**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 уравнений итерационными методами на системах с общей памятью

|  |  |  |
| --- | --- | --- |
| Студенты гр. 0301 |  | Прохоров Б.В. |
|  |  | Михайлов В.А. |
|  |  | Козлов Т.В. |
|  |  | Логунов О.Ю. |
|  |  | Машенков И.А. |
| Преподаватель |  | Костичев С.В. |

Санкт-Петербург

2024

## Цель работы.

Практическое освоение методов решения систем линейных алгебраических уравнений (СЛАУ) итерационными методами на вычислительных системах с общей памятью.

## Задание.

1. В зависимости от номера варианта задания разработать алгоритмы решения СЛАУ для последовательных и параллельных вычислений.
2. Написать и отладить программы на языке С++, реализующие разработанные алгоритмы последовательных и параллельных вычислений с использованием биб-лиотек OpenMP и mpi.
3. Запустить программы для следующих значений размерности СЛАУ: 5, 10, 100, 500, 1000, 5000, 10000
4. Оценить размерность СЛАУ, при которой эффективнее использовать алгоритмы последовательного и параллельного вычислений для разного числа потоков (по крайней мере для меньшего, равного и большего, чем число процессоров). Под эффективностью понимается время работы программы на матрице.

Вариант 3.

Решение СЛАУ Ax = b методом Зейделя (Гаусса-Зейделя) с использованием OpenMP.

## Выполнение работы.

*Программное и аппаратное окружение*

Программное окружение при выполнении работы:

1. Операционная система: Windows 10 Pro 64bit.
2. Программа выполняется в среде WSL (Windows Subsystem for Linux), что позволяет запускать Linux-программы в Windows.
3. На WSL установлена версия дистрибутива Linux (Ubuntu 20.04).
4. Компилятор g++ (версии GCC), поддерживающий флаг -fopenmp для работы с OpenMP.
5. Библиотека MPI для распараллеливания вычислений (пакеты openmpi-bin openmpi-common libopenmpi-dev).
6. Python 3.11.4 (пакеты pandas и matplotlib).
7. IDE для разработки – Visual Studio Code с подключением к WSL.
8. Управление компиляцией и запуском программ осуществляется через командную строку WSL.

Аппаратное окружение:

1. Процессор 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz.
2. Установленная память (ОЗУ) 48 ГБ (47,7 ГБ доступно).
3. Тип системы 64-разрядная операционная система, процессор x64.

*Описание метода снятия метрик производительности*

Происходит выполнение следующих шагов для измерения времени решения системы линейных уравнений (СЛАУ) методом Зейделя (методом Гаусса-Зейделя) как для последовательного, так и для параллельного алгоритмов с использованием OpenMP:

Инициализируется массив sizes содержит различные значения размерности СЛАУ (5, 10, 100, 500, 1000, 5000, 10000), для которых будет выполнено измерение времени.

Инициализируется массив thread\_counts определяет количество потоков (1, 2, 4, 8), которые будут использоваться для параллельной версии метода.

Инициализируется переменная для хранения результатов results — вектор кортежей, который будет хранить данные для каждого эксперимента: размерность, количество потоков, время выполнения последовательного алгоритма и время выполнения параллельного алгоритма.

Происходит запуск экспериментов. Для каждого значения size из массива sizes вызывается функция generate\_random\_matrix, которая создаёт случайную матрицу с указанной размерностью и правую часть (вектор b). Эта матрица представляет собой СЛАУ, которую нужно решить. Решение СЛАУ и измерение времени. Для каждого количества потоков из массива thread\_counts создаются начальные векторы x\_serial и x\_parallel, инициализированные нулями, для хранения решений СЛАУ. Измерение времени последовательного алгоритма: функция measure\_time замеряет время выполнения последовательной функции gauss\_seidel\_serial и сохраняет результат в serial\_time. Измерение времени параллельного алгоритма: measure\_time замеряет время выполнения параллельной функции gauss\_seidel\_parallel с использованием текущего количества потоков и сохраняет результат в parallel\_time. Сохранение результатов для текущей размерности и количества потоков: кортеж (size, threads, serial\_time, parallel\_time) добавляется в вектор results. Результаты выводятся в консоль для наглядности.

По завершении всех экспериментов данные из results сохраняются в файл gauss\_seidel\_results.csv с помощью функции save\_results\_to\_csv. Этот CSV-файл содержит информацию о времени выполнения последовательного и параллельного методов для каждой размерности и каждого количества потоков.

Программа завершает работу после сохранения всех результатов.

После выполнения всех вычислений данные из CSV-файлов обрабатываются с помощью Python-библиотеки pandas, и с помощью matplotlib строится график, на котором визуализируются зависимости времени выполнения от размерности СЛАУ для каждого количества потоков, где сравнивается время выполнения последовательного и параллельного методов Зейделя. График сохраняется в формате PNG (см. рис. 1).

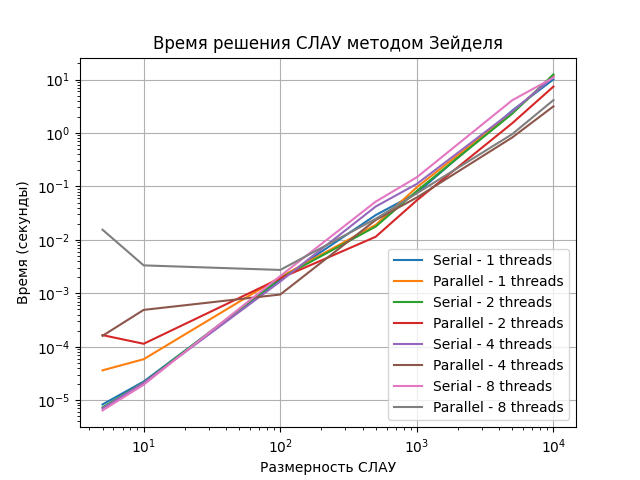


Рисунок 1 – График зависимостей времени выполнения последовательного и параллельного методов Зейделя для разного количества потоков от размерности СЛАУ

Блок-схемы алгоритмов с пояснения

Блок-схема последовательного алгоритма метода Зейделя на рис. 2.



Рисунок 2 – Блок-схема последовательного алгоритма метода Зейделя

Задаются исходные значения для матрицы A, вектора b и вектора решения x, который заполняется нулями. На каждой итерации происходит сохранение текущего решения в prev\_x. Затем для каждой строки матрицы A рассчитывается новый элемент x[i], исключая диагональный элемент. По завершении итерации рассчитывается ошибка на основе разницы между x и prev\_x. Цикл продолжается, пока ошибка не станет меньше TOLERANCE или не превысит MAX\_ITER. Алгоритм возвращает полученный вектор x как решение.

Блок-схема параллельного алгоритма метода Зейделя представлена на рис. 3.

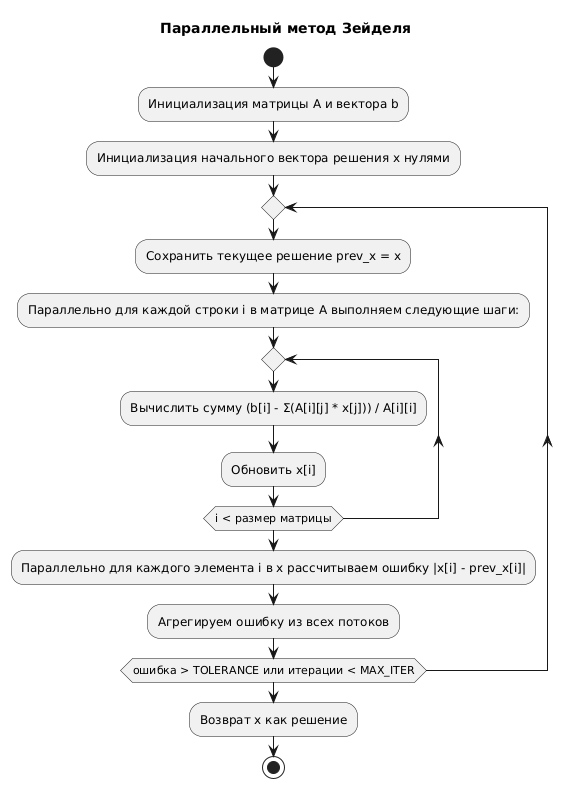


Рисунок 3 – Блок-схема параллельного алгоритма сортировки выбором

Задаются матрица A, вектор b, и вектор x с начальными нулями. Каждая строка i обрабатывается параллельно для вычисления нового значения x[i]. Ошибка также рассчитывается параллельно по элементам вектора x, и результаты суммируются. Цикл завершается, если ошибка становится меньше TOLERANCE или превысит MAX\_ITER. Возвращается вектор x как результат решения.

Сравнительная оценка эффективности

Расчёт ускорения (Speedup) программы для параллельного алгоритма по сравнению с последовательным выполнялся c помощью формулы .

Таблица 1 – Сравнительная оценка эффективности программы для различных значений размерности СЛАУ для разного числа потоков

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность СЛАУ | Количество потоков | Время последовательного алгоритма (сек) | Время параллельного алгоритма (сек) | Ускорение |
| 5 | 1 | 8.328e-06 | 3.6217e-05 | 0.23 |
| 5 | 2 | 7.327e-06 | 0.000165023 | 0.04 |
| 5 | 4 | 7.043e-06 | 0.000160279 | 0.04 |
| 5 | 8 | 6.441e-06 | 0.0154748 | 0.0004 |
| 10 | 1 | 2.2325e-05 | 5.8446e-05 | 0.38 |
| 10 | 2 | 2.1392e-05 | 0.00011371 | 0.19 |
| 10 | 4 | 2.1114e-05 | 0.000489406 | 0.04 |
| 10 | 8 | 1.946e-05 | 0.00333085 | 0.006 |
| 100 | 1 | 0.0018142 | 0.00201365 | 0.90 |
| 100 | 2 | 0.00191333 | 0.00191032 | 1.00 |
| 100 | 4 | 0.00166483 | 0.000948791 | 1.75 |
| 100 | 8 | 0.00210596 | 0.00273793 | 0.77 |
| 500 | 1 | 0.0291095 | 0.0191315 | 1.52 |
| 500 | 2 | 0.0177679 | 0.011444 | 1.55 |
| 500 | 4 | 0.0420932 | 0.0235883 | 1.78 |
| 500 | 8 | 0.0520065 | 0.0247947 | 2.10 |
| 1000 | 1 | 0.0752927 | 0.0966314 | 0.78 |
| 1000 | 2 | 0.0822728 | 0.0549792 | 1.50 |
| 1000 | 4 | 0.111473 | 0.0621111 | 1.80 |
| 1000 | 8 | 0.148539 | 0.0744816 | 2.00 |
| 5000 | 1 | 2.63521 | 2.41538 | 1.09 |
| 5000 | 2 | 2.30485 | 1.52943 | 1.51 |
| 5000 | 4 | 2.54068 | 0.82664 | 3.07 |
| 5000 | 8 | 4.11082 | 0.954967 | 4.30 |
| 10000 | 1 | 10.0643 | 11.8188 | 0.85 |
| 10000 | 2 | 12.4825 | 7.39134 | 1.69 |
| 10000 | 4 | 11.2136 | 3.15345 | 3.56 |
| 10000 | 8 | 10.8477 | 4.12252 | 2.63 |

## Тестирование.

На рис. 4 представлен пример работы программы.

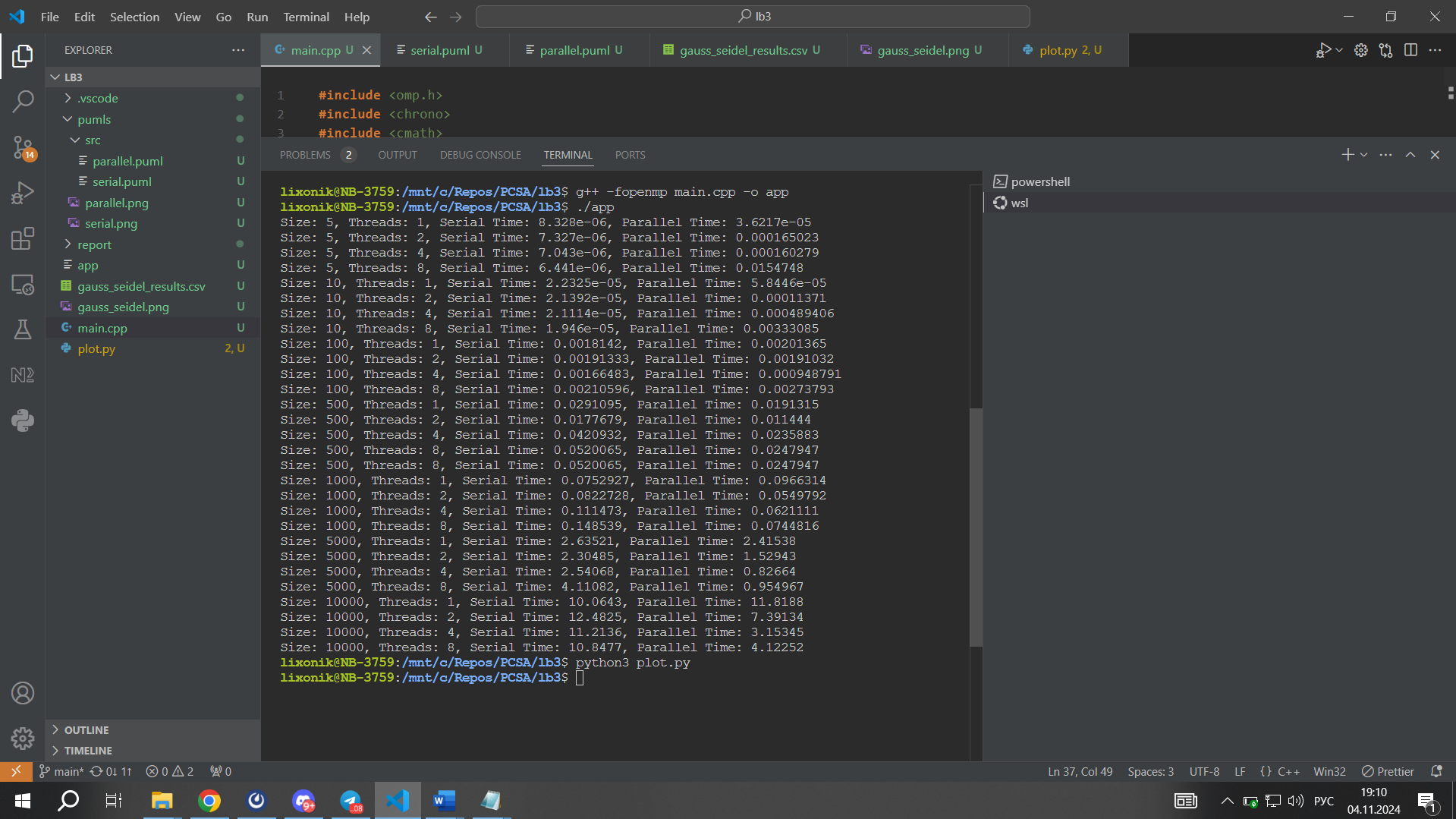


Рисунок 2 – Пример работы программы

## Выводы.

На практике были освоены методы решения систем линейных алгебраических уравнений (СЛАУ) итерационными методами на вычислительных системах с общей памятью.

Для небольших матриц использование параллельного метода оказывается менее эффективным, чем последовательный метод. Это связано с накладными расходами на создание и синхронизацию потоков, которые превосходят выигрыш от параллельного выполнения. Ускорение в этом случае ниже 1, что указывает на замедление.

Начиная с размерности 100, параллельный метод начинает показывать ускорение, хотя оно и не всегда устойчиво. Для матрицы размерности 500 при использовании 4 и 8 потоков наблюдается стабильное ускорение, превышающее 1, что указывает на прирост производительности параллельного метода. Однако на этой стадии накладные расходы на управление потоками всё ещё имеют заметное влияние, и ускорение хоть и положительное, но незначительное.

Для больших матриц параллельный метод Гаусса-Зейделя показывает существенное ускорение, особенно при 4 и 8 потоках. В случае размерности 5000 и 10000 ускорение достигает наибольших значений (до 4.3 раз быстрее для 5000 при 8 потоках и 3.56 раз для 10000 при 4 потоках). Это говорит о том, что параллельные вычисления действительно начинают оправдываться при высоких объёмах данных. При этом следует отметить, что при 8 потоках для некоторых матриц ускорение немного падает, что может быть связано с избыточной конкуренцией за ресурсы и накладными расходами на синхронизацию.

Параллельный метод эффективен для решения СЛАУ больших размерностей, где выигрыш от распределения вычислений между потоками перекрывает накладные расходы. Для малых и средних матриц рекомендуется использовать последовательный метод, так как он демонстрирует более стабильное время выполнения и не требует дополнительной синхронизации.

# Приложение А Исходный код программы

Название файла: main.cpp

#include <omp.h>

#include <chrono>

#include <cmath>

#include <fstream>

#include <functional>

#include <iostream>

#include <vector>

const double TOLERANCE = 1e-10; // критерий сходимости

const int MAX\_ITER = 10000; // максимальное количество итераций

template <typename Function>

double measure\_time(Function fn) {

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

fn();

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

std::chrono::duration<double> duration = end - start;

return duration.count();

}

std::vector<std::vector<double>> generate\_random\_matrix(int size) {

std::vector<std::vector<double>> matrix(size, std::vector<double>(size + 1));

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

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

matrix[i][j] =

(i == j) ? 1.0 + size : static\_cast<double>(rand()) / RAND\_MAX;

}

matrix[i][size] = static\_cast<double>(rand()) / RAND\_MAX; // вектор b

}

return matrix;

}

bool gauss\_seidel\_serial(const std::vector<std::vector<double>> &matrix,

std::vector<double> &x) {

int size = matrix.size();

std::vector<double> prev\_x(size, 0.0);

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

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

double sum = matrix[i][size];

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

if (i != j) sum -= matrix[i][j] \* x[j];

}

x[i] = sum / matrix[i][i];

}

double error = 0.0;

for (int i = 0; i < size; ++i) error += std::abs(x[i] - prev\_x[i]);

if (error < TOLERANCE) return true;

prev\_x = x;

}

return false;

}

bool gauss\_seidel\_parallel(const std::vector<std::vector<double>> &matrix,

std::vector<double> &x, int num\_threads) {

int size = matrix.size();

std::vector<double> prev\_x(size, 0.0);

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

#pragma omp parallel for num\_threads(num\_threads)

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

double sum = matrix[i][size];

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

if (i != j) sum -= matrix[i][j] \* x[j];

}

x[i] = sum / matrix[i][i];

}

double error = 0.0;

#pragma omp parallel for reduction(+ : error) num\_threads(num\_threads)

for (int i = 0; i < size; ++i) error += std::abs(x[i] - prev\_x[i]);

if (error < TOLERANCE) return true;

prev\_x = x;

}

return false;

}

void save\_results\_to\_csv(

const std::string &filename,

const std::vector<std::tuple<int, int, double, double>> &results) {

std::ofstream file(filename);

file << "Size,Threads,SerialTime,ParallelTime\n";

for (const auto &[size, threads, serial\_time, parallel\_time] : results) {

file << size << "," << threads << "," << serial\_time << ","

<< parallel\_time << "\n";

}

}

int main() {

std::vector<int> sizes = {5, 10, 100, 500, 1000, 5000, 10000};

std::vector<int> thread\_counts = {1, 2, 4, 8};

std::vector<std::tuple<int, int, double, double>> results;

for (int size : sizes) {

auto matrix = generate\_random\_matrix(size);

for (int threads : thread\_counts) {

std::vector<double> x\_serial(size, 0.0), x\_parallel(size, 0.0);

double serial\_time =

measure\_time([&]() { gauss\_seidel\_serial(matrix, x\_serial); });

double parallel\_time = measure\_time(

[&]() { gauss\_seidel\_parallel(matrix, x\_parallel, threads); });

results.emplace\_back(size, threads, serial\_time, parallel\_time);

std::cout << "Size: " << size << ", Threads: " << threads

<< ", Serial Time: " << serial\_time

<< ", Parallel Time: " << parallel\_time << "\n";

}

}

save\_results\_to\_csv("gauss\_seidel\_results.csv", results);

return 0;

}

Название файла: plot.py

import pandas as pd

import matplotlib.pyplot as plt

data = pd.read\_csv('gauss\_seidel\_results.csv')

for threads in data['Threads'].unique():

subset = data[data['Threads'] == threads]

plt.plot(subset['Size'], subset['SerialTime'], label=f'Serial - {threads} threads')

plt.plot(subset['Size'], subset['ParallelTime'], label=f'Parallel - {threads} threads')

plt.xlabel('Размерность СЛАУ')

plt.ylabel('Время (секунды)')

plt.title('Время решения СЛАУ методом Зейделя')

plt.legend()

plt.xscale('log')

plt.yscale('log')

plt.grid(True)

plt.savefig('gauss\_seidel.png')