МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра вычислительных технологий

ЛАБОРАТОРНАЯ РАБОТА №1
Дисциплина: Теория параллельных алгоритмов

Работу выполнил: _____ А. А. Костров

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: _____ Е. А. Нигодин

Краснодар
2026

Тема. Умножение матриц.

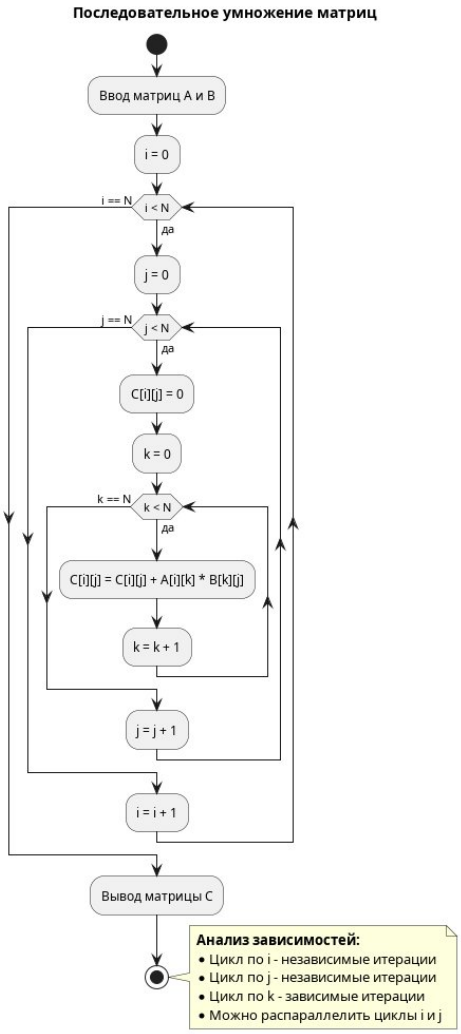
Цель. Распараллелить процесс перемножения матриц.

Задание. Составить последовательный алгоритм. Построить граф алгоритма. Исследовать на нём зависимости и возможности распараллеливания. Выполнить параллельную реализацию для случаев ленточного и блочного разделения матриц. Составить таблицу, отражающую сравнительное время выполнения последовательного и параллельных алгоритмов для разных размеров матриц на 2-х, 3-х и 4-х ядрах.

Входные данные: матрица A размерности $n \times n$ и матрица B размерности $n \times n$. Необходимо получить матрицу C путём перемножения A на B. Элементы искомой матрицы C вычисляются следующим образом:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad 0 \le i < n, 0 \le j < n$$

Последовательный алгоритм перемножения можно схематично изобразить в следующем виде:

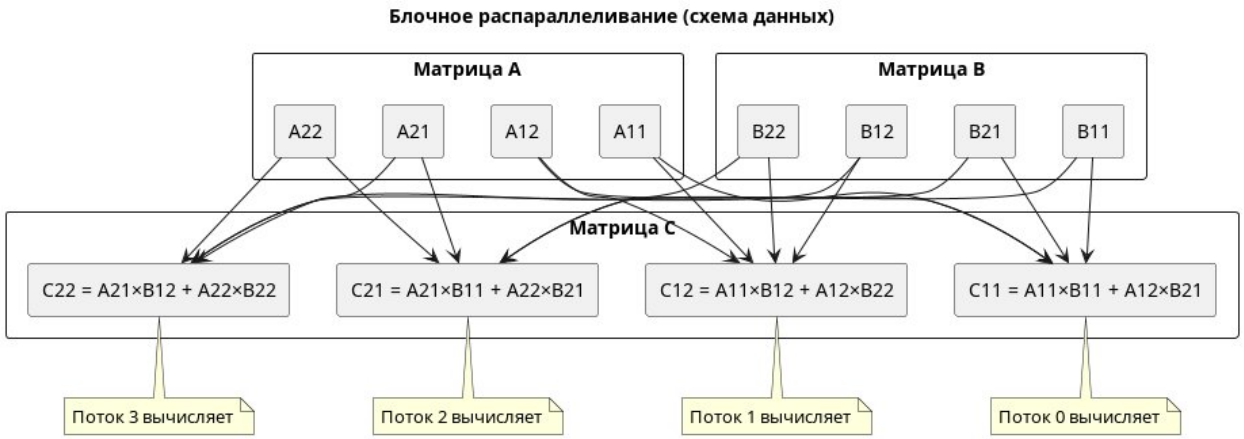


Возможность распараллеливания – представляем первую матрицу в виде нескольких последовательностей горизонтальных лент, каждая из которых параллельно перемножается со второй матрицей, после чего все результирующие ленты мы объединяем в одну искомую матрицу:

Реализуем параллельное ленточное умножение матриц:

```
void parallel_strip_multiply(const vector<vector<double>>& A,
                             const vector<vector<double>>& B,
                             vector<vector<double>>& C, int n) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            C[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Помимо рассмотрения матриц в виде наборов строк и столбцов, также можно использовать блочное представление матриц:



Напишем функцию для блочного перемножения матриц:

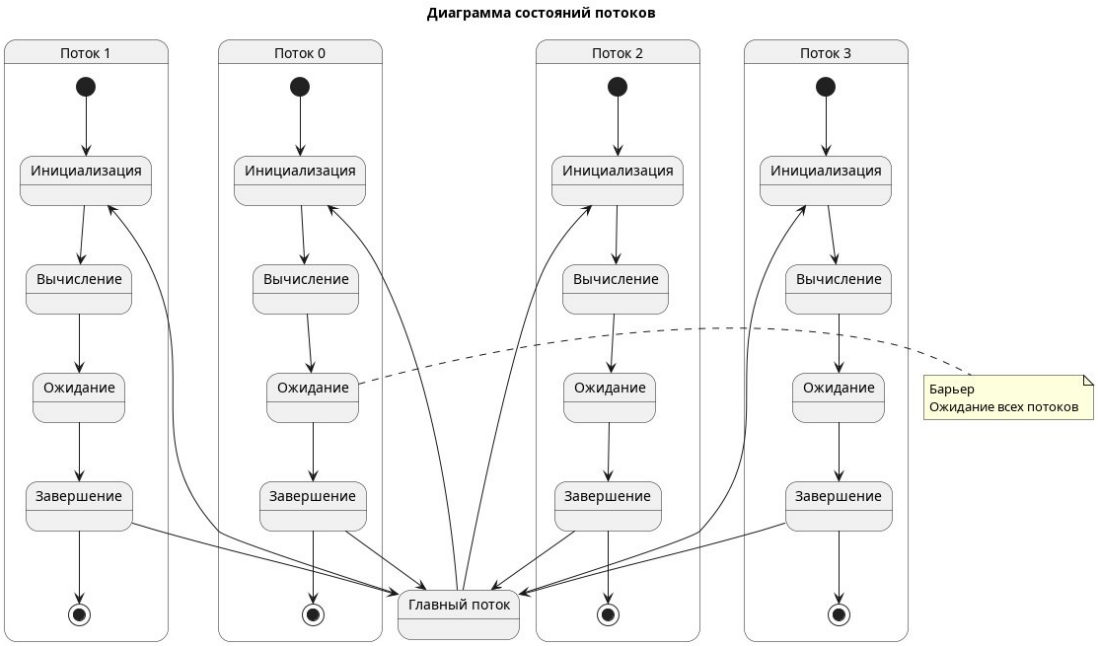
```
void parallel_block_multiply(const vector<vector<double>>& A,
                             const vector<vector<double>>& B,
                             vector<vector<double>>& C, int n, int block_size = 64) {
    #pragma omp parallel for collapse(2)
    for (int ii = 0; ii < n; ii += block_size) {
        for (int jj = 0; jj < n; jj += block_size) {
            for (int kk = 0; kk < n; kk += block_size) {
                for (int i = ii; i < min(ii + block_size, n); ++i) {
                    for (int j = jj; j < min(jj + block_size, n); ++j) {
                        double sum = 0.0;
                        for (int k = kk; k < min(kk + block_size, n); ++k) {
                            sum += A[i][k] * B[k][j];
                        }
                        #pragma omp atomic
                        C[i][j] += sum;
                    }
                }
            }
        }
    }
}
```

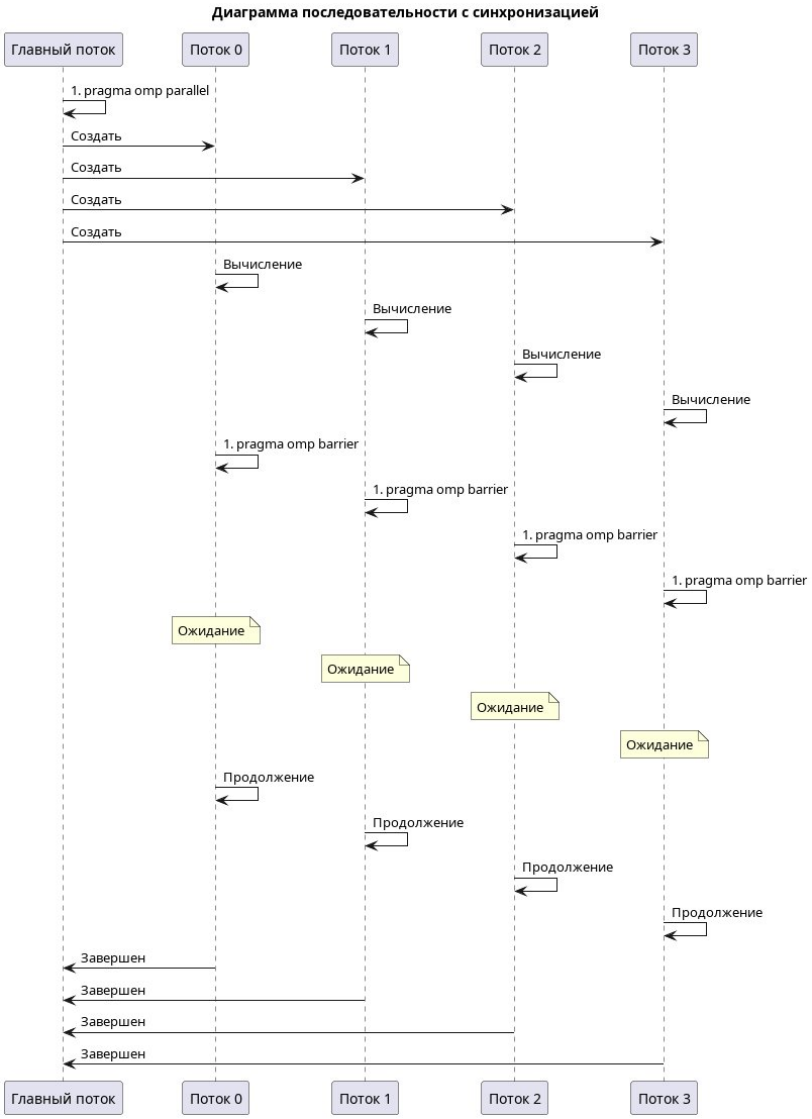
Полный код итоговой программы:

```
#include <iostream>
#include <vector>
#include <chrono>
#include <iomanip>
#include <omp.h>

using namespace std;
using namespace chrono;

void initialize_matrices(vector<vector<double>>& A, vector<vector<double>>& B, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }
}

void sequential_multiply(const vector<vector<double>>& A,
                         const vector<vector<double>>& B,
                         vector<vector<double>>& C, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            C[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void parallel_strip_multiply(const vector<vector<double>>& A,
                             const vector<vector<double>>& B,
                             vector<vector<double>>& C, int n) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            C[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void parallel_block_multiply(const vector<vector<double>>& A,
                             const vector<vector<double>>& B,
                             vector<vector<double>>& C, int n, int block_size = 64) {
    #pragma omp parallel for collapse(2)
    for (int ii = 0; ii < n; ii += block_size) {
        for (int jj = 0; jj < n; jj += block_size) {
            for (int kk = 0; kk < n; kk += block_size) {
                for (int i = ii; i < min(ii + block_size, n); ++i) {
                    for (int j = jj; j < min(jj + block_size, n); ++j) {
                        double sum = 0.0;
                        for (int k = kk; k < min(kk + block_size, n); ++k) {
                            sum += A[i][k] * B[k][j];
                        }
                        #pragma omp atomic
                        C[i][j] += sum;
                    }
                }
            }
        }
    }
}
```

```
void run_benchmark(int n, int num_threads) {
    vector<vector<double>> A(n, vector<double>(n));
    vector<vector<double>> B(n, vector<double>(n));
    vector<vector<double>> C_seq(n, vector<double>(n, 0.0));
    vector<vector<double>> C_strip(n, vector<double>(n, 0.0));
    vector<vector<double>> C_block(n, vector<double>(n, 0.0));

    initialize_matrices(A, B, n);
    omp_set_num_threads(num_threads);

    if (num_threads == 1) {
        auto start = high_resolution_clock::now();
        sequential_multiply(A, B, C_seq, n);
        auto end = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(end - start).count();
        cout << setw(10) << n << setw(15) << num_threads << setw(25) << duration << " ms";
    }

    fill(C_strip.begin(), C_strip.end(), vector<double>(n, 0.0));
    auto start_strip = high_resolution_clock::now();
    parallel_strip_multiply(A, B, C_strip, n);
    auto end_strip = high_resolution_clock::now();
    auto duration_strip = duration_cast<milliseconds>(end_strip - start_strip).count();

    fill(C_block.begin(), C_block.end(), vector<double>(n, 0.0));
    auto start_block = high_resolution_clock::now();
    parallel_block_multiply(A, B, C_block, n);
    auto end_block = high_resolution_clock::now();
    auto duration_block = duration_cast<milliseconds>(end_block - start_block).count();

    if (num_threads == 1) {
        cout << setw(25) << duration_strip << " ms" << setw(25) << duration_block << " ms" <<
endl;
    } else {
        cout << setw(10) << n << setw(15) << num_threads
             << setw(25) << "-" << setw(25) << duration_strip << " ms"
             << setw(25) << duration_block << " ms" << endl;
    }
}

int main() {
    vector<int> sizes = {256, 512, 1024};
    vector<int> threads = {1, 2, 3, 4};

    cout <<
"=============================================================================================
=======" << endl;
    cout << setw(10) << "Size" << setw(15) << "Threads" << setw(25) << "Sequential (ms)"
         << setw(25) << "Strip (ms)" << setw(25) << "Block (ms)" << endl;
    cout <<
"=============================================================================================
=======" << endl;

    for (int n : sizes) {
        for (int t : threads) {
            run_benchmark(n, t);
        }
        cout << "-----------------------------------------------------------------------------
----------------------" << endl;
    }

    return 0;
}
```

Приведём диаграммы, отражающие состояния потоков:





Зафиксируем показатели работы алгоритмов:

Size	Threads	Sequential (ms)	Strip (ms)	Block (ms)
256	1	44 ms	41 ms	46 ms
256	2	-	13 ms	10 ms
256	3	-	17 ms	16 ms
256	4	-	10 ms	12 ms
512	1	190 ms	172 ms	120 ms
512	2	-	89 ms	61 ms
512	3	-	103 ms	57 ms
512	4	-	90 ms	56 ms
1024	1	2410 ms	2442 ms	1180 ms
1024	2	-	1149 ms	557 ms
1024	3	-	770 ms	495 ms
1024	4	-	604 ms	374 ms

Вывод. Мы рассмотрели процесс перемножения матриц, изобразили его схематично, после чего, изучив возможности распараллеливания, выполнили ленточное и блочное перемножение матриц с замером времени для случаев последовательного и параллельного на 2-х, 3-х и 4-х потоках.