###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Умножение матрицы на матрицу в MPI 2D решетка»

студента 2 курса, 23210 группы

**Лаухина Егора Денисовича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Артюхов А.А.

Новосибирск 2025

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc101966097)

[ЗАДАНИЕ 3](#_Toc101966098)

ОПИСАНИЕ РАБОТЫ 3

ЗАКЛЮЧЕНИЕ 4

Приложение 1. Графики 5

Приложение 2. Код программы 6

[Приложение 3. Профилирование 9](#_Toc101966103)

# **ЦЕЛЬ**

Изучить и реализовать **параллельный алгоритм умножения матриц** с использованием **двумерной (2D) решетки процессов** в среде MPI. Исследовать эффективность распараллеливания вычислений в зависимости от конфигурации решетки.

# **ЗАДАНИЕ**

1. **Реализация алгоритма:**
   * Разработать программу на MPI, выполняющую умножение матриц  *C*=*A*×*B* с использованием **2D-распределения данных**:
     + Матрица A разбивается на p1​ горизонтальных полос.
     + Матрица B разбивается на p2​ вертикальных полос.
     + Каждый процесс вычисляет свою подматрицу Ci,j​ как произведение соответствующих полос Ai​ и Bj​.
2. **Исследование производительности:**
   * Провести тесты для разных конфигураций решетки.
   * Измерить время выполнения и сравнить с последовательной версией.
   * Построить графики зависимости времени от размера матрицы и числа процессов.
3. **Профилирование программы:**
   * Выполнить профилирование программы с использованием решетки 4х3

# **ОПИСАНИЕ РАБОТЫ**

1. **Реализация и проверка алгоритма**
   * Был успешно реализован **параллельный алгоритм умножения матриц** с использованием **2D-решетки процессов** в MPI.
   * Корректность работы программы подтверждена **тестированием**: результаты умножения совпали с эталонными (последовательными вычислениями).
2. **Исследование производительности**
   * Проведены запуски программы с разными конфигурациями 2D-решеток (например, 2×2, 3×2, 3×4 и тд.).
   * Составлены **таблицы зависимости времени выполнения** от:
     + Формы решетки (числа процессов p1×p2​).
   * Построены **графики ускорения (Sp​)** и **эффективности (Ep​)** в зависимости от числа процессов
3. **Профилирование программы**
   * Выполнено **профилирование на 16 процессах**
   * Проанализированы:
     + Распределение нагрузки между процессами.
     + Время, затраченное на **коммуникации** (передачи данных) и **вычисления**.
     + Узкие места производительности
4. **Вывод**
   * На основе анализа полученных таблиц и графиков был сформулирован вывод.

Размер матрицы: 2400×2400

|  |  |  |  |
| --- | --- | --- | --- |
| ***Полученные результаты*** | | | |
| **Решетка** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1х1 | 48,4963 | 100% | 1 |
| 2х1 | 26,9651 | 90% | 1,79 |
| 3х1 | 29,2474 | 83% | 1,65 |
| 4х1 | 21,0859 | 57% | 2,3 |
| 2х2 | 20,0664 | 60% | 2,42 |
| 6х1 | 17,24 | 46% | 2,81 |
| 2х3 | 16,5738 | 48% | 2,92 |
| 3х2 | 16,3458 | 49% | 2,96 |
| 3х3 | 13,6039 | 40% | 3,56 |
| 5х2 | 13,0552 | 37% | 3,71 |
| 4х3 | 12,1781 | 33% | 3,98 |
| 12х1 | 12,6082 | 32% | 3,84 |

# ЗАКЛЮЧЕНИЕ

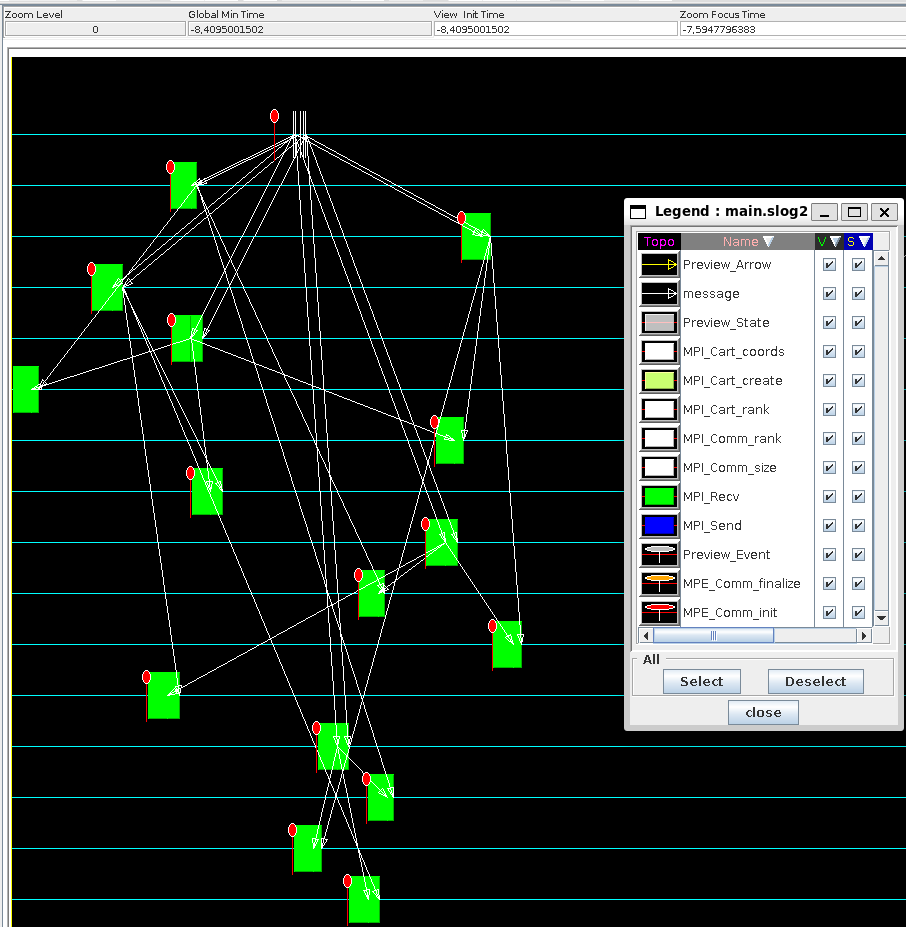
Анализ таблицы позволяет сделать вывод, что использование 2D-решетки в разы уменьшает время выполнения программы, но при этом видно, что увеличение числа процессов приводит к снижению эффективности.

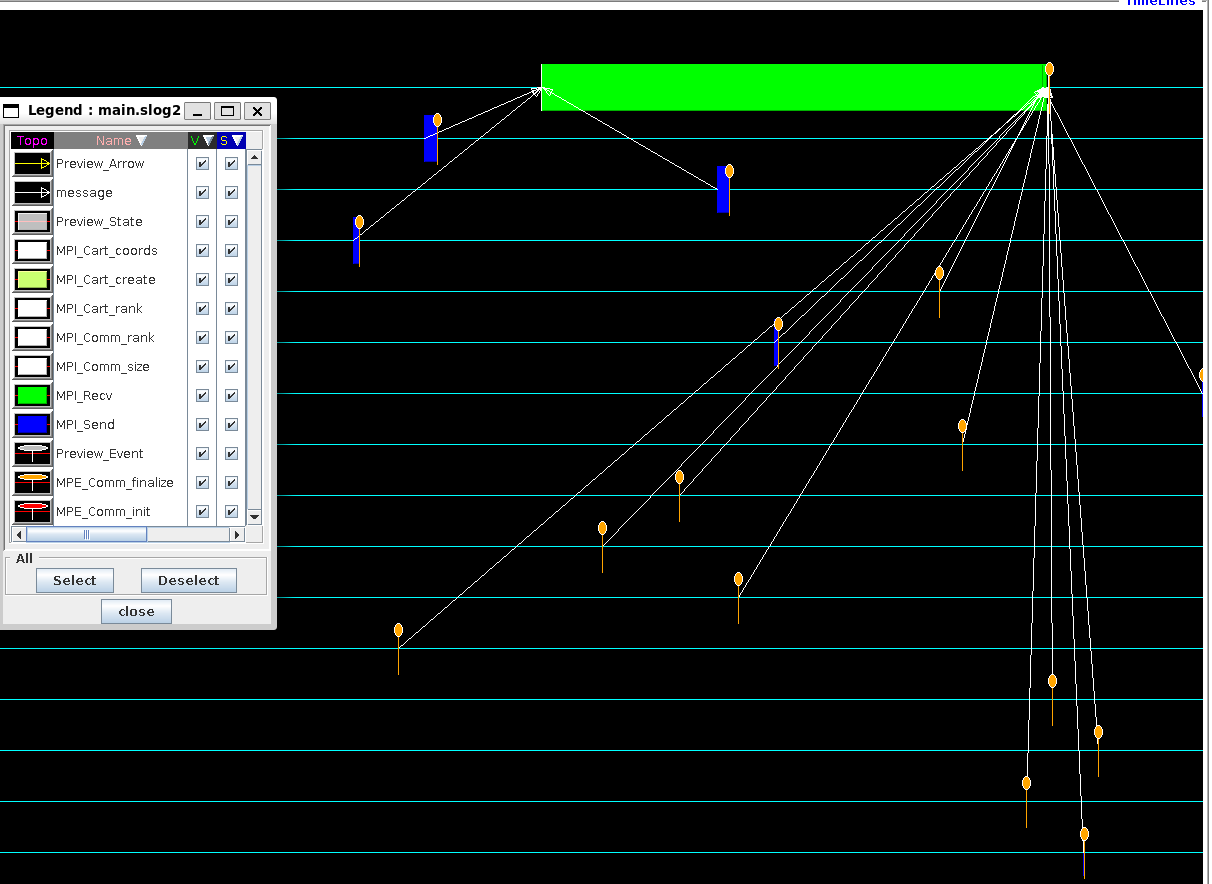
# ГРАФИК

**Код Программы**

#include <mpi.h>  
#include <iostream>  
#include <vector>  
#include <cstdlib>  
#include <ctime>  
#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);  
 auto startTime = MPI\_Wtime();  
  
 int n1 = 2400, n2 = 2400, n3 = 2400;  
 int proc\_rows = 4, proc\_cols = 3;  
  
 int dims[2] = {proc\_rows, proc\_cols};  
 int periods[2] = {0, 0};  
 MPI\_Comm grid\_comm;  
 MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, 1, &grid\_comm);  
  
 int coords[2];  
 MPI\_Cart\_coords(grid\_comm, rank, 2, coords);  
 int i = coords[0], j = coords[1];  
  
 int block\_n1 = n1 / proc\_rows;  
 int block\_n2 = n2;  
 int block\_n3 = n3 / proc\_cols;  
  
 std::vector<double> A, B;  
 if (rank == 0) {  
 A.resize(n1 \* n2);  
 B.resize(n2 \* n3);  
 std::srand(std::time(nullptr));  
  
  
 for (int i = 0; i < n1 \* n2; ++i) A[i] = std::rand() % 10;  
 for (int i = 0; i < n2 \* n3; ++i) B[i] = std::rand() % 10;  
 }  
  
 std::vector<double> temp\_A(block\_n1 \* n2);  
 std::vector<double> temp\_B(n2 \* block\_n3);  
 if (j == 0) {  
 if (rank == 0) {  
 for (int pi = 0; pi < proc\_rows; ++pi) {  
 std::vector<double> temp\_block(block\_n1 \* n2);  
 for (int row = 0; row < block\_n1; ++row) {  
 std::copy(A.begin() + (pi \* block\_n1 + row) \* n2,  
 A.begin() + (pi \* block\_n1 + row + 1) \* n2,  
 temp\_block.begin() + row \* n2);  
 }  
 if (pi == 0) {  
 temp\_A = temp\_block;  
 } else {  
 int dest\_rank;  
 int coords\_send[2] = {pi, 0};  
 MPI\_Cart\_rank(grid\_comm, coords\_send, &dest\_rank);  
 MPI\_Send(temp\_block.data(), block\_n1 \* n2, MPI\_DOUBLE, dest\_rank, 1, grid\_comm);  
 }  
 }  
 } else {  
 MPI\_Recv(temp\_A.data(), block\_n1 \* n2, MPI\_DOUBLE, 0, 1, grid\_comm, MPI\_STATUS\_IGNORE);  
 }  
  
 for (int pj = 1; pj < proc\_cols; ++pj) {  
 int dest\_rank;  
 int coords\_send[2] = {i, pj};  
 MPI\_Cart\_rank(grid\_comm, coords\_send, &dest\_rank);  
 MPI\_Send(temp\_A.data(), block\_n1 \* n2, MPI\_DOUBLE, dest\_rank, 3, grid\_comm);  
 }  
 } else {  
 MPI\_Recv(temp\_A.data(), block\_n1 \* n2, MPI\_DOUBLE, MPI\_ANY\_SOURCE, 3, grid\_comm, MPI\_STATUS\_IGNORE);  
 }  
  
  
 if (i == 0) {  
 if (rank == 0) {  
 for (int pj = 0; pj < proc\_cols; ++pj) {  
 std::vector<double> temp\_block(n2 \* block\_n3);  
 for (int col = 0; col < block\_n3; ++col) {  
 for (int row = 0; row < n2; ++row) {  
 temp\_block[row \* block\_n3 + col] = B[row \* n3 + pj \* block\_n3 + col];  
 }  
 }  
 if (pj == 0) {  
 temp\_B = temp\_block;  
 } else {  
 int dest\_rank;  
 int coords\_send[2] = {0, pj};  
 MPI\_Cart\_rank(grid\_comm, coords\_send, &dest\_rank);  
 MPI\_Send(temp\_block.data(), n2 \* block\_n3, MPI\_DOUBLE, dest\_rank, 2, grid\_comm);  
 }  
 }  
 } else {  
 MPI\_Recv(temp\_B.data(), n2 \* block\_n3, MPI\_DOUBLE, 0, 2, grid\_comm, MPI\_STATUS\_IGNORE);  
 }  
  
  
 for (int pi = 1; pi < proc\_rows; ++pi) {  
 int dest\_rank;  
 int coords\_send[2] = {pi, j};  
 MPI\_Cart\_rank(grid\_comm, coords\_send, &dest\_rank);  
 MPI\_Send(temp\_B.data(), n2 \* block\_n3, MPI\_DOUBLE, dest\_rank, 4, grid\_comm);  
 }  
 } else {  
 MPI\_Recv(temp\_B.data(), n2 \* block\_n3, MPI\_DOUBLE, MPI\_ANY\_SOURCE, 4, grid\_comm, MPI\_STATUS\_IGNORE);  
 }  
  
 std::vector<double> local\_C(block\_n1 \* block\_n3, 0.0);  
 for (int i1 = 0; i1 < block\_n1; ++i1)  
 for (int k = 0; k < block\_n2; ++k)  
 for (int j1 = 0; j1 < block\_n3; ++j1)  
 local\_C[i1 \* block\_n3 + j1] += temp\_A[i1 \* block\_n2 + k] \* temp\_B[k \* block\_n3 + j1];  
  
 if (rank == 0) {  
 std::vector<double> C(n1 \* n3, 0.0);  
   
 for (int row = 0; row < block\_n1; ++row)  
 for (int col = 0; col < block\_n3; ++col)  
 C[row \* n3 + col] = local\_C[row \* block\_n3 + col];  
  
  
 for (int pr = 1; pr < size; ++pr) {  
 int coords\_recv[2];  
 MPI\_Cart\_coords(grid\_comm, pr, 2, coords\_recv);  
 int ii = coords\_recv[0], jj = coords\_recv[1];  
 std::vector<double> temp\_C(block\_n1 \* block\_n3);  
 MPI\_Recv(temp\_C.data(), block\_n1 \* block\_n3, MPI\_DOUBLE, pr, 5, grid\_comm, MPI\_STATUS\_IGNORE);  
 for (int row = 0; row < block\_n1; ++row)  
 for (int col = 0; col < block\_n3; ++col)  
 C[(ii \* block\_n1 + row) \* n3 + jj \* block\_n3 + col] = temp\_C[row \* block\_n3 + col];  
 }  
  
  
// std::cout << "Matrix A:\n";  
// for (int i = 0; i < n1; ++i) {  
// for (int j = 0; j < n2; ++j)  
// std::cout << A[i \* n2 + j] << " ";  
// std::cout << "\n";  
// }  
//  
// std::cout << "\nMatrix B:\n";  
// for (int i = 0; i < n2; ++i) {  
// for (int j = 0; j < n3; ++j)  
// std::cout << B[i \* n3 + j] << " ";  
// std::cout << "\n";  
// }  
//  
// std::cout << "\nMatrix C = A \* B:\n";  
// for (int i = 0; i < n1; ++i) {  
// for (int j = 0; j < n3; ++j)  
// std::cout << C[i \* n3 + j] << " ";  
// std::cout << "\n";  
// }  
  
 std::cout << "Время выполнения: " << (MPI\_Wtime() - startTime) << " с\n";  
 } else {  
 MPI\_Send(local\_C.data(), block\_n1 \* block\_n3, MPI\_DOUBLE, 0, 5, grid\_comm);  
 }  
  
 MPI\_Finalize();  
 return 0;  
}

**Профилирование**

****

****