**Технология программирования MPI**

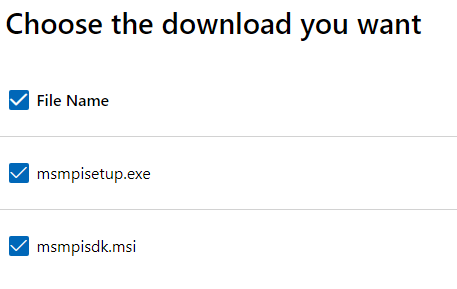
**Задание для самостоятельной работы**

Решить задания с 14 - 32

**Ход работы**

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

Создайте проект в среде Visual Studio 2010 с поддержкой MPI

1. Скачать и установить MPI  
   
2. Создайте консольный проект в Visual Studio.
3. Настройка поддержки MPI:

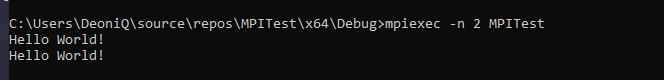
Добавьте путь к заголовочным файлам MPICH2:

Properties → C/C++ → General → Additional Include Directories: "C:\Program Files (x86)\Microsoft SDKs\MPI\Include" и “C:\Program Files (x86)\Microsoft SDKs\MPI\Include\x64 (или x86) “

1. Добавьте путь к библиотекам:

Properties → Linker → General → Additional Library Directories: "C:\Program Files (x86)\Microsoft SDKs\MPI\Lib".

1. Указать библиотеку:
2. Properties → Linker → Input → Additional Dependencies: mpi.lib.



**Задание 15. Программа «I am!»**

#include <mpi.h>

#include <stdio.h>

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

MPI\_Init(&argc, &argv);

int rank, size;

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

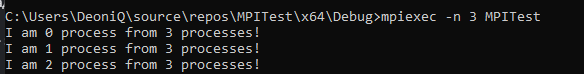
MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

printf("I am %d process from %d processes!\n", rank, size);

MPI\_Finalize();

return 0;

}



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

#include <mpi.h>

#include <stdio.h>

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) {

printf("%d processes.\n", size);

}

else if (rank % 2 == 0) {

printf("I am %d process: SECOND!\n", rank);

}

else {

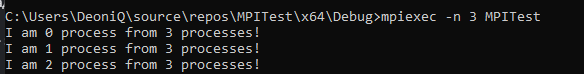
printf("I am %d process: FIRST!\n", rank);

}

MPI\_Finalize();

return 0;

}



**Задание 17. Блокирующие обмены «точка-точка»**

#include <mpi.h>

#include <stdio.h>

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

MPI\_Init(&argc, &argv);

int rank;

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

if (rank == 0) {

int message = 123;

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

}

else if (rank == 1) {

int message;

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

printf("Received message: %d\n", message);

}

MPI\_Finalize();

return 0;

}



**Задание 18. Схема «эстафетная палочка»**

#include <mpi.h>

#include <stdio.h>

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) {

message = 0;

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

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

}

else {

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

message += 1;

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

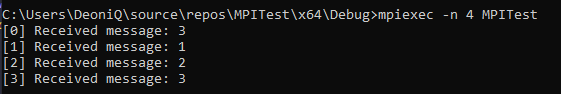
}

printf("[%d] Received message: %d\n", rank, message);

MPI\_Finalize();

return 0;

}



**Задание 19. Схема «мастер-рабочие»**

#include <mpi.h>

#include <stdio.h>

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) {

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

int message;

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

printf("receive message '%d' from %d\n", message, i);

}

}

else {

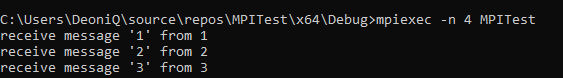
MPI\_Send(&rank, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}



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

#include <mpi.h>

#include <stdio.h>

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

MPI\_Init(&argc, &argv);

int rank;

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

if (rank == 0) {

int message = 45;

MPI\_Request request;

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

MPI\_Wait(&request, MPI\_STATUS\_IGNORE);

}

else if (rank == 1) {

int message;

MPI\_Request request;

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

MPI\_Wait(&request, MPI\_STATUS\_IGNORE);

printf("receive message '%d'\n", message);

}

MPI\_Finalize();

return 0;

}



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

#include <mpi.h>

#include <stdio.h>

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\_data = rank;

int recv\_data;

MPI\_Request requests[2];

// Отправляем данные следующему процессу

MPI\_Isend(&send\_data, 1, MPI\_INT, (rank + 1) % size, 0, MPI\_COMM\_WORLD, &requests[0]);

// Получаем данные от предыдущего процесса

MPI\_Irecv(&recv\_data, 1, MPI\_INT, (rank - 1 + size) % size, 0, MPI\_COMM\_WORLD, &requests[1]);

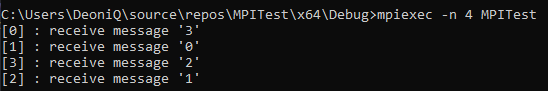
MPI\_Waitall(2, requests, MPI\_STATUSES\_IGNORE);

printf("[%d] : receive message '%d'\n", rank, recv\_data);

MPI\_Finalize();

return 0;

}



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

#include <mpi.h>

#include <stdio.h>

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\_data = new int[size];

int\* recv\_data = new int[size];

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

send\_data[i] = rank;

}

MPI\_Alltoall(send\_data, 1, MPI\_INT, recv\_data, 1, MPI\_INT, MPI\_COMM\_WORLD);

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

if (i != rank) {

printf("[%d] Received message: %d from %d\n", rank, recv\_data[i], i);

}

}

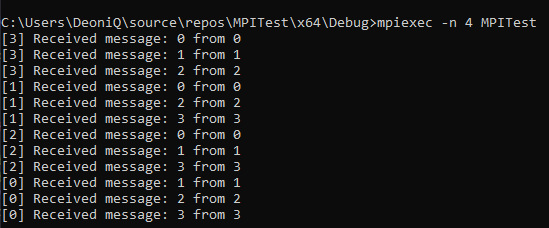
delete[] send\_data;

delete[] recv\_data;

MPI\_Finalize();

return 0;

}



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

1. С использованием MPI\_Bcast

#include <mpi.h>

#include <iostream>

#include <vector>

#include <cstring>

#include <ctime>

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::string str;

std::vector<int> counts(26, 0);

double start\_time, end\_time;

if (rank == 0) {

// Ввод данных

std::cout << "Введите длину строки n: ";

std::cin >> n;

std::cout << "Введите строку: ";

std::cin >> str;

start\_time = MPI\_Wtime(); // Начало измерения времени

}

// Рассылка длины строки

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

if (rank != 0) {

str.resize(n);

}

// Рассылка строки

MPI\_Bcast(&str[0], n, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

// Локальный подсчет символов

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

if (rank == i % size) {

counts[str[i] - 'a']++;

}

}

// Сбор результатов

std::vector<int> global\_counts(26, 0);

MPI\_Reduce(counts.data(), global\_counts.data(), 26, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

end\_time = MPI\_Wtime(); // Конец измерения времени

// Вывод результатов

for (int i = 0; i < 26; i++) {

if (global\_counts[i] > 0) {

std::cout << static\_cast<char>('a' + i) << " = " << global\_counts[i] << std::endl;

}

}

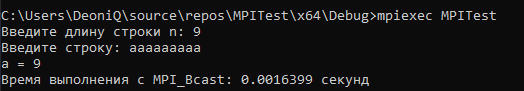
std::cout << "Время выполнения с MPI\_Bcast: " << (end\_time - start\_time) << " секунд" << std::endl;

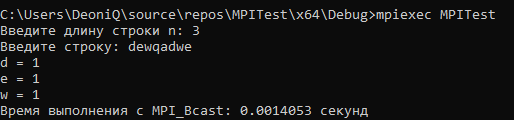
}

MPI\_Finalize();

return 0;

}





1. C использованием коммуникаций "точка-точка"

#include <mpi.h>

#include <iostream>

#include <vector>

#include <cstring>

#include <ctime>

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::string str;

std::vector<int> counts(26, 0);

double start\_time, end\_time;

if (rank == 0) {

// Ввод данных

std::cout << "Введите длину строки n: ";

std::cin >> n;

std::cout << "Введите строку: ";

std::cin >> str;

start\_time = MPI\_Wtime(); // Начало измерения времени

// Рассылка длины строки

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

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

}

// Рассылка строки

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

MPI\_Send(str.data(), n, MPI\_CHAR, i, 0, MPI\_COMM\_WORLD);

}

}

else {

// Получение длины строки

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

str.resize(n);

// Получение строки

MPI\_Recv(&str[0], n, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

// Локальный подсчет символов

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

if (rank == i % size) {

counts[str[i] - 'a']++;

}

}

// Сбор результатов

std::vector<int> global\_counts(26, 0);

MPI\_Reduce(counts.data(), global\_counts.data(), 26, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

end\_time = MPI\_Wtime(); // Конец измерения времени

// Вывод результатов

for (int i = 0; i < 26; i++) {

if (global\_counts[i] > 0) {

std::cout << static\_cast<char>('a' + i) << " = " << global\_counts[i] << std::endl;

}

}

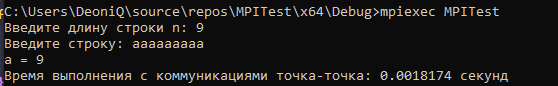
std::cout << "Время выполнения с коммуникациями точка-точка: " << (end\_time - start\_time) << " секунд" << std::endl;

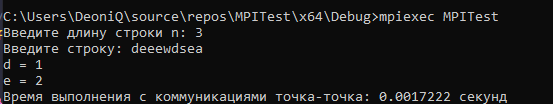
}

MPI\_Finalize();

return 0;

}





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

1. C использованием MPI\_Reduce

#include <mpi.h>

#include <iostream>

#include <cmath>

#include <chrono>

double calculatePiPartial(int start, int end, double step) {

double sum = 0.0;

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);

int N = 1000000; // Точность вычисления

double step = 1.0 / N;

int chunk = N / size;

int start = rank \* chunk;

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

auto startTime = std::chrono::high\_resolution\_clock::now(); // Начало измерения времени

double partialSum = calculatePiPartial(start, end, step);

double totalSum;

MPI\_Reduce(&partialSum, &totalSum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

auto endTime = std::chrono::high\_resolution\_clock::now(); // Конец измерения времени

if (rank == 0) {

std::chrono::duration<double> elapsed = endTime - startTime;

std::cout << "Pi: " << totalSum \* step << std::endl;

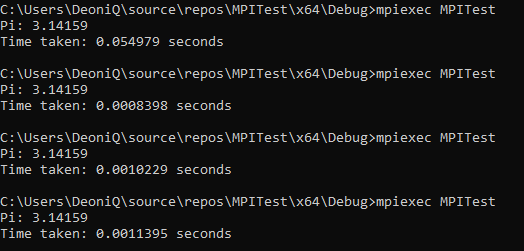
std::cout << "Time taken: " << elapsed.count() << " seconds" << std::endl;

}

MPI\_Finalize();

return 0;

}



1. С использованием точечных коммуникаций

#include <mpi.h>

#include <iostream>

#include <cmath>

#include <chrono>

double calculatePiPartial(int start, int end, double step) {

double sum = 0.0;

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);

int N = 1000000; // Точность вычисления

double step = 1.0 / N;

int chunk = N / size;

int start = rank \* chunk;

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

auto startTime = std::chrono::high\_resolution\_clock::now(); // Начало измерения времени

double partialSum = calculatePiPartial(start, end, step);

if (rank != 0) {

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

}

else {

double totalSum = partialSum;

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

double tempSum;

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

totalSum += tempSum;

}

auto endTime = std::chrono::high\_resolution\_clock::now(); // Конец измерения времени

std::chrono::duration<double> elapsed = endTime - startTime;

std::cout << "Pi: " << totalSum \* step << std::endl;

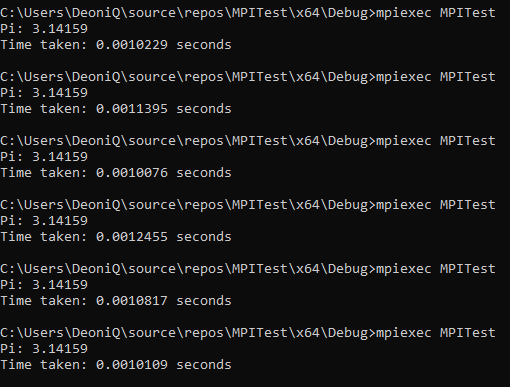
std::cout << "Time taken: " << elapsed.count() << " seconds" << std::endl;

}

MPI\_Finalize();

return 0;

}



**Задание 25. Коллективные коммуникации: функции распределения и сбора данных**

1. Использование MPI\_Scatter и MPI\_Gather

#include <iostream>

#include <mpi.h>

#include <chrono>

void matrixMultiply(double\* A, double\* B, double\* C, int n, int local\_n) {

for (int i = 0; i < local\_n; ++i) {

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

C[i \* n + j] = 0;

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

C[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;

double\* A = nullptr, \* B = nullptr, \* C = nullptr;

double\* local\_A = nullptr, \* local\_C = nullptr;

if (rank == 0) {

std::cout << "Enter the matrix size n (1 <= n <= 10): ";

std::cin >> n;

A = new double[n \* n];

B = new double[n \* n];

C = new double[n \* n];

std::cout << "Enter the elements of the matrix A (by lines): ";

for (int i = 0; i < n \* n; ++i) std::cin >> A[i];

std::cout << "Enter the elements of the matrix B (by lines): ";

for (int i = 0; i < n \* n; ++i) std::cin >> B[i];

}

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

int local\_n = n / size;

local\_A = new double[local\_n \* n];

local\_C = new double[local\_n \* n];

double start\_time = MPI\_Wtime(); // Начало измерения времени

MPI\_Scatter(A, local\_n \* n, MPI\_DOUBLE, local\_A, local\_n \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

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

matrixMultiply(local\_A, B, local\_C, n, local\_n);

MPI\_Gather(local\_C, local\_n \* n, MPI\_DOUBLE, C, local\_n \* n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

double end\_time = MPI\_Wtime(); // Конец измерения времени

if (rank == 0) {

std::cout << "Result of matrix multiplication C = A \* B:" << std::endl;

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

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

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

}

std::cout << std::endl;

}

std::cout << "Result time: " << end\_time - start\_time << " sec " << std::endl;

delete[] A;

delete[] B;

delete[] C;

}

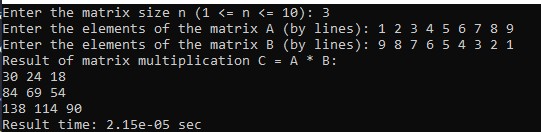
delete[] local\_A;

delete[] local\_C;

MPI\_Finalize();

return 0;

}



1. Использование точечных коммуникаций с измерением времени

#include <iostream>

#include <mpi.h>

#include <chrono>

void matrixMultiply(double\* A, double\* B, double\* C, int n, int local\_n) {

for (int i = 0; i < local\_n; ++i) {

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

C[i \* n + j] = 0;

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

C[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;

double\* A = nullptr, \* B = nullptr, \* C = nullptr;

double\* local\_A = nullptr, \* local\_C = nullptr;

if (rank == 0) {

std::cout << "Enter the matrix size n (1 <= n <= 10): ";

std::cin >> n;

A = new double[n \* n];

B = new double[n \* n];

C = new double[n \* n];

std::cout << "Enter the elements of the matrix A (by lines): ";

for (int i = 0; i < n \* n; ++i) std::cin >> A[i];

std::cout << "Enter the elements of the matrix B (by lines): ";

for (int i = 0; i < n \* n; ++i) std::cin >> B[i];

}

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

int local\_n = n / size;

local\_A = new double[local\_n \* n];

local\_C = new double[local\_n \* n];

double start\_time = MPI\_Wtime(); // Начало измерения времени

if (rank == 0) {

// Отправка данных другим процессам

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

MPI\_Send(A + dest \* local\_n \* n, local\_n \* n, MPI\_DOUBLE, dest, 0, MPI\_COMM\_WORLD);

}

// Копирование данных для процесса 0

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

local\_A[i] = A[i];

}

}

else {

// Получение данных от процесса 0

MPI\_Recv(local\_A, local\_n \* n, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

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

matrixMultiply(local\_A, B, local\_C, n, local\_n);

if (rank == 0) {

// Сбор данных от других процессов

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

MPI\_Recv(C + src \* local\_n \* n, local\_n \* n, MPI\_DOUBLE, src, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

// Копирование данных от процесса 0

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

C[i] = local\_C[i];

}

}

else {

// Отправка данных процессу 0

MPI\_Send(local\_C, local\_n \* n, MPI\_DOUBLE, 0, 1, MPI\_COMM\_WORLD);

}

double end\_time = MPI\_Wtime(); // Конец измерения времени

if (rank == 0) {

std::cout << "Result of matrix multiplication C = A \* B:" << std::endl;

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

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

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

}

std::cout << std::endl;

}

std::cout << "Result time: " << end\_time - start\_time << " sec " << std::endl;

delete[] A;

delete[] B;

delete[] C;

}

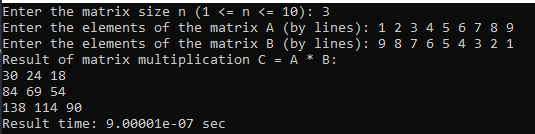
delete[] local\_A;

delete[] local\_C;

MPI\_Finalize();

return 0;

}



**Задание 26. Группы и коммуникаторы**

#include <mpi.h>

#include <iostream>

#include <string>

#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);

int color = (world\_rank % 2 == 0) ? 0 : MPI\_UNDEFINED;

MPI\_Comm even\_comm;

MPI\_Comm\_split(MPI\_COMM\_WORLD, color, world\_rank, &even\_comm);

char msg\_buffer[11] = { 0 };

if (world\_rank == 0) {

if (argc < 2) {

std::cerr << "-n <num\_procs> ./program <message>\n";

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

}

std::string input\_msg = argv[1];

if (input\_msg.size() < 1 || input\_msg.size() > 10) {

std::cerr << "Message length must be between 1 and 10 characters.\n";

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

}

strncpy\_s(msg\_buffer, sizeof(msg\_buffer), input\_msg.c\_str(), 10);

msg\_buffer[10] = '\0';

}

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

int even\_rank, even\_size;

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

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

MPI\_Bcast(msg\_buffer, 10, MPI\_CHAR, 0, even\_comm);

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

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

<< ". Message = " << msg\_buffer << std::endl;

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

}

else {

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

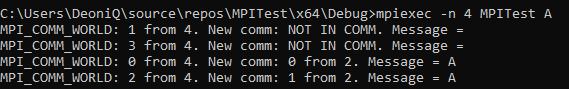
<< ". New comm: NOT IN COMM. Message = " << std::endl;

}

MPI\_Finalize();

return 0;

}



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

#include <mpi.h>

#include <iostream>

#include <vector>

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);

MPI\_Comm parent\_comm;

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

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

int n = 3;

if (argc > 1) n = atoi(argv[1]);

char\* command = argv[0];

char\* args[] = { command, nullptr };

std::vector<int> errcodes(n);

MPI\_Comm child\_intercomm;

MPI\_Comm\_spawn(command, args, n, MPI\_INFO\_NULL, 0, MPI\_COMM\_WORLD, &child\_intercomm, errcodes.data());

int child\_remote\_size;

MPI\_Comm\_remote\_size(child\_intercomm, &child\_remote\_size);

std::cout << "Parent process: " << world\_rank << " from " << world\_size

<< " processes. Created " << child\_remote\_size << " child processes." << std::endl;

MPI\_Comm\_free(&child\_intercomm);

}

else {

// Дочерний процесс

int child\_rank, child\_size;

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

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

int parent\_rank\_in\_parent\_comm;

MPI\_Comm\_rank(parent\_comm, &parent\_rank\_in\_parent\_comm);

std::cout << "I am " << child\_rank << " process from " << child\_size

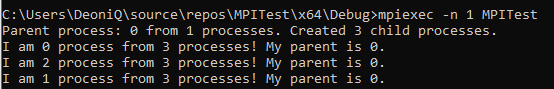
<< " processes! My parent is 0." << std::endl;

}

MPI\_Finalize();

return 0;

}



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

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <mpi.h>

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

int rank, size;

int N;

MPI\_Win win;

int\* total\_hits = NULL;

MPI\_Init(&argc, &argv);

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

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

if (rank == 0) {

if (argc < 2) {

printf("Usage: %s N\n", argv[0]);

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

}

N = atoi(argv[1]);

}

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

int local\_n = N / size;

int remainder = N % size;

if (rank < remainder) local\_n++;

srand(time(NULL) ^ rank);

int local\_hits = 0;

for (int i = 0; i < local\_n; i++) {

double x = (double)rand() / RAND\_MAX;

double y = (double)rand() / RAND\_MAX;

if (x \* x + y \* y <= 1.0) local\_hits++;

}

if (rank == 0) {

total\_hits = (int\*)malloc(sizeof(int));

\*total\_hits = 0;

}

MPI\_Win\_create(total\_hits, (rank == 0) ? sizeof(int) : 0,

sizeof(int), MPI\_INFO\_NULL, MPI\_COMM\_WORLD, &win);

MPI\_Win\_fence(0, win);

if (rank != 0) {

MPI\_Accumulate(&local\_hits, 1, MPI\_INT, 0, 0, 1, MPI\_INT,

MPI\_SUM, win);

}

else {

\*total\_hits += local\_hits;

}

MPI\_Win\_fence(0, win);

if (rank == 0) {

double pi = 4.0 \* (\*total\_hits) / N;

printf("Calculated pi = %.6f\n", pi);

}

MPI\_Win\_free(&win);

if (rank == 0) free(total\_hits);

MPI\_Finalize();

return 0;

}



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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Программа | Параметр N | Кол-во нитей | Время (сек) |
| 57 | «Число Pi» | 100 | 1 | 0.000002 |
| 58 | 10 000 000 | 1 | 0.028998 |
| 59 | 4 000 000 000 | 1 | 10.998178 |
| 60 | 100 | 2 | 0.000002 |
| 61 | 10 000 000 | 2 | 0.027573 |
| 62 | 4 000 000 000 | 2 | 10.953461 |
| 63 | 100 | 4 | 0.000002 |
| 64 | 10 000 000 | 4 | 0.027171 |
| 65 | 4 000 000 000 | 4 | 10.974918 |
| 66 | 100 | 6 | 0.000002 |
| 67 | 10 000 000 | 6 | 0.026935 |
| 68 | 4 000 000 000 | 6 | 10.967320 |
| 69 | 100 | 8 | 0.000002 |
| 70 | 10 000 000 | 8 | 0.027117 |
| 71 | 4 000 000 000 | 8 | 10.930862 |
| 72 | 100 | 10 | 0.000002 |
| 73 | 10 000 000 | 10 | 0.027508 |
| 74 | 4 000 000 000 | 10 | 10.940098 |
| 75 | 100 | 12 | 0.000002 |
| 76 | 10 000 000 | 12 | 0.027303 |
| 77 | 4 000 000 000 | 12 | 10.951109 |

#include <iostream>

#include <cmath>

#include <chrono>

#include <omp.h>

#include <cstdlib>

#include <iomanip>

double calculatePiPartial(long long int start, long long int end, double step) {

double sum = 0.0;

#pragma omp parallel for reduction(+:sum)

for (long long 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) {

if (argc != 3) {

std::cerr << "Usage: ./program <N> <num\_threads>" << std::endl;

return 1;

}

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

int threads = std::atoi(argv[2]);

omp\_set\_num\_threads(threads);

double step = 1.0 / N;

long long int start = 0;

long long int end = N;

auto startTime = std::chrono::high\_resolution\_clock::now();

double partialSum = calculatePiPartial(start, end, step);

auto endTime = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = endTime - startTime;

double piValue = partialSum \* step;

std::cout << "N: " << N << ", Threads: " << threads << ", Pi: " << std::fixed << std::setprecision(10) << piValue

<< ", Time: " << std::fixed << std::setprecision(6) << elapsed.count() << " seconds" << std::endl;

return 0;

}

**Задание 30. Проект с поддержкой MPI + OpenMP**

1. Добавьте поддержку OpenMP:  
   Properties → C/C++ → Language → OpenMP Support: Yes (/openmp).

**Задание 31: Программа "I am"**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <omp.h>

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

{

int num\_threads = 4;

int rank, size;

// Initialize MPI

MPI\_Init(&argc, &argv);

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

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

if (argc > 1) {

num\_threads = atoi(argv[1]);

if (num\_threads <= 0) num\_threads = 4;

}

omp\_set\_num\_threads(num\_threads);

#pragma omp parallel

{

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

int total\_hybrid\_threads = num\_threads \* size;

#pragma omp critical

{

printf("I am %d thread from %d process. Number of hybrid threads = %d\n",

thread\_id, rank, total\_hybrid\_threads);

}

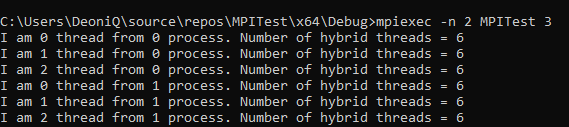
}

MPI\_Finalize();

return 0;

}

Если запустить с 2 процессами и 3 нитями



**Задание 32. Программа «Число 𝜋»**

#include <iostream>

#include <iomanip>

#include <cmath>

#include <mpi.h>

#include <omp.h>

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

MPI\_Init(&argc, &argv);

int world\_size, world\_rank;

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

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

if (argc != 2) {

if (world\_rank == 0) {

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

}

MPI\_Finalize();

return 1;

}

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

if (N <= 0) {

if (world\_rank == 0) {

std::cerr << "N must be a positive integer" << std::endl;

}

MPI\_Finalize();

return 1;

}

long long chunk\_size = N / world\_size;

long long remainder = N % world\_size;

long long start = world\_rank \* chunk\_size + std::min(world\_rank, (int)remainder);

long long end = start + chunk\_size + (world\_rank < remainder ? 1 : 0);

double local\_sum = 0.0;

const double inv\_N = 1.0 / N;

#pragma omp parallel for reduction(+:local\_sum)

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

double x = (i + 0.5) \* inv\_N;

local\_sum += 4.0 / (1.0 + x \* x);

}

double global\_sum = 0.0;

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

if (world\_rank == 0) {

double pi = global\_sum \* inv\_N;

std::cout << std::setprecision(8) << std::fixed << pi << std::endl;

}

MPI\_Finalize();

return 0;

}

