

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»

(Самарский университет)

Естественнонаучный институт

Механико-математический факультет

Отчёт

Лабораторная работа №3

«Высокопроизводительные вычисления в механике»

Выполнил: Ушатов Д.О.

Группа 4446-010303D   
 Проверил: Неженский М.С.

      «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2025 г.

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Самара 2025

# Задание

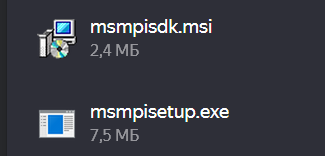
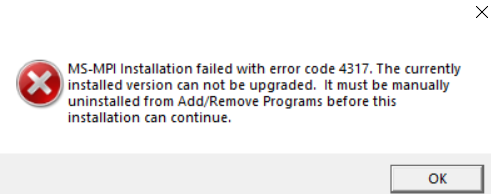
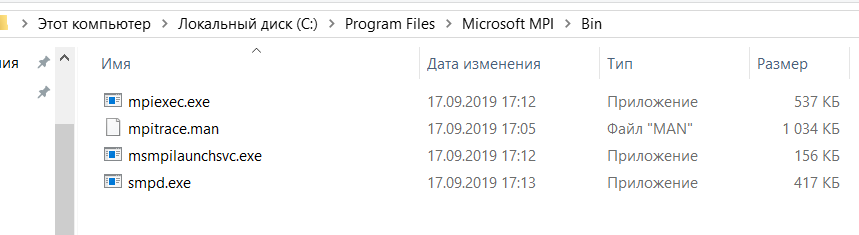
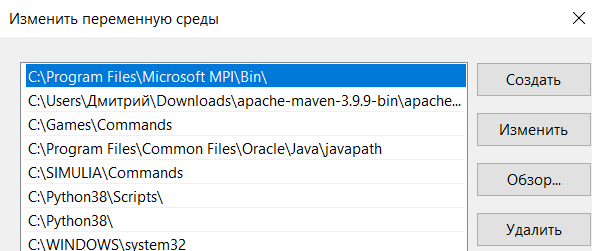
Исследовать модификацию программы из л/р №1 для параллельной работы по технологии MPI.

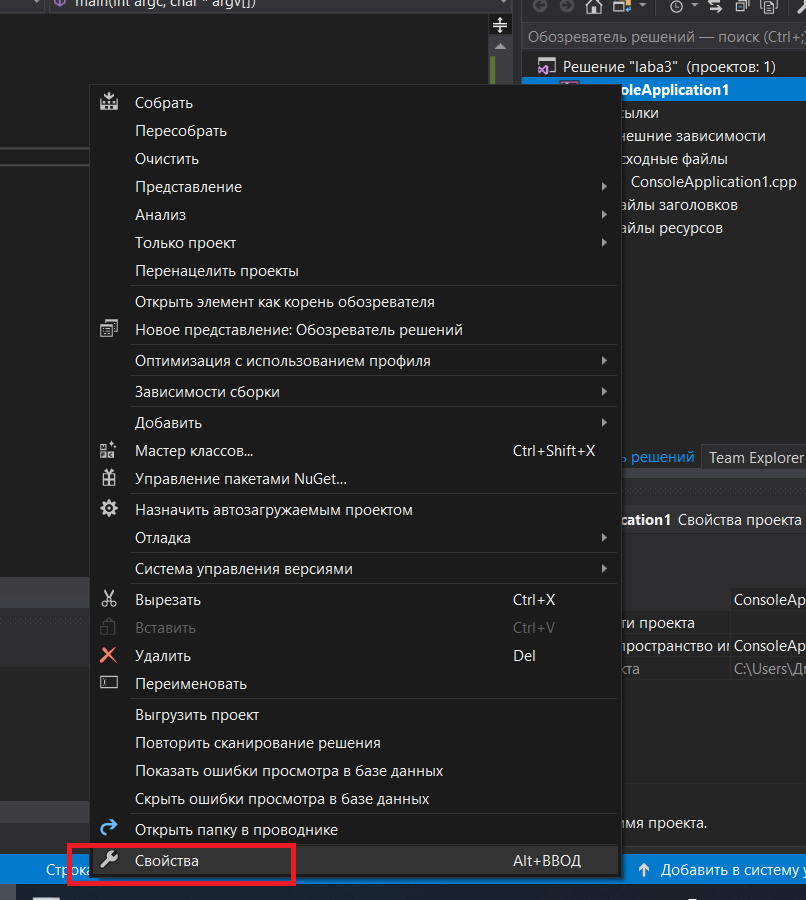
matMulMPI.cpp – исходники прилагаются (приложение 1)

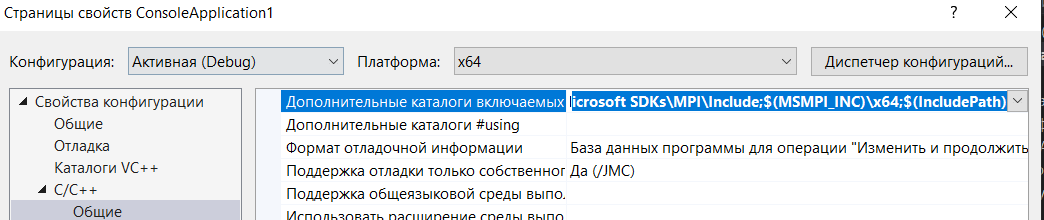
Аппаратные характеристики, программное обеспечение описаны в приложении 3.

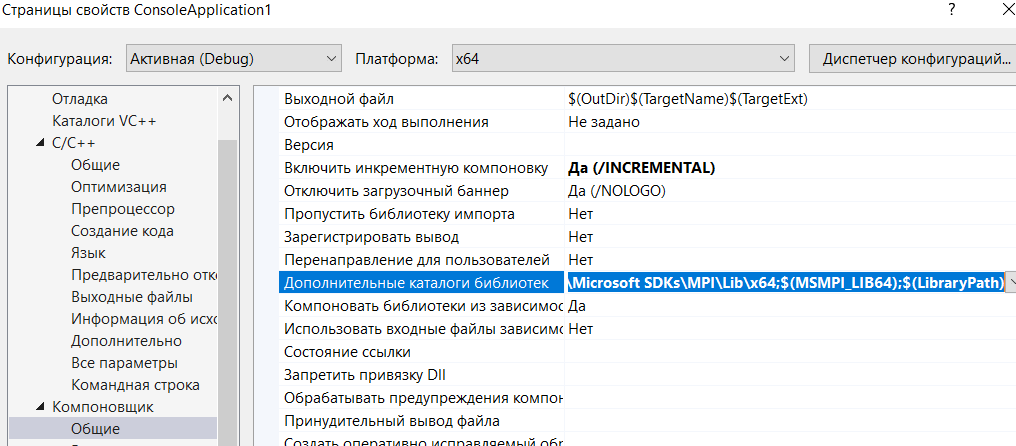
Для начала работы в Microsort Visual Studio 2017 Professional необходимо:

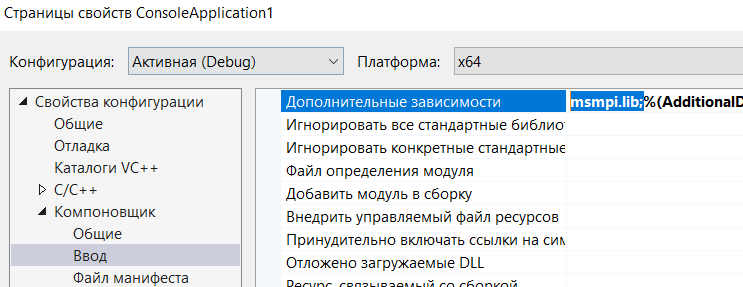
Установить Microsoft MPI (обязательно)

Скачать можно MS-MPI с официального сайта https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi (но придется использовать VPN, в противном случае эти файлы можно загрузить с GitHub: <https://github.com/microsoft/Microsoft-MPI/releases)> :   
MS-MPI v10.1.1:  
  
  
При установке msmpisetup.exe может возникнуть следующая ошибка:  
  
 Это значит, что на компьютере уже есть старая версия MPI, и установщик не может её автоматически обновить. В моём случае версия MPI была связана с несколькими программами, например, с Microsoft Visual Studio Express Edition 2008.   
После ручного ввода в переменных средах, установки значений в VS 2017, ошибка сохранялась. Помогло полное удаление старых версий MS-MPI и установка версии 10.1.1.   
  
После установки потребовалось вручную изменить переменные среды (path):  


После этого необходимо изменить в VS 2017 следующее:  
 В свойствах проекта:  


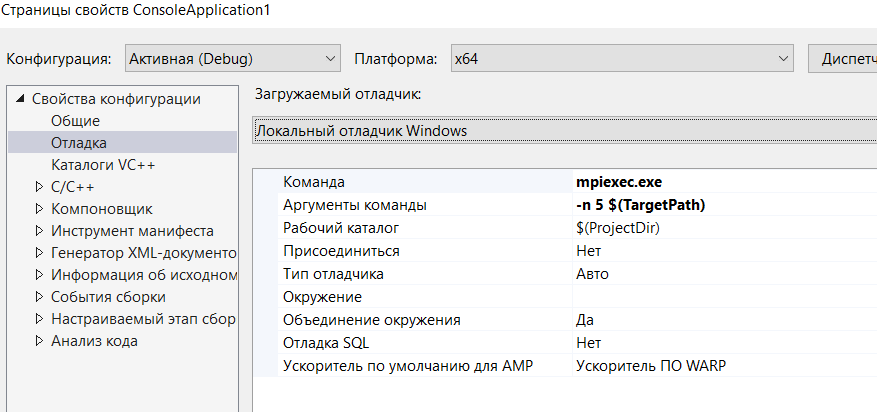
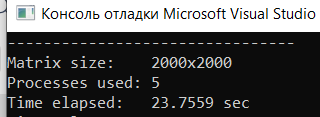
C/C++ → Общие → Доп. каталоги включаемых файлов: $(MSMPI\_INC)  


Компоновщик → Общие → Доп. каталоги библиотек: $(MSMPI\_LIB64);$(LibraryPath)  


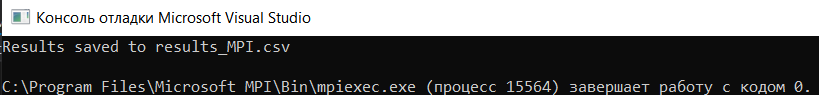
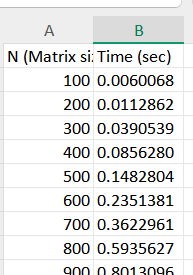
Компоновщик → Ввод → Доп. зависимости: msmpi.lib;%(AdditionalDependencies)  


В свойствах проекта:

Отладка → Команда: mpiexec.exe

Аргументы команды: -n 5 $(TargetPath)  
  
  
 Запускаем программу, получаем ошибки, из-за которых код незначительно модернизируем (см.приложение 2). После изменений вновь производим запуск, получаем при N=2000 и при n(кол-во ядер процессора)=5:  


У нас уже есть данные из лабораторной работы №2 по времени выполения при последовательных ввычислениях, поэтому продолжим работать исключительно с версией MPI. Для оптимизации человеческого труда, вносим изменения в код для

Далее запускаем программу с размерностью матриц N=100..2000 с шагом 100. Выводим результаты в файл формата .cvs «result\_MPI.csv»:  
  


То же самое делаем при N=5..95 с шагом 5. Данные по времени при N=5..2000 представлены в таблицах 1,2, где также осуществлено сравнение времени выполнения вычислений.

Таблица 1. Сравнение вычислений при N от 5 до 100 с шагом 5

|  |  |  |
| --- | --- | --- |
|  | Последовательное вычисление | MPI |
| 5 | 0.0000113 | 0.0000006 |
| 10 | 0.0000098 | 0.0000012 |
| 15 | 0.0000219 | 0.0000022 |
| 20 | 0.0000382 | 0.0000079 |
| 25 | 0.0000589 | 0.0000159 |
| 30 | 0.0014761 | 0.0000226 |
| 35 | 0.0001613 | 0.0000370 |
| 40 | 0.0002605 | 0.0000474 |
| 45 | 0.0004763 | 0.0001073 |
| 50 | 0.0005524 | 0.0001464 |
| 55 | 0.0007341 | 0.0001801 |
| 60 | 0.0008838 | 0.0001943 |
| 65 | 0.0011067 | 0.0003093 |
| 70 | 0.0013527 | 0.0003016 |
| 75 | 0.0014934 | 0.0004037 |
| 80 | 0.0022463 | 0.0006254 |
| 85 | 0.0025019 | 0.0005946 |
| 90 | 0.0030884 | 0.0006923 |
| 95 | 0.0034998 | 0.0010377 |
| 100 | 0.0033512 | 0.0000006 |

Сравнение последовательной и параллельной версий:

Видим, что использование технологии MPI выгоднее по времени, однако их эффективность достаточно низка (выигрыш по времени не соответствует затраченным ресурсам). Значит, последовательная версия (1 поток) эффективна для N < 100, так как нет накладных расходов на создание потоков.

Посмотрим, что будет, если размерность матрицы значительно увеличить. Так получаем таблицу 2.

Таблица 2. N=100..2000 с шагом 100

|  |  |  |
| --- | --- | --- |
| Размерность матрицы N | Полследовательное вычисление | MPI |
| 100 | 0.0017869 | 0.0000006 |
| 200 | 0.0536293 | 0.0141363 |
| 300 | 0.1859679 | 0.0403806 |
| 400 | 0.4850560 | 0.0837038 |
| 500 | 1.3044851 | 0.1528482 |
| 600 | 1.9892603 | 0.2744501 |
| 700 | 3.1790760 | 0.3827995 |
| 800 | 2.7097745 | 0.5873585 |
| 900 | 5.4355068 | 0.8828559 |
| 1000 | 6.2476031 | 1.102102 |
| 1100 | 10.1608067 | 1.502885 |
| 1200 | 14.3275076 | 1.992255 |
| 1300 | 19.1640031 | 2.531742 |
| 1400 | 17.4531502 | 3.041824 |
| 1500 | 35.9598479 | 3.904516 |
| 1600 | 24.7978589 | 5.123034 |
| 1700 | 40.0282811 | 7.430563 |
| 1800 | 48.7874487 | 8.596651 |
| 1900 | 73.3097600 | 10.232560 |
| 2000 | 57.8104152 | 11.776311 |

Заметим по графику, что последовательная версия (1 поток) при N > 400 становится неоптимальной, а при N> 800 разница становится весьма значимой в пользу параллельного метода. Также отметим, что время выполнения растёт кубически (O(N³)), так как для каждой ячейки результирующей матрицы выполняется N операций умножения и сложения.

Итог: Вычисления с MPI ускоряют перемножение матриц в 5 раз для больших размеров (N > 1000), но для малых матриц (N < 100) выгоднее использовать последовательный код из-за накладных расходов. Оптимальный результат при сравнении с последовательными вычислениями достигается при 5 процессах (что соответствует 5 физическим ядрам процессора), а для задач с N > 1000 главным ограничением становится не вычислительная мощность, а эффективность работы с памятью и минимизация коммуникационных затрат MPI. Оптимальные результаты достигаются при тщательном балансе между количеством процессов (не больше физических ядер) и локализацией данных.

Приложение 1

Исходный код с пояснениями:

#include "mpi.h" // Основная библиотека MPI

#include <cstdlib> // Для работы с памятью и atoi()

#include <iostream> // Ввод/вывод

#include "time.h" // Для замера времени (clock())

using namespace std;

// Глобальные переменные MPI

int ProcNum; // Общее количество процессов

int ProcRank; // Ранг (номер) текущего процесса

// Функция транспонирования матрицы

void Flip(double \*B, int Size)

{

double temp = 0.0;

for (int i = 0; i < Size; i++)

{

for (int j = i + 1; j < Size; j++)

{

temp = B[i\*Size + j];

B[i\*Size + j] = B[j\*Size + i];

B[j\*Size + i] = temp;

}

}

}

// Основная функция умножения матриц с использованием MPI

void MatrixMultiplicationMPI(double \*A, double \*B, double \*C, int Size)

{

int dim = Size;

int i, j, k, p, ind;

double temp;

MPI\_Status Status;

int ProcPartSize = dim / ProcNum; // Количество строк на процесс

int ProcPartElem = ProcPartSize \* dim; // Количество элементов на процесс

// Выделение памяти для буферов

double\* bufA = new double[ProcPartElem];

double\* bufB = new double[ProcPartElem];

double\* bufC = new double[ProcPartElem];

if (ProcRank == 0) {

Flip(B, Size);

}

// Разделение данных между процессами

MPI\_Scatter(A, ProcPartElem, MPI\_DOUBLE, bufA, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(B, ProcPartElem, MPI\_DOUBLE, bufB, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Локальное умножение блоков

temp = 0.0;

for (i = 0; i < ProcPartSize; i++) {

for (j = 0; j < ProcPartSize; j++) {

for (k = 0; k < dim; k++)

temp += bufA[i\*dim + k] \* bufB[j\*dim + k]; / // Сохранение результата с учетом смещения для текущего процесса

bufC[i\*dim + j + ProcPartSize \* ProcRank] = temp;

temp = 0.0;

}

}

// пересылка данных между процессами

int NextProc; int PrevProc;

for (p = 1; p < ProcNum; p++) {

NextProc = ProcRank + 1;

if (ProcRank == ProcNum - 1) NextProc = 0;

PrevProc = ProcRank - 1;

if (ProcRank == 0) PrevProc = ProcNum - 1;

// Отправка своего блока следующему процессу и получение от предыдущего

MPI\_Sendrecv\_replace(bufB, ProcPartElem, MPI\_DOUBLE,

NextProc, 0, PrevProc, 0, MPI\_COMM\_WORLD, &Status);

temp = 0.0;

for (i = 0; i < ProcPartSize; i++) {

for (j = 0; j < ProcPartSize; j++) {

for (k = 0; k < dim; k++) {

temp += bufA[i\*dim + k] \* bufB[j\*dim + k];

}

// Вычисление для записи результата

if (ProcRank - p >= 0)

ind = ProcRank - p;

else

ind = (ProcNum - p + ProcRank);

bufC[i\*dim + j + ind \* ProcPartSize] = temp;

temp = 0.0;

}

}

}

// Сбор результатов от всех процессов

MPI\_Gather(bufC, ProcPartElem, MPI\_DOUBLE, C, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Освобождение памяти

delete[] bufA;

delete[] bufB;

delete[] bufC;

}

// Функция вывода матрицы (шаблонная для работы с разными типами)

template <typename T>

int matrixOutput(T \*Mat, int size, string name)

{

cout << "\"" << name << "\" matrix output:" << endl;

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

{

for (int j = 0; j < size; j++)

{

cout << Mat[i\*size + j] << " ";

}

cout << endl;

}

return 1;

}

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

{

clock\_t start; // Для замера времени выполнения

// Проверка аргументов командной строки

if (argc != 2)

{

cout << "Program usage: " << endl

<< "./" << argv[0] << " <n>" << endl

<< "where <n> is the size of square matrix" << endl;

return -1;

}

const int N = atoi(argv[1]); // Получаем размер матрицы из аргументов

cout << "Begin initializing ..." << endl;

double \*A, \*B; // Указатели на исходные матрицы

// Выделение памяти под матрицы

A = new double[N\*N];

B = new double[N\*N];

// Инициализация матриц A и B по заданным формулам

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

{

A[i\*N + j] = (i + 1) \* (j + 1); // A[i][j] = (i+1)\*(j+1)

B[i\*N + j] = (i + 1) + 2 \* (j + 1); // B[i][j] = (i+1) + 2\*(j+1)

}

// Вывод матриц (закомментирован для больших размеров)

//matrixOutput<double>(A, N, "A");

//matrixOutput<double>(B, N, "B");

double \*C = new double[N\*N]; // Память под результирующую матрицу

cout << "Begin calculating ..." << endl;

start = clock(); // Замер времени начала вычислений

MPI\_Init(&argc, &argv);

// Получаем количество процессов и ранг текущего процесса

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

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

// Основная функция умножения матриц

MatrixMultiplicationMPI(A, B, C, N);

// Вывод времени выполнения

if (ProcRank == 0) {

cout << endl << "Calculation time: "

<< double(clock() - start) / CLOCKS\_PER\_SEC

<< " seconds" << endl;

}

//matrixOutput<double>(C, N, "resulting C");

// Освобождение памяти

delete[] A;

delete[] B;

delete[] C;

// Завершение работы MPI

MPI\_Finalize();

return 0;

}

# Приложение 2

Следующий код позволяет задать изменение размерности матрицы N [от .. до] с некоторым шагом для вычислений на нескольких ядрах процессора (в данном случае для 5) и полученные результаты (время выполнения) записать для удобства работы в файл «results\_MPI.csv». Откуда извлекаем данные для построения графиков и дальнейшего анализа.  
#include "mpi.h" // Основная библиотека MPI

#include <cstdlib> // Для работы с памятью

#include <iostream> // Ввод/вывод

#include <fstream> // Работа с файлами

#include <iomanip> // Форматирование вывода

using namespace std;

// Глобальные переменные MPI

int ProcNum; // Общее количество процессов

int ProcRank; // Ранг (номер) текущего процесса

// Функция транспонирования матрицы (оптимизация доступа к памяти)

void Flip(double \*B, int Size) {

double temp;

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

for (int j = i + 1; j < Size; j++) {

// Классический swap элементов

temp = B[i \* Size + j];

B[i \* Size + j] = B[j \* Size + i];

B[j \* Size + i] = temp;

}

}

}

// Основная функция умножения матриц с кольцевым алгоритмом

void MatrixMultiplicationMPI(double \*A, double \*B, double \*C, int Size) {

int dim = Size;

double temp;

MPI\_Status Status;

// Расчет размера блока для каждого процесса

int ProcPartSize = dim / ProcNum; // Количество строк на процесс

int ProcPartElem = ProcPartSize \* dim; // Всего элементов на процесс

// Выделение памяти для буферов

double\* bufA = new double[ProcPartElem];

double\* bufB = new double[ProcPartElem];

double\* bufC = new double[ProcPartElem];

if (ProcRank == 0) {

Flip(B, Size);

}

// Разделение данных между процессами

MPI\_Scatter(A, ProcPartElem, MPI\_DOUBLE, bufA, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(B, ProcPartElem, MPI\_DOUBLE, bufB, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Локальное умножение блоков

temp = 0.0;

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

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

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

temp += bufA[i \* dim + k] \* bufB[j \* dim + k];

}

// Сохранение для текущего процесса

bufC[i \* dim + j + ProcPartSize \* ProcRank] = temp;

temp = 0.0;

}

}

// (основной алгоритм)

int NextProc, PrevProc;

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

NextProc = ProcRank + 1;

if (ProcRank == ProcNum - 1) NextProc = 0;

PrevProc = ProcRank - 1;

if (ProcRank == 0) PrevProc = ProcNum - 1;

// Отправка и прием данных

MPI\_Sendrecv\_replace(bufB, ProcPartElem, MPI\_DOUBLE,

NextProc, 0, PrevProc, 0, MPI\_COMM\_WORLD, &Status);

// Умножение с новым блоком

temp = 0.0;

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

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

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

temp += bufA[i \* dim + k] \* bufB[j \* dim + k];

}

// Расчет для записи результата

int ind = (ProcRank - p >= 0) ? ProcRank - p : (ProcNum - p + ProcRank);

bufC[i \* dim + j + ind \* ProcPartSize] = temp;

temp = 0.0;

}

}

}

// Сбор результатов

MPI\_Gather(bufC, ProcPartElem, MPI\_DOUBLE, C, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Освобождение памяти

delete[] bufA;

delete[] bufB;

delete[] bufC;

}

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

// Инициализация MPI

MPI\_Init(&argc, &argv);

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

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

// Создание файла результатов

ofstream output\_file;

if (ProcRank == 0) {

output\_file.open("results\_MPI.csv");

output\_file << "N (Matrix size);Time (sec)\n";

}

// Основной цикл по размерам матриц (5-95 с шагом 5)

for (int N = 5; N <= 95; N += 5) {

// Пропускаем размеры, которые не делятся на число процессов

if (N % ProcNum != 0) {

if (ProcRank == 0) {

output\_file << N << ";SKIPPED (N not divisible by ProcNum)\n";

}

continue;

}

// Выделение памяти

double \*A = nullptr, \*B = nullptr, \*C = new double[N \* N];

// Только root-процесс заполняет исходные матрицы

if (ProcRank == 0) {

A = new double[N \* N];

B = new double[N \* N];

// Заполнение по схеме

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

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

A[i \* N + j] = (i + 1) \* (j + 1); // A[i][j] = (i+1)\*(j+1)

B[i \* N + j] = (i + 1) + 2 \* (j + 1); // B[i][j] = (i+1) + 2\*(j+1)

}

}

}

// Замер времени выполнения

double start\_time = MPI\_Wtime();

MatrixMultiplicationMPI(A, B, C, N);

double elapsed\_time = MPI\_Wtime() - start\_time;

// Запись результатов

if (ProcRank == 0) {

output\_file << N << ";" << fixed << setprecision(7) << elapsed\_time << "\n";

delete[] A;

delete[] B;

}

delete[] C; // Все процессы освобождают память для C

}

// Завершение работы

if (ProcRank == 0) {

output\_file.close();

cout << "Results saved to results\_MPI.csv\n";

}

MPI\_Finalize();

return 0;

}

Приложение 3

Аппаратные харакеристики:  
Процессор: AMD Ryzen 5 5500U (6 ядер / 12 потоков, 2.1 – 4.0 GHz, TDP 15 Вт)

* Графика\*\*: Radeon Vega 7 (встроенная)
* Техпроцесс: 7 нм
* Кэш\*\*: L2 — 3 MB, L3 — 8 MB

Оперативная память:

* 1 модуль DDR4 8 ГБ, 3200 МГц  
  Программное обеспечение:

Название ОС: Microsoft Windows 10 Pro

Версия 22H2: Сборка ОС 19045.5247

Тип системы: x64-based PC

Для работы с кодом было использовано: Visual Studio 2017 Pro, при работе питание осуществялось от сети.