МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ВТ

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Архитектура параллельных вычислительных систем»
Тема: Решение систем линейных алгебраических уравнений прямыми 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 прямыми 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 C++, реализующие разработанные алгоритмы последовательных и параллельных вычислений с использованием библиотек 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.
5. Недостаток прямых методов заключается в том, что погрешность накапливается в процессе вычисления. Оцените погрешность вычислений при различных размерностях СЛАУ.

Вариант 3.

Решение СЛАУ Ax = b методом Гаусса-Жордана с использованием библиотеки MPI.

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

Для оценки производительности алгоритма выбираются ключевые метрики, которые наиболее полно отражают эффективность работы программы. Основной метрикой является время выполнения (execution time), которое измеряется для каждого алгоритма (последовательного и параллельного) для различных входных данных и разного числа потоков/процессов. Это позволяет понять, как изменяется производительность с ростом размерности задачи и числа потоков.

Для корректной оценки производительности важно использовать разнообразные входные данные, которые могут отражать реальные условия работы алгоритма. В данном случае данные генерируются случайным образом для различных размерностей матрицы системы (5, 10, 100, 500, 1000).

Время выполнения алгоритма замеряется с использованием высокоточных часов, таких как std::chrono::high_resolution_clock в C++. Это позволяет точно измерить продолжительность работы алгоритма, включая все его этапы (разложение матрицы и обратный ход). Для каждого размера матрицы и количества потоков/процессов замеряется время работы как для последовательного, так и для параллельного выполнения.

Проводятся замеры времени для различных конфигураций системы:

- Для каждой размерности матрицы (5, 10, 100, 500, 1000) запускается алгоритм как в последовательном, так и в параллельном варианте. Размерности 5000 и 10000 не рассматривались в силу чрезвычайно длительного времени работы программы.
- Параллельная версия тестируется с разным количеством процессов/потоков (1, 2, 4, 8), чтобы понять, как производительность зависит от числа используемых вычислительных ресурсов.

В случае параллельных вычислений, использующих MPI, важно правильно синхронизировать процессы между собой. Например, каждый процесс должен корректно обмениваться данными с другими процессами (при делении работы и сборе результатов) и синхронизировать шаги алгоритма, чтобы избежать ошибок. Это также должно быть учтено при измерении времени, чтобы все процессы завершились одновременно и время замера было точным.

Результаты замеров времени выполнения для каждой комбинации размерности матрицы и числа потоков/процессов записываются в CSV-файл. Это позволяет сохранить данные в структурированном виде для дальнейшего анализа. После этого можно использовать средства визуализации, такие как графики (с помощью Python и библиотеки matplotlib), чтобы проанализировать зависимость времени выполнения от размерности задачи и числа используемых потоков/процессов. График сохраняется в формате PNG (см. рис. 1).

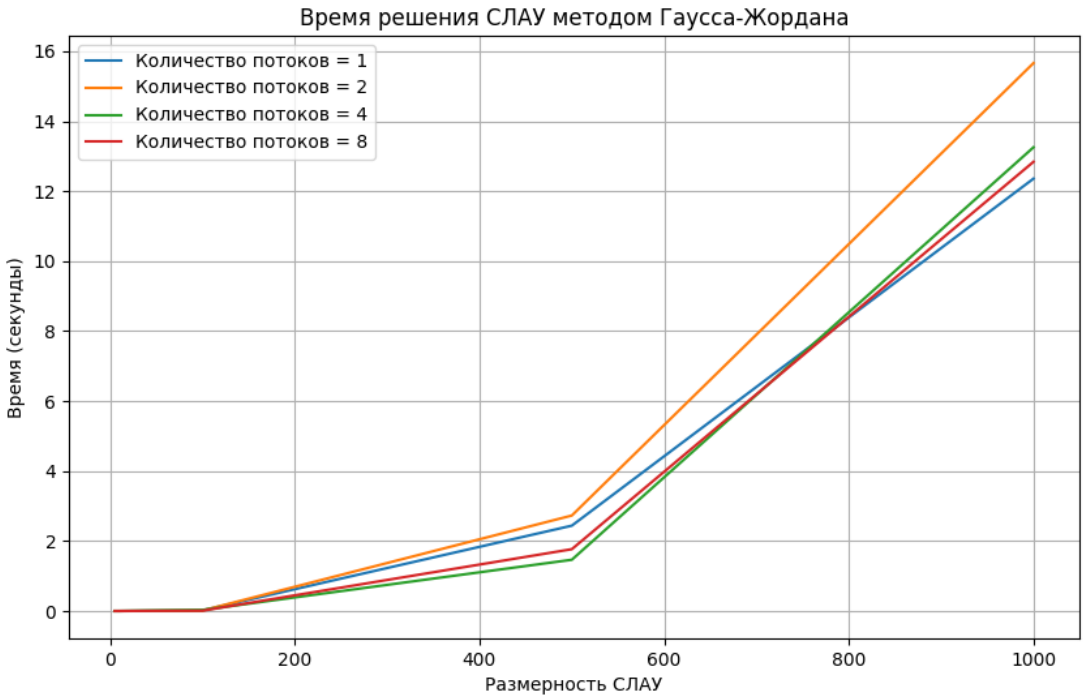

Рисунок 1 – График зависимостей времени выполнения последовательного и параллельного методов Гаусса-Жордана для разного количества потоков от размерности СЛАУ

Блок-схемы алгоритмов с пояснения

Блок-схема последовательного алгоритма метода Гаусса-Жордана на рис. 2.

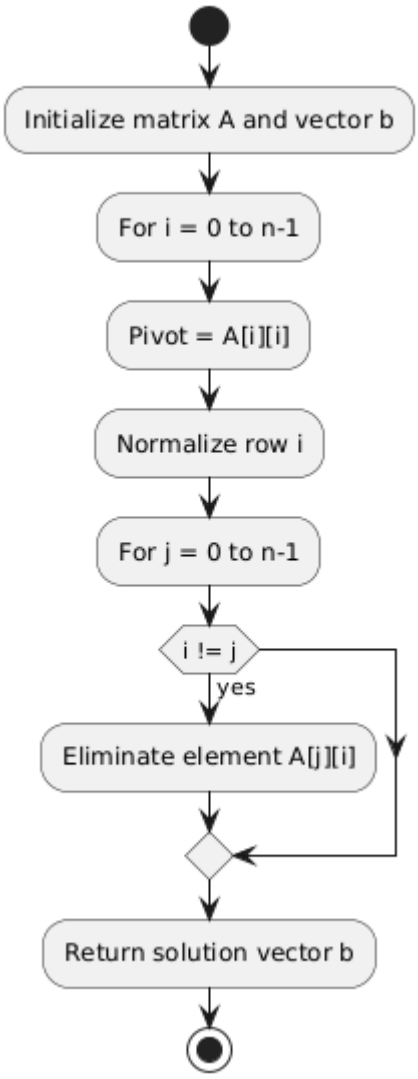

Рисунок 2 – Блок-схема последовательного алгоритма метода Гаусса-Жордана

Сначала происходит инициализация матрицы и вектора. Загружаются данные системы линейных уравнений, представленные матрицей A и вектором правых частей b.

Далее для каждой строки матрицы выполняются операции приведения с помощью цикл по строкам (i = 0 to n-1).

Каждый элемент строки нормализуется так, чтобы элемент на главной диагонали стал равен 1.

Для каждой строки, не являющейся текущей, вычитаются пропорциональные значения, чтобы элементы ниже диагонали стали равны 0.

После выполнения всех шагов получается решение системы в векторе b.

Блок-схема параллельного алгоритма метода Гаусса-Жордана представлена на рис. 3.

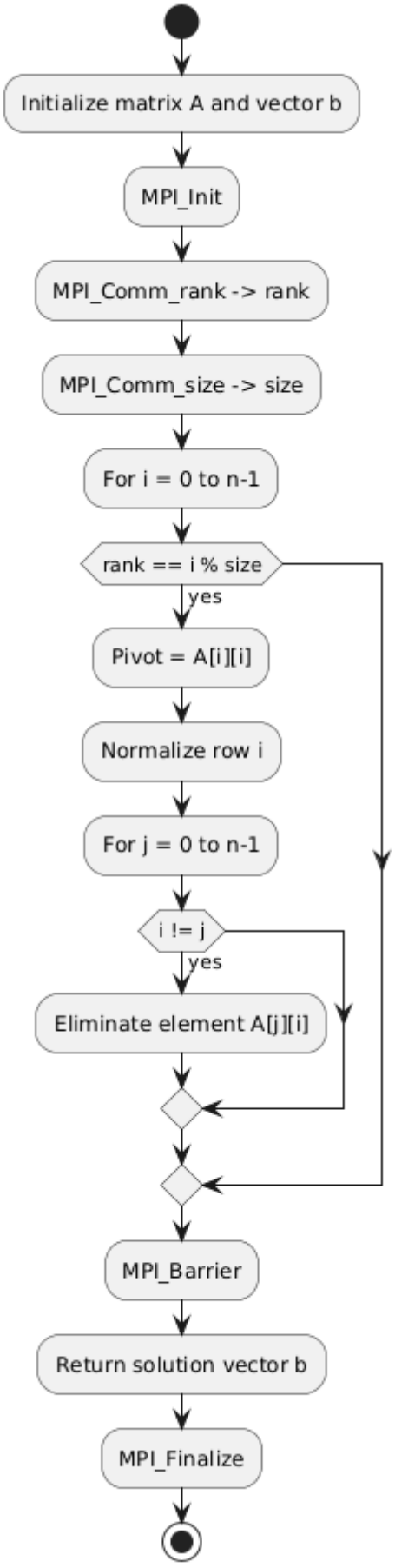

Рисунок 3 – Блок-схема параллельного алгоритма метода Гаусса-Жордана

Сначала происходит инициализация матрицы и вектора. Загружаются данные системы линейных уравнений, как и в последовательном алгоритме.

Инициализируются процессы MPI, каждый процесс получает свой уникальный идентификатор (rank), а также общее количество процессов (size).

Для каждой строки посредством цикла по строкам (i = 0 to n-1) проверяется, должен ли текущий процесс работать с этой строкой (по модулю от номера строки и общего количества процессов).

Процесс выполняет нормализацию строки и исключение элементов из других строк, как в последовательном алгоритме.

Используется MPI_Barrier для синхронизации всех процессов, чтобы каждый процесс завершил свою работу до перехода к следующей строке.

После выполнения всех шагов, каждый процесс завершает выполнение и отправляет результаты.

Сравнительная оценка эффективности

Расчёт ускорения (Speedup) программы для параллельного алгоритма по сравнению с последовательным выполнялся с помощью формулы $S = \frac{T_{serial}}{T_{parallel}}$.

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

Размерность СЛАУ	Количество потоков	Время последовательного алгоритма (секунды)	Время параллельного алгоритма (секунды)	Ускорение
5	1	0.000232464	0.000232464	1.00
5	2	0.000232464	0.00002628	8.85
5	4	0.000232464	0.000021979	10.58
5	8	0.000232464	0.00007222	3.22

10	1	0.00007133	0.00007133	1.00
10	2	0.00007133	0.000106341	0.67
10	4	0.00007133	0.000064747	1.10
10	8	0.00007133	0.000055433	1.29
100	1	0.0183878	0.0183878	1.00
100	2	0.0183878	0.014387	1.28
100	4	0.0183878	0.0303942	0.60
100	8	0.0183878	0.00900652	2.04
500	1	2.44	2.44	1.00
500	2	2.44	2.72968	0.89
500	4	2.44	1.46547	1.66
500	8	2.44	1.76733	1.38
1000	1	12.3579	12.3579	1.00
1000	2	12.3579	15.6608	0.79
1000	4	12.3579	13.2534	0.93
1000	8	12.3579	12.8392	0.96

Тестирование.

Программа отрабатывает демонстрационный сценарий при передаче флага demo во время запуска программы.

На рис. 4-7 представлен пример работы программы.

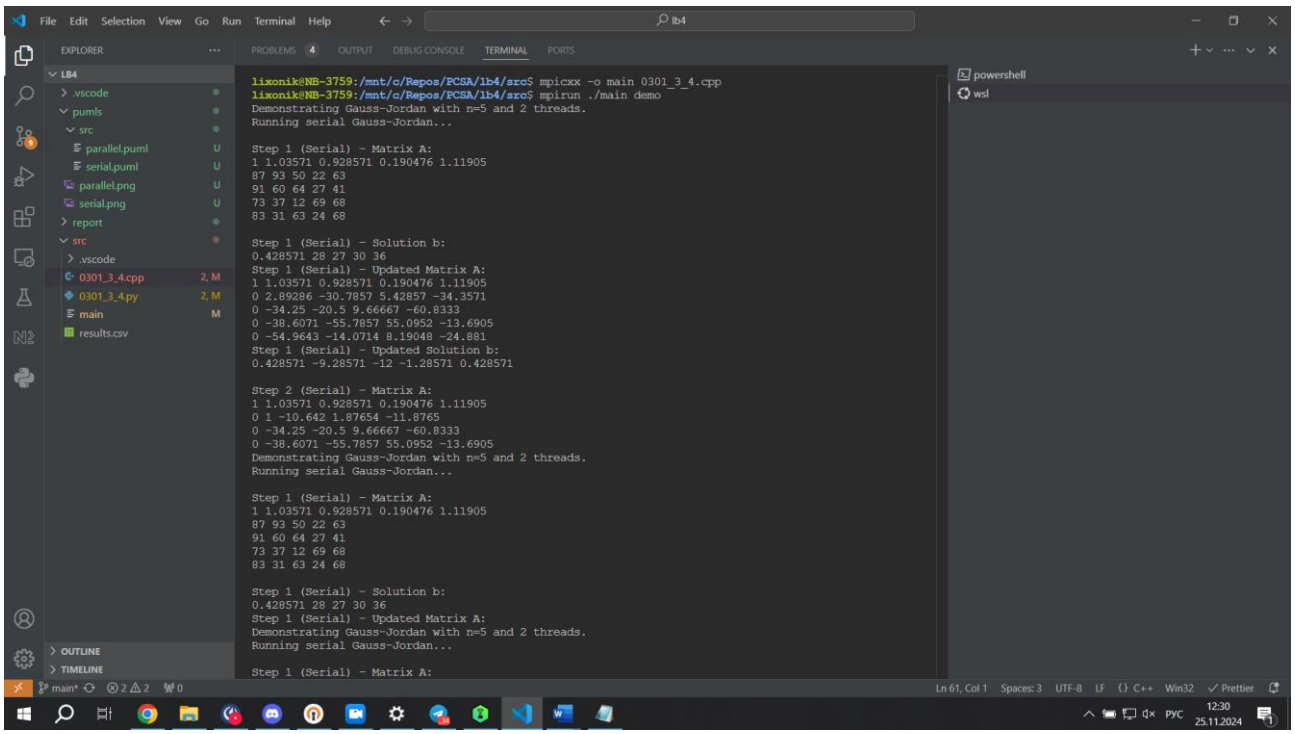

Рисунок 4 – Пример работы программы. Начало демонстрации работы последовательного метода Гаусса-Жордана

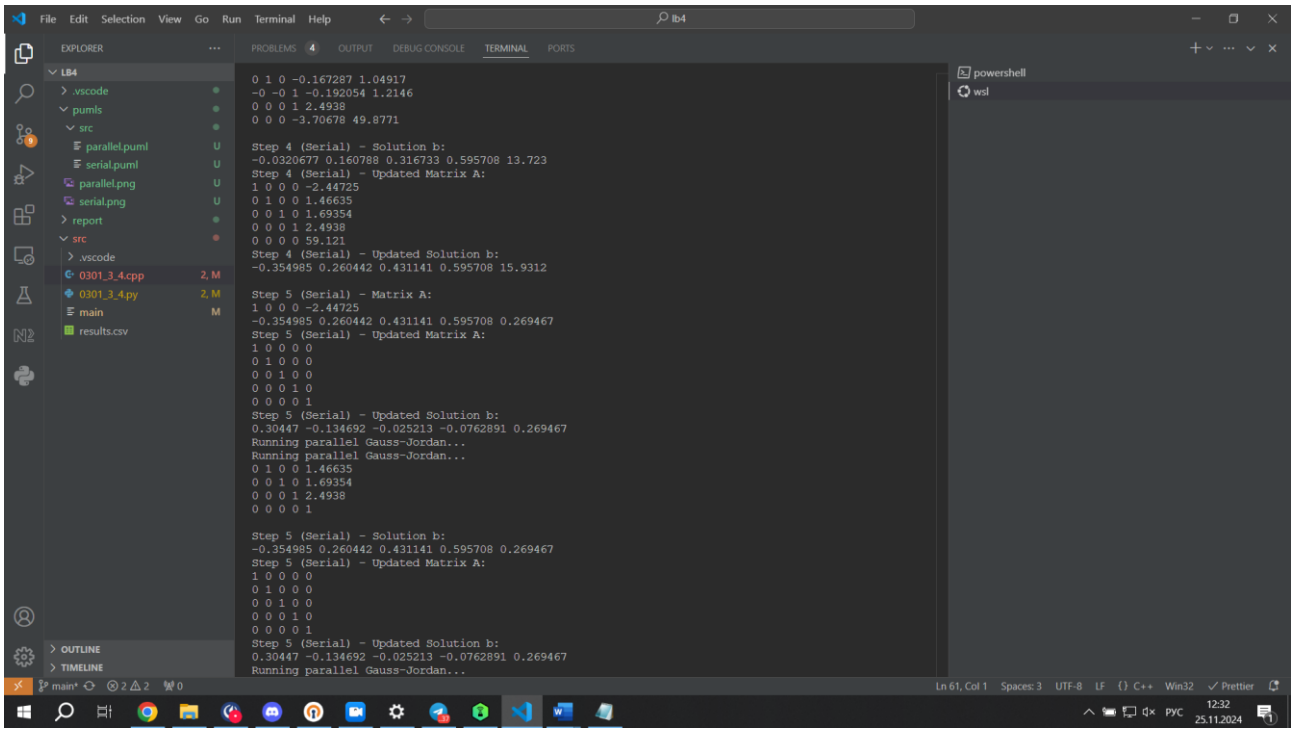

Рисунок 5 – Пример работы программы. Конец демонстрации работы последовательного метода Гаусса-Жордана

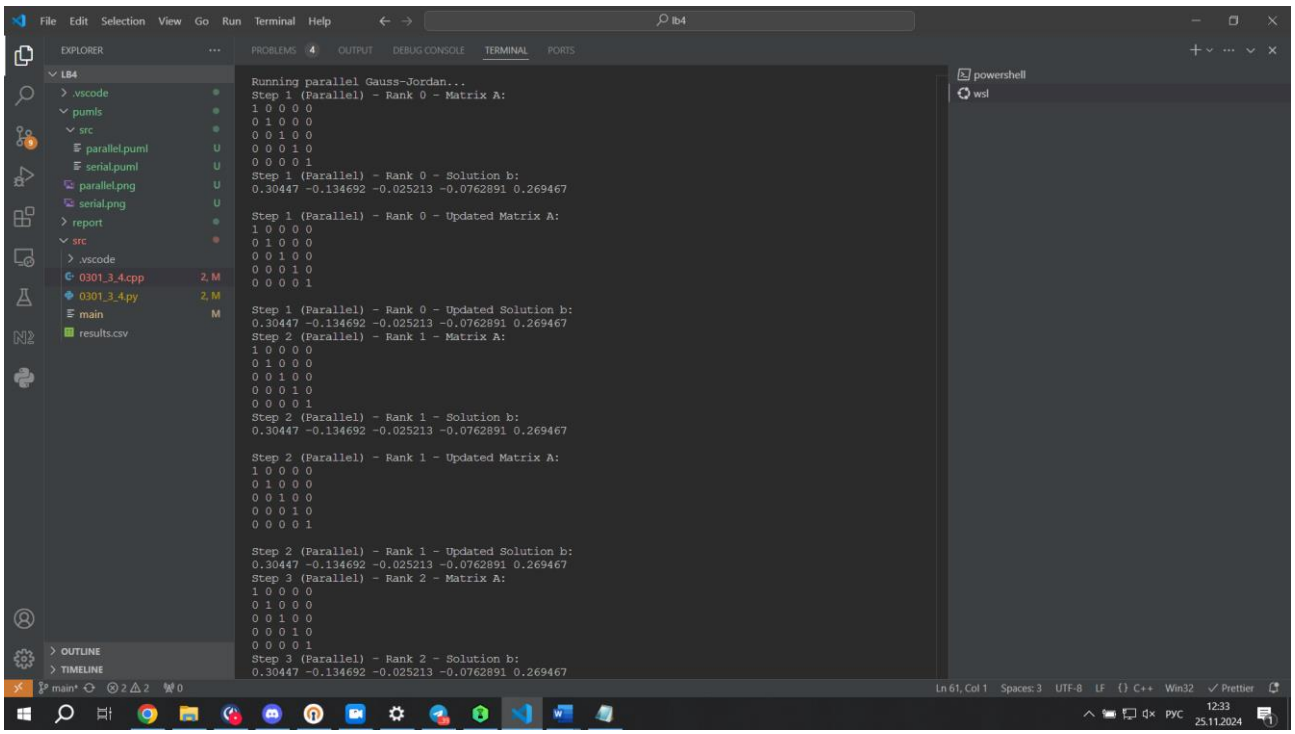

Рисунок 6 – Пример работы программы. Начало демонстрации работы параллельного метода Гаусса-Жордана

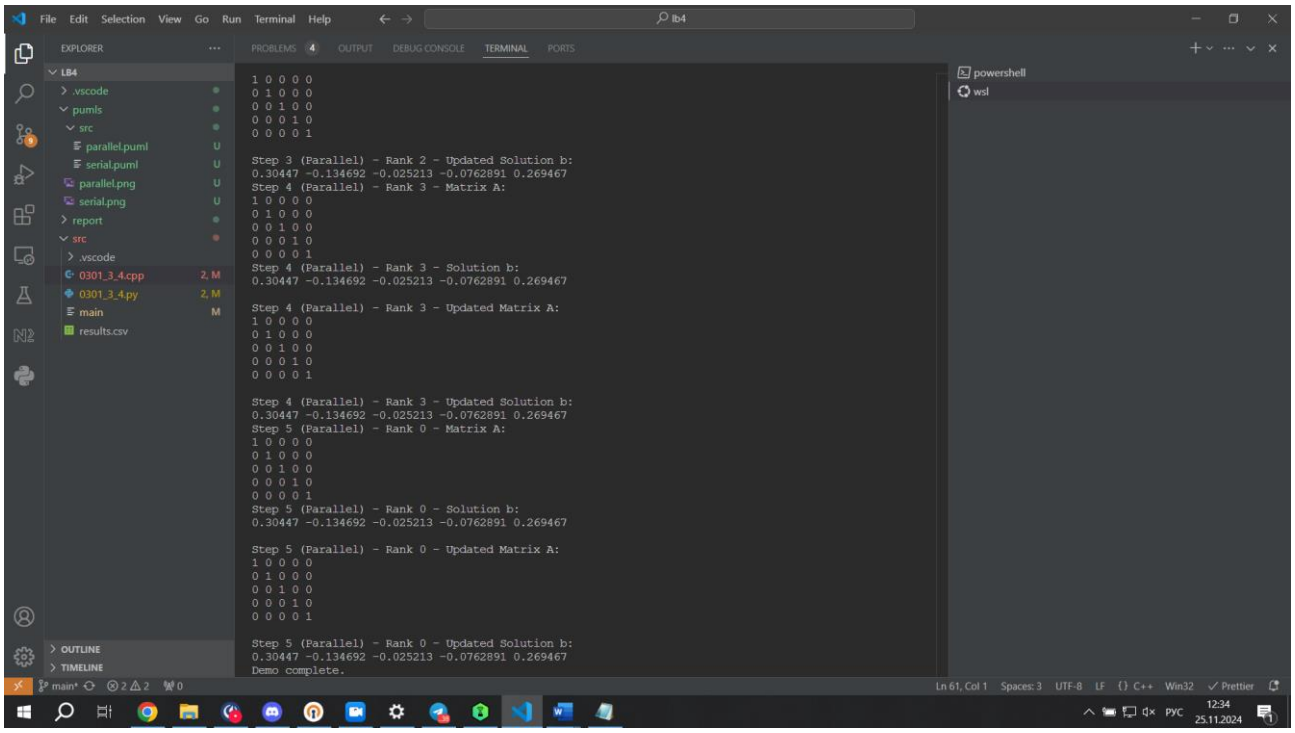

Рисунок 7 – Пример работы программы. Конец демонстрации работы параллельного метода Гаусса-Жордана

Выводы.

На практике были освоены методы решения систем линейных алгебраических уравнений прямыми методами на вычислительных системах с общей памятью.

Ускорение значительно зависит от размерности задачи и количества потоков. Для малых размерностей (5 или 10) ускорение незначительное, так как накладные расходы на параллельную обработку данных могут превышать выйгрыш от параллельного выполнения.

Для больших размерностей (1000) ускорение заметно возрастает, особенно при 8 потоках, когда выйгрыш от параллельности становится очевидным.

В некоторых случаях (для размерности 100) увеличение числа потоков приводит к ухудшению производительности, что может быть связано с недостаточной эффективностью параллелизма для данного размера задачи или накладными расходами на синхронизацию процессов.

Параллельный метод эффективен для решения СЛАУ больших размерностей, где выйгрыш 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: 0301_3_4.cpp

```
#include <mpi.h>
#include <chrono>
#include <cstring>
#include <fstream>
#include <iostream>
#include <vector>

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();
}

void print_matrix(const std::vector<std::vector<double>>& A) {
   int n = A.size();
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         std::cout << A[i][j] << " ";
      }
      std::cout << std::endl;
   }
}

void print_solution(const std::vector<double>& b) {
   for (const double& value : b) {
      std::cout << value << " ";
   }
   std::cout << std::endl;
}

void gauss_jordan_serial(std::vector<std::vector<double>>& A,
                         std::vector<double>& b) {
   int n = A.size();

   for (int i = 0; i < n; ++i) {
      double pivot = A[i][i];
      for (int j = 0; j < n; ++j) {
         A[i][j] /= pivot;
      }
      b[i] /= pivot;

      std::cout << std::endl
                << "Step " << i + 1 << " (Serial) - Matrix A:" <<
std::endl;
      print_matrix(A);
      std::cout << std::endl
```

```
                    << "Step " << i + 1 << " (Serial) - Solution b:" <<
std::endl;
            print_solution(b);

            for (int j = 0; j < n; ++j) {
               if (i != j) {
                  double factor = A[j][i];
                  for (int k = 0; k < n; ++k) {
                     A[j][k] -= A[i][k] * factor;
                  }
                  b[j] -= b[i] * factor;
               }
            }

            std::cout << "Step " << i + 1
                      << " (Serial) - Updated Matrix A:" << std::endl;
            print_matrix(A);
            std::cout << "Step " << i + 1
                      << " (Serial) - Updated Solution b:" << std::endl;
            print_solution(b);
         }
      }

      void gauss_jordan_parallel(std::vector<std::vector<double>>& A,
                                 std::vector<double>&  b,  int  rank,  int
size) {
         int n = A.size();

         for (int i = 0; i < n; ++i) {
            if (rank == i % size) {
               double pivot = A[i][i];
               for (int j = 0; j < n; ++j) {
                  A[i][j] /= pivot;
               }
               b[i] /= pivot;

               std::cout << "Step " << i + 1 << " (Parallel) - Rank " <<
rank
                         << " - Matrix A:" << std::endl;
               print_matrix(A);
               std::cout << "Step " << i + 1 << " (Parallel) - Rank " <<
rank
                         << " - Solution b:" << std::endl;
               print_solution(b);

               for (int j = 0; j < n; ++j) {
                  if (i != j) {
                     double factor = A[j][i];
                     for (int k = 0; k < n; ++k) {
                        A[j][k] -= A[i][k] * factor;
                     }
                     b[j] -= b[i] * factor;
                  }
               }

               std::cout << std::endl
                         << "Step " << i + 1 << " (Parallel) - Rank " <<
rank
```

```
                        << " - Updated Matrix A:" << std::endl;
                print_matrix(A);
                std::cout << std::endl
                        << "Step " << i + 1 << " (Parallel) - Rank " <<
rank
                        << " - Updated Solution b:" << std::endl;
                print_solution(b);
            }
            MPI_Barrier(MPI_COMM_WORLD);
        }
    }

    void generate_random_system(int n, std::vector<std::vector<double>>&
A,
                                std::vector<double>& b) {
        A.resize(n, std::vector<double>(n));
        b.resize(n);

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                A[i][j] = rand() % 100 + 1;
            }
            b[i] = rand() % 100 + 1;
        }
    }

    void run_test(int n, int num_threads) {
        std::vector<std::vector<double>> A;
        std::vector<double> b;

        generate_random_system(n, A, b);

        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        double time_taken = measure_time([&]() {
            if (size == 1) {
                gauss_jordan_serial(A, b);
            } else {
                gauss_jordan_parallel(A, b, rank, size);
            }
        });

        std::cout << "Time for n=" << n << " and threads=" << num_threads
<< ": "
                  << time_taken << " seconds" << std::endl;
    }

    void write_to_csv(const std::string& filename,
                      const std::vector<std::vector<double>>& data) {
        std::ofstream file(filename);

        for (const auto& row : data) {
            for (size_t i = 0; i < row.size(); ++i) {
                file << row[i];
                if (i < row.size() - 1) file << ",";
            }
```

```
            file << "\n";
        }
    }

    void demo() {
        int n = 5;
        int num_threads = 2;

        std::cout << "Demonstrating Gauss-Jordan with n=" << n << " and "
                  << num_threads << " threads." << std::endl;

        std::vector<std::vector<double>> A;
        std::vector<double> b;

        generate_random_system(n, A, b);

        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        std::cout << "Running serial Gauss-Jordan..." << std::endl;
        gauss_jordan_serial(A, b);
        std::cout << "Running parallel Gauss-Jordan..." << std::endl;
        gauss_jordan_parallel(A, b, rank, size);

        std::cout << "Demo complete." << std::endl;
    }

    int main(int argc, char* argv[]) {
        MPI_Init(&argc, &argv);

        bool is_demo = false;
        for (int i = 1; i < argc; ++i) {
            if (strcmp(argv[i], "demo") == 0) {
                is_demo = true;
                break;
            }
        }

        if (is_demo) {
            demo();
        } else {
            std::vector<int> sizes = {5, 10, 100, 500, 1000};
            std::vector<int> threads = {1, 2, 4, 8};
            std::vector<std::vector<double>> results;

            for (int n : sizes) {
                for (int num_threads : threads) {
                    double time_taken =
                        measure_time([&]() { run_test(n, num_threads); });
                    results.push_back({(double)n,       (double)num_threads,
time_taken});
                }
            }

            write_to_csv("results.csv", results);
        }
```

```
    MPI_Finalize();
    return 0;
}
```

Название файла: 0301_3_4.py

```
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv('results.csv', header=None, names=["Size", "Threads", "Time"])

plt.figure(figsize=(10, 6))

for threads in [1, 2, 4, 8]:
    subset = data[data["Threads"] == threads]
    plt.plot(subset["Size"], subset["Time"], label=f'Количество потоков = {threads}')

plt.xlabel('Размерность СЛАУ')
plt.ylabel('Время (секунды)')
plt.title('Время решения СЛАУ методом Гаусса-Жордана')
plt.legend()
plt.grid(True)
plt.show()
```