15

#include <mpi.h>

#include <stdio.h>

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

int ProcRank, ProcNum;

MPI\_Init(&argc, &argv);

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

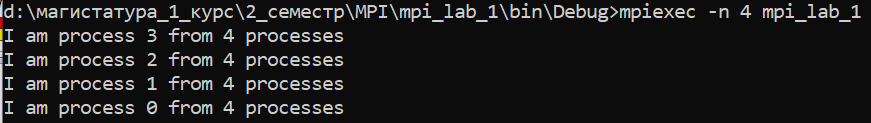
MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);

printf("I am process %d from %d processes\n", ProcRank, ProcNum);

MPI\_Finalize();

return 0;

}



16

#include <mpi.h>

#include <iostream>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

if (ProcRank == 0) {

// Процесс с номером 0 выводит общее количество процессов

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

} else {

// Остальные процессы выводят сообщения в зависимости от четности их номера

if (ProcRank % 2 == 0) {

// Процессы с четным номером выводят "SECOND!"

std::cout << "I am " << ProcRank << ": SECOND!" << std::endl;

} else {

// Процессы с нечетным номером выводят "FIRST!"

std::cout << "I am " << ProcRank << ": FIRST!" << std::endl;

}

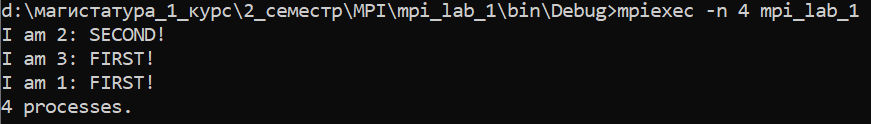
}

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



17

#include <mpi.h>

#include <iostream>

#include <cstring>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

if (ProcNum < 2) {

// Программа требует как минимум два процесса

std::cerr << "Программа требует как минимум два процесса." << std::endl;

MPI\_Finalize();

return 1;

}

if (ProcRank == 0) {

// Процесс 0 отправляет сообщение процессу 1

const char\* message = "Hello from process 0!";

MPI\_Send(message, strlen(message) + 1, MPI\_CHAR, 1, 0, MPI\_COMM\_WORLD);

} else if (ProcRank == 1) {

// Процесс 1 получает сообщение от процесса 0

char received\_message[100];

MPI\_Status status;

MPI\_Recv(received\_message, 100, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, &status);

// Вывод полученного сообщения

std::cout << "Receive message: '" << received\_message << "'" << std::endl;

}

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



18

#include <mpi.h>

#include <iostream>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

int message = 0; // Переменная для хранения сообщения

if (ProcRank == 0) {

// Процесс 0 начинает передачу с отправки своего номера

message = ProcRank;

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

std::cout << "Process " << ProcRank << " sent message: " << message << std::endl;

} else {

// Остальные процессы получают сообщение от предыдущего процесса

MPI\_Status status;

MPI\_Recv(&message, 1, MPI\_INT, ProcRank - 1, 0, MPI\_COMM\_WORLD, &status);

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

// Инкрементируем сообщение

message++;

if (ProcRank < ProcNum - 1) {

// Если это не последний процесс, отправляем сообщение следующему

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

std::cout << "Process " << ProcRank << " sent message: " << message << std::endl;

} else {

// Если это последний процесс, выводим окончательное значение

std::cout << "Final message received by process " << ProcRank << ": " << message << std::endl;

}

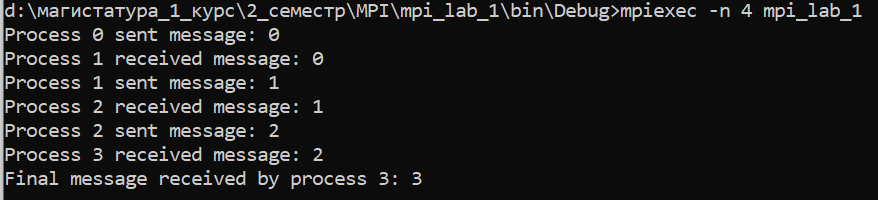
}

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



19

#include <mpi.h>

#include <iostream>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

if (ProcRank == 0) {

// Процесс 0 (master) принимает сообщения от всех остальных процессов

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

int received\_message;

MPI\_Status status;

MPI\_Recv(&received\_message, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, &status);

std::cout << "Receive message: '" << received\_message << "' from process " << i << std::endl;

}

} else {

// Остальные процессы (slave) отправляют свои номера процессу 0

int message = ProcRank;

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

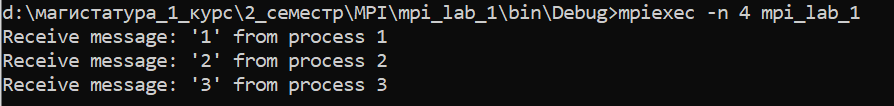
}

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



20

#include <mpi.h>

#include <iostream>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

if (ProcNum < 2) {

// Программа требует как минимум два процесса

std::cerr << "Программа требует как минимум два процесса." << std::endl;

MPI\_Finalize();

return 1;

}

if (ProcRank == 0) {

// Процесс 0 отправляет сообщение процессу 1

int message = 42; // Сообщение для отправки

MPI\_Request send\_request;

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

// Ожидание завершения отправки

MPI\_Status send\_status;

MPI\_Wait(&send\_request, &send\_status);

std::cout << "Process 0 sent message: " << message << std::endl;

} else if (ProcRank == 1) {

// Процесс 1 получает сообщение от процесса 0

int received\_message;

MPI\_Request recv\_request;

MPI\_Irecv(&received\_message, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &recv\_request);

// Ожидание завершения приема

MPI\_Status recv\_status;

MPI\_Wait(&recv\_request, &recv\_status);

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

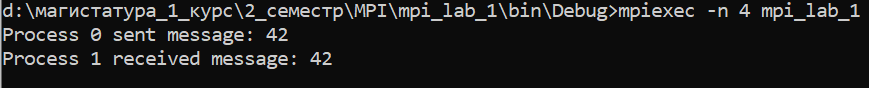
}

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



21

#include <mpi.h>

#include <iostream>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

int send\_message = ProcRank; // Сообщение для отправки

int recv\_message; // Буфер для приема сообщения

// Определение рангов для отправки и приема

int send\_to = (ProcRank + 1) % ProcNum; // Следующий процесс

int recv\_from = (ProcRank - 1 + ProcNum) % ProcNum; // Предыдущий процесс

// Одновременная отправка и прием сообщения

MPI\_Sendrecv(&send\_message, 1, MPI\_INT, send\_to, 0,

&recv\_message, 1, MPI\_INT, recv\_from, 0,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

// Вывод результата

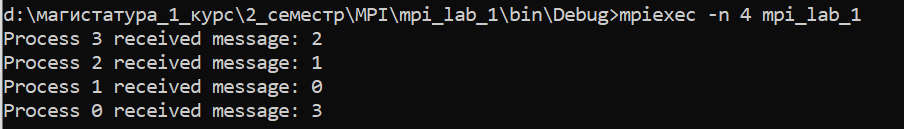
std::cout << "Process " << ProcRank << " received message: " << recv\_message << std::endl;

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



22

#include <mpi.h>

#include <iostream>

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

int ProcRank, ProcNum;

// Инициализация MPI

MPI\_Init(&argc, &argv);

// Получение количества процессов

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

// Получение ранга текущего процесса

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

// Каждый процесс отправляет свой номер всем остальным процессам

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

if (dest != ProcRank) {

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

}

}

// Каждый процесс получает сообщения от всех остальных процессов

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

if (src != ProcRank) {

int received\_message;

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

// Вывод полученного сообщения

std::cout << "[" << ProcRank << "]: receive message '" << received\_message << "' from " << src << std::endl;

}

}

// Вывод собственного сообщения

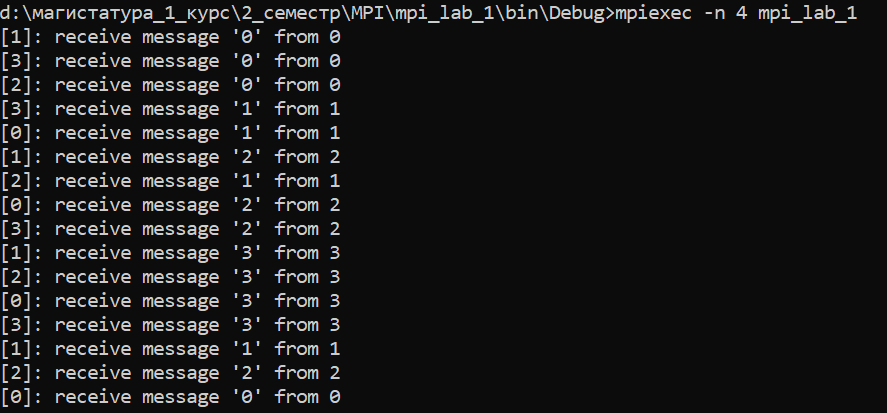
std::cout << "[" << ProcRank << "]: receive message '" << ProcRank << "' from " << ProcRank << std::endl;

// Завершение работы с MPI

MPI\_Finalize();

return 0;

}



23

#include <iostream>

#include <mpi.h>

#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;

std::string input\_string;

std::vector<int> global\_counts(26, 0); // Для хранения общего количества каждой буквы

if (rank == 0) {

// Ввод данных

std::cout << "Введите длину строки (1 <= n <= 100): ";

std::cin >> n;

std::cout << "Введите строку из " << n << " строчных английских букв: ";

std::cin >> input\_string;

}

// Рассылка длины строки всем процессам

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

// Рассылка строки всем процессам

if (rank != 0) {

input\_string.resize(n);

}

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

// Подсчет вхождений символов в каждом процессе

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

int start = rank \* n / size;

int end = (rank + 1) \* n / size;

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

if (input\_string[i] >= 'a' && input\_string[i] <= 'z') {

local\_counts[input\_string[i] - 'a']++;

}

}

// Сбор результатов от всех процессов

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

// Вывод результатов

if (rank == 0) {

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

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

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

}

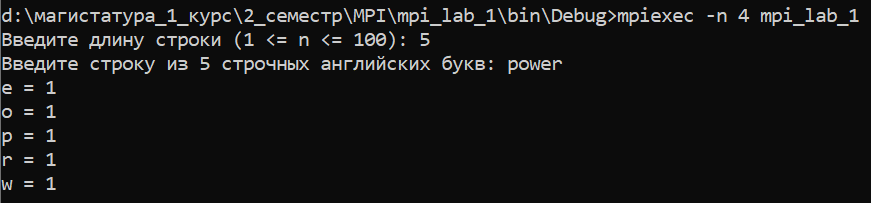
}

}

MPI\_Finalize();

return 0;

}



#include <iostream>

#include <mpi.h>

#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;

std::string input\_string;

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

if (rank == 0) {

// Ввод данных

std::cout << "Введите длину строки (1 <= n <= 100): ";

std::cin >> n;

std::cout << "Введите строку из " << n << " строчных английских букв: ";

std::cin >> input\_string;

// Отправка длины строки и строки каждому процессу

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

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

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

}

} else {

// Получение длины строки и строки

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

input\_string.resize(n);

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

}

// Подсчет вхождений символов в каждом процессе

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

int start = rank \* n / size;

int end = (rank + 1) \* n / size;

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

if (input\_string[i] >= 'a' && input\_string[i] <= 'z') {

local\_counts[input\_string[i] - 'a']++;

}

}

// Сбор результатов от всех процессов

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

// Вывод результатов

if (rank == 0) {

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

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

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

}

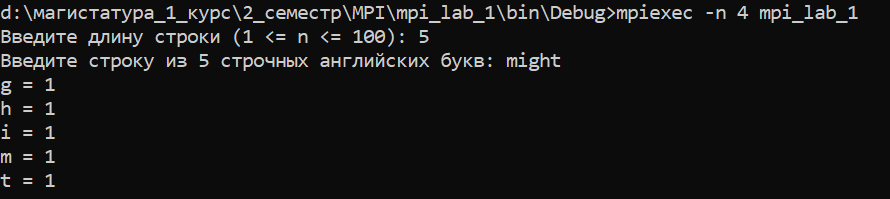
}

}

MPI\_Finalize();

return 0;

}



**Сравнение эффективности**

**Коллективные коммуникации (MPI\_Bcast):**

* **Преимущества:**
  + Простота реализации.
  + Оптимизированы для широковещательной рассылки, что может быть быстрее для большого числа процессов.
  + Меньше кода и меньше ошибок, связанных с передачей данных.
* **Недостатки:**
  + Может быть менее гибким для сложных сценариев передачи данных.

**Точечные коммуникации (MPI\_Send/MPI\_Recv):**

* **Преимущества:**
  + Большая гибкость в управлении передачей данных.
  + Может быть эффективнее для малого числа процессов или специфических сценариев.
* **Недостатки:**
  + Более сложная реализация, особенно для большого числа процессов.
  + Высокий риск ошибок, таких как тупиковые ситуации (deadlocks).

24

#include <iostream>

#include <iomanip>

#include <mpi.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 N;

if (rank == 0) {

std::cout << "Введите точность вычисления (N): ";

std::cin >> N;

}

// Рассылаем значение N всем процессам

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

double local\_pi = 0.0;

const double step = 1.0 / N;

// Каждый процесс вычисляет свою часть суммы

int start = rank \* N / size;

int end = (rank + 1) \* N / size;

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

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

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

}

local\_pi \*= step;

// Собираем частичные суммы с помощью MPI\_Reduce

double global\_pi;

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

// Вывод результата в процессе 0

if (rank == 0) {

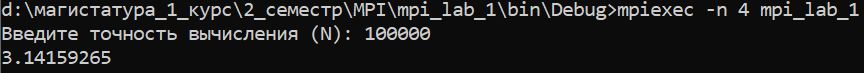
std::cout << std::fixed << std::setprecision(8) << global\_pi << std::endl;

}

MPI\_Finalize();

return 0;

}



#include <iostream>

#include <mpi.h>

#include <iomanip>

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;

if (rank == 0) {

std::cout << "Введите точность вычисления (N): ";

std::cin >> N;

}

// Рассылаем значение N всем процессам

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

double local\_pi = 0.0;

const double step = 1.0 / N;

// Каждый процесс вычисляет свою часть суммы

int start = rank \* N / size;

int end = (rank + 1) \* N / size;

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

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

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

}

local\_pi \*= step;

// Сбор результатов с использованием MPI\_Send и MPI\_Recv

if (rank != 0) {

// Процессы отправляют свои результаты процессу 0

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

} else {

// Процесс 0 собирает результаты от всех процессов

double global\_pi = local\_pi;

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

double received\_pi;

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

global\_pi += received\_pi;

}

// Вывод результата

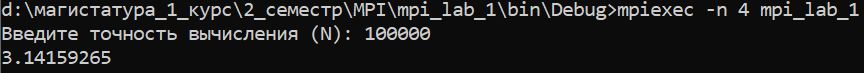
std::cout << std::fixed << std::setprecision(8) << global\_pi << std::endl;

}

MPI\_Finalize();

return 0;

}



## ****Коллективные коммуникации (MPI\_Reduce):****

* **Преимущества:**
  + Простота реализации: одна функция заменяет множество операций MPI\_Send и MPI\_Recv.
  + Оптимизация: коллективные операции обычно более эффективны, так как они используют внутренние оптимизации MPI для выполнения редукции.
  + Меньше кода и меньше ошибок.
* **Недостатки:**
  + Меньшая гибкость по сравнению с точечными коммуникациями.

## ****Точечные коммуникации (MPI\_Send/MPI\_Recv):****

* **Преимущества:**
  + Гибкость: можно контролировать каждый шаг передачи данных.
  + Полезно для нестандартных сценариев, где коллективные операции не подходят.
* **Недостатки:**
  + Сложность: требуется больше кода и внимания к деталям.
  + Риск ошибок: например, тупиковые ситуации (deadlocks), если порядок вызовов MPI\_Send и MPI\_Recv неправильный.
  + Меньшая производительность: ручная реализация редукции обычно менее эффективна, чем использование MPI\_Reduce.

25

#include <iostream>

#include <mpi.h>

#include <vector>

#include <iomanip>

void matrixMultiply(const std::vector<double>& A, const std::vector<double>& B, std::vector<double>& C, int n, int rows, int offset) {

for (int i = 0; i < rows; ++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 + offset) \* 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::cout << "Введите размер матрицы (n x n): ";

std::cin >> n;

if (n <= 0) {

std::cerr << "Ошибка: размер матрицы должен быть положительным!" << std::endl;

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

}

A.resize(n \* n);

B.resize(n \* n);

C.resize(n \* n);

std::cout << "Введите элементы матрицы A:" << std::endl;

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

std::cin >> A[i];

}

std::cout << "Введите элементы матрицы B:" << std::endl;

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

std::cin >> B[i];

}

}

// Рассылаем размер матрицы всем процессам

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

if (n <= 0) {

MPI\_Finalize();

return 1;

}

// Вычисляем, сколько строк должен обработать текущий процесс

int rows\_per\_process = n / size;

int remainder = n % size;

int local\_rows = (rank < remainder) ? rows\_per\_process + 1 : rows\_per\_process;

int offset = (rank < remainder)

? rank \* (rows\_per\_process + 1)

: remainder \* (rows\_per\_process + 1) + (rank - remainder) \* rows\_per\_process;

// Если процессу не досталось строк, завершаем его работу

if (local\_rows <= 0) {

MPI\_Finalize();

return 0;

}

// Выделяем память под локальные данные

std::vector<double> local\_A(local\_rows \* n);

std::vector<double> local\_C(local\_rows \* n);

// Рассылаем матрицу B всем процессам

if (rank != 0) {

B.resize(n \* n);

}

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

// Разделяем матрицу A с помощью MPI\_Scatterv (только если процессу достались строки)

if (local\_rows > 0) {

std::vector<int> sendcounts(size);

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

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

sendcounts[i] = (i < remainder) ? (rows\_per\_process + 1) \* n : rows\_per\_process \* n;

displs[i] = (i == 0) ? 0 : displs[i - 1] + sendcounts[i - 1];

}

MPI\_Scatterv(

A.data(),

sendcounts.data(),

displs.data(),

MPI\_DOUBLE,

local\_A.data(),

local\_rows \* n,

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

// Вычисляем произведение

matrixMultiply(local\_A, B, local\_C, n, local\_rows, 0);

// Собираем результаты

MPI\_Gatherv(

local\_C.data(),

local\_rows \* n,

MPI\_DOUBLE,

C.data(),

sendcounts.data(),

displs.data(),

MPI\_DOUBLE,

0,

MPI\_COMM\_WORLD

);

}

// Вывод результата (только root-процесс)

if (rank == 0) {

std::cout << "Результирующая матрица C:" << std::endl;

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

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

std::cout << std::setw(10) << C[i \* n + j] << " ";

}

std::cout << std::endl;

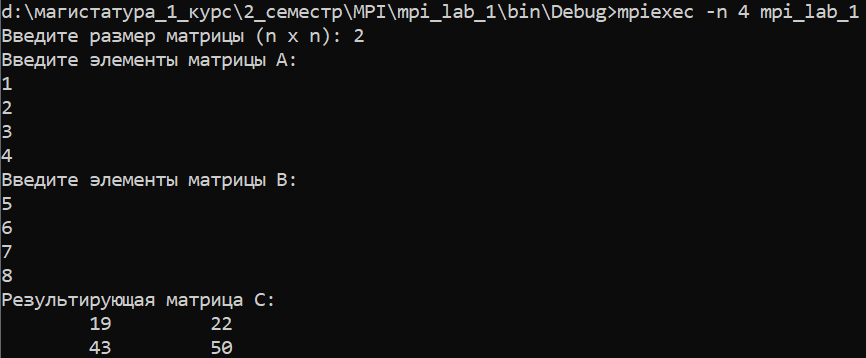
}

}

MPI\_Finalize();

return 0;

}



#include <iostream>

#include <mpi.h>

#include <vector>

#include <iomanip>

void matrixMultiply(const std::vector<double>& A, const std::vector<double>& B,

std::vector<double>& C, int n, int rows, int offset) {

for (int i = 0; i < rows; ++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 + offset) \* 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;

// Ввод данных в процессе 0

if (rank == 0) {

std::cout << "Введите размер матрицы (n x n): ";

std::cin >> n;

if (n <= 0) {

std::cerr << "Ошибка: размер матрицы должен быть положительным!" << std::endl;

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

}

A.resize(n \* n);

B.resize(n \* n);

C.resize(n \* n);

std::cout << "Введите элементы матрицы A:" << std::endl;

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

std::cin >> A[i];

}

std::cout << "Введите элементы матрицы B:" << std::endl;

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

std::cin >> B[i];

}

}

// Рассылка размера матрицы всем процессам

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

if (n <= 0) {

MPI\_Finalize();

return 1;

}

// Вычисление количества строк для каждого процесса

int rows\_per\_process = n / size;

int remainder = n % size;

int local\_rows = (rank < remainder) ? rows\_per\_process + 1 : rows\_per\_process;

int offset = (rank < remainder)

? rank \* (rows\_per\_process + 1)

: remainder \* (rows\_per\_process + 1) + (rank - remainder) \* rows\_per\_process;

// Выделение памяти под локальные данные

std::vector<double> local\_A(local\_rows \* n);

std::vector<double> local\_C(local\_rows \* n);

// Рассылка матрицы B всем процессам

if (rank != 0) {

B.resize(n \* n);

}

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

// Распределение строк матрицы A с помощью MPI\_Send/MPI\_Recv

if (rank == 0) {

// Отправка данных другим процессам

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

int dest\_rows = (dest < remainder) ? rows\_per\_process + 1 : rows\_per\_process;

int dest\_offset = (dest < remainder)

? dest \* (rows\_per\_process + 1)

: remainder \* (rows\_per\_process + 1) + (dest - remainder) \* rows\_per\_process;

if (dest\_rows > 0) {

MPI\_Send(&A[dest\_offset \* n], dest\_rows \* n, MPI\_DOUBLE, dest, 0, MPI\_COMM\_WORLD);

}

}

// Копирование своих данных

if (local\_rows > 0) {

std::copy(A.begin() + offset \* n,

A.begin() + (offset + local\_rows) \* n,

local\_A.begin());

}

} else {

// Получение данных от процесса 0

if (local\_rows > 0) {

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

}

}

// Вычисление локальной части произведения

if (local\_rows > 0) {

matrixMultiply(local\_A, B, local\_C, n, local\_rows, 0);

}

// Сбор результатов с помощью MPI\_Send/MPI\_Recv

if (rank == 0) {

// Копирование своих результатов

if (local\_rows > 0) {

std::copy(local\_C.begin(),

local\_C.end(),

C.begin() + offset \* n);

}

// Получение результатов от других процессов

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

int src\_rows = (src < remainder) ? rows\_per\_process + 1 : rows\_per\_process;

int src\_offset = (src < remainder)

? src \* (rows\_per\_process + 1)

: remainder \* (rows\_per\_process + 1) + (src - remainder) \* rows\_per\_process;

if (src\_rows > 0) {

MPI\_Recv(&C[src\_offset \* n], src\_rows \* n, MPI\_DOUBLE, src, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

}

} else {

// Отправка результатов процессу 0

if (local\_rows > 0) {

MPI\_Send(local\_C.data(), local\_rows \* n, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

}

// Вывод результата

if (rank == 0) {

std::cout << "Результирующая матрица C:" << std::endl;

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

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

std::cout << std::setw(10) << C[i \* n + j] << " ";

}

std::cout << std::endl;

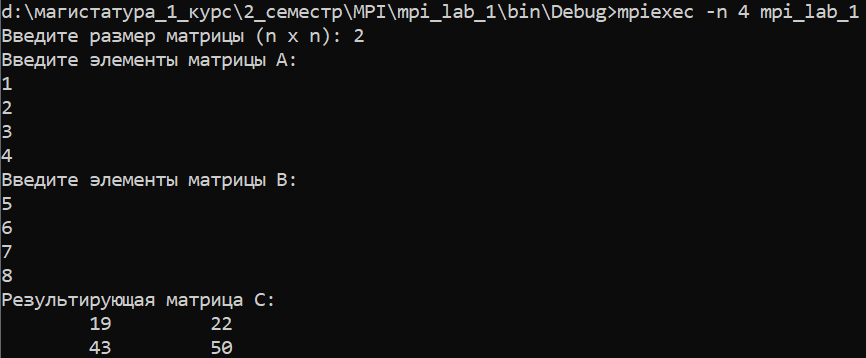
}

}

MPI\_Finalize();

return 0;

}



26

#include <cstring>

#include <iostream>

#include <mpi.h>

#include <string>

#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);

// Чтение сообщения процессом 0

char message[11] = {0}; // Максимум 10 символов + '\0'

if (world\_rank == 0) {

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

std::cin >> message;

// Проверка длины сообщения

if (strlen(message) < 1 || strlen(message) > 10) {

std::cerr << "Error: message length must be 1-10 characters" << std::endl;

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

}

}

// Создаем группу для процессов с четными номерами

MPI\_Group world\_group;

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

// Выбираем процессы с четными номерами

std::vector<int> even\_ranks;

for (int i = 0; i < world\_size; i += 2) {

even\_ranks.push\_back(i);

}

// Создаем новую группу

MPI\_Group even\_group;

MPI\_Group\_incl(world\_group, even\_ranks.size(), even\_ranks.data(), &even\_group);

// Создаем новый коммуникатор

MPI\_Comm even\_comm;

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

// Получаем информацию о новом коммуникаторе

int even\_rank = MPI\_UNDEFINED, even\_size = 0;

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

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

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

// Рассылаем сообщение только в новом коммуникаторе

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

}

// Синхронизация перед выводом

MPI\_Barrier(MPI\_COMM\_WORLD);

// Вывод информации

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

<< ". New comm: " << (even\_rank == MPI\_UNDEFINED ? -1 : even\_rank)

<< " from " << even\_size << ". Message = ";

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

std::cout << message;

} else {

std::cout << "no";

}

std::cout << std::endl;

// Освобождение ресурсов

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

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

}

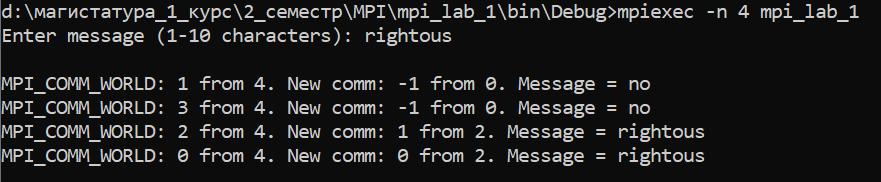
MPI\_Group\_free(&even\_group);

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

MPI\_Finalize();

return 0;

}



27

#include <iostream>

#include <mpi.h>

#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) {

// Родительский процесс

if (world\_rank == 0) {

int n;

std::cout << "Enter number of child processes to spawn: ";

std::cin >> n;

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

<< world\_size << " processes!\n"

<< "My parent is none." << std::endl;

// Создаем дочерние процессы

MPI\_Comm intercomm;

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

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

// Обмен данными с дочерними процессами

int data\_to\_send = 100;

MPI\_Bcast(&data\_to\_send, 1, MPI\_INT, MPI\_ROOT, intercomm);

// Собираем результаты от дочерних процессов

if (n > 0) {

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

MPI\_Comm child\_comm;

MPI\_Intercomm\_merge(intercomm, 0, &child\_comm);

MPI\_Gather(MPI\_IN\_PLACE, 0, MPI\_INT,

results.data(), 1, MPI\_INT,

0, child\_comm);

std::cout << "Results from children:";

for (int res : results) {

std::cout << " " << res;

}

std::cout << std::endl;

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

}

MPI\_Comm\_free(&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);

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

<< child\_size << " processes!\n"

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

// Получаем данные от родителя

int received\_data;

MPI\_Bcast(&received\_data, 1, MPI\_INT, 0, parent\_comm);

// Вычисляем результат

int result = received\_data + child\_rank;

// Отправляем результат родителю

MPI\_Comm child\_comm;

MPI\_Intercomm\_merge(parent\_comm, 1, &child\_comm);

MPI\_Gather(&result, 1, MPI\_INT,

nullptr, 0, MPI\_INT,

0, child\_comm);

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

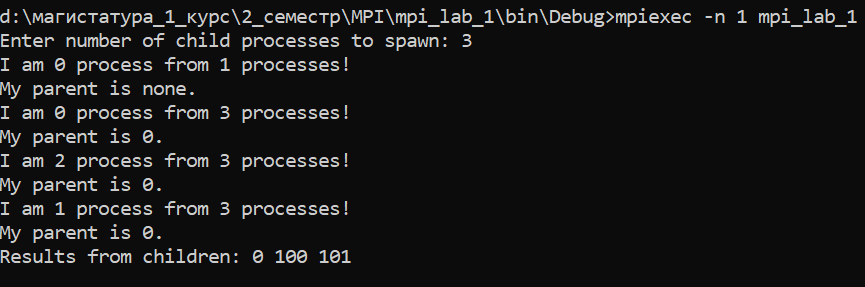
MPI\_Comm\_free(&parent\_comm);

}

MPI\_Finalize();

return 0;

}



28

#include <iostream>

#include <iomanip>

#include <mpi.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 N;

if (rank == 0) {

std::cout << "Введите точность вычисления (N): ";

std::cin >> N;

}

// Рассылаем значение N всем процессам

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

double local\_pi = 0.0;

const double step = 1.0 / N;

// Каждый процесс вычисляет свою часть суммы

int start = rank \* N / size;

int end = (rank + 1) \* N / size;

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

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

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

}

local\_pi \*= step;

// Создаем окно для односторонней коммуникации

double global\_pi = 0.0;

MPI\_Win win;

if (rank == 0) {

// Процесс 0 выделяет память для результата

MPI\_Win\_create(&global\_pi, sizeof(double), sizeof(double), MPI\_INFO\_NULL, MPI\_COMM\_WORLD, &win);

} else {

// Остальные процессы не выделяют память

MPI\_Win\_create(nullptr, 0, 1, MPI\_INFO\_NULL, MPI\_COMM\_WORLD, &win);

}

// Начинаем доступ к окну

MPI\_Win\_fence(0, win);

if (rank != 0) {

// Процессы, кроме 0, добавляют свой локальный результат в global\_pi процесса 0

MPI\_Accumulate(&local\_pi, 1, MPI\_DOUBLE, 0, 0, 1, MPI\_DOUBLE, MPI\_SUM, win);

} else {

// Процесс 0 добавляет свой локальный результат напрямую

global\_pi += local\_pi;

}

// Завершаем доступ к окну

MPI\_Win\_fence(0, win);

if (rank == 0) {

std::cout << std::fixed << std::setprecision(8) << global\_pi << std::endl;

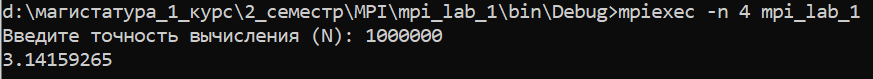
}

MPI\_Win\_free(&win);

MPI\_Finalize();

return 0;

}



29

ПРОДУБЛИРУЮ ЗДЕСЬ КОД ИЗ 24 ЛАБЫ, С ТОЙ ЛИШЬ РАЗНИЦЕЙ ЧТО ТУТ ДОБАВЛЕНО ОТОБРАЖЕНИЕ ВРЕМЕНИ

#include <iostream>

#include <iomanip>

#include <mpi.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 N;

if (rank == 0) {

std::cout << "Введите точность вычисления (N): ";

std::cin >> N;

}

// Рассылаем значение N всем процессам

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

// Засекаем время начала вычислений

double start\_time = MPI\_Wtime();

double local\_pi = 0.0;

const double step = 1.0 / N;

// Каждый процесс вычисляет свою часть суммы

int start = rank \* N / size;

int end = (rank + 1) \* N / size;

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

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

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

}

local\_pi \*= step;

// Собираем частичные суммы с помощью MPI\_Reduce

double global\_pi;

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

// Засекаем время окончания вычислений

double end\_time = MPI\_Wtime();

// Вывод результата и времени выполнения в процессе 0

if (rank == 0) {

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

std::cout << "Pi = " << global\_pi << std::endl;

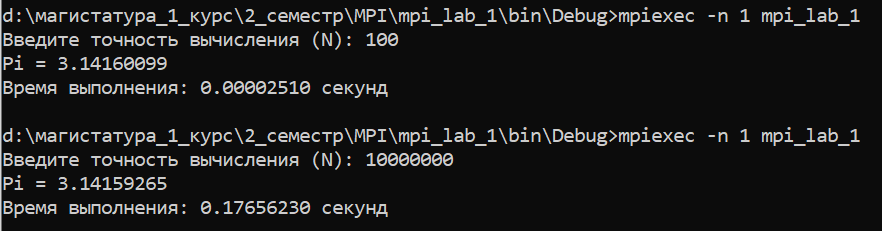
std::cout << "Время выполнения: " << (end\_time - start\_time) << " секунд" << std::endl;

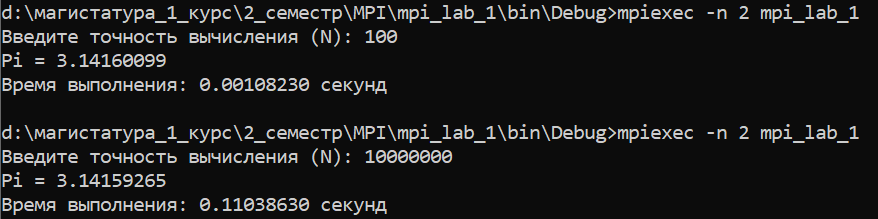
}

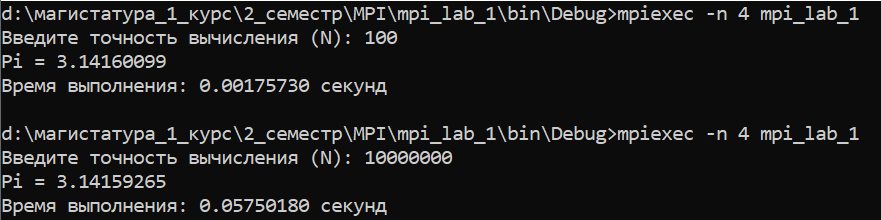
MPI\_Finalize();

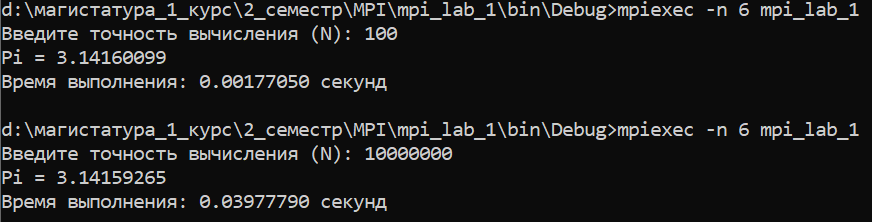
return 0;

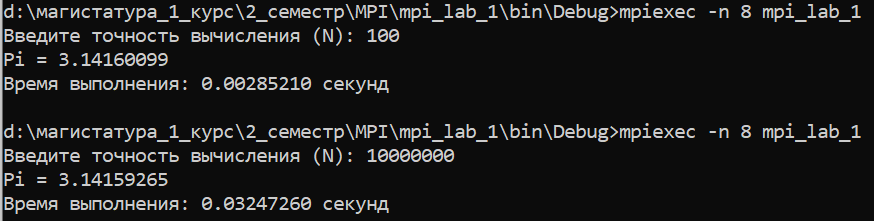
}

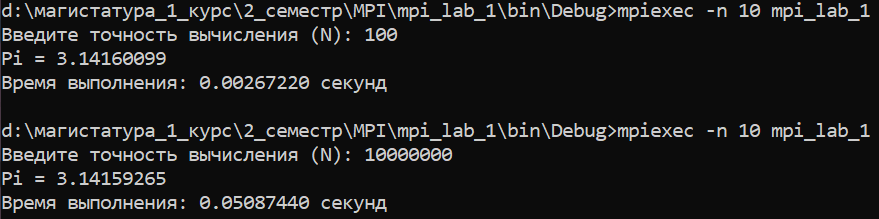


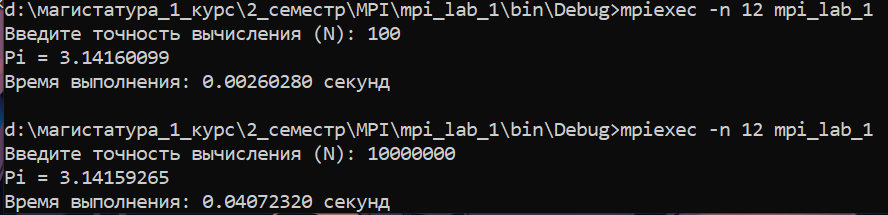












30

#include <iostream>

#include <mpi.h>

#include <omp.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 nthreads = 0;

#pragma omp parallel

{

#pragma omp master

nthreads = omp\_get\_num\_threads();

}

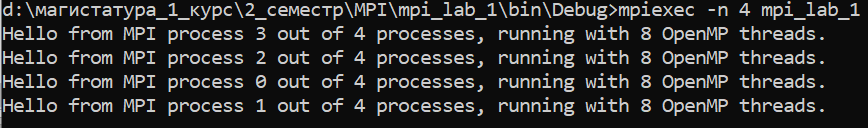
std::cout << "Hello from MPI process " << rank << " out of " << size

<< " processes, running with " << nthreads << " OpenMP threads." << std::endl;

MPI\_Finalize();

return 0;

}



31

#include <iostream>

#include <mpi.h>

#include <omp.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);

int n;

if (world\_rank == 0) {

std::cout << "Введите количество нитей: ";

std::cin >> n;

}

// Рассылаем значение n всем процессам

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

int total\_threads = n \* world\_size;

#pragma omp parallel num\_threads(n)

{

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

// Печатаем требуемую строку

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

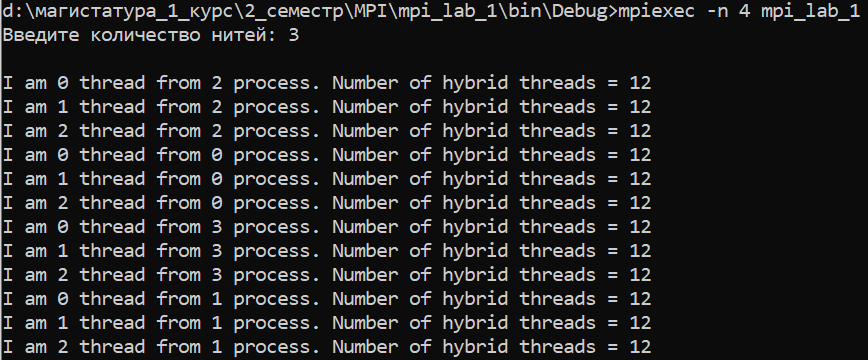
thread\_num, world\_rank, total\_threads);

}

MPI\_Finalize();

return 0;

}



32

#include <iostream>

#include <iomanip>

#include <mpi.h>

#include <omp.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 N;

if (rank == 0) {

std::cout << "Введите точность вычисления (N): ";

std::cin >> N;

}

// Рассылаем значение N всем процессам

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

double step = 1.0 / N;

double local\_sum = 0.0;

// Определяем границы работы для каждого процесса

int start = rank \* N / size;

int end = (rank + 1) \* N / size;

// Параллельный цикл OpenMP внутри процесса

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

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

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

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

}

local\_sum \*= step;

// Суммируем результаты всех процессов с помощью MPI\_Reduce

double global\_pi = 0.0;

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

if (rank == 0) {

std::cout << std::fixed << std::setprecision(8) << global\_pi << std::endl;

}

MPI\_Finalize();

return 0;

}

