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

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Архитектура параллельных вычислительных систем»
Тема: Операции над элементами векторов и матриц 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.

Получить знания о конструировании простых параллельных алгоритмов на системах с общей памятью, а также общее представление о масштабируемости задач. На практике освоить основные директивы OpenMP и mpi, способах распределения вычислений между потоками, способах распределения вычислений итерационных циклов между потоками.

Задание.

1. В зависимости от номера варианта задания разработать соответствующие алгоритмы операций над элементами векторов и матриц.

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 матрицы и вектора: 10, 100, 500, 1000, 5000.

4. Оценить размерности матрицы и вектора, при которых 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.

Умножение матрицы на матрицу с использованием директивы распараллеливания параметрических циклов #pragma omp for и с использованием “ручного” задания работ (распараллеливания циклов без директивы for).

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 (если установлены необходимые библиотеки и компилятор).
5. Библиотека OpenMP для распараллеливания вычислений.
6. IDE для разработки – Visual Studio Code с подключением к WSL.
7. 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 каждой комбинации размерности матрицы и количества потоков создаются две исходные матрицы (например, firstMatrix и secondMatrix), которые будут перемножаться, и пустая результирующая матрица для хранения результата. Матрицы заполняются случайными значениями для имитации реальных данных.

Время выполнения каждой функции перемножения матриц измеряется с использованием стандартной библиотеки C++ chrono. Для этого создается обертка, которая фиксирует момент начала и завершения выполнения алгоритма, а затем вычисляет длительность операции. В качестве метрики времени используется высокоточное время выполнения, предоставляемое функцией std::chrono::high_resolution_clock::now().

Для каждой комбинации (размер матрицы и количество потоков) последовательно вызываются три метода:

- Последовательное умножение (serial) – алгоритм перемножения матриц выполняется без использования параллельных вычислений.

- Параллельное умножение с использованием OpenMP и директивы #pragma omp for – выполняется параллельное умножение матриц с автоматическим распределением нагрузки с помощью OpenMP.
- Параллельное умножение с ручным распределением (manual) – каждый поток вручную обрабатывает свой диапазон строк матрицы.

Для каждого метода время выполнения записывается в переменную и затем сохраняется для последующего анализа.

После выполнения каждого из алгоритмов перемножения проводится проверка на корректность: результаты всех методов сравниваются между собой. Если результаты отличаются, выводится соответствующее сообщение.

Вся информация о времени выполнения для каждой комбинации записывается в CSV-файл. Каждая строка файла содержит:

- Размерность матрицы.
- Количество потоков.
- Время выполнения последовательного умножения.
- Время выполнения параллельного умножения с использованием #pragma omp for.
- Время выполнения параллельного умножения с ручным распределением нагрузки.

Формат записи данных в CSV-файл позволяет легко использовать их для дальнейшей обработки и анализа. Каждая строка файла содержит результаты для конкретной комбинации размера матрицы и количества потоков, а также времена выполнения всех трех алгоритмов.

После завершения всех измерений данные читаются из созданного CSV-файла с помощью Python и библиотеки csv. Информация о размере матрицы и времени выполнения для каждого метода группируется по количеству потоков.

Для каждой группы (в зависимости от количества потоков) строятся три графика, отображающие зависимости времени выполнения от размера матрицы:

- Для последовательного умножения.

- Для параллельного умножения с #pragma omp for.
- Для параллельного умножения с ручным распределением.

Ось X представляет размер матрицы, а ось Y показывает время выполнения в секундах.

Для каждой комбинации количества потоков создается отдельный график, который сохраняется в виде PNG-файла (см. рис. 1-3).

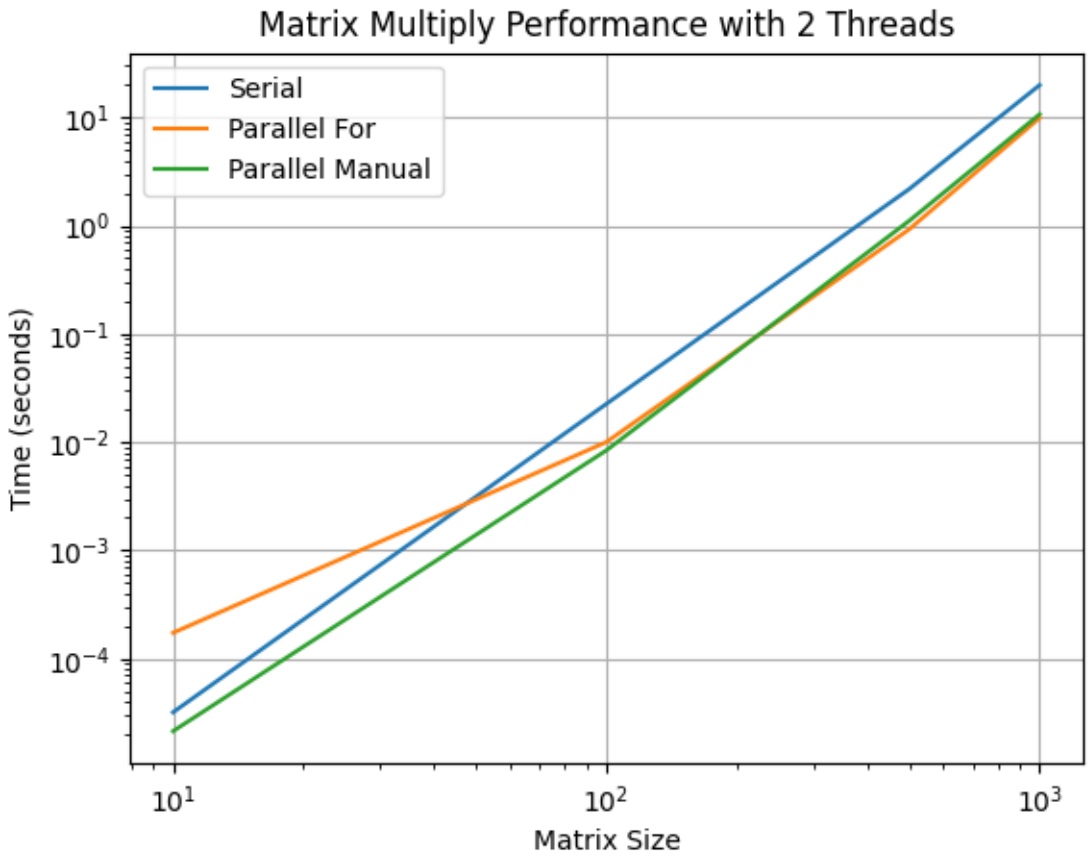

Рисунок 1 – График зависимости времени работы алгоритмов от размерности матрицы при использовании 2 потоков

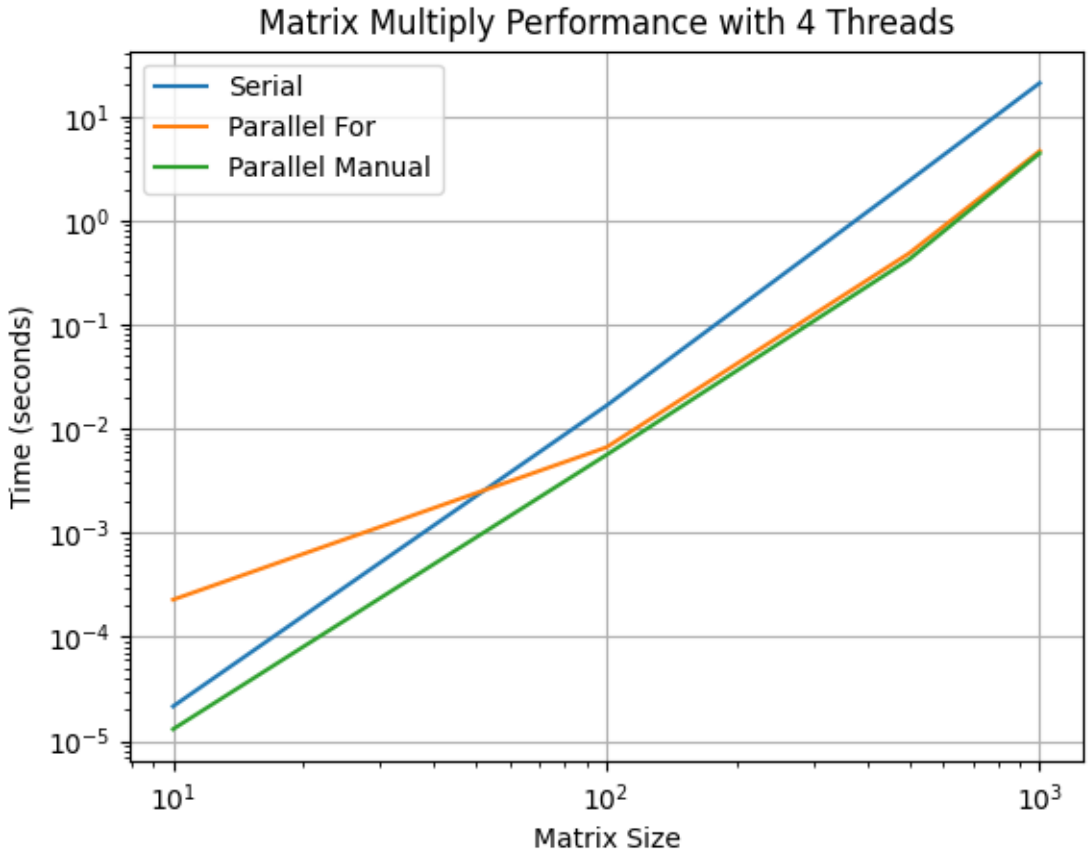

Рисунок 2 – График зависимости времени работы алгоритмов от размерности матрицы при использовании 4 потоков

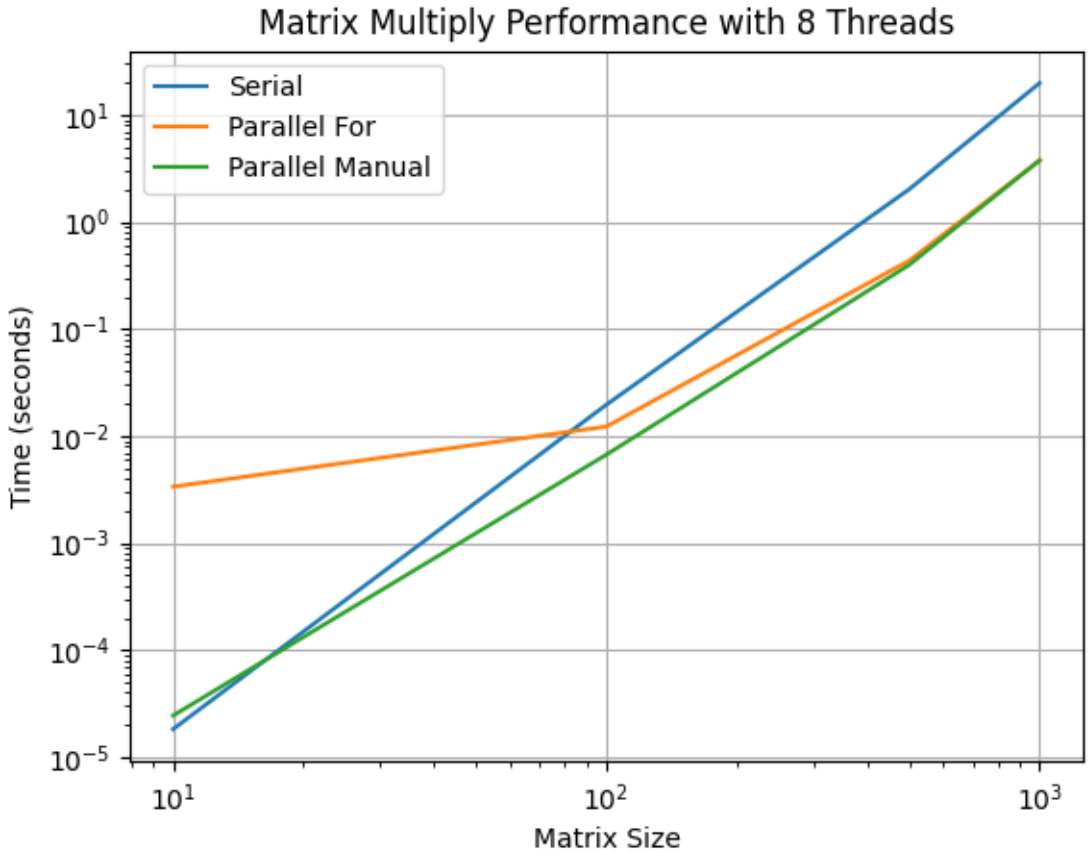

Рисунок 3 – График зависимости времени работы алгоритмов от размерности матрицы при использовании 8 потоков

Блок-схемы алгоритмов с пояснения

Блок-схема алгоритма автоматического параллельного перемножения матриц представлена на рис. 4.

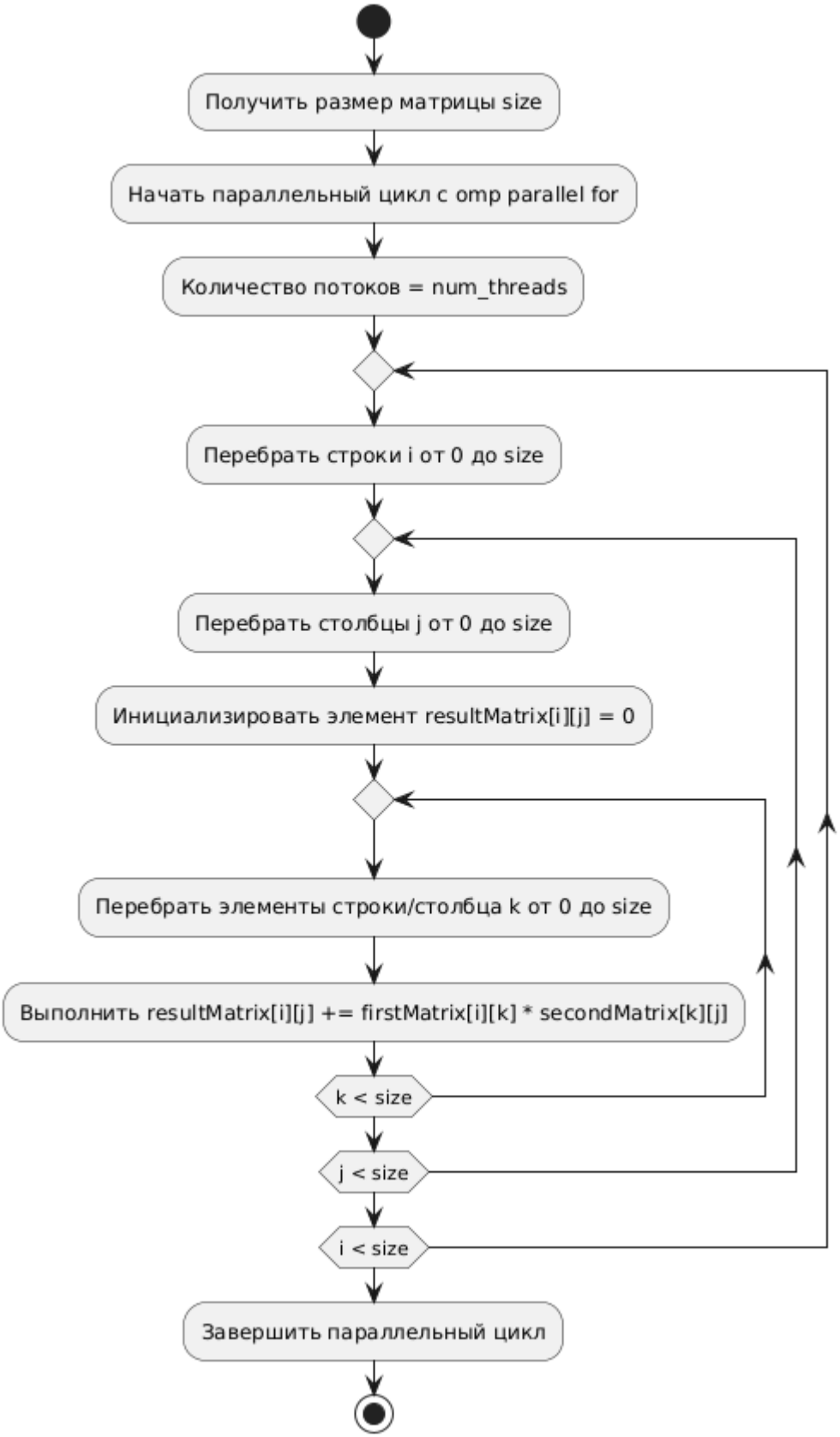

Рисунок 4 – Блок-схема алгоритма автоматического параллельного перемножения матриц

Алгоритм использует директиву #pragma omp parallel for для распараллеливания основного цикла по строкам матрицы. Каждый поток обрабатывает разные строки матрицы, производя вычисления для каждого элемента результирующей матрицы. Основной цикл перебирает строки, вложенные циклы – столбцы и элементы для перемножения.

Блок-схема алгоритма «ручного» параллельного перемножения матриц представлена на рис. 5.

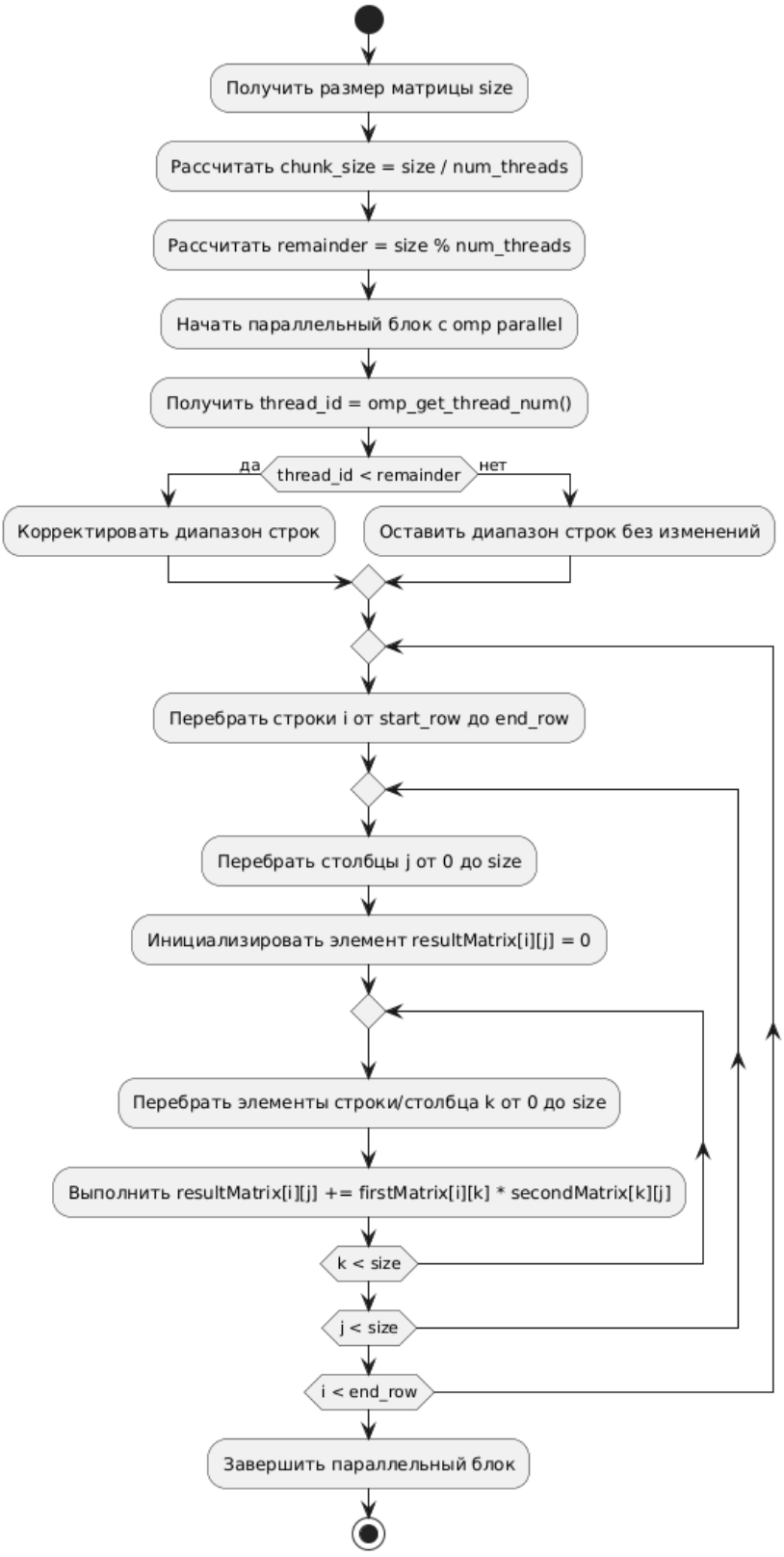

Рисунок 5 – Блок-схема алгоритма «ручного» параллельного перемножения матриц

В этой версии алгоритма происходит ручное распределение строк матрицы между потоками. Каждому потоку назначается свой диапазон строк для

обработки, при этом учитываются остаточные строки (remainder), которые распределяются между потоками. Как и в случае с автоматическим распараллеливанием, потоки выполняют умножение строк первой матрицы на столбцы второй матрицы и записывают результат.

Блок-схема алгоритма последовательного перемножения матриц представлена на рис. 5.

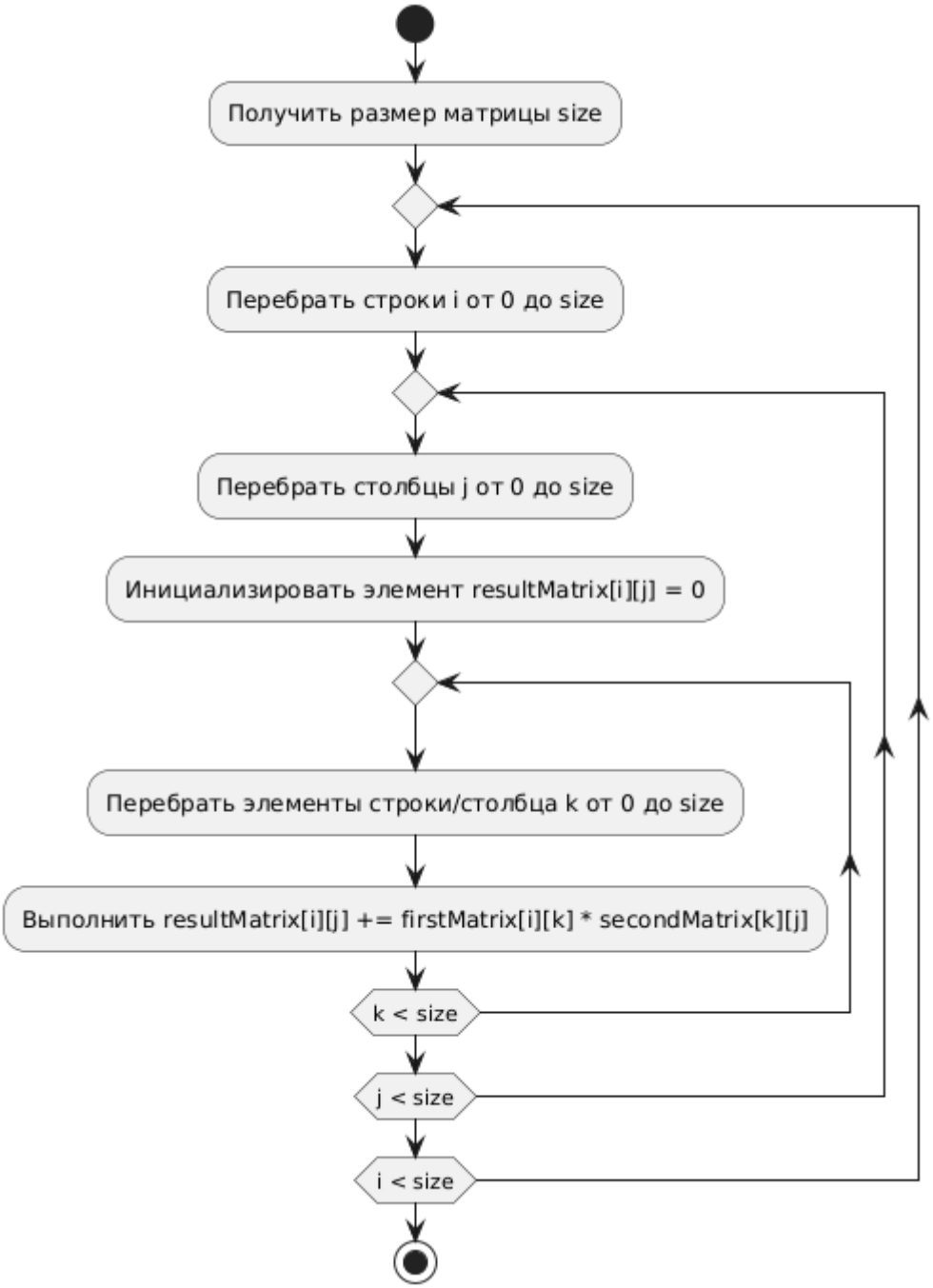

Рисунок 6 – Блок-схема алгоритма последовательного параллельного перемножения матриц

Этот алгоритм выполняется последовательно, без распараллеливания. Три вложенных цикла перебирают строки, столбцы и элементы матриц для выполнения умножения и суммирования.

Оценка производительности алгоритмов

Таблица 1 – Время выполнения программы для различных значений размерности матрицы и матрицы для разного числа потоков

Размер матрицы	Количество потоков	Время выполнения последовательного перемножения (сек)	Время выполнения автоматического параллельного перемножения (сек)	Время выполнения «*ручного*» параллельного перемножения (сек)
10	2	3.2051e-05	0.00017358	2.1446e-05
100	2	0.0225973	0.0100394	0.00841049
500	2	2.17999	0.926371	1.10981
1000	2	19.8952	9.89262	10.684
10	4	2.1524e-05	0.000227797	1.2991e-05
100	4	0.0166378	0.0066183	0.00559385
500	4	2.40869	0.484423	0.423146
1000	4	20.8864	4.65229	4.41835
10	8	1.8276e-05	0.00336627	2.447e-05
100	8	0.0196322	0.0122646	0.00670898
500	8	2.02389	0.435325	0.397351
1000	8	19.8545	3.77056	3.76524

Если рассчитать ускорение и эффективность для параллельных алгоритмов по сравнению с последовательным с помощью следующих формул:

- Ускорение (Speedup): $S = \frac{T_{serial}}{T_{parallel}}$.
- Эффективность (Efficiency): $E = \frac{S}{threads}$.

Таблица 2 – Сравнительная оценка эффективности программы для различных значений размерности матрицы и матрицы для разного числа потоков

Размер матрицы	Количество потоков	Ускорение автоматического распараллеливания	Эффективность автоматического распараллеливания	Ускорение «*ручного*» распараллеливания	Эффективность «*ручного*» распараллеливания
10	2	0.1846	0.0923	1.4941	0.1846
100	2	2.2513	1.1256	2.6854	1.3427
500	2	2.3539	1.1769	1.9648	0.9824
1000	2	10.684	2.0104	1.0052	1.8626
10	4	0.0945	0.0236	1.6569	0.4142
100	4	2.5144	0.6286	2.9735	0.7434
500	4	4.9711	1.2428	5.6927	1.4232
1000	4	4.4917	1.1229	4.7283	1.1821
10	8	0.0054	0.0007	0.7468	0.0934
100	8	1.6005	0.2001	2.9264	0.3658
500	8	4.6484	0.5811	5.0933	0.6367
1000	8	5.2655	0.6582	5.2719	0.6590

Тестирование.

На рис. 7 представлен пример работы программы для перемножения матриц размерности 4 и количеством потоков 6.

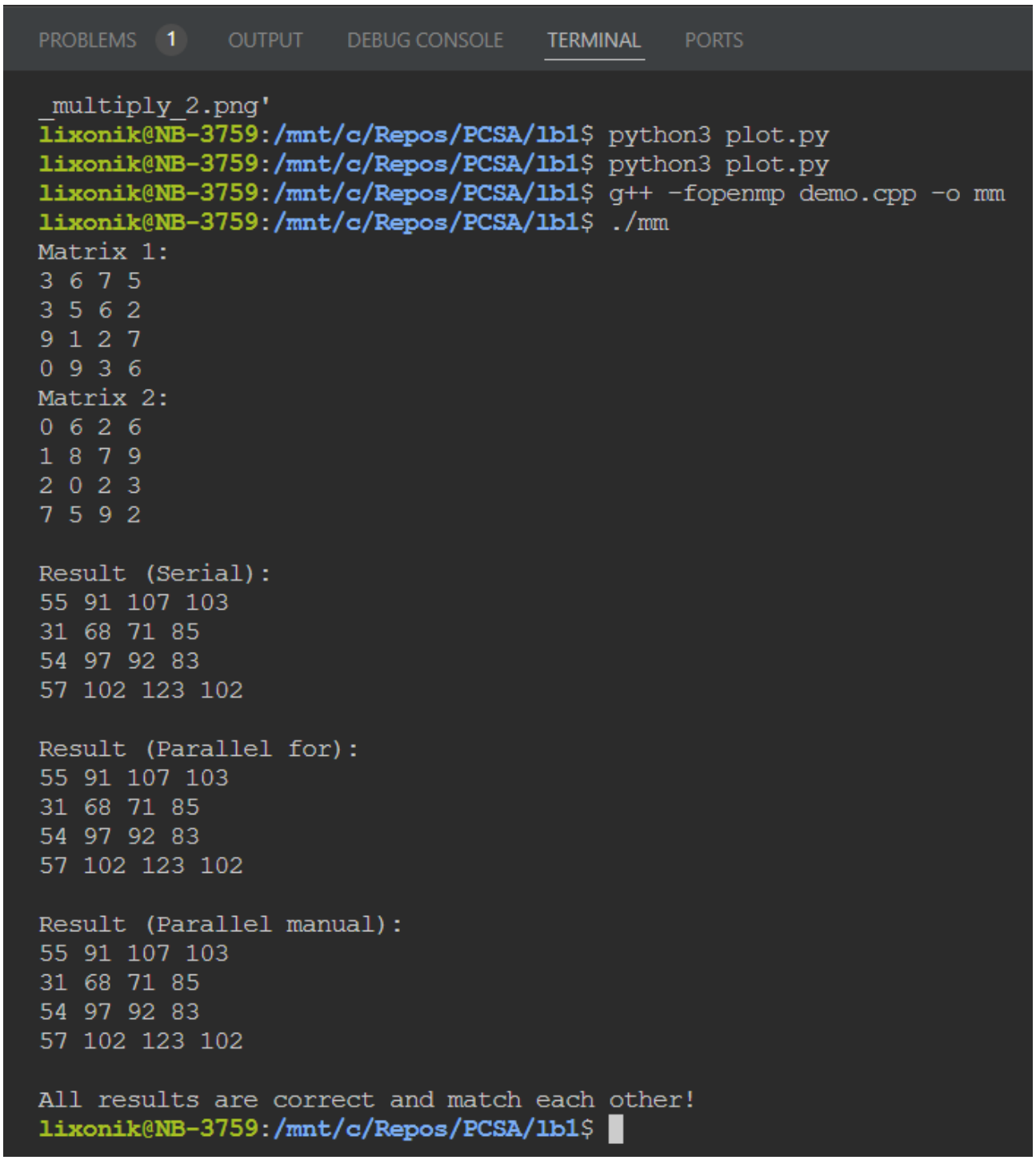

Рисунок 7 – Пример работы программы

Выводы.

Были получены знания о конструировании простых параллельных алгоритмов на системах с общей памятью, а также общее представление о масштабируемости задач. На практике освоены основные директивы OpenMP и

mpi, способы распределения вычислений между потоками, способы распределения вычислений итерационных циклов между потоками.

Для малых матриц (например, размер 10x10) параллельные алгоритмы практически не дают ускорения или даже могут показывать замедление из-за накладных расходов на распараллеливание.

Для больших матриц (например, 500x500 и 1000x1000) наблюдается значительное ускорение. При увеличении числа потоков до 8 скорость выполнения возрастает в 5 раз и более.

Эффективность параллельных алгоритмов резко падает при малых размерностях матриц. Это связано с тем, что размер задачи недостаточен для того, чтобы оправдать затраты на организацию многопоточности.

Для больших матриц эффективность увеличивается, особенно при 4 потоках. Для некоторых случаев она превышает 1, что говорит о хорошем использовании параллельных ресурсов.

Алгоритм с ручным управлением потоками показывает лучшие результаты для больших матриц, чем вариант с #pragma omp for, что можно объяснить более гибким распределением работы между потоками.

Для малых задач этот алгоритм иногда проигрывает, поскольку дополнительные вычисления для определения диапазонов строк между потоками увеличивают накладные расходы.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <chrono>
#include <fstream>
#include "./matrix_utils/matrix.cpp"
#include "./parallel_mult_utils/for_multiply.cpp"
#include "./parallel_mult_utils/manual_multiply.cpp"
#include "./parallel_mult_utils/serial_multiply.cpp"

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();
}

int main() {
   std::ofstream results_file("matrix_multiply.csv");
   results_file                                                    <<
"MatrixSize,Threads,Serial,ParallelFor,ParallelManual\n";

   int sizes[] = {10, 100, 500, 1000};
   int threads[] = {2, 4, 8};

   for (int num_threads : threads) {
      for (int size : sizes) {
         std::cout << "Testing matrix size: " << size << " with " <<
num_threads
                   << " threads" << std::endl;

         Matrix firstMatrix(size), secondMatrix(size);
         Matrix result_serial(size), result_parallel_for(size),
             result_parallel_manual(size);

         firstMatrix.initialize();
         secondMatrix.initialize();

         // Serial multiplication
         double serial_time = measure_time([&]() {
            serial_multiply(firstMatrix,              secondMatrix,
result_serial);
         });

         // Parallel with #pragma omp for
         double parallel_for_time = measure_time([&]() {
            for_multiply(firstMatrix,                 secondMatrix,
result_parallel_for,
                         num_threads);
         });
```

```
            // Parallel manual multiplication
            double parallel_manual_time = measure_time([&]() {
               manual_multiply(firstMatrix,              secondMatrix,
result_parallel_manual,
                               num_threads);
            });

            if (result_serial.isEqual(result_parallel_for) &&
                result_serial.isEqual(result_parallel_manual)) {
               std::cout << "Results match for size " << size << " and
threads "
                         << num_threads << std::endl;
            } else {
               std::cout << "Error: Results do not match for size " <<
size
                         << " and threads " << num_threads << std::endl;
            }

            results_file  <<  size  <<  ","  <<  num_threads  <<  ","  <<
serial_time << ","
                          <<     parallel_for_time     <<     ","     <<
parallel_manual_time
                          << "\n";
         }
      }

      results_file.close();
      return 0;
   }
```

Название файла: demo.cpp

```
#include <omp.h>
#include <iostream>
#include "./matrix_utils/matrix.cpp"
#include "./parallel_mult_utils/for_multiply.cpp"
#include "./parallel_mult_utils/manual_multiply.cpp"
#include "./parallel_mult_utils/serial_multiply.cpp"

#define SIZE 4

int main() {
   Matrix firstMatrix(SIZE), secondMatrix(SIZE);
   Matrix result_serial(SIZE), result_parallel_for(SIZE),
       result_parallel_manual(SIZE);

   int threads_count = 6;

   firstMatrix.initialize();
   secondMatrix.initialize();

   std::cout << "Matrix 1:" << std::endl;
   firstMatrix.print();
   std::cout << "Matrix 2:" << std::endl;
   secondMatrix.print();

   // parallel serial
   serial_multiply(firstMatrix, secondMatrix, result_serial);
```

```
        std::cout << "\nResult (Serial):" << std::endl;
        result_serial.print();

        // parallel #pragma omp for
        for_multiply(firstMatrix,   secondMatrix,   result_parallel_for,
threads_count);
        std::cout << "\nResult (Parallel for):" << std::endl;
        result_parallel_for.print();

        // parallel manal
        manual_multiply(firstMatrix, secondMatrix, result_parallel_manual,
                        threads_count);
        std::cout << "\nResult (Parallel manual):" << std::endl;
        result_parallel_manual.print();

        // assert
        if (result_serial.isEqual(result_parallel_for) &&
            result_serial.isEqual(result_parallel_manual)) {
           std::cout << "\nAll results are correct and match each other!"
                     << std::endl;
        } else {
           std::cout << "\nError: The results do not match!" << std::endl;
        }

        return 0;
     }
```

Название файла: matrix.cpp

```
     #ifndef MATRIX_CPP
     #define MATRIX_CPP

     #include <iostream>

     class Matrix {
       private:
        int size;
        double **data;

       public:
        Matrix(int n) : size(n) {
           data = new double *[size];
           for (int i = 0; i < size; ++i) {
              data[i] = new double[size];
           }
        }

        ~Matrix() {
           for (int i = 0; i < size; ++i) {
              delete[] data[i];
           }
           delete[] data;
        }

        void initialize() {
           for (int i = 0; i < size; ++i) {
              for (int j = 0; j < size; ++j) {
                 data[i][j] = rand() % 10;
```

```
            }
        }
    }

    void print() const {
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                std::cout << data[i][j] << " ";
            }
            std::cout << std::endl;
        }
    }

    // indexed access
    double *operator[](int index) const { return data[index]; }

    bool isEqual(const Matrix &other) const {
        if (size != other.size) return false;
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                if (data[i][j] != other[i][j]) {
                    return false;
                }
            }
        }
        return true;
    }

    int getSize() const { return size; }
};

#endif
```

Название файла: for_multiply.cpp

```
#include <omp.h>
#include "../matrix_utils/matrix.cpp"

void for_multiply(const Matrix& firstMatrix, const Matrix&
secondMatrix,
                  Matrix& resultMatrix, int num_threads) {

    int size = resultMatrix.getSize();

#pragma omp parallel for num_threads(num_threads)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            resultMatrix[i][j] = 0;
            for (int k = 0; k < size; k++) {
                resultMatrix[i][j] += firstMatrix[i][k] *
secondMatrix[k][j];
            }
        }
    }
}
```

Название файла: manual_multiply.cpp

```
#include <omp.h>
#include "../matrix_utils/matrix.cpp"

void manual_multiply(const Matrix& firstMatrix, const Matrix&
secondMatrix,
                     Matrix& resultMatrix, int num_threads) {
    int size = resultMatrix.getSize();
    int chunk_size = size / num_threads;
    int remainder = size % num_threads;

#pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        int start_row = thread_id * chunk_size;
        int end_row = (thread_id + 1) * chunk_size;

        if (thread_id < remainder) {
            start_row += thread_id;
            end_row += thread_id + 1;
        } else {
            start_row += remainder;
            end_row += remainder;
        }

        for (int i = start_row; i < end_row; ++i) {
            for (int j = 0; j < size; ++j) {
                resultMatrix[i][j] = 0;
                for (int k = 0; k < size; ++k) {
                    resultMatrix[i][j]     +=     firstMatrix[i][k]     *
secondMatrix[k][j];
                }
            }
        }
    }
}
```

Название файла: serial_multiply.cpp

```
#include "../matrix_utils/matrix.cpp"

void serial_multiply(const Matrix &firstMatrix, const Matrix
&secondMatrix,
                     Matrix &resultMatrix) {
    int size = firstMatrix.getSize();

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            resultMatrix[i][j] = 0;
            for (int k = 0; k < size; k++) {
                resultMatrix[i][j]      +=      firstMatrix[i][k]      *
secondMatrix[k][j];
            }
        }
    }
}
```

Название файла: plot.py

```
import matplotlib.pyplot as plt
import csv

data = {}

with open('matrix_multiply.csv', 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        size = int(row['MatrixSize'])
        threads = int(row['Threads'])
        if threads not in data:
            data[threads] = {'sizes': [], 'serial': [],
'parallel_for': [], 'parallel_manual': []}
        data[threads]['sizes'].append(size)
        data[threads]['serial'].append(float(row['Serial']))

data[threads]['parallel_for'].append(float(row['ParallelFor']))

data[threads]['parallel_manual'].append(float(row['ParallelManual']))

for threads, results in data.items():
    plt.figure()
    plt.plot(results['sizes'], results['serial'], label='Serial')
    plt.plot(results['sizes'], results['parallel_for'],
label='Parallel For')
    plt.plot(results['sizes'], results['parallel_manual'],
label='Parallel Manual')
    plt.xlabel('Matrix Size')
    plt.ylabel('Time (seconds)')
    plt.title(f'Matrix Multiply Performance with {threads} Threads')
    plt.legend()
    plt.grid(True)
    plt.xscale('log')
    plt.yscale('log')
    plt.savefig(f'./plots/matrix_multiply_{threads}.png')
```