Лабораторная работа №2

Коллективные операции передачи данных

Максим Ромашкин

ПИН-34   Номер в списке: 18

Оглавление

[**Задание, код программы и пояснение:** 1](#_Toc95838769)

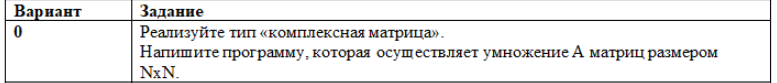
[**Листинг программы:** 1](#_Toc95838770)

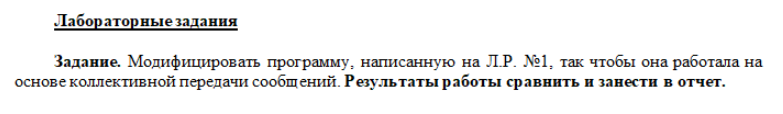
[**Результат программы:** 1](#_Toc95838771)

[**Выполнение примера** 1](#_Toc95838772)

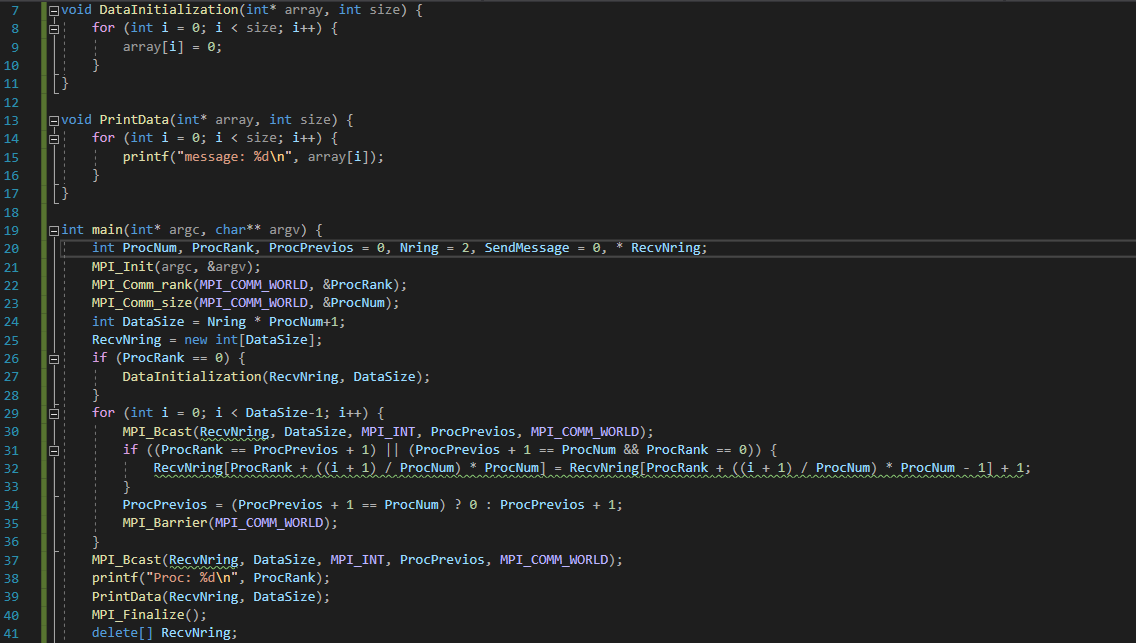
[**Ответы на контрольные вопросы** 1](#_Toc95838773)

# **Задание, код программы и пояснение:**





В процессе 0 создается массив (размером [Количество процессов] \*[Количество кругов] +1) и заполняется 0. Далее 0 процесс отправляет всем процессам этот массив. После этого первый процесс увеличивает «свою» ячейку массива на 1 от ячейки массива, принадлежащей предыдущему процесс (то есть 0) и передает всем остальным процессам измененный массив. И так далее каждый следующий процесс увеличивает на 1 от предыдущего. После цикла изначальный процесс передает всем остальным процессам получившийся в итоге массив.



# **Листинг программы:**

#include <stdio.h>

#include <mpi.h>

#include <iostream>

using namespace std;

void DataInitialization(int\* array, int size) {

for (int i = 0; i < size; i++) {

array[i] = 0;

}

}

void PrintData(int\* array, int size) {

for (int i = 0; i < size; i++) {

printf("message: %d\n", array[i]);

}

}

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

int ProcNum, ProcRank, ProcPrevios = 0, Nring = 2, SendMessage = 0, \* RecvNring;

MPI\_Init(argc, &argv);

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

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

int DataSize = Nring \* ProcNum+1;

RecvNring = new int[DataSize];

if (ProcRank == 0) {

DataInitialization(RecvNring, DataSize);

}

for (int i = 0; i < DataSize-1; i++) {

MPI\_Bcast(RecvNring, DataSize, MPI\_INT, ProcPrevios, MPI\_COMM\_WORLD);

if ((ProcRank == ProcPrevios + 1) || (ProcPrevios + 1 == ProcNum && ProcRank == 0)) {

RecvNring[ProcRank + ((i + 1) / ProcNum) \* ProcNum] = RecvNring[ProcRank + ((i + 1) / ProcNum) \* ProcNum - 1] + 1;

}

ProcPrevios = (ProcPrevios + 1 == ProcNum) ? 0 : ProcPrevios + 1;

MPI\_Barrier(MPI\_COMM\_WORLD);

}

MPI\_Bcast(RecvNring, DataSize, MPI\_INT, ProcPrevios, MPI\_COMM\_WORLD);

printf("Proc: %d\n", ProcRank);

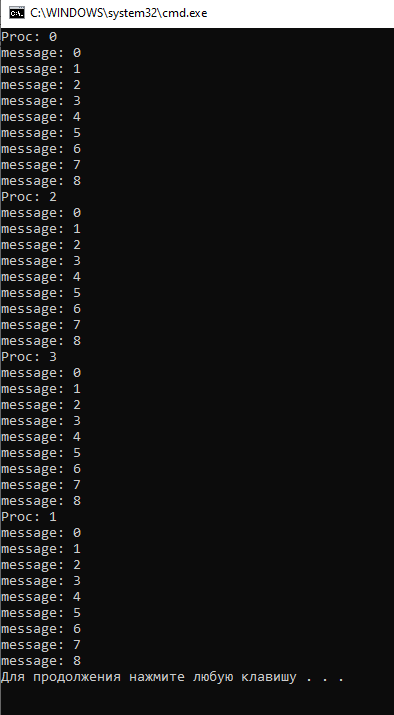
PrintData(RecvNring, DataSize);

MPI\_Finalize();

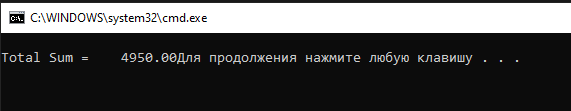
delete[] RecvNring;

}

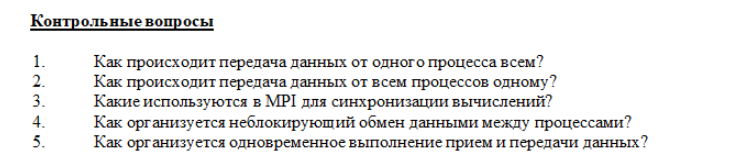
# **Результат программы:**



# **Выполнение примера**



# **Ответы на контрольные вопросы**



1. 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);
2. Коллективной операция 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);
3. 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.
4. 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);  
     
   Проверка состояния выполняемой неблокирующей операции передачи данных выполняется при помощи функции:  
     
   int **MPI\_Test**(MPI\_Request \*request, int \*flag, MPI\_status \*status);  
     
   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)
5. 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);  
     
   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)