSum;

*for* ( int i=1; i < ProcNum; i++ )

{

MPI\_Recv(*&ProcSum*, 1, MPI\_DOUBLE, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD,

*&Status*);

TotalSum = TotalSum + ProcSum;

}

} *else* *//* *все* *процессы* *отсылают* *свои* *частичные* *суммы*

MPI\_Send(&ProcSum, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

*//* *вывод* *результата*

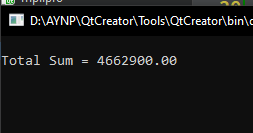
*if* ( ProcRank == 0 )

printf("\nTotal Sum = %10.2f",TotalSum);

MPI\_Finalize();

}

Результат выполнения примера:



Задание - Вариант 0.

Реализуйте функцию ring, которая создаёт N процессов и посылает сообщение первому процессу, который посылает сообщение второму, второй - третьему, и так далее. Наконец, процесс N посылает сообщение обратно процессу 1. После того, как сообщение обежало вокруг кольца M раз, все процессы заканчивают работу.

**Задание.** Модифицировать программу, написанную на Л.Р. №1, так чтобы она работала на основе коллективной передачи сообщений. **Результаты работы сравнить и занести в отчет.**

Код программы:

*//--------------------Подключаемые библиотеки--------------------//*

#include <headers/mpi.h>

#include <iostream>

#include <chrono>

*using* *namespace* std;

int **main**(int argc, char\* argv[])

{

*//--------------------Иициализация--------------------//*

int procNum, procRank, recv;

int m=5;*//кол-во* *итераций*

MPI\_Init(&argc, *&argv*);

MPI\_Comm\_size(MPI\_COMM\_WORLD, *&procNum*);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, *&procRank*);

*auto* start = std::chrono::steady\_clock::now();

*//--------------------Запуск* *цикла--------------------//*

*if* (procRank == 0) {

recv = 0;

cout << "Num of processors: " << procNum << endl;

cout << "Main process with rank: " << procRank << endl;

cout << "Start broadcasting" << endl;

MPI\_Bcast(*&recv*, 1, MPI\_INT, procRank, MPI\_COMM\_WORLD);

}

*//--------------------Главный* *цикл--------------------//*

*while*(*true*)

{

*//Вычисляем* *значение* *root,* *из* *которого* *будем* *получать* *сообщение*

int fromProc = (procNum+procRank-1) % procNum;

cout << "Processor " << procRank << " recv the message from root " << fromProc << endl;

*//Получаем* *сообщение*

MPI\_Bcast(*&recv*, 1, MPI\_INT, fromProc, MPI\_COMM\_WORLD);

cout << "Processor " << procRank << " recv the message " << recv << endl;

*//Увеличиваем* *значение* *счётчика* *на* *1*

int nextCount = recv + 1;

cout << "Processor " << procRank << " send the message " << nextCount << endl;

*//Потом* *отправляем* *сообщение*

MPI\_Bcast(*&nextCount*, 1, MPI\_INT, procRank, MPI\_COMM\_WORLD);

*//Проверяем* *завершение* *цикла,* *m* *кол-во* *итераций,* *а* *(recv-1)/procNum* *текующая* *итерация*

*if*((recv-1)/procNum == m-1){

*break*;

}

}

*//--------------------Завершение* *работы* *MPI--------------------//*

MPI\_Finalize();

*auto* end = std::chrono::steady\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end-start;

std::cout << "elapsed time: " << elapsed\_seconds.count() << "s\n";

*return* 0;

}

Вывод программы (4 процесса):

Processor 3 recv the message from root 2

Processor 1 recv the message from root 0

Processor 2 recv the message from root 1

Num of processors: 4

Main process with rank: 0

Start broadcasting

Processor 0 recv the message from root 3

Processor 1 recv the message 0

Processor 1 send the message 1

Processor 1 recv the message from root 0

Processor 2 recv the message 1

Processor 2 send the message 2

Processor 2 recv the message from root 1

Processor 3 recv the message 2

Processor 3 send the message 3

Processor 3 recv the message from root 2

Processor 0 recv the message 3

Processor 0 send the message 4

Processor 0 recv the message from root 3

Processor 1 recv the message 4

Processor 1 send the message 5

Processor 1 recv the message from root 0

Processor 2 recv the message 5

Processor 2 send the message 6

Processor 2 recv the message from root 1

Processor 3 recv the message 6

Processor 3 send the message 7

Processor 3 recv the message from root 2

Processor 0 recv the message 7

Processor 0 send the message 8

Processor 0 recv the message from root 3

Processor 1 recv the message 8

Processor 1 send the message 9

Processor 1 recv the message from root 0

Processor 2 recv the message 9

Processor 2 send the message 10

Processor 3 recv the message 10

Processor 2 recv the message from root 1

Processor 3 send the message 11

Processor 0 recv the message 11

Processor 3 recv the message from root 2

Processor 0 send the message 12

Processor 1 recv the message 12

Processor 0 recv the message from root 3

Processor 1 send the message 13

Processor 2 recv the message 13

Processor 1 recv the message from root 0

Processor 2 send the message 14

Processor 3 recv the message 14

Processor 2 recv the message from root 1

Processor 3 send the message 15

Processor 0 recv the message 15

Processor 3 recv the message from root 2

Processor 0 send the message 16

Processor 1 recv the message 16

Processor 0 recv the message from root 3

Processor 1 send the message 17

Processor 2 recv the message 17

Processor 1 recv the message from root 0

Processor 2 send the message 18

Processor 3 recv the message 18

Processor 3 send the message 19

Processor 0 recv the message 19

Processor 0 send the message 20

Processor 1 recv the message 20

Processor 1 send the message 21

Код программы из 1 лабораторной работы:

*//--------------------Подключаемые* *библиотеки--------------------//*

#include <headers/mpi.h>

#include <iostream>

#include <cstring>

*using* *namespace* std;

int **main**(int argc, char\* argv[]) {

*//--------------------Иициализация--------------------//*

int procNum, procRank, recv;

int m = 2;*//количество* *итераций*

MPI\_Status Status;

MPI\_Init(&argc, *&argv*);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &*procNum*);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &*procRank*);

*//--------------------Главные* *процесс--------------------//*

*if* (procRank == 0) {

printf(" Num of processors: %3d", procNum);

printf("\n Main process with rank: %3d", procRank);

int id = 0;

printf("\n Send message to process: %3d", 1);

*//отправляем* *сообщение* *"0"* *процессу* *1*

MPI\_Send(&id, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

}

*//--------------------Остальные* *процессы--------------------//*

*else* {

*for*(int i = 0; i < m; i++) {

printf("\n Process %3d receiving message... ", procRank);

printf("\n Current time is: %3d", MPI\_Wtime());

*//процесс* *ождидает* *сообщение* *из* *любого* *источника*

*//первый* *попавшийся* *процесс* *перехватит* *управление*

MPI\_Recv(*&recv*, 1, MPI\_INT, MPI\_ANY\_SOURCE,

MPI\_ANY\_TAG, MPI\_COMM\_WORLD, *&Status*);

printf("\n Process recieved massage: %3d", recv);

printf("\n Current time is: %3d", MPI\_Wtime());

*//recv+1* *станет* *отправляемым* *сообщением*

++recv;

*//последний* *процесс* *в* *цепочке*

*if*(procRank == procNum-1) {

*if*(m != i+1){

*//отправляет* *сообщение* *recv* *процессу* *с* *рангом* *на* *1* *больше* *своего*

printf("\n Process sending massage: %3d", recv);

printf("\n Current time is: %3d", MPI\_Wtime());

MPI\_Send(&recv, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

}

}

*//любой* *другой* *процесс*

*else* {

*//отправляет* *сообщение* *recv* *процессу* *с* *рангом* *на* *1* *больше* *своего*

printf("\n Process %3d sending massage: %3d", recv);

printf("\n Current time is: %3d", MPI\_Wtime());

MPI\_Send(&recv, 1, MPI\_INT, procRank+1, 0, MPI\_COMM\_WORLD);

}

}

}

*//--------------------Завершение* *работы* *MPI--------------------//*

MPI\_Finalize();

*return* 0;

*}*

Замеры времени программы из 2 лабораторной (4 процесса, cout выключен):

elapsed time: 0.0035891s

elapsed time: 0.0035889s

elapsed time: 0.0035796s

elapsed time: 0.003748s

Замеры времени программы из 1 лабораторной (4 процесса, printf выключен):

elapsed time: 0.0246935s

elapsed time: 0.0482143s

elapsed time: 0.0379965s

elapsed time: 0.0315231s

Большой-большой Постскриптум

Изначально была попытка реализовать MPI\_Bcast для кольцевого обмена всех потоков кроме основного. Однако по какой-то причине после первой итерации обмена данными между процессами получаемое ими число начинает дублироваться, что приводит к некорректной работе программы (см. вывод). Либо в программе где-то находится очень незаметная ошибка (которую я за 3 с лишним часа так и не смог обнаружить), либо это связано с тем, что MPI\_Bcast обязан работать со всеми процессами одновременно и в реализации что-то ломается, когда один из процессов не получает сообщения. Решением во втором случае будет изменение группы процессоров, которые будут выполнять MPI\_Bcast.

Код программы:

*//--------------------Подключаемые* *библиотеки--------------------//*

#include <headers/mpi.h>

#include <iostream>

#include <cstring>

*using* *namespace* std;

int **main**(int argc, char\* argv[]) {

*//--------------------Иициализация--------------------//*

int procNum, procRank, recv;

int m = 3;*//кол-во* *итераций*

MPI\_Status status;

MPI\_Init(&argc, *&argv*);

MPI\_Comm\_size(MPI\_COMM\_WORLD, *&procNum*);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, *&procRank*);

*//--------------------Главный* *процесс--------------------//*

*if* (procRank == 0) {

cout << "Num of processors: " << procNum << endl;

cout << "Main process with rank: " << procRank << endl;

cout << "Send rank of cycle start processor to all processors" << endl;

int firstCount = 1;

*//MPI\_Send(&firstCount,* *1,* *MPI\_INT,* *1,* *0,* *MPI\_COMM\_WORLD);*

MPI\_Bcast(&*firstCount*, 1, MPI\_INT, procRank, MPI\_COMM\_WORLD);

}

*//--------------------Остальные* *процессы--------------------//*

*else* {

*if*(procRank == 1){

cout << "Take first" << endl;

*//MPI\_Recv(&recv,* *1,* *MPI\_INT,* *MPI\_ANY\_SOURCE,* *0,* *MPI\_COMM\_WORLD,* *&status);*

MPI\_Bcast(&*recv*, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

int nextCount = recv + 1;

MPI\_Bcast(*&nextCount*, 1, MPI\_INT, procRank, MPI\_COMM\_WORLD);

}

int currIter = -1;

*while*(*true*) {

int fromProc = (procNum+procRank-3)%(procNum-1)+1;

cout << "Processor " << procRank << " take from " << fromProc << endl;

MPI\_Bcast(*&recv*, 1, MPI\_INT, fromProc, MPI\_COMM\_WORLD);

cout << "Processor " << procRank << " took from " << fromProc << endl;

int nextCount = recv + 1;

cout << "Processor " << procRank << " recv the message " << recv << endl;

cout << "Processor " << procRank << " send the message " << nextCount << endl;

currIter = recv / (procNum-1);

*//cout* *<<* *"Now* *iter* *"* *<<* *currIter* *<<* *"* *processor* *"* *<<* *procRank* *<<* *endl;*

*if*(currIter == m){

cout << "End " << procRank << " processor " << procRank << endl;

*break*;

}

MPI\_Bcast(*&nextCount*, 1, MPI\_INT, procRank, MPI\_COMM\_WORLD);

cout << "Processor " << procRank << " end loop\n";

}

}

*//--------------------Завершение* *работы* *MPI--------------------//*

MPI\_Finalize();

*return* 0;

}

Вывод программы:

//Первый цикл

Take first//Берёт число из root 0

Num of processors: 4

Processor 3 take from 2

Main process with rank: 0

Processor 2 take from 1//Берёт число из root 1

Send rank of cycle start processor to all processors

Processor 1 take from 3//Берёт число из root 3

Processor 2 took from 1

Processor 2 recv the message 2

Processor 2 send the message 3

Processor 2 end loop

Processor 2 take from 1//Берёт число из root 1

Processor 3 took from 2

Processor 3 recv the message 3

Processor 3 send the message 4

Processor 1 took from 3

Processor 2 took from 1

//Начало второго цикла

Processor 1 recv the message 4//Сообщение дублируется, когда берёт из root 3

Processor 2 recv the message 4//Тоже берёт из root 3, хотя должно брать из root 1

Processor 1 send the message 5

Processor 3 end loop

Processor 3 take from 2

Processor 2 send the message 5

Processor 1 end loop

Processor 1 take from 3

Processor 3 took from 2

Processor 2 end loop

Processor 2 take from 1

Processor 3 recv the message 5

Processor 2 took from 1

Processor 3 send the message 6

Processor 1 took from 3

Processor 2 recv the message 5

Processor 3 end loop

Processor 3 take from 2

Processor 1 recv the message 6

Processor 2 send the message 6

Processor 3 took from 2

Processor 1 send the message 7

Processor 2 end loop

Processor 2 take from 1

Processor 3 recv the message 6

Processor 1 end loop

Processor 1 take from 3

Processor 2 took from 1

Processor 3 send the message 7

Processor 1 took from 3

Processor 2 recv the message 6

Processor 3 end loop

Processor 3 take from 2

Processor 1 recv the message 7

Processor 2 send the message 7

Processor 1 send the message 8

Processor 3 took from 2

Processor 2 end loop

Processor 2 take from 1

Processor 1 end loop

Processor 1 take from 3

Processor 3 recv the message 7

Processor 2 took from 1

Processor 3 send the message 8

Processor 2 recv the message 7

Processor 1 took from 3

Processor 3 end loop

Processor 3 take from 2

Processor 2 send the message 8

Processor 1 recv the message 8

Processor 3 took from 2

Processor 2 end loop

Processor 2 take from 1

Processor 1 send the message 9

Processor 3 recv the message 8

Processor 2 took from 1

Processor 1 end loop

Processor 1 take from 3

Processor 3 send the message 9

Processor 2 recv the message 7

Processor 1 took from 3

Processor 3 end loop

Processor 3 take from 2

Processor 2 send the message 8

Processor 1 recv the message 9

Processor 3 took from 2

Processor 2 end loop

Processor 2 take from 1

Processor 1 send the message 10

Processor 3 recv the message 8

Processor 2 took from 1

End 1 processor 1

Processor 3 send the message 9

Processor 2 recv the message 8

Processor 3 end loop

Processor 3 take from 2

Processor 2 send the message 9

Processor 3 took from 2

Processor 2 end loop

Processor 2 take from 1

Processor 3 recv the message 9

Processor 2 took from 1

Processor 3 send the message 10

Processor 2 recv the message 8

End 3 processor 3

Processor 2 send the message 9

Processor 2 end loop

Processor 2 take from 1

Processor 2 took from 1

Processor 2 recv the message 9

Processor 2 send the message 10

End 2 processor 2

Контрольные вопросы:

**Как происходит передача данных от одного процесса всем?**

Функции MPI\_Send и MPI\_Recv, обеспечивают возможность выполнения парных операций передачи данных между двумя процессами параллельной программы. Для выполнения коммуникационных коллективных операций, в которых принимают участие все процессы коммуникатора, в MPI предусмотрен специальный набор функций.

Достижение эффективного выполнения операции передачи данных от одного процесса всем процессам программы (широковещательная рассылка данных) может быть обеспечено при помощи функции MPI:

int MPI\_Bcast(void \*buf,int count,MPI\_Datatype type,int root,MPI\_Comm comm);

где

buf, count, type – буфер памяти с отправляемым сообщением (для процесса с рангом 0),

и для приема сообщений для всех остальных процессов,

root - ранг процесса, выполняющего рассылку данных,

comm - коммуникатор, в рамках которого выполняется передача данных.

Функция MPI\_Bcast осуществляет рассылку данных из буфера buf, содержащего count элементов типа type с процесса, имеющего номер root, всем процессам, входящим в коммуникатор comm.

- функция MPI\_Bcast определяет коллективную операцию и, тем самым, при выполнении необходимых рассылок данных вызов функции MPI\_Bcast должен быть осуществлен всеми процессами указываемого коммуникатора;

- указываемый в функции MPI\_Bcast буфер памяти имеет различное назначение в разных процессах. Для процесса с рангом root, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение. Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

**Как происходит передача данных от всем процессов одному?**

С помощью операции передачи данных от всех процессов одному процессу. В этой операции над собираемыми значениями осуществляется та или иная обработка данных (данная операция еще именуется операцией редукции данных). Реализация операции редукции при помощи обычных парных операций передачи данных является неэффективной и достаточно трудоемкой. Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция:

int MPI\_Reduce(void \*sendbuf, void \*recvbuf,int count,MPI\_Datatype type,

MPI\_Op op,int root,MPI\_Comm comm);

где

- sendbuf - буфер памяти с отправляемым сообщением,

- recvbuf – буфер памяти для результирующего сообщения (только для процесса с рангом root),

- count - количество элементов в сообщениях,

- type – тип элементов сообщений,

- op - операция, которая должна быть выполнена над данными,

- root - ранг процесса, на котором должен быть получен результат,

- comm - коммуникатор, в рамках которого выполняется операция.

- функция MPI\_Reduce определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров count, type, op, root, comm;

- передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом root;

- выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования MPI\_SUM, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно.

**Какие используются в MPI для синхронизации вычислений?**

Синхронизация процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

int MPI\_Barrier(MPI\_Comm comm);

Функция MPI\_Barrier должна вызываться всеми процессами используемого коммуникатора. При вызове функции MPI\_Barrier выполнение процесса блокируется, продолжение вычислений процесса происходит только после вызова функции MPI\_Barrier всеми процессами коммуникатора.

**Как организуется неблокирующий обмен данными между процессами?**

Функции отправки и приема сообщений, рассмотренные в 1 лабораторной являются блокирующими. При выполнении параллельных вычислений часть сообщений может быть отправлена и принята заранее до момента реальной потребности в пересылаемых данных. В таких ситуациях желательно иметь возможность выполнения функций обмена данными без блокировки процессов для совмещения процессов передачи сообщений и вычислений. Такой неблокирующий способ выполнения обменов помогает уменьшать потери эффективности параллельных вычислений из-за коммуникационных операций.

MPI обеспечивает возможность неблокированного выполнения операций передачи данных между двумя процессами. Наименование неблокирующих аналогов образуется из названий соответствующих функций путем добавления префикса I (Immediate). Список параметров неблокирующих функций содержит весь набор параметров исходных функций и один дополнительный параметр request с типом MPI\_Request (в функции MPI\_Irecv отсутствует также параметр status):

int MPI\_Isend(void \*buf, int count, MPI\_Datatype type, int dest,

int tag, MPI\_Comm comm, MPI\_Request \*request);

int MPI\_Issend(void \*buf, int count, MPI\_Datatype type, int dest,

int tag, MPI\_Comm comm, MPI\_Request \*request);

int MPI\_Ibsend(void \*buf, int count, MPI\_Datatype type, int dest,

int tag, MPI\_Comm comm, MPI\_Request \*request);

int MPI\_Irsend(void \*buf, int count, MPI\_Datatype type, int dest,

int tag, MPI\_Comm comm, MPI\_Request \*request);

int MPI\_Irecv(void \*buf, int count, MPI\_Datatype type, int source,

int tag, MPI\_Comm comm, MPI\_Request \*request);

Вызов неблокирующей функции приводит к инициации запрошенной операции передачи, после чего выполнение функции завершается и процесс может продолжить свои действия.

Перед своим завершением неблокирующая функция определяет переменную request, которая далее может использоваться для проверки завершения инициированной операции обмена.

Проверка состояния выполняемой неблокирующей операции передачи данных выполняется при помощи функции:

int MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_status \*status);

где

request - дескриптор операции, определенный при вызове неблокирующей функции,

flag - результат проверки (= true, если операция завершена),

status - результат выполнения операции обмена (только для завершенной операции).

Если при выполнении неблокирующей операции окажется, что продолжение вычислений невозможно без получения передаваемых данных, то может быть использована блокирующая операция ожидания завершения операции:

int MPI\_Wait(MPI\_Request \*request, MPI\_status \*status)

Кроме рассмотренных, MPI содержит ряд дополнительных функций проверки и ожидания неблокирующих операций обмена:

- MPI\_Testall - проверка завершения всех перечисленных операций обмена;

- MPI\_Waitall – ожидание завершения всех операций обмена;

- MPI\_Testany - проверка завершения хотя бы одной из перечисленных операций обмена;

- MPI\_Waitany – ожидание завершения любой из перечисленных операций обмена;

- MPI\_Testsome - проверка завершения каждой из перечисленных операций обмена;

- MPI\_Waitsome - ожидание завершения хотя бы одной из перечисленных операций обмена и оценка состояния по всем операциям.

**Как организуется одновременное выполнение прием и передачи данных?**

Достижение эффективного и гарантированного одновременного выполнения операций передачи и приема данных может быть обеспечено при помощи функции MPI:

int MPI\_Sendrecv(void \*sbuf,int scount,MPI\_Datatype stype,int dest, int stag, void \*rbuf,int rcount,MPI\_Datatype rtype,int source,int rtag, MPI\_Comm comm, MPI\_Status \*status);

где

sbuf, scount, stype, dest, stag - параметры передаваемого сообщения;

rbuf, rcount, rtype, source, rtag - параметры принимаемого сообщения;

comm - коммуникатор, в рамках которого выполняется передача данных;

status – структура данных с информацией о результате выполнения операции.

Функция MPI\_Sendrecv передает сообщение, описываемое параметрами (sbuf, scount, stype, dest, stag), процессу с рангом dest и принимает сообщение в буфер, определяемый параметрами (rbuf, rcount, rtype, source, rtag), от процесса с рангом source.

В функции MPI\_Sendrecv для передачи и приема сообщений применяются разные буфера. В случае же, когда сообщения имеют одинаковый тип, в MPI имеется возможность использования единого буфера:

int MPI\_Sendrecv\_replace (void \*buf, int count, MPI\_Datatype type, int dest, int stag, int source, int rtag, MPI\_Comm comm, MPI\_Status\* status)