**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №3**

**по дисциплине «Архитектура параллельных вычислительных систем»**

Тема: «РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАНЕНИЙ ИТЕРАЦИОННЫМИ МЕТОДАМИ НА СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ»

|  |  |  |
| --- | --- | --- |
| Студенты гр. 0302 |  | Бикеев М.Р. |
|  |  | Штейнберг Э.Э. |
|  |  | Камоликов В.А. |
|  |  | Билоблоцкий В.В. |
| Преподаватель |  | Костичев С.В. |

Санкт-Петербург

2024

# ВВЕДЕНИЕ

*Цель работы*: Реализация программы по решению СЛАУ Ax = b методом Зейделя (Гаусса-Зейделя) (вариант 4)

Используемая библиотека – MPI.

*Конфигурация системы.*

Программное обеспечение:

* Windows 11 Pro 23H2 22631.4169
* Visual Studio 2022 (v143)
* Microsoft MPI v10.1.12498.52

Аппаратное окружение:

* Процессор с тактовой частотой 3.30 GHz (4.2 GHz)
* Количество ядер – 6, потоков – 12.

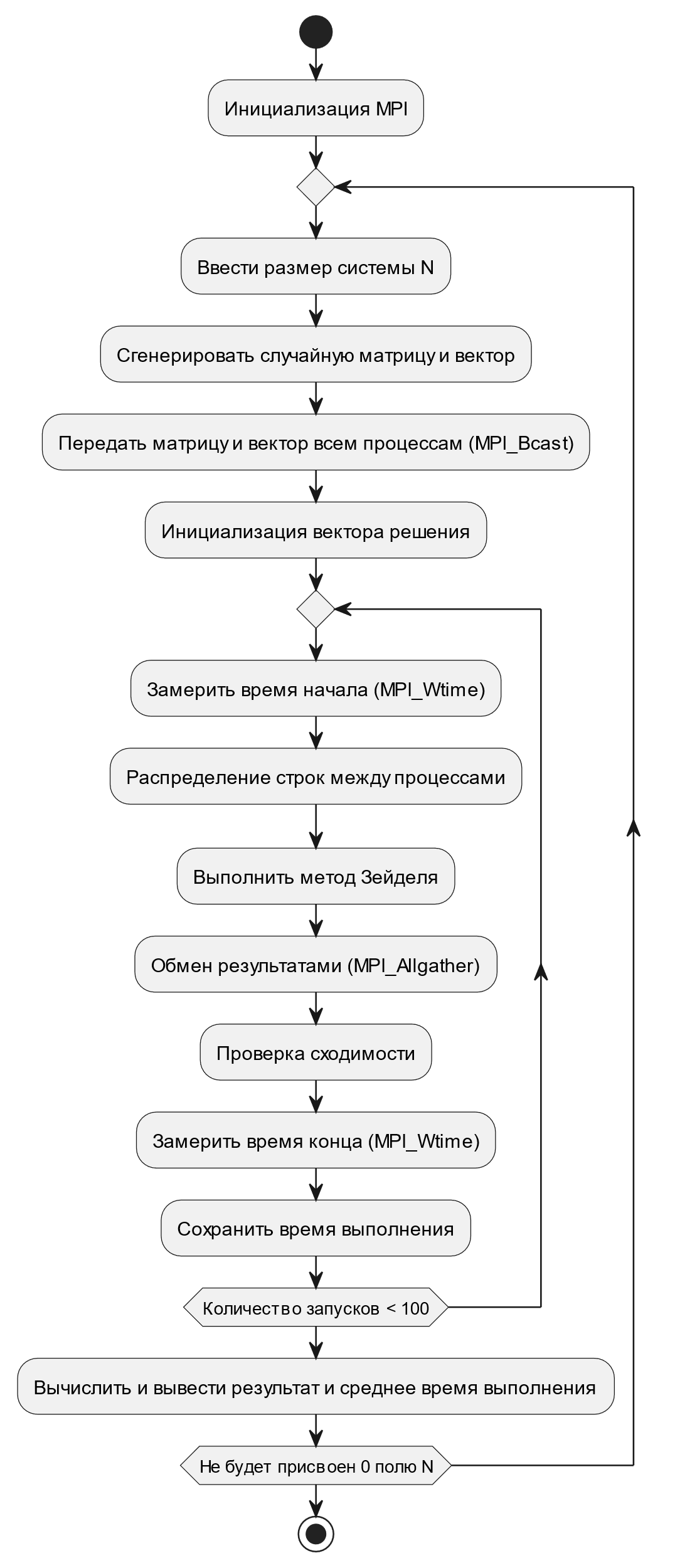
# Реализация программы

## Описание метода

Программа решает систему линейных уравнений методом Зейделя с использованием параллельных вычислений через MPI. Пользователь вводит размер системы уравнений. Программа генерирует случайную матрицу и вектор, решает задачу методом Зейделя 100 раз и выводит среднее время выполнения и результат. Время выполнения измеряется с помощью MPI\_Wtime, данные передаются между процессами через MPI-функции.

## Блок схема алгоритмов

На Рисунке 1 показана блок схема выполнения приложения.

  
Рисунок 1 – Блок схема алгоритма программы

# Результаты работы программы

## Примеры запуска программы

При заполнении данных пользователь может указать как размер системы. После ввода данных пользователь получает среднее время выполнения сортировки за 100 повторений, как показано на Рисунке 2.

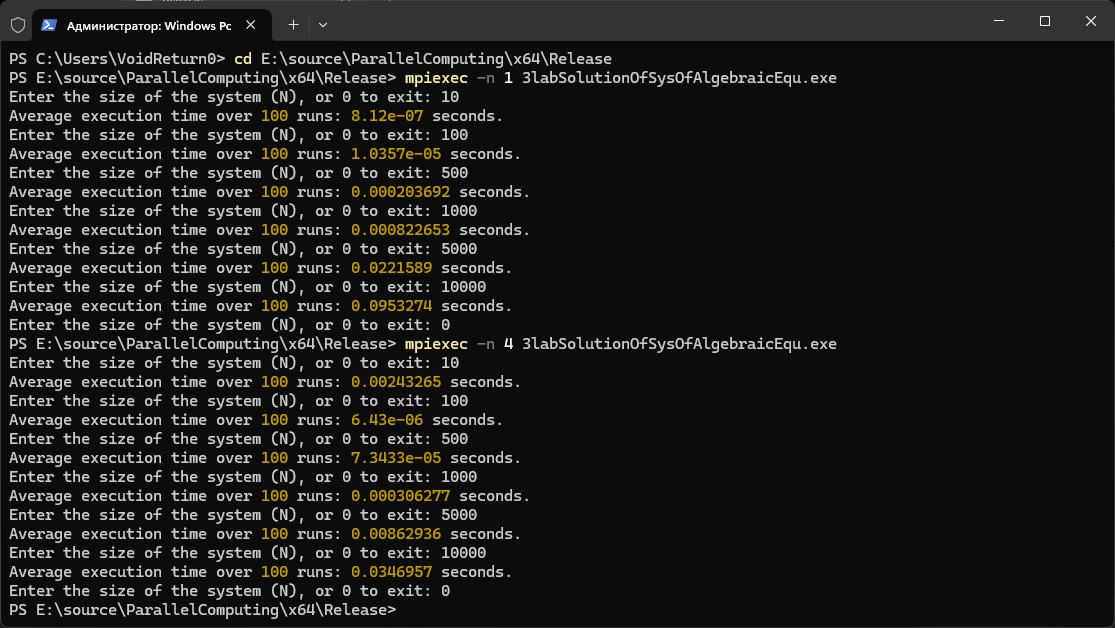


Рисунок 2 – Запуск и получение результатов на 1 процессе

## Сравнительная оценка эффективности программы

Рисунок 3 показывает результаты решение СЛАУ с разным количеством процессов. Программа становится эффективнее в зависимости от размеров решаемой системы (размерности матрицы) и количества потоков.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер системы** | **1 процесс** | **2 процесса** | **4 процесса** |  |
| 1 | 0 | 0 | 0 |  |
| 10 | 8,12E-07 | 5,00E-06 | 2,43E-03 |  |
| 100 | 1,04E-05 | 7,63E-06 | 6,43E-06 |  |
| 500 | 2,04E-04 | 1,19E-04 | 7,34E-05 |  |
| 1000 | 8,23E-04 | 4,82E-04 | 3,06E-04 |  |
| 5000 | 0,0221589 | 1,39E-02 | 0,0086294 |  |
| 10000 | 0,0953274 | 0,0548788 | 0,0346957 |  |

Рисунок 3 – Отношение выполняемого времени в зависимости от числа процессов и размеров системы

Ключевые моменты:

* На малых значениях (N ≤ 100) параллелизация не дает значительных преимуществ из-за накладных расходов на синхронизацию и обмен данными.
* На средних значениях (N = 500 - 1000) наблюдается заметное ускорение с увеличением числа потоков до 8.
* На больших значениях (N > 1000) параллелизм наиболее эффективен. Время решения значительно сокращается при использовании до 12 потоков, но чрезмерное количество потоков может повысить накладные расходы на коммуникацию.

# Текст программы

Для данной лабораторной работы было разработано консольное приложение включающая в себя 1 файл программного кода.

Файл 1 - *3lab.cpp*

#include <mpi.h>

#include <iostream>

#include <vector>

#include <cstdlib>

#include <cmath>

// Параметры задачи

const double TOLERANCE = 1e-6; // Точность

const int MAX\_ITERATIONS = 1000; // Максимальное количество итераций

const int NUM\_RUNS = 100; // Количество повторений

// Функция для генерации случайной матрицы и вектора

void generate\_random\_matrix\_and\_vector(std::vector<std::vector<double>>& A, std::vector<double>& b, int N) {

srand(time(0));

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

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

A[i][j] = rand() % 10 + 1; // случайное число от 1 до 10

}

A[i][i] += N \* 10; // увеличиваем диагональные элементы для улучшения сходимости

b[i] = rand() % 10 + 1; // случайное число от 1 до 10

}

}

// Функция для решения СЛАУ методом Зейделя

void gauss\_seidel\_parallel(const std::vector<std::vector<double>>& A, const std::vector<double>& b, std::vector<double>& x, int rank, int size) {

int N = A.size();

int local\_start = rank \* N / size; // Начало области, обрабатываемой процессом

int local\_end = (rank + 1) \* N / size; // Конец области, обрабатываемой процессом

std::vector<double> x\_old(N, 0.0); // Вектор предыдущих значений

for (int iter = 0; iter < MAX\_ITERATIONS; ++iter) {

double norm = 0.0;

for (int i = local\_start; i < local\_end; ++i) {

double sigma = 0.0;

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

if (j != i)

sigma += A[i][j] \* x[j];

}

x\_old[i] = x[i]; // Сохраняем старое значение

x[i] = (b[i] - sigma) / A[i][i]; // Обновляем значение

norm += std::pow(x[i] - x\_old[i], 2);

}

// Обмен данными между процессами

MPI\_Allgather(MPI\_IN\_PLACE, 0, MPI\_DATATYPE\_NULL, x.data(), N / size, MPI\_DOUBLE, MPI\_COMM\_WORLD);

// Считаем глобальную норму (точность)

double global\_norm;

MPI\_Allreduce(&norm, &global\_norm, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

// Проверка сходимости

if (sqrt(global\_norm) < TOLERANCE) {

break;

}

}

}

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);

while (true) {

int N;

// Ввод размера системы

std::cout << "Enter the size of the system (N), or 0 to exit: ";

std::cin >> N;

// Распространяем значение N на все процессы

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

if (N == 0) break; // Выход из цикла, если введено 0

// Инициализация матрицы и вектора

std::vector<std::vector<double>> A(N, std::vector<double>(N));

std::vector<double> b(N);

std::vector<double> x(N, 0.0); // Начальное приближение

// Генерируем случайную матрицу и вектор

generate\_random\_matrix\_and\_vector(A, b, N);

// Вывод начальных значений опционально

//std::cout << "Initial system:" << std::endl;

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

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

// std::cout << A[i][j] << " ";

// }

// std::cout << " | " << b[i] << std::endl;

//}

// Распространяем матрицу и вектор b на все процессы

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

MPI\_Bcast(A[i].data(), N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

MPI\_Bcast(b.data(), N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Переменная для хранения общего времени

double total\_time = 0.0;

// Выполняем решение 100 раз

for (int run = 0; run < NUM\_RUNS; ++run) {

// Замер времени с использованием MPI

double start\_time = MPI\_Wtime();

// Решаем СЛАУ методом Зейделя

gauss\_seidel\_parallel(A, b, x, rank, size);

// Замер времени завершен

double end\_time = MPI\_Wtime();

// Суммируем время выполнения

total\_time += (end\_time - start\_time);

}

// Вывод результата

// Выводим решение (после последнего запуска) опционально

//std::cout << "Solution: ";

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

// std::cout << x[i] << " ";

//}

//std::cout << std::endl;

// Вычисляем и выводим среднее время выполнения

double average\_time = total\_time / NUM\_RUNS;

std::cout << "Average execution time over " << "\033[33m" << NUM\_RUNS << "\033[0m" << " runs: " << "\033[33m" << average\_time << "\033[0m" << " seconds." << std::endl;

}

MPI\_Finalize();

return 0;

}

# Выводы

В процессе выполнения лабораторной работы была разработана программа для решения системы линейных уравнений методом Зейделя с использованием параллельных вычислений с помощью библиотеки MPI.

Программа эффективно решает системы линейных уравнений методом Зейделя с использованием MPI, особенно для задач средней и большой размерности. При малых размерностях затраты на межпроцессорные коммуникации могут нивелировать прирост от параллелизации. Оптимальное количество потоков для большинства задач лежит в диапазоне от 4 до 8, при этом для крупных систем использование до 12 потоков дает заметные улучшения в производительности.