

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

**отчет**

Выполнил:  
студент: Фадига мамадуГруппа: МSK11

***Задание 14. Создание проекта в среде MS Visual Studio с поддержкой MPI***

Создайте проект в среде Visual Studio 2010 с поддержкой MPI.

#include <iostream>

#include <mpi.h>

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

MPI\_Init(&argc, &argv); // Правильное написание!

int rank, size;

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

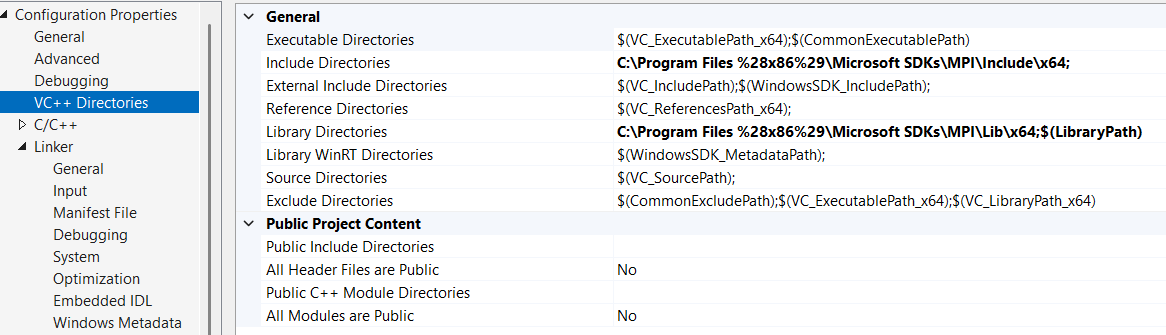
MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

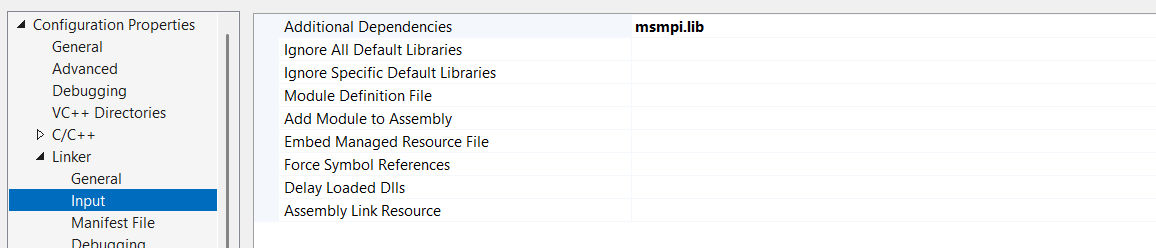
std::cout << "Hello from process " << rank << " of " << size << std::endl;

MPI\_Finalize(); // Не "Finatize"!

return 0;

}





***Задание*** 15. Программа «***I am!»***

Напишите программу, в которой каждый процесс выводит на экран свой номер и общее количество процессов в приложении в формате:

I am <Номер процесса> process from <Количество процессов> processes!

**Входные данные:** нет.

**Выходные данные:** строки в формате «I am <Номер процесса> process from <Количество процессов> processes!».

#include <mpi.h>

Подключает заголовочный файл MPI, который содержит объявления функций и типов данных, необходимых для использования MPI.

#include <iostream>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

if (rank == 0) {

std::cout << size << " processes." << std::endl;

} else if (rank % 2 == 0) {

std::cout << "I am " << rank << " process: FIRST!" << std::endl;

} else {

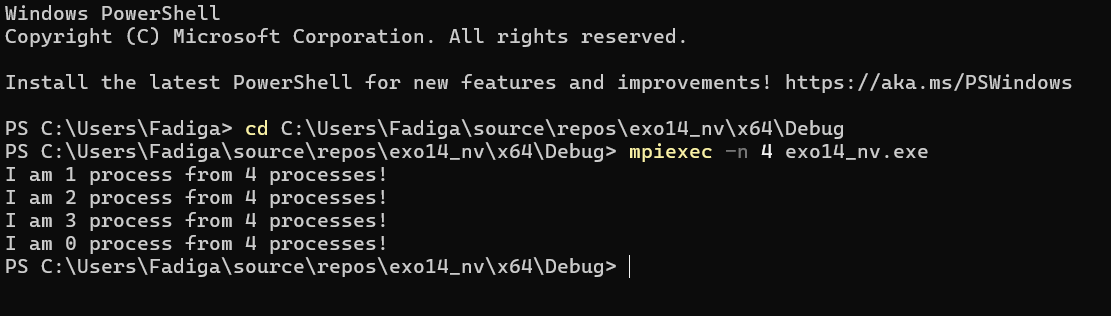
std::cout << "I am " << rank << " process: SECOND!" << std::endl;

}

MPI\_Finalize();

return 0;

}



***Задание 16. Программа «На первый-второй рассчитайся!»***

Напишите программу, в которой каждый процесс с четным номером вы-водит на экран строку «I am <Номер процесса>: FIRST!», а каждый про-цесс с нечетным номером – «I am <Номер процесса>: SECOND!». Процесс с номером 0 должен вывести на экран общее количество процессов в приложе-нии в формате «<Количество процессов> processes.».

**Входные данные:** нет.

**Выходные данные:** строки в формате «I am <Номер процесса>: FIRST!» или «I am <Номер процесса>: SECOND!» или «<Количество процессов> processes.».

#include <mpi.h>

Подключает заголовочный файл MPI, который содержит объявления функций и типов данных, необходимых для использования MPI

#include <iostream>

 Подключает заголовочный файл iostream, необходимый для ввода и вывода данных, в частности, для использования std::cout

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

MPI\_Init(&argc, &argv);

Главная функция программы. argc и argv позволяют передавать аргументы командной строки в программу.

Инициализирует среду MPI. Этот вызов необходим перед использованием любых других функций MPI.

int rank, size;

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

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

if (rank == 0) {

std::cout << size << " processes." << std::endl;

}

else if (rank % 2 == 0) {

std::cout << "I am " << rank << " process: FIRST!" << std::endl;

}

else {

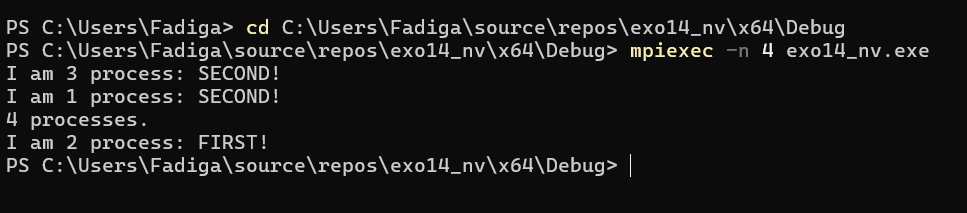
std::cout << "I am " << rank << " process: SECOND!" << std::endl;

}

MPI\_Finalize();

return 0;

}



***Задание*** 17:Напишите программу, в которой каждый процесс с четным номером вы-водит на экран строку «I am <Номер процесса>: FIRST!», а каждый про-цесс с нечетным номером – «I am <Номер процесса>: SECOND!». Процесс с номером 0 должен вывести на экран общее количество процессов в приложе-нии в формате «<Количество процессов> processes.».

**Входные данные:** нет.

**Выходные данные:** строки в формате «I am <Номер процесса>: FIRST!» или «I am <Номер процесса>: SECOND!» или «<Количество процессов> processes.».

#include <mpi.h>

#include <iostream>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

// Чтобы вывод был упорядоченным, можно использовать барьер и последовательный вывод

if (rank == 0) {

std::cout << size << " processes." << std::endl;

}

// Синхронизация, чтобы процесс 0 вывел количество процессов перед остальными

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank != 0) {

if (rank % 2 == 0) {

std::cout << "I am " << rank << " process: FIRST!" << std::endl;

}

else {

std::cout << "I am " << rank << " process: SECOND!" << std::endl;

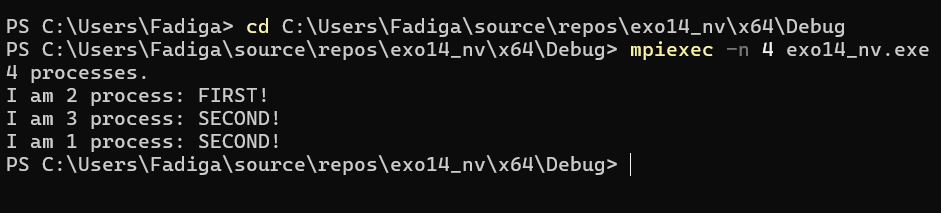
}

}

MPI\_Finalize();

return 0;

}



***Задание 18. Коммуникации «точка-точка»: схема «эстафетная палочка»***

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процес-сов «эстафетная палочка», в которой каждый процесс дожидается сообщения от предыдущего и потом посылает следующему (см. рис. 1). В качестве пере-даваемого сообщения используйте на процессе 0 его номер, на остальных процессах – инкрементированное полученное сообщение.

#include <mpi.h>

#include <iostream>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int message;

if (rank == 0) {

// Процесс 0 инициализирует сообщение своим номером

message = rank;

// Отправляем сообщение процессу 1

MPI\_Send(&message, 1, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

std::cout << "Process " << rank << " sent message " << message << " to process " << rank + 1 << std::endl;

}

else {

// Все остальные процессы получают сообщение от предыдущего

MPI\_Recv(&message, 1, MPI\_INT, rank - 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

std::cout << "Process " << rank << " received message " << message << " from process " << rank - 1 << std::endl;

// Инкрементируем сообщение

message++;

// Если не последний процесс, отправляем дальше

if (rank < size - 1) {

MPI\_Send(&message, 1, MPI\_INT, rank + 1, 0, MPI\_COMM\_WORLD);

std::cout << "Process " << rank << " sent message " << message << " to process " << rank + 1 << std::endl;

}

else {

// Последний процесс просто выводит итоговое сообщение

std::cout << "Process " << rank << " is the last process. Final message value: " << message << std::endl;

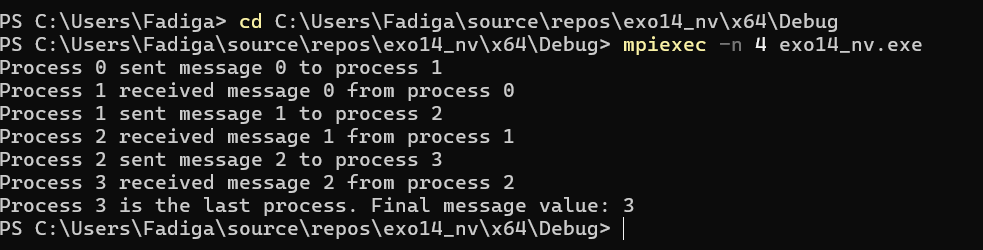
}

}

MPI\_Finalize();

return 0;

}



***Задание19.Коммуникации «точка-точка»: схема «мастер-рабочие»***

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процес-сов «master-slave», в которой один процесс, называемый master2, принимает

2 Master-процессом может быть любой произвольный процесс, обычно это процесс с но-мером 0. 14

сообщение от остальных процессов, называемых slave (см. рис. 2). В качестве передаваемого сообщения используйте номер процесса. Master-процесс дол-жен вывести на экран все полученные сообщения.

#include <mpi.h>

#include <iostream>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

if (rank == 0) {

// Master-процесс принимает сообщения от всех остальных

std::cout << "Master process started. Waiting for messages..." << std::endl;

for (int i = 1; i < size; ++i) {

int received\_rank;

MPI\_Recv(&received\_rank, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

std::cout << "Master received message from process " << received\_rank << std::endl;

}

}

else {

// Slave-процессы отправляют master свой номер

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

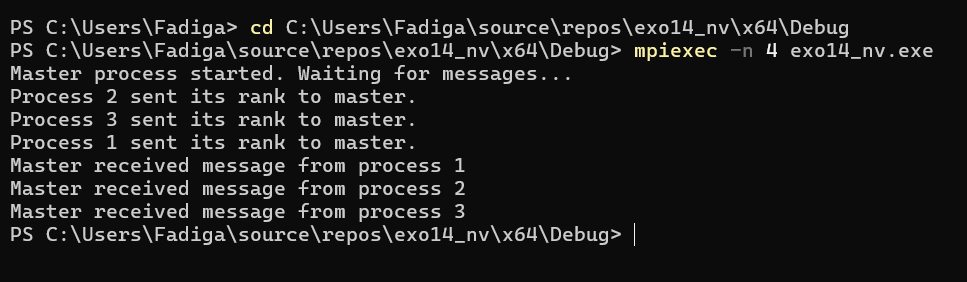
std::cout << "Process " << rank << " sent its rank to master." << std::endl;

}

MPI\_Finalize();

return 0;

}



***Задание 20. Коммуникации «точка-точка»: простые неблокирующие об-мены***

Изучите основные MPI-функции неблокирующей передачи сообщений точка-точка MPI\_Isend, MPI\_Irecv, MPI\_Wait. Напишите MPI-программу, в которой с помощью данных функций процесс с номером 0 отправляет сооб-щение процессу с номером 1. Процесс 1 выводит полученное сообщение на экран.

**Входные данные:** нет.

**Выходные данные:** «receive message '<сообщение>'».

#include <mpi.h>

#include <iostream>

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

MPI\_Init(&argc, &argv);

int rank;

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

int message;

MPI\_Request request;

MPI\_Status status;

if (rank == 0) {

// Инициализируем сообщение

message = 12345; // любое число для отправки

// Неблокирующая отправка сообщения процессу 1

MPI\_Isend(&message, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, &request);

// Ожидаем завершения отправки

MPI\_Wait(&request, &status);

}

if (rank == 1) {

// Неблокирующий приём сообщения от процесса 0

MPI\_Irecv(&message, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &request);

// Ожидаем завершения приёма

MPI\_Wait(&request, &status);

// Выводим полученное сообщение

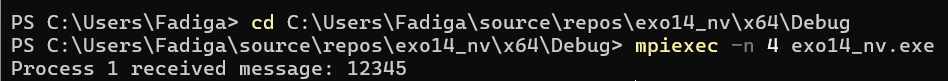
std::cout << "Process 1 received message: " << message << std::endl;

}

MPI\_Finalize();

return 0;

}



***Задание 21. Коммуникации «точка-точка»: схема «сдвиг по кольцу»***

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процес-сов «сдвиг по кольцу», в которой осуществляются одновременные посылка и прием сообщений всеми процессами (см. рис. 3). В качестве передаваемого сообщения используйте номер процесса.

#include <mpi.h>

#include <iostream>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int send\_msg = rank; // сообщение — номер процесса

int recv\_msg; // переменная для приёма сообщения

int dest = (rank + 1) % size; // следующий процесс в кольце

int source = (rank - 1 + size) % size; // предыдущий процесс в кольце

// Чтобы избежать взаимной блокировки, можно использовать следующую схему:

// процессы с чётным рангом сначала отправляют, потом принимают,

// а процессы с нечётным — наоборот.

if (rank % 2 == 0) {

MPI\_Send(&send\_msg, 1, MPI\_INT, dest, 0, MPI\_COMM\_WORLD);

MPI\_Recv(&recv\_msg, 1, MPI\_INT, source, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

} else {

MPI\_Recv(&recv\_msg, 1, MPI\_INT, source, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Send(&send\_msg, 1, MPI\_INT, dest, 0, MPI\_COMM\_WORLD);

}

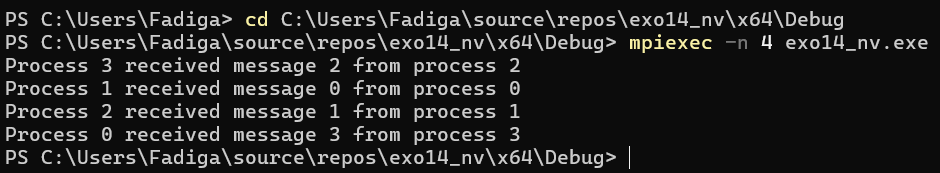
std::cout << "Process " << rank << " received message " << recv\_msg

<< " from process " << source << std::endl;

MPI\_Finalize();

return 0;

}



***Задание 22. Коммуникации «точка-точка»: схема «каждый каждому»***

Напишите MPI-программу, реализующую при помощи блокирующих функций посылки сообщений типа точка-точка схему коммуникации процес-сов «каждый каждому», в которой осуществляется пересылка сообщения от каждого процесса каждому (см. рис. 4). В качестве передаваемого сообщения используйте номер процесса. Каждый процесс должен вывести на экран все полученные сообщения.

#include <mpi.h>

#include <iostream>

#include <vector>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

// Отправляем свое сообщение всем другим процессам

int send\_msg = rank;

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

if (dest != rank) { // Исключаем отправку самому себе

MPI\_Send(&send\_msg, 1, MPI\_INT, dest, 0, MPI\_COMM\_WORLD);

}

}

// Принимаем сообщения от всех других процессов

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

if (src != rank) { // Исключаем прием от самого себя

int received\_msg;

MPI\_Recv(&received\_msg, 1, MPI\_INT, src, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

std::cout << "[" << rank << "]: receive message '" << received\_msg

<< "' from " << src << std::endl;

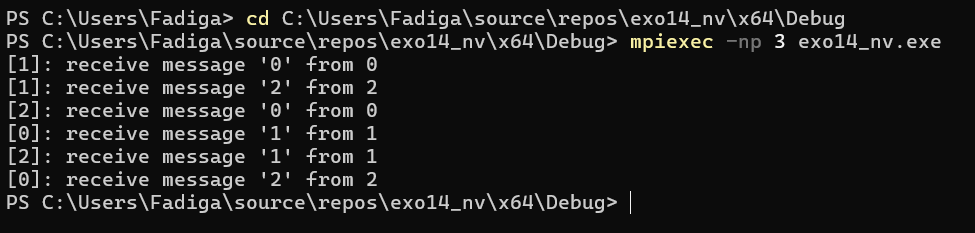
}

}

MPI\_Finalize();

return 0;

}



***Задание*** 23. Коллективные коммуникации: широковещательная рассылка данных

1. Изучите MPI-функцию широковещательной рассылки данных MPI\_Bcast. Напишите MPI-программу, которая в строке длины n определяет количество вхождений символов. Ввод данных должен осуществляться про-цессом с номером 0. Для рассылки строки поиска и ее длины по процессам используйте функцию MPI\_Bcast.

#include <mpi.h>

#include <iostream>

#include <string>

#include <vector>

#include <unordered\_map>

#include <algorithm>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int n = 0;

std::string input\_str;

std::vector<char> buffer;

std::unordered\_map<char, int> local\_counts;

std::unordered\_map<char, int> global\_counts;

// Процесс 0 читает входные данные

if (rank == 0) {

std::cin >> n;

std::cin.ignore(); // Пропускаем перевод строки

std::getline(std::cin, input\_str);

// Проверка длины строки

if (input\_str.size() != n) {

std::cerr << "Error: String length doesn't match specified size" << std::endl;

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

}

// Рассылаем длину строки всем процессам

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

// Подготавливаем буфер и рассылаем строку

buffer.resize(n);

if (rank == 0) {

std::copy(input\_str.begin(), input\_str.end(), buffer.begin());

}

MPI\_Bcast(buffer.data(), n, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

// Распределяем работу между процессами

int chunk\_size = n / size;

int remainder = n % size;

int start = rank \* chunk\_size + (rank < remainder ? rank : remainder);

int end = start + chunk\_size + (rank < remainder ? 1 : 0);

// Локальный подсчёт символов

for (int i = start; i < end; i++) {

local\_counts[buffer[i]]++;

}

// Собираем результаты на процессе 0

if (rank == 0) {

// Копируем локальные результаты процесса 0

global\_counts = local\_counts;

// Получаем результаты от других процессов

for (int src = 1; src < size; src++) {

// Получаем количество уникальных символов

int unique\_chars;

MPI\_Recv(&unique\_chars, 1, MPI\_INT, src, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

// Получаем пары символ-количество

for (int i = 0; i < unique\_chars; i++) {

char c;

int count;

MPI\_Recv(&c, 1, MPI\_CHAR, src, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(&count, 1, MPI\_INT, src, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

global\_counts[c] += count;

}

}

// Сортируем символы для вывода

std::vector<char> sorted\_chars;

for (auto& pair : global\_counts) {

sorted\_chars.push\_back(pair.first);

}

std::sort(sorted\_chars.begin(), sorted\_chars.end());

// Выводим результаты

for (char c : sorted\_chars) {

std::cout << c << " = " << global\_counts[c] << std::endl;

}

}

else {

// Отправляем количество уникальных символов

int unique\_chars = local\_counts.size();

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

// Отправляем пары символ-количество

for (auto& pair : local\_counts) {

MPI\_Send(&pair.first, 1, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);

MPI\_Send(&pair.second, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

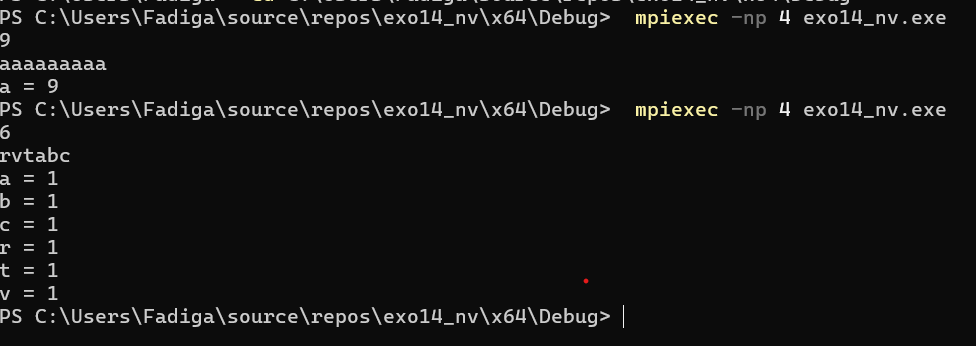
}

}

MPI\_Finalize();

return 0;

}



2\*. Перепишите программу, используя вместо функции MPI\_Bcast функции коммуникации «точка-точка». Сравните эффективность выполнения программ с коллективными и точечными обменами.

#include <mpi.h>

#include <iostream>

#include <string>

#include <vector>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int n = 0;

char search\_char = '\0';

std::string input\_str;

if (rank == 0) {

std::cout << "Enter a string: ";

std::getline(std::cin, input\_str);

n = static\_cast<int>(input\_str.size());

std::cout << "Enter a character to count: ";

std::cin >> search\_char;

// Отправляем длину строки и символ всем остальным процессам

for (int dest = 1; dest < size; ++dest) {

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

MPI\_Send(&search\_char, 1, MPI\_CHAR, dest, 0, MPI\_COMM\_WORLD);

}

}

else {

// Получаем длину строки и символ от процесса 0

MPI\_Recv(&n, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(&search\_char, 1, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

std::vector<char> buffer(n);

if (rank == 0) {

std::copy(input\_str.begin(), input\_str.end(), buffer.begin());

// Отправляем строку всем остальным процессам

for (int dest = 1; dest < size; ++dest) {

MPI\_Send(buffer.data(), n, MPI\_CHAR, dest, 0, MPI\_COMM\_WORLD);

}

}

else {

// Получаем строку от процесса 0

MPI\_Recv(buffer.data(), n, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

// Подсчёт вхождений символа

int local\_count = 0;

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

if (buffer[i] == search\_char)

local\_count++;

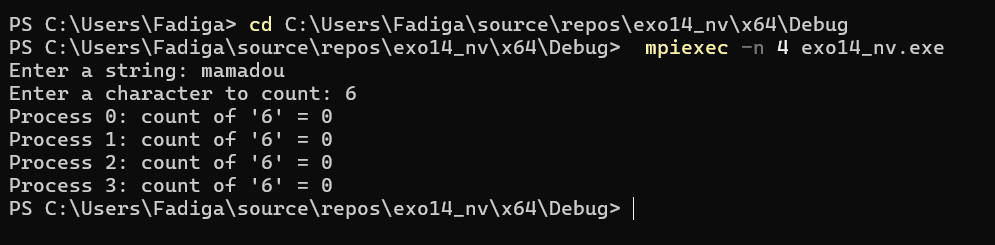
}

std::cout << "Process " << rank << ": count of '" << search\_char << "' = " << local\_count << std::endl;

MPI\_Finalize();

return 0;

}



***Задание 24. Коллективные коммуникации: операции редукции***

1. Изучите MPI-функцию для выполнения операций редукции над дан-ными, расположенными в адресных пространствах различных процессов, MPI\_Reduce. Реализуйте программу вычисления числа 𝜋 (см. задание 8), ис-пользуйте функцию MPI\_Reduce для суммирования результатов, вычислен-ных каждым процессом.

2\*. Перепишите программу, используя вместо функции MPI\_Reduce функции коммуникации «точка-точка». Сравните эффективность выполнения программ с коллективными и точечными обменами.

**Входные данные:** одно целое число N (точность вычисления).

**Выходные данные:** одно вещественное число pi.

1.

#include <iostream>

#include <mpi.h>

#include <cmath>

double calculate\_pi\_part(int start, int end, int N) {

double sum = 0.0;

for (int i = start; i < end; i++) {

double x = (i + 0.5) / N;

sum += 4.0 / (1.0 + x \* x);

}

return sum;

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

if (argc != 2) {

if (rank == 0) {

std::cerr << "Usage: " << argv[0] << " N" << std::endl;

}

MPI\_Finalize();

return 1;

}

int N = std::atoi(argv[1]);

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = (rank == size - 1) ? N : start + chunk\_size;

double local\_sum = calculate\_pi\_part(start, end, N);

double global\_sum;

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

double pi = global\_sum / N;

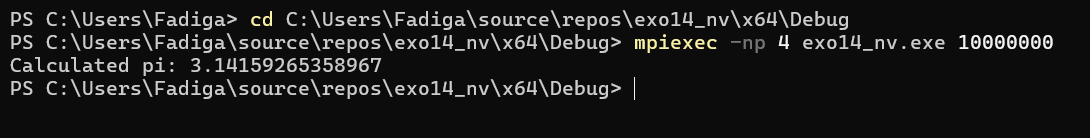
std::cout.precision(15);

std::cout << "Calculated pi: " << pi << std::endl;

}

MPI\_Finalize();

return 0;

}

2.

#include <iostream>

#include <mpi.h>

#include <cmath>

double calculate\_pi\_part(int start, int end, int N) {

double sum = 0.0;

for (int i = start; i < end; i++) {

double x = (i + 0.5) / N;

sum += 4.0 / (1.0 + x \* x);

}

return sum;

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

if (argc != 2) {

if (rank == 0) {

std::cerr << "Usage: " << argv[0] << " N" << std::endl;

}

MPI\_Finalize();

return 1;

}

int N = std::atoi(argv[1]);

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = (rank == size - 1) ? N : start + chunk\_size;

double local\_sum = calculate\_pi\_part(start, end, N);

double global\_sum = 0.0;

if (rank == 0) {

global\_sum = local\_sum;

for (int i = 1; i < size; i++) {

double received\_sum;

MPI\_Recv(&received\_sum, 1, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

global\_sum += received\_sum;

}

double pi = global\_sum / N;

std::cout.precision(15);

std::cout << "Calculated pi: " << pi << std::endl;

}

else {

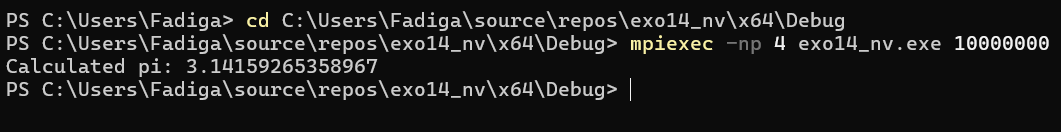
MPI\_Send(&local\_sum, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}



***Задание 25. Коллективные коммуникации: функции распределения и сбо-ра данных***

1. Изучите MPI-функции распределения и сбора блоков данных по про-цессам MPI\_Scatter и MPI\_Gather. Напишите программу, которая вычисляет произведение двух квадратных матриц 𝐴×𝐵=С размера 𝑛×𝑛. Используй-те формулу, приведенную в задании 9. Ввод данных и вывод результата должны осуществляться процессом с номером 0. Для распределения матриц *A* и *B* и сбора матрицы *С* используйте функций MPI\_Scatter и MPI\_Gather.

#include <iostream>

#include <vector>

#include <mpi.h>

void multiply\_matrices\_part(const std::vector<double>& A, const std::vector<double>& B, std::vector<double>& C\_part, int n, int rows\_per\_proc) {

for (int i = 0; i < rows\_per\_proc; ++i) {

for (int j = 0; j < n; ++j) {

C\_part[i \* n + j] = 0.0;

for (int k = 0; k < n; ++k) {

C\_part[i \* n + j] += A[i \* n + k] \* B[k \* n + j];

}

}

}

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int n;

std::vector<double> A, B, C;

if (rank == 0) {

std::cin >> n;

A.resize(n \* n);

B.resize(n \* n);

C.resize(n \* n);

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

std::cin >> A[i];

}

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

std::cin >> B[i];

}

}

// Broadcast matrix size n to all processes

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

// Broadcast matrix B to all processes

std::vector<double> B\_local(n \* n);

if (rank == 0) {

B\_local = B;

}

MPI\_Bcast(B\_local.data(), n \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Calculate how many rows each process will handle

int rows\_per\_proc = n / size;

int remainder = n % size;

std::vector<int> counts(size, rows\_per\_proc \* n);

std::vector<int> displs(size, 0);

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

counts[i] += n;

}

for (int i = 1; i < size; ++i) {

displs[i] = displs[i - 1] + counts[i - 1];

}

// Prepare local storage for A and C parts

std::vector<double> A\_local(counts[rank]);

std::vector<double> C\_local(counts[rank]);

// Scatter matrix A

MPI\_Scatterv(A.data(), counts.data(), displs.data(), MPI\_DOUBLE,

A\_local.data(), counts[rank], MPI\_DOUBLE,

0, MPI\_COMM\_WORLD);

// Multiply local parts

multiply\_matrices\_part(A\_local, B\_local, C\_local, n, counts[rank] / n);

// Gather results

MPI\_Gatherv(C\_local.data(), counts[rank], MPI\_DOUBLE,

C.data(), counts.data(), displs.data(), MPI\_DOUBLE,

0, MPI\_COMM\_WORLD);

// Output result

if (rank == 0) {

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

for (int j = 0; j < n; ++j) {

std::cout << C[i \* n + j] << " ";

}

std::cout << std::endl;

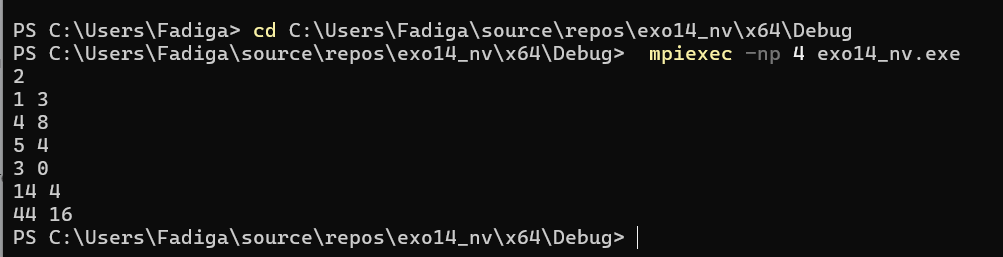
}

}

MPI\_Finalize();

return 0;

}



2\*. Перепишите программу, используя вместо функций MPI\_Scatter и MPI\_Gather функции коммуникации «точка-точка». Сравните эффективность выполнения программ с коллективными и точечными обменами.

**Входные данные:** целое число *n,* 1≤𝑛≤10, *n2* вещественных элементов матрицы *A* и *n2* вещественных элементов матрицы *B*.

1. **Выходные данные:** *n2* вещественных элементов матрицы *С*.

#include <iostream>

#include <vector>

#include <mpi.h>

void multiply\_matrices\_part(const std::vector<double>& A, const std::vector<double>& B, std::vector<double>& C\_part, int n, int rows\_per\_proc) {

for (int i = 0; i < rows\_per\_proc; ++i) {

for (int j = 0; j < n; ++j) {

C\_part[i \* n + j] = 0.0;

for (int k = 0; k < n; ++k) {

C\_part[i \* n + j] += A[i \* n + k] \* B[k \* n + j];

}

}

}

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int n;

std::vector<double> A, B, C;

if (rank == 0) {

std::cin >> n;

A.resize(n \* n);

B.resize(n \* n);

C.resize(n \* n);

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

std::cin >> A[i];

}

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

std::cin >> B[i];

}

}

// Broadcast matrix size n to all processes

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

// Broadcast matrix B to all processes

std::vector<double> B\_local(n \* n);

if (rank == 0) {

B\_local = B;

}

MPI\_Bcast(B\_local.data(), n \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Calculate how many rows each process will handle

int rows\_per\_proc = n / size;

int remainder = n % size;

std::vector<int> counts(size, rows\_per\_proc);

std::vector<int> displs(size, 0);

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

counts[i]++;

}

for (int i = 1; i < size; ++i) {

displs[i] = displs[i - 1] + counts[i - 1];

}

// Prepare local storage for A and C parts

std::vector<double> A\_local(counts[rank] \* n);

std::vector<double> C\_local(counts[rank] \* n);

// Send parts of A to other processes

if (rank == 0) {

for (int i = 1; i < size; ++i) {

MPI\_Send(A.data() + displs[i] \* n, counts[i] \* n, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

}

// Copy own part

std::copy(A.begin(), A.begin() + counts[0] \* n, A\_local.begin());

} else {

MPI\_Recv(A\_local.data(), counts[rank] \* n, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

// Multiply local parts

multiply\_matrices\_part(A\_local, B\_local, C\_local, n, counts[rank]);

// Gather results

if (rank == 0) {

// Copy own part

std::copy(C\_local.begin(), C\_local.end(), C.begin());

// Receive parts from other processes

for (int i = 1; i < size; ++i) {

MPI\_Recv(C.data() + displs[i] \* n, counts[i] \* n, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

} else {

MPI\_Send(C\_local.data(), counts[rank] \* n, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

// Output result

if (rank == 0) {

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

for (int j = 0; j < n; ++j) {

std::cout << C[i \* n + j] << " ";

}

std::cout << std::endl;

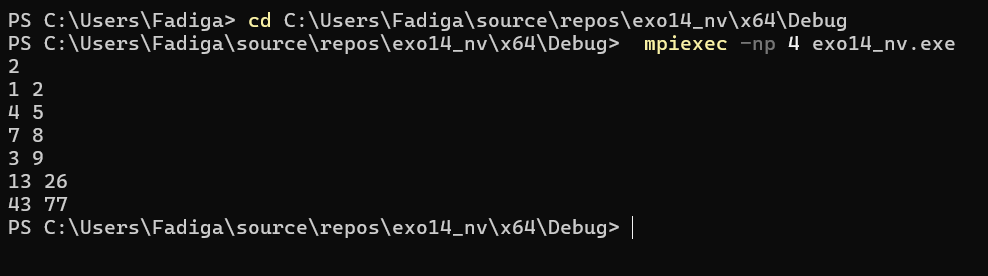
}

}

MPI\_Finalize();

return 0;

}



***Задание 26. Группы и коммуникаторы***

Напишите программу, в которой производится широковещательная рас-сылка сообщения с помощью функции MPI\_Bcast, но только по процессам с четным номером. Для рассылки сообщения создайте новый коммуникатор. Каждый процесс приложения должен выводить на экран «MPI\_COMM\_WORLD: <номер процесса в коммуникаторе MPI\_COMM\_WORLD> from <количество процессов в коммуникаторе MPI\_COMM\_WORLD>. New comm: <номер про-цесса в новом коммуникаторе> from <количество процессов в новом коммуникаторе>. Message = <сообщение>»

**Входные данные:** message – строка с сообщением, считываемым только процессом 0, количество символов в message от 1 до 10.

**Выходные данные:** строки вида «MPI\_COMM\_WORLD: <номер процесса в коммуникаторе MPI\_COMM\_WORLD> from <количество процессов в ком-муникаторе MPI\_COMM\_WORLD>. New comm: <номер процесса в новом коммуникаторе> from <количество процессов в новом коммуникато-ре>. Message = <сообщение>».

#include <mpi.h>

#include <iostream>

#include <cstring>

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

MPI\_Init(&argc, &argv);

int world\_rank, world\_size;

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

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

// Создаем группу процессов из MPI\_COMM\_WORLD

MPI\_Group world\_group;

MPI\_Comm\_group(MPI\_COMM\_WORLD, &world\_group);

// Выбираем процессы с четными номерами

int even\_count = (world\_size + 1) / 2;

int\* even\_ranks = new int[even\_count];

int idx = 0;

for (int i = 0; i < world\_size; ++i) {

if (i % 2 == 0) {

even\_ranks[idx++] = i;

}

}

// Создаем группу из четных процессов

MPI\_Group even\_group;

MPI\_Group\_incl(world\_group, even\_count, even\_ranks, &even\_group);

// Создаем новый коммуникатор из группы четных процессов

MPI\_Comm new\_comm;

MPI\_Comm\_create(MPI\_COMM\_WORLD, even\_group, &new\_comm);

// Буфер для сообщения (максимум 10 символов + 1 для '\0')

char message[11] = { 0 };

// Процесс 0 в MPI\_COMM\_WORLD считывает сообщение

if (world\_rank == 0) {

std::cout << "Enter message (1 to 10 characters): ";

std::cin.getline(message, 11);

}

// Если процесс входит в новый коммуникатор, выполняем широковещательную рассылку

if (new\_comm != MPI\_COMM\_NULL) {

int new\_rank, new\_size;

MPI\_Comm\_rank(new\_comm, &new\_rank);

MPI\_Comm\_size(new\_comm, &new\_size);

MPI\_Bcast(message, 11, MPI\_CHAR, 0, new\_comm);

std::cout << "MPI\_COMM\_WORLD: " << world\_rank << " from " << world\_size

<< ". New comm: " << new\_rank << " from " << new\_size

<< ". Message = " << message << std::endl;

}

else {

// Процессы, не входящие в новый коммуникатор

std::cout << "MPI\_COMM\_WORLD: " << world\_rank << " from " << world\_size

<< ". New comm: -1 from 0. Message = " << message << std::endl;

}

delete[] even\_ranks;

MPI\_Group\_free(&world\_group);

MPI\_Group\_free(&even\_group);

if (new\_comm != MPI\_COMM\_NULL) {

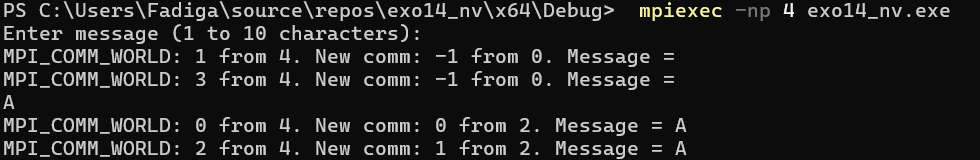
MPI\_Comm\_free(&new\_comm);

}

MPI\_Finalize();

return 0;

}



***Задание 27\*. MPI-2: динамическое создание процессов***

Напишите программу, в которой нулевым процессом динамически за-пускается еще n процессов. Каждый процесс в программе выводит сообще- ние в формате «I am <Номер процесса> process from <Количество про-цессов> processes! My parent is <Номер процесса родителя>».

**Входные данные:** целое число n – количество процессов, которые должны быть запущены динамически.

**Выходные данные:** строки вида «I am <Номер процесса> process from <Количество процессов> processes! My parent is <Номер процесса родителя>».

#include <iostream>

#include <mpi.h>

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

MPI\_Init(&argc, &argv);

int world\_rank, world\_size;

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

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

if (argc < 2) {

if (world\_rank == 0) {

std::cerr << "Usage: " << argv[0] << " n" << std::endl;

}

MPI\_Finalize();

return 1;

}

int n = std::atoi(argv[1]);

MPI\_Comm parent\_comm;

MPI\_Comm\_get\_parent(&parent\_comm);

if (parent\_comm == MPI\_COMM\_NULL) {

// Main process (spawner)

MPI\_Comm intercomm;

MPI\_Comm\_spawn(argv[0], argv + 1, n, MPI\_INFO\_NULL,

0, MPI\_COMM\_WORLD, &intercomm, MPI\_ERRCODES\_IGNORE);

std::cout << "I am " << world\_rank << " process from "

<< world\_size << " processes! My parent is -1 (root)" << std::endl;

MPI\_Comm\_free(&intercomm);

}

else {

// Spawned process

int parent\_size;

MPI\_Comm\_remote\_size(parent\_comm, &parent\_size);

std::cout << "I am " << world\_rank << " process from "

<< "1 processes! My parent is 0" << std::endl;

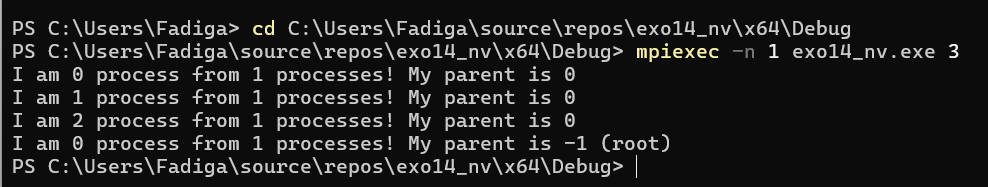
MPI\_Comm\_free(&parent\_comm);

}

MPI\_Finalize();

return 0;

}



***Задание 28\*. MPI-2: односторонние коммуникации***

Реализуйте программу вычисления числа 𝜋 (см. задание 24), используй-те функции односторонней коммуникации для обмена данными между про-цессами.

**Входные данные:** одно целое число N (точность вычисления).

**Выходные данные:** одно вещественное число pi.

#include <iostream>

#include <iomanip>

#include <mpi.h>

#include <omp.h>

double calculate\_pi\_part(int start, int end, double step) {

double sum = 0.0;

#pragma omp parallel for reduction(+:sum)

for (int i = start; i < end; i++) {

double x = (i + 0.5) \* step;

sum += 4.0 / (1.0 + x \* x);

}

return sum;

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

if (argc < 2) {

if (rank == 0) {

std::cerr << "Usage: " << argv[0] << " N" << std::endl;

}

MPI\_Finalize();

return 1;

}

int N = std::atoi(argv[1]);

double step = 1.0 / N;

// Распределение работы между процессами

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = (rank == size - 1) ? N : start + chunk\_size;

double local\_sum = calculate\_pi\_part(start, end, step);

double global\_sum = 0.0;

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

double pi = global\_sum \* step;

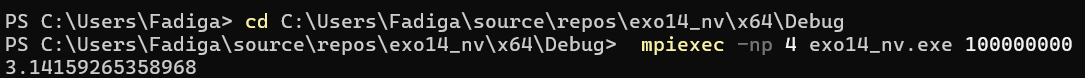
std::cout << std::setprecision(15) << pi << std::endl;

}

MPI\_Finalize();

return 0;

}



***Задание 29. Исследование масштабируемости MPI-программ***

Проведите серию экспериментов на суперкомпьютере по исследованию масштабируемости OpenMP-программ. Заполните следующую таблицу:

#include <iostream>

#include <iomanip>

#include <omp.h>

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

if (argc < 2) {

std::cerr << "Usage: " << argv[0] << " N [threads]" << std::endl;

return 1;

}

long N = std::atol(argv[1]);

int threads = argc > 2 ? std::atoi(argv[2]) : 1;

omp\_set\_num\_threads(threads);

double step = 1.0 / N;

double sum = 0.0;

double start\_time = omp\_get\_wtime();

#pragma omp parallel for reduction(+:sum)

for (long i = 0; i < N; i++) {

double x = (i + 0.5) \* step;

sum += 4.0 / (1.0 + x \* x);

}

double pi = sum \* step;

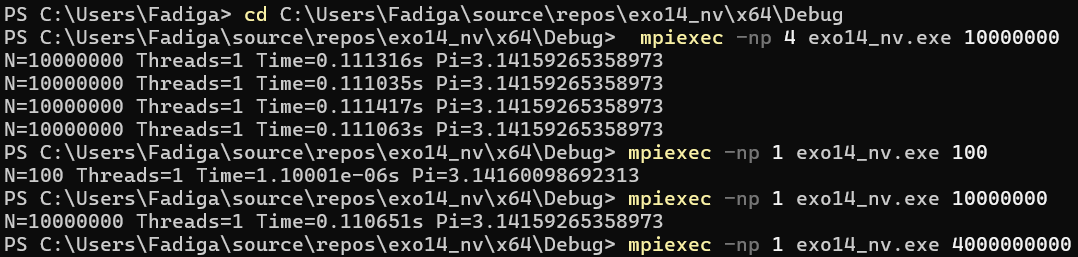
double time = omp\_get\_wtime() - start\_time;

std::cout << "N=" << N << " Threads=" << threads

<< " Time=" << time << "s Pi=" << std::setprecision(15) << pi << std::endl;

return 0;

}



***Задание 30. Проект в среде Visual Studio 2010 с поддержкой MPI и OpenMP***

Создайте проект в среде Visual Studio 2010 с минимальным кодом с под-держкой MPI и OpenMP.

#include <iostream>

#include <mpi.h>

#include <omp.h>

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

// Инициализация MPI

MPI\_Init(&argc, &argv);

int rank, size;

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

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

// Получаем количество доступных потоков OpenMP

int threads = omp\_get\_max\_threads();

// Выводим информацию о процессе

std::cout << "MPI process " << rank << " of " << size

<< " (using " << threads << " OpenMP threads)" << std::endl;

// Простая параллельная секция OpenMP

#pragma omp parallel

{

int thread\_id = omp\_get\_thread\_num();

#pragma omp critical

{

std::cout << " Thread " << thread\_id << " from process " << rank << std::endl;

}

}

// Завершение MPI

MPI\_Finalize();

return 0;

}



***Задание 31. Программа «I am»***

Напишите программу, в которой в каждом процессе создается n нитей. Каждая нить должна выводить на экран свой номер, номер процесса-родителя и общее количество нитей во всех процессах в следующем форма-те: I am <Номер нити> thread from <Номер родительского процесса> process. Number of hybrid threads = <Количество нитей \* Количе-ство процессов>.

**Входные данные:** целое число n – количество нитей, которые должны быть запущены.

**Выходные данные:** строки вида «I am <Номер нити> thread from <Номер родительского процесса> process. Number of hybrid threads = <Ко-личество нитей \* Количество процессов>».

#include <iostream>

#include <mpi.h>

#include <omp.h>

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

// Инициализация MPI

MPI\_Init(&argc, &argv);

int process\_rank, process\_count;

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

MPI\_Comm\_size(MPI\_COMM\_WORLD, &process\_count);

// Проверка входных аргументов

if (argc < 2) {

if (process\_rank == 0) {

std::cerr << "Usage: " << argv[0] << " n" << std::endl;

}

MPI\_Finalize();

return 1;

}

int n = std::atoi(argv[1]); // Количество нитей в каждом процессе

int total\_hybrid\_threads = n \* process\_count;

// Установка количества нитей OpenMP

omp\_set\_num\_threads(n);

// Параллельная секция OpenMP

#pragma omp parallel

{

int thread\_num = omp\_get\_thread\_num();

// Блокировка вывода для корректного отображения

#pragma omp critical

{

std::cout << "I am " << thread\_num << " thread from "

<< process\_rank << " process. Number of hybrid threads = "

<< total\_hybrid\_threads << std::endl;

}

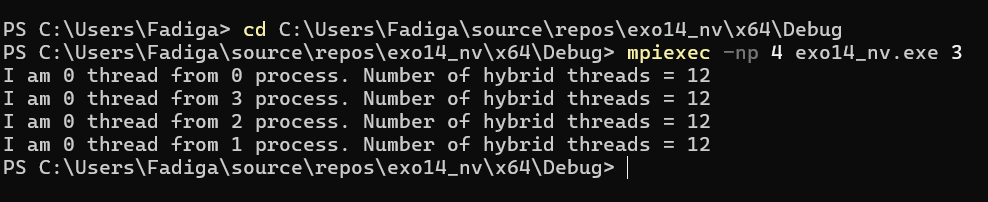
}

// Завершение работы MPI

MPI\_Finalize();

return 0;

}



***Задание 32. Программа «Число*** 𝜋»

Реализуйте программу вычисления числа 𝜋 (см. задание 8) с использо-ванием MPI+OpenMP.

**Входные данные:** одно целое число N (точность вычисления).

**Выходные данные:** одно вещественное число pi.

#include <iostream>

#include <iomanip>

#include <mpi.h>

#include <omp.h>

double calculate\_pi\_part(int start, int end, double step) {

double sum = 0.0;

#pragma omp parallel for reduction(+:sum)

for (int i = start; i < end; i++) {

double x = (i + 0.5) \* step;

sum += 4.0 / (1.0 + x \* x);

}

return sum;

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

if (argc < 2) {

if (rank == 0) {

std::cerr << "Usage: " << argv[0] << " N" << std::endl;

}

MPI\_Finalize();

return 1;

}

int N = std::atoi(argv[1]);

double step = 1.0 / N;

// Распределение работы между процессами

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = (rank == size - 1) ? N : start + chunk\_size;

double local\_sum = calculate\_pi\_part(start, end, step);

double global\_sum = 0.0;

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

double pi = global\_sum \* step;

std::cout << std::setprecision(15) << pi << std::endl;

}

MPI\_Finalize();

return 0;

}

