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

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Архитектура параллельных вычислительных систем»
Тема: Решение систем линейных алгебраических уравнений с матрицами специального вида 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 СЛАУ с матрицами специального вида 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.

Вариант 3.

Решение СЛАУ Ax = b методом квадратного корня (разложение Холецкого) с использованием библиотеки 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

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. 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 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 потоков (1, 2, 4, 8), чтобы понять, как производительность зависит от числа используемых вычислительных ресурсов.

Для параллельного алгоритма используется OpenMP, позволяющий изменять количество потоков, задействованных в вычислениях. Для каждой комбинации размерности матрицы и числа потоков измеряется производительность.

Результаты замеров времени выполнения для каждой комбинации размерности матрицы и числа потоков 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. График сохраняется в формате PNG (см. рис. 1).

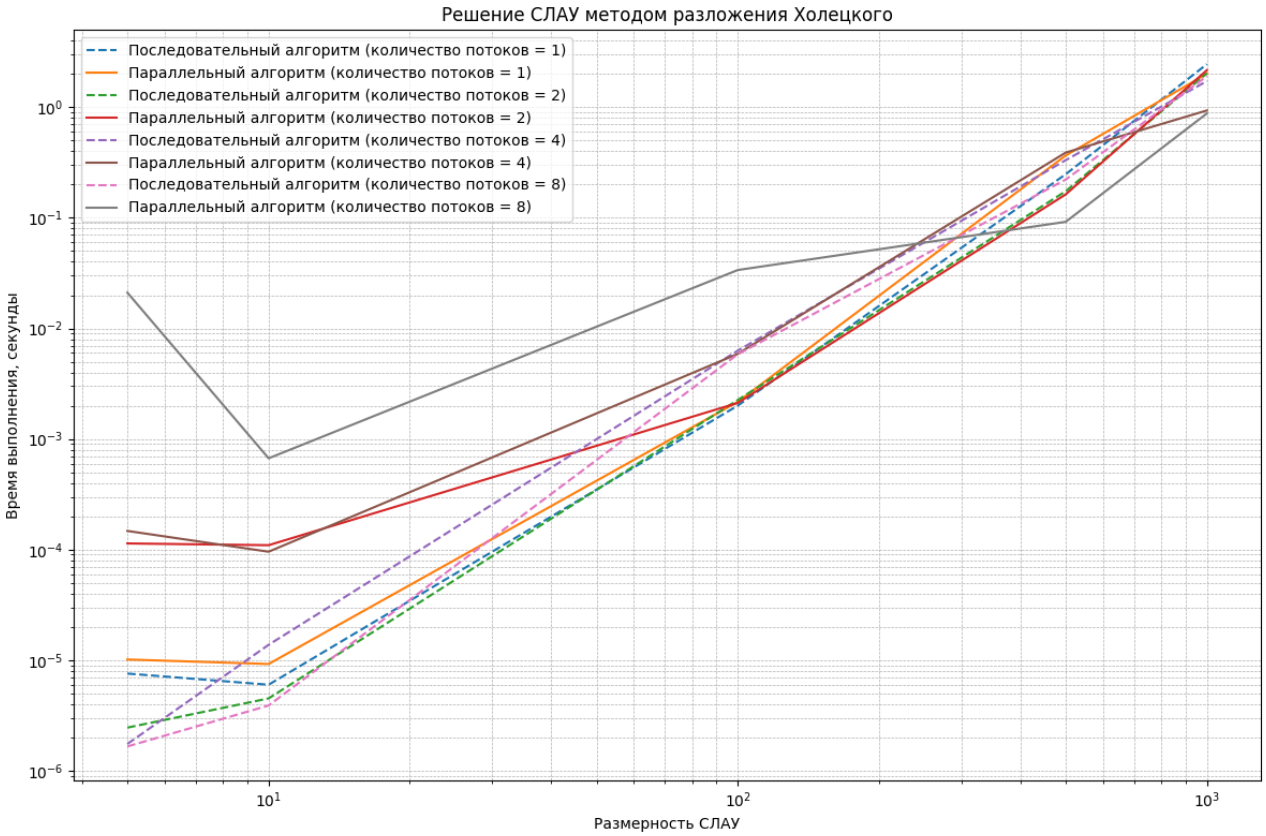

Рисунок 1 – График зависимостей времени выполнения последовательного и параллельного методов разложения Холецкого для разного количества потоков от размерности СЛАУ

Блок-схемы алгоритмов с пояснения

Блок-схема последовательного алгоритма метода разложения Холецкого на рис. 2.

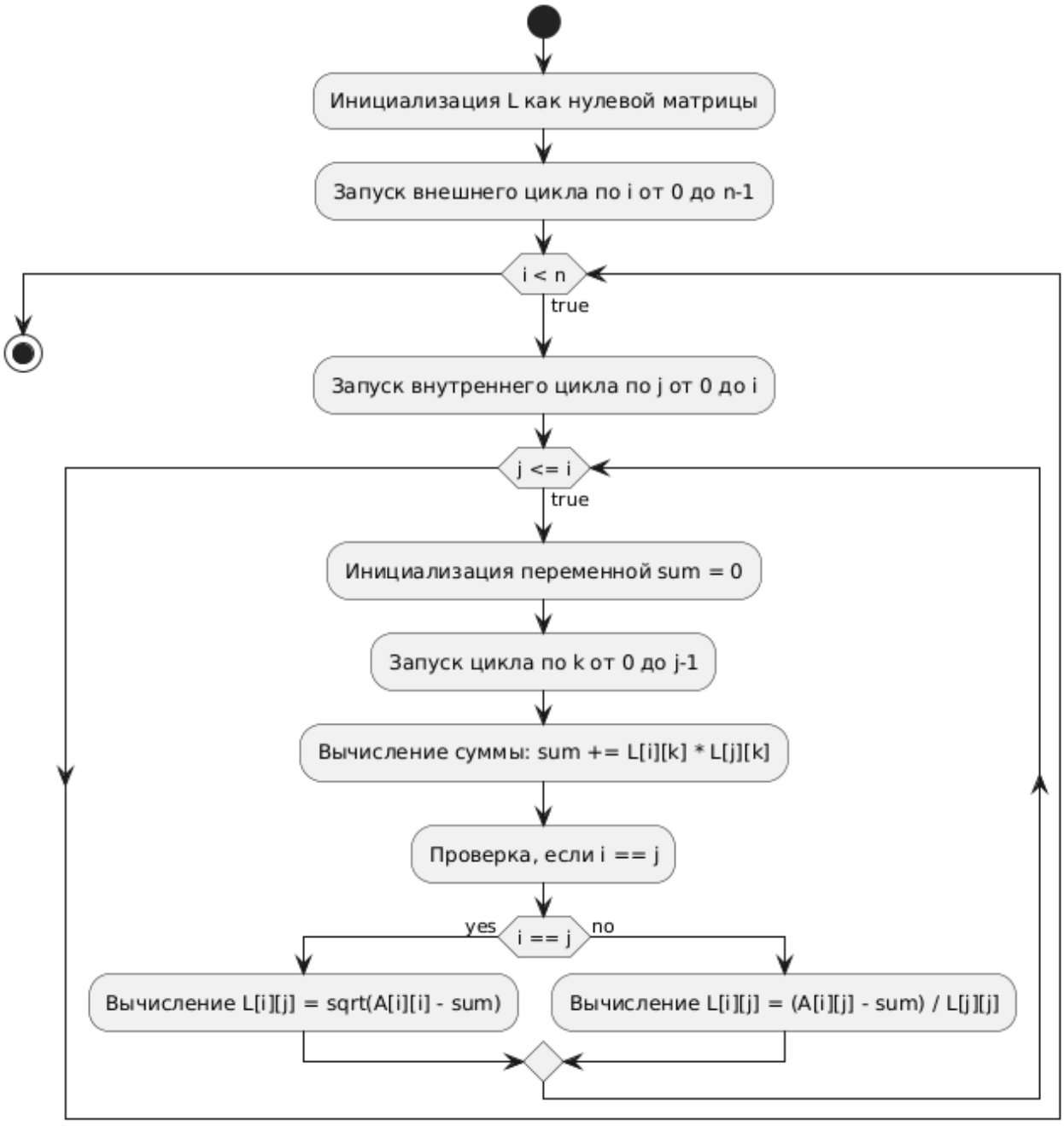

Рисунок 2 – Блок-схема последовательного алгоритма метода разложения Холецкого

Инициализируется матрица L размера $n \times n$ с нулями.

Происходит цикл по i внешний цикл проходит по строкам (или столбцам) матрицы A от 0 до n-1.

Происходит цикл по j внутренний цикл проходит по столбцам (или строкам) для каждого индекса i.

Далее вычисляется сумма для каждого элемента матрицы L рассчитывается сумма, используя уже вычисленные элементы матрицы L.

После обновляются элементы L. Если i == j, то вычисляется диагональный элемент L[i][i] как квадратный корень из A[i][i] минус сумма. Иначе вычисляется элемент L[i][j] как разность A[i][j] минус сумма, делённая на диагональный элемент L[j][j].

Блок-схема параллельного алгоритма метода разложения Холецкого представлена на рис. 3.

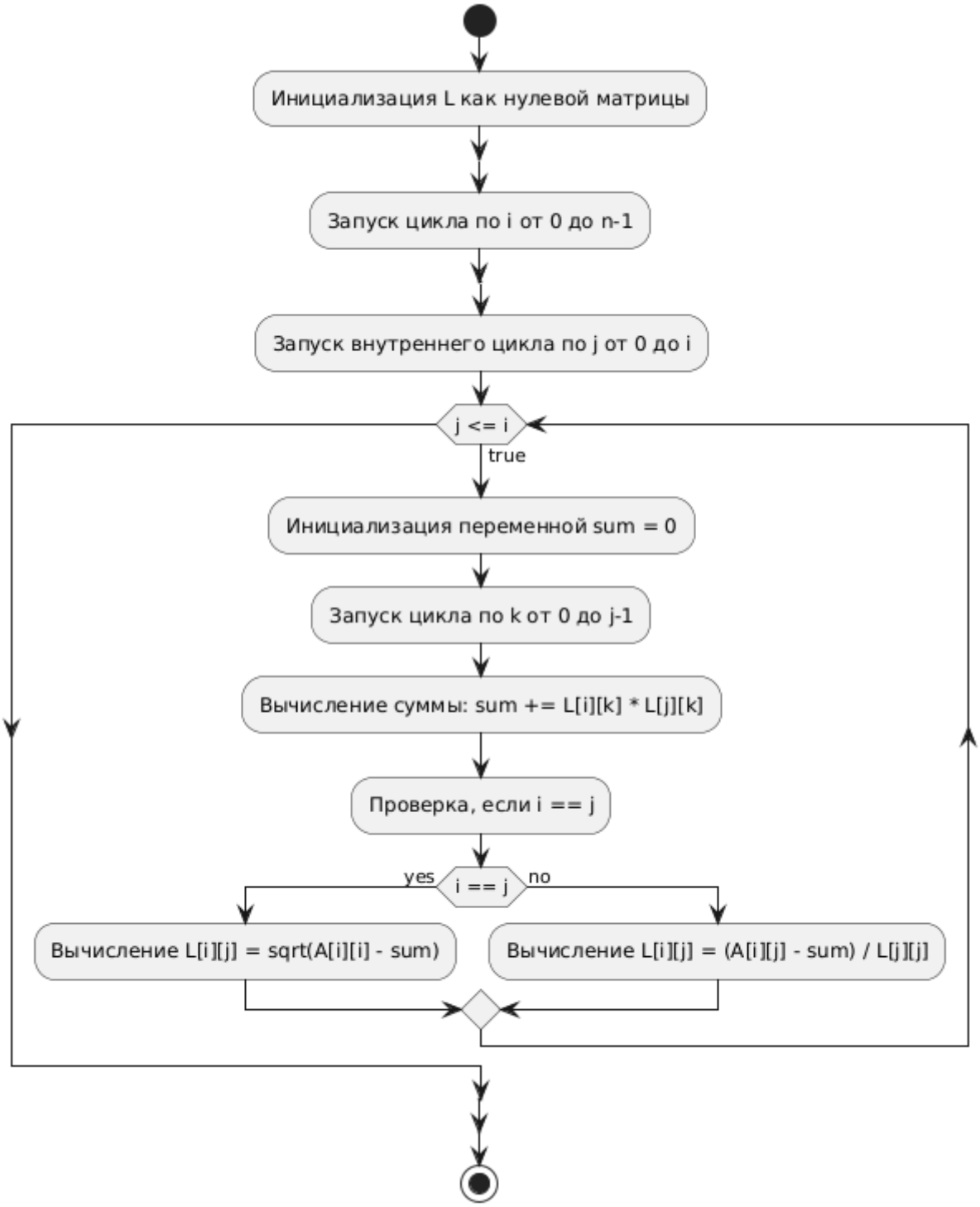

Рисунок 3 – Блок-схема параллельного алгоритма метода разложения Холецкого

Инициализируется матрица L размера $n \times n$ с нулями.

Происходит параллельный цикл по i. Параллельный цикл по строкам (или столбцам) матрицы A, где каждый поток обрабатывает разные строки.

Далее происходит цикл по j и вычисления. Для каждого элемента матрицы L производится аналогичное вычисление, как в последовательном варианте, с учётом параллельной обработки элементов.

Основное отличие от последовательного алгоритма заключается в том, что вычисления для разных элементов матрицы L выполняются параллельно. Это позволяет ускорить процесс для больших матриц.

Обновление элементов L происходит точно такие же вычисления для каждого элемента, но они выполняются параллельно, если это возможно. Важно, чтобы потоки не изменяли одни и те же элементы одновременно (для этого используется правильная синхронизация).

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	7.571e-06	1.0146e-05	0.745
5	2	2.457e-06	0.000113688	21.59
5	4	1.748e-06	0.000147534	11.85
5	8	1.66e-06	0.0210637	0.079
10	1	6.002e-06	9.24e-06	0.649
10	2	4.513e-06	0.000109621	41.2
10	4	1.3775e-05	9.5678e-05	0.144

10	8	3.894e-06	0.000669287	5.82
100	1	0.00200528	0.002169	0.92
100	2	0.0022528	0.00211842	1.06
100	4	0.00629779	0.00588947	1.07
100	8	0.00583632	0.0337273	0.17
500	1	0.24783	0.365963	0.68
500	2	0.174008	0.162942	1.07
500	4	0.332208	0.389086	0.85
500	8	0.221781	0.0920005	2.41
1000	1	2.4523	2.00846	1.22
1000	2	2.02638	2.16349	0.94
1000	4	1.72433	0.935351	1.84
1000	8	1.90844	0.879324	2.17

Тестирование.

Программа отрабатывает демонстрационный сценарий при передаче флага DDEMO_MODE во время компиляции программы.

На рис. 4 представлен демонстрационный сценарий работы программы. На рис. 5 представлен пример работы программы.

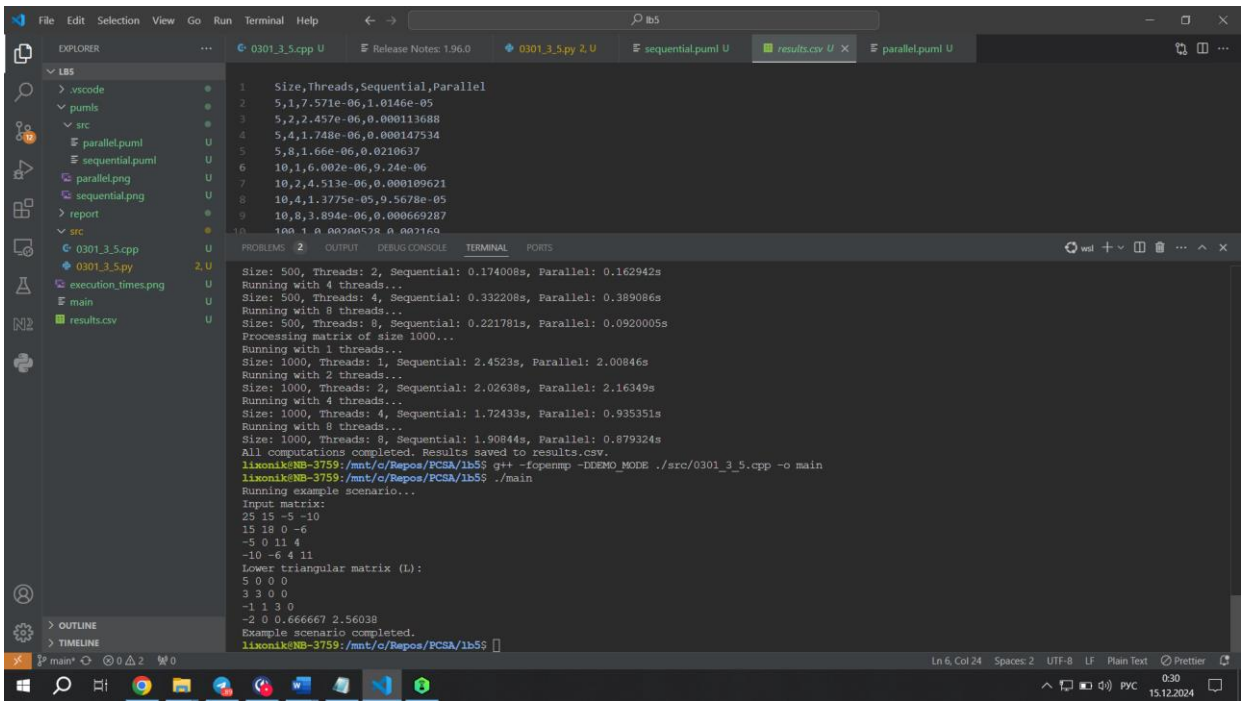

Рисунок 4 – Демонстрационный сценарий работы программы

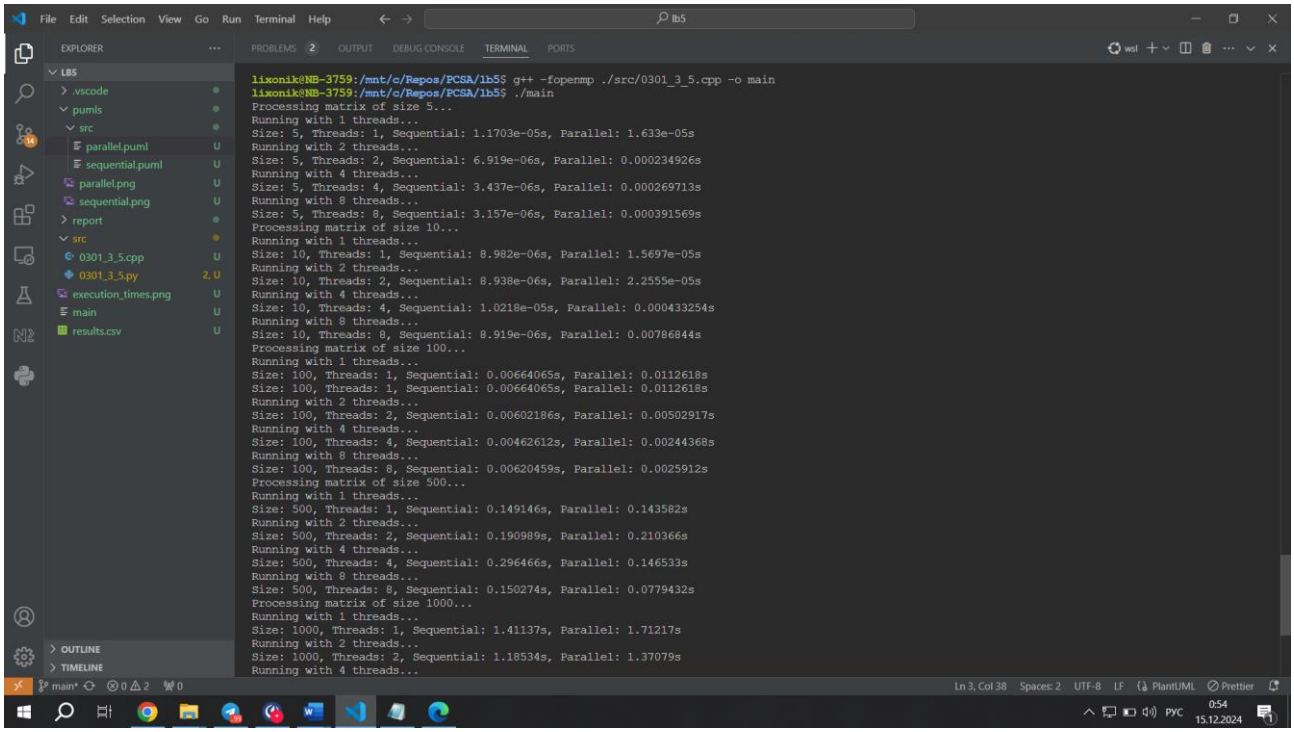

Рисунок 5 – Пример работы программы

Выводы.

На практике были освоены методы решения СЛАУ с матрицами специального вида на вычислительных системах с общей памятью.

Для малых размерностей (5 и 10) параллельное выполнение не дает заметного ускорения из-за накладных расходов на управление потоками и недостаточной сложности вычислений. Использование нескольких потоков может даже замедлить выполнение программы из-за этих расходов.

Для средних размерностей (100 и 500) параллельное выполнение начинает показывать небольшие улучшения в производительности. Однако увеличение числа потоков выше 4 может не давать значительного прироста, а в некоторых случаях может замедлить выполнение.

Для больших размерностей (1000) параллельные методы начинают показывать существенные преимущества. Ускорение становится более очевидным, и увеличение числа потоков до 8 существенно ускоряет выполнение программы. Это подтверждает, что параллельное выполнение эффективно при решении задач с большими размерностями СЛАУ.

В целом, для задач с маленькими и средними матрицами параллельный алгоритм может не оправдать ожиданий из-за накладных расходов, в то время как для больших задач (особенно с размерностями 1000 и выше) параллельный метод показывает существенное улучшение производительности.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: 0301_3_5.cpp

```
#include <omp.h>
#include <chrono>
#include <cmath>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

void log_step(const std::string &message) { std::cout << message << std::endl; }

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 cholesky_decomposition_sequential(
    const std::vector<std::vector<double>> &A,
    std::vector<std::vector<double>> &L) {
   int n = A.size();
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j <= i; ++j) {
         double sum = 0.0;
         for (int k = 0; k < j; ++k) sum += L[i][k] * L[j][k];

         if (i == j)
            L[i][j] = std::sqrt(A[i][i] - sum);
         else
            L[i][j] = (A[i][j] - sum) / L[j][j];
      }
   }
}

void cholesky_decomposition_parallel(const std::vector<std::vector<double>> &A,

std::vector<std::vector<double>> &L) {
   int n = A.size();
#pragma omp parallel for
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j <= i; ++j) {
         double sum = 0.0;
         for (int k = 0; k < j; ++k) sum += L[i][k] * L[j][k];

         if (i == j)
            L[i][j] = std::sqrt(A[i][i] - sum);
```

```
            else
               L[i][j] = (A[i][j] - sum) / L[j][j];
         }
      }
   }

   void forward_substitution(const std::vector<std::vector<double>> &L,
                             const std::vector<double> &b,
                             std::vector<double> &y) {
      int n = L.size();
      for (int i = 0; i < n; ++i) {
         double sum = 0.0;
         for (int j = 0; j < i; ++j) sum += L[i][j] * y[j];
         y[i] = (b[i] - sum) / L[i][i];
      }
   }

   void backward_substitution(const std::vector<std::vector<double>> &L,
                              const std::vector<double> &y,
                              std::vector<double> &x) {
      int n = L.size();
      for (int i = n - 1; i >= 0; --i) {
         double sum = 0.0;
         for (int j = i + 1; j < n; ++j) sum += L[j][i] * x[j];
         x[i] = (y[i] - sum) / L[i][i];
      }
   }

   std::vector<std::vector<double>> generate_spd_matrix(int n) {
      std::vector<std::vector<double>>   A(n,   std::vector<double>(n,
0.0));
      for (int i = 0; i < n; ++i) {
         for (int j = 0; j <= i; ++j) {
            double value = (rand() % 100) + 1;
            A[i][j] = value;
            A[j][i] = value;
         }
         A[i][i] += n * 10;
      }
      return A;
   }

   void run_example() {
      log_step("Running example scenario...");

      int n = 4;
      std::vector<std::vector<double>> A = {
          {25, 15, -5, -10}, {15, 18, 0, -6}, {-5, 0, 11, 4}, {-10, -6,
4, 11}};

      log_step("Input matrix:");
      for (const auto &row : A) {
         for (double val : row) std::cout << val << " ";
         std::cout << std::endl;
      }

      std::vector<std::vector<double>>   L(n,   std::vector<double>(n,
0.0));
```

```
        cholesky_decomposition_sequential(A, L);

        log_step("Lower triangular matrix (L):");
        for (const auto &row : L) {
           for (double val : row) std::cout << val << " ";
           std::cout << std::endl;
        }

        log_step("Example scenario completed.");
     }

     int main() {
     #ifdef DEMO_MODE
        run_example();
     #else
        std::vector<int> sizes = {5, 10, 100, 500, 1000};
        std::vector<int> threads = {1, 2, 4, 8};

        std::ofstream csv_file("results.csv");
        csv_file << "Size,Threads,Sequential,Parallel" << std::endl;

        for (int size : sizes) {
           log_step("Processing matrix of size " + std::to_string(size) +
"...");
           auto A = generate_spd_matrix(size);
           std::vector<double> b(size, 1.0);

           for (int t : threads) {
              omp_set_num_threads(t);

              log_step("Running   with   "   +   std::to_string(t)   +   "
threads...");

              std::vector<std::vector<double>> L_seq(size,

std::vector<double>(size, 0.0));
              double seq_time = measure_time(
                  [&]() { cholesky_decomposition_sequential(A, L_seq); });

              std::vector<std::vector<double>> L_par(size,

std::vector<double>(size, 0.0));
              double par_time =
                  measure_time([&]() { cholesky_decomposition_parallel(A,
L_par); });

              csv_file << size << "," << t << "," << seq_time << "," <<
par_time
                       << std::endl;
              std::cout << "Size: " << size << ", Threads: " << t
                        << ", Sequential: " << seq_time
                        << "s, Parallel: " << par_time << "s\n";
           }
        }

        csv_file.close();
        log_step("All   computations   completed.   Results   saved   to
results.csv.");
```

```
#endif
   return 0;
}
```

Название файла: 0301_3_4.py

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

csv_file = "results.csv"
data = pd.read_csv(csv_file)

sizes = data['Size'].unique()
threads = data['Threads'].unique()

plt.figure(figsize=(12, 8))

for t in threads:
    subset = data[data['Threads'] == t]
    plt.plot(subset['Size'], subset['Sequential'],
label=f"Последовательный алгоритм (количество потоков = {t})",
linestyle='--')
    plt.plot(subset['Size'], subset['Parallel'],
label=f"Параллельный алгоритм (количество потоков = {t})")

plt.xscale('log')
plt.yscale('log')
plt.xlabel("Размерность СЛАУ")
plt.ylabel("Время выполнения, секунды")
plt.title("Решение СЛАУ методом разложения Холецкого")
plt.legend()
plt.grid(True, which="both", linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.savefig("execution_times.png")
plt.show()
```