**Лабораторная работа №2**

**Коллективные операции передачи данных**

**Отчет и код выполнил**

**студент группы ПИН-34**

**Устян Давид**

Цель: изучить основные принципы коллективных операций передачи данных в технологии MPI на примере использования в рамках языка С++.

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);

- 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.

2. Передача данных от всех процессов одному процессу. Операции редукции

В рассмотренной программе суммирования числовых значений имеющаяся процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной 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;

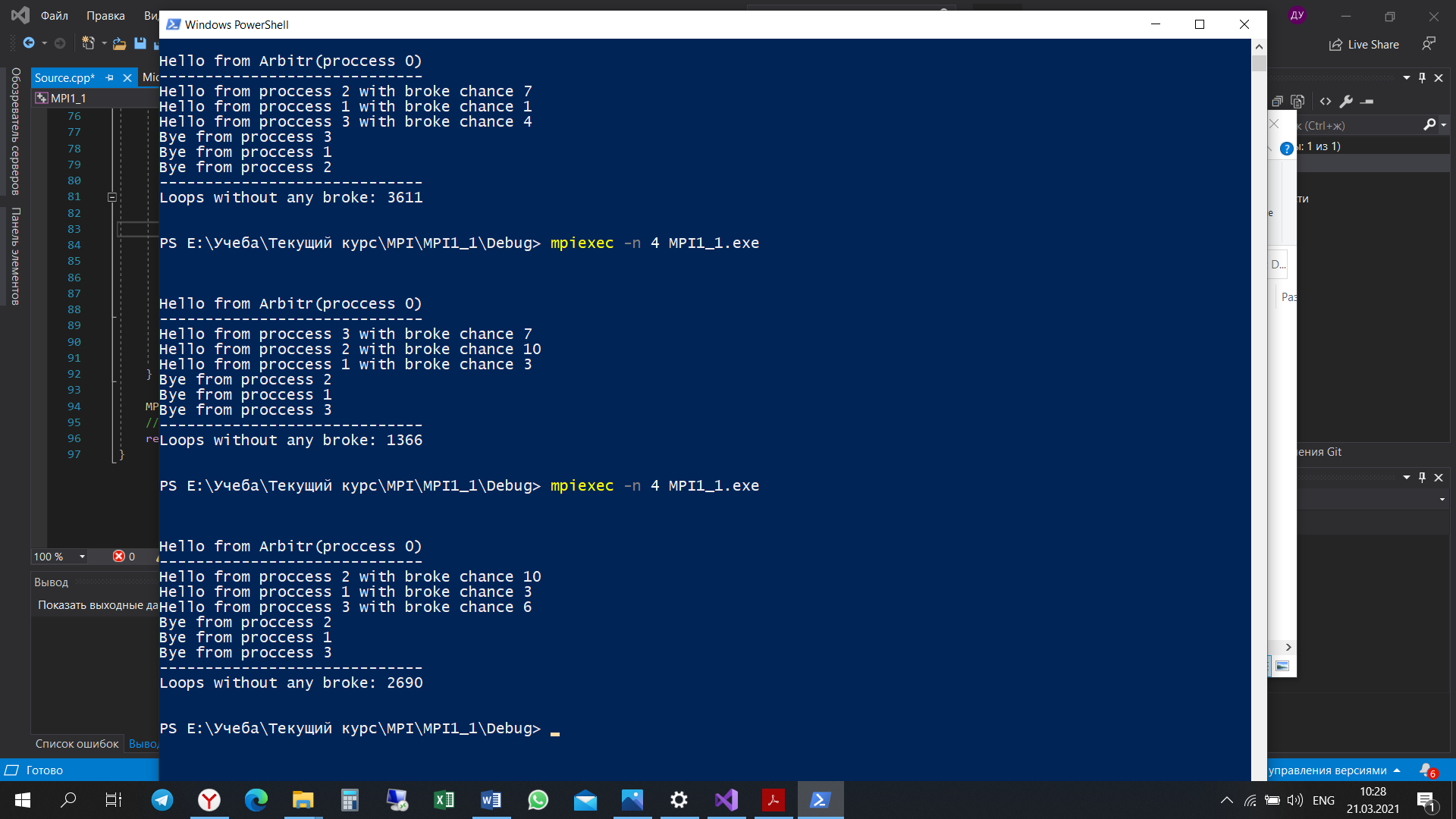
- передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом root

**Задание (вариант 2):**

Реализуйте процесс-«счётчик», (который запускается со значением 0) и

1) если получена -1, то он выводит в текущее значение и заканчивает работу;

2) если получено любое другое сообщение, то значение увеличивается на 1

Результат запуска для 4-х процессов 

Добиться разных значений удалось за счет введения функции зерна генератора, зависящего от ранга процесса.

Код программы:

#include "mpi.h"

#include "time.h"

#include "stdlib.h"

#include "string"

#include "stdio.h"

#include <iostream>;

using namespace std;

// Программа симулирует подсчет выполненных работ до первой поломки.

// За симуляцию работы отвечает функция work(int crit), где crit - критическое значение от 0 до 100

// В случае поломки функция возвращает -1

// Процесс 0 анализирует выполнение работы другими процессами и завершает работу, если хотя бы один сломался.

int work(int crit)

{

int value = rand() % 100000 - crit;

if (value < 0)

return -1;

return value;

}

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

{

const int broke\_radius = 10;

const int min\_broke\_chance = 1;

int buffer = 0, numtasks, rank;

int done\_work = 0, work\_value = 0;

int crit\_value = 0;

MPI\_Status Status;

// <программный код без использования MPI функций>

MPI\_Init(&argc, &argv);

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

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

// <программный код с использованием MPI функций>

if (rank == 0)

{

//printf("Hello from Arbitr (proccess 0)\n");

cout << "\n\n\nHello from Arbitr(proccess 0)\n"

<< "-----------------------------\n";

fflush(stdout);

MPI\_Barrier(MPI\_COMM\_WORLD);

work\_value = 1;

done\_work = 0;

while (work\_value == 1)

{

MPI\_Bcast(&work\_value, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&work\_value, &buffer, 1, MPI\_DOUBLE, MPI\_MIN, 0, MPI\_COMM\_WORLD);

if (buffer == -1)

{

work\_value = -1;

MPI\_Bcast(&work\_value, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

else done\_work++;

}

MPI\_Barrier(MPI\_COMM\_WORLD);

//printf("Loops without any broke: %d\n", done\_work);

cout << "-----------------------------\n"

<< "Loops without any broke: " << done\_work << "\n\n\n";

}

else

{

srand(time(NULL) + 100 \* rank);

crit\_value = rand() % broke\_radius + min\_broke\_chance;

MPI\_Barrier(MPI\_COMM\_WORLD);

//printf("Hello from proccess %d with broke chance %d\n", rank, crit\_value);

cout << "Hello from proccess " << rank << " with broke chance " << crit\_value << endl;

while (true)

{

if (work\_value == -1) break;

work\_value = work(crit\_value);

MPI\_Reduce(&work\_value, &buffer, 1, MPI\_DOUBLE, MPI\_MIN, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&work\_value, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

//printf("Hello from proccess %d", rank);

cout << "Bye from proccess " << rank << endl;

MPI\_Barrier(MPI\_COMM\_WORLD);

}

MPI\_Finalize();

// <программный код без использования MPI функций>

return 0;

}