



Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования  
«Московский государственный технический университет имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**Лабораторная работа №4  
по дисциплине «Анализ алгоритмов»**

Тема Программирование параллельных потоков

Студент Байгарин Алан

Группа ИУ7-52Б

Преподаватели Волкова Л.Л., Строганов Д.В., Строганов Ю.В.

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ</b>	4
<b>1 Аналитическая часть</b>	5
1.1 Определения	5
1.2 Задание к лабораторной работе	6
1.3 Описание алгоритмов	6
1.3.1 Основные положения последовательного алгоритма	6
1.3.2 Основные положения параллельного алгоритма	6
<b>2 Конструкторская часть</b>	9
2.1 Схемы алгоритмов	9
2.1.1 Последовательного алгоритм	9
2.1.2 Параллельный алгоритм	12
<b>3 Технологическая часть</b>	15
3.1 Средства реализации	15
3.2 Реализация алгоритмов	15
3.3 Функциональные тесты	20
<b>4 Исследовательская часть</b>	21
4.1 Характеристики ЭВМ	21
4.2 Время выполнения алгоритмов	21
<b>ЗАКЛЮЧЕНИЕ</b>	25
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	26

# ВВЕДЕНИЕ

Целью данной лабораторной работы является разработка и сравнительный анализ последовательного и параллельного алгоритмов.

Для достижения поставленной цели необходимо решить следующие задачи:

- 1) описать последовательный алгоритм решения задачи согласно персональному варианту;
- 2) разработать параллельную версию алгоритма;
- 3) реализовать обе версии алгоритма;
- 4) выполнить сравнительный анализ зависимостей времени решения задачи от размерности входа для реализации последовательного алгоритма и для реализации модифицированного алгоритма, запущенного с единственным рабочим потоком;
- 5) выполнить сравнительный анализ зависимостей времени решения задачи от размерности входа для реализации модифицированного алгоритма при  $k$  рабочих потоках,  $k$  принимает значения  $1, 2, \dots, 8 \cdot q$ , где  $q$  — количество логических ядер процессора ЭВМ;
- 6) сформулировать рекомендацию о выборе  $k$  для решения задачи на ЭВМ.

# 1 Аналитическая часть

В данной части будут приведены задание к лабораторной работе, необходимые для его выполнения определения, а также описаны используемые для его решения алгоритмы.

## 1.1 Определения

*Простым графом*  $G(V, E)$  называется совокупность двух множеств — непустого множества  $V$  и множества  $E$  неупорядоченных пар различных элементов множества  $V$ . Множество  $V$  называется множеством *вершин*, множество  $E$  называется множеством *ребер*. [1, с. 5]

$$G(V, E) = \langle V, E \rangle, V \neq \emptyset, E \in V \times V$$

Вместо термина "*простой граф*" далее будет употребляться термин "*граф*".

Число вершин графа далее будет обозначаться как  $p$ , число ребер — как  $q$ .

Вершины графа далее будут обозначаться буквой  $v$  с указанием номера вершины ( $v_1, v_2, \dots, v_p$ ), ребра графа — буквой  $e$  с указанием номера ребра ( $e_1, e_2, \dots, e_q$ ).

Если элементами множества  $E$  являются *упорядоченные* пары (т.е. пары, в которых фиксирован порядок элементов), то граф называется *ориентированным* (или *орграфом*). В этом случае элементы множества  $V$  называются *узлами*, а элементы множества  $E$  — *дугами*. Первую вершину упорядоченной пары называют *началом дуги*, вторую — *концом*. [1, с. 5]

*Маршрутом* в графе называется последовательность вершин и ребер вида  $v_0e_1v_1e_2 \dots e_kv_k$  в которой  $e_i = (v_{i-1}, v_i)$ . Если все ребра в маршруте различны, то маршрут называется *цепью*. В цепи  $v_0e_1v_1e_2 \dots e_kv_k$ , вершины  $v_0, v_k$  называются *концами цепи*. Говорят, что цепь с концами  $u, v$  соединяет вершины  $u, v$  (обозначается  $\langle u, v \rangle$ ). Для орграфов цепь называется *путём*. [1, с. 15]

*Нагруженным*, или *взвешенным* графом  $G(V, E)$  называется граф, каждому ребру которого сопоставлено некоторое число, называемое *весом*. [1, с. 48]

Взвешенный граф может быть *матрицей смежности*  $C_{p \times p}$ , для которой элемент  $c_{i,j}$  равен весу ребра  $(i, j)$ . Если  $(i, j) \notin E$ , то  $c_{i,j} = \inf$ . [1, с. 48]

*Минимальным путём*  $\langle u, v \rangle_{min}$  в графе называется путь с наименьшим суммарным весом ребер. [1, с. 48]

Вес минимального пути  $\langle u, v \rangle_{min}$  далее будет обозначаться как  $w(u, v)$ .

*Поток* — это динамический объект, в процессе представленный отдельной точкой управления и выполняющий последовательность команд, отчитывающую от этой точки (выполняемая параллельно в рамках процесса часть программы). [2, с. 103]

*Condition variable* (переменная условия) — примитив синхронизации для блокировки потоков на переменной в ожидании модификации разделяемой памяти (условия) и

уведомления от другого потока. [3, с. 1231]

*Мьюотекс* — примитив синхронизации, используемый для организации монопольного доступа к разделенному ресурсу для защиты от гонок и синхронизации между несколькими потоками. [3, с. 1220]

## 1.2 Задание к лабораторной работе

Дан орграф. Найти все пары вершин на расстоянии (сумма меток дуг на пути) не большем, чем заданная на входе действительная величина.

Иначе — для данных  $G = (V, E)$ ,  $s \in \mathbb{R}$  найти все  $(v_i, v_j) : w(v_i, v_j) \leq s$

## 1.3 Описание алгоритмов

### 1.3.1 Основные положения последовательного алгоритма

Алгоритм поиска всех пар вершин с весом пути не большей, чем заданная величина  $s$ , основывается на *алгоритме Флойда* для поиска минимальных путей между всеми парами вершин графа [1, с. 64], с последующей выборкой тех, для которых такое расстояние меньше или равно  $s$ .

Для решения поставленной задачи нет необходимости хранить информацию о самом пути, только о его длине, из-за чего в алгоритме Флойда будет использоваться только матрица минимальных путей.

### 1.3.2 Основные положения параллельного алгоритма

Алгоритм Флойда на  $k$ -й итерации рассчитывает матрицу  $D_{p \times p}$  минимальных путей, проходящих через вершины  $1, \dots, k$ , в которой  $d_{i,j}$  — вес пути. Суть поиска состоит в сравнении веса пути  $v_i(v_i, v_k)v_k(v_k, v_j)v_j$  с весом  $v_i(v_i, v_j)v_j \equiv d_{i,j}$ , и замене  $d_{i,j}$  на вес первого пути в случае, если он оказался меньше.

Для параллельного выполнения алгоритма, матрицу  $D$  можно разбить на блоки размером  $b \times b$ ,  $b = \lfloor p \div \sqrt{n} \rfloor$ , где  $n$  - число потоков (разбиение на блоки меньшего размера бессмысленно, т.к. иначе максимальное число блоков будет больше числа потоков), и применить алгоритм Флойда к каждому такому блоку. При таком подходе итерации будут происходить не по вершинам графа, а по числу блоков  $b$ .

Обозначим блок как  $W_{x,y}$ , где  $x, y$  - его координаты в матрице  $D$ . Для  $k$ -й итерации по блокам, согласно алгоритму, значения в любом блоке  $W_{i,j}$ ,  $i \neq k, j \neq k$  зависят от значений внутри блока  $W_{i,k}$  и значений внутри блока  $W_{k,j}$ , значения в блоке  $W_{i,k}$  зависят от других значений внутри самого себя и от значений внутри блока  $W_{k,k}$ , значения внутри

блока  $W_{k,j}$  также зависят от значений внутри самого себя и внутри блока  $W_{k,k}$ , а значения внутри блока  $W_{k,k}$  зависят только от значений внутри самого себя. Таким образом, при рассчитанных  $W_{i,k}$  и  $W_{k,j}$  можно параллельно рассчитать все блоки  $W_{i,j}$ , а при имеющемся  $W_{k,k}$  можно рассчитать все  $W_{i,k}$  и  $W_{k,j}$ . Следовательно, для корректного расчета  $k$ -й итерации по блокам, нужно применить последовательный алгоритм Флойда к блоку  $W_{k,k}$ , затем параллельно применить его ко всем блокам  $W_{i,k}$  и  $W_{k,j}$ ,  $i \neq k, j \neq k$ , затем применить алгоритм ко всем остальным блокам разбиения. Проделав это для каждого значения  $k$ , получится идентичная последовательному алгоритму матрица кратчайших расстояний.

Иллюстрация информационных зависимостей на каждой фазе параллельного алгоритма при  $k = 1$  и  $n = 16$  приведена на рисунке 1.1.

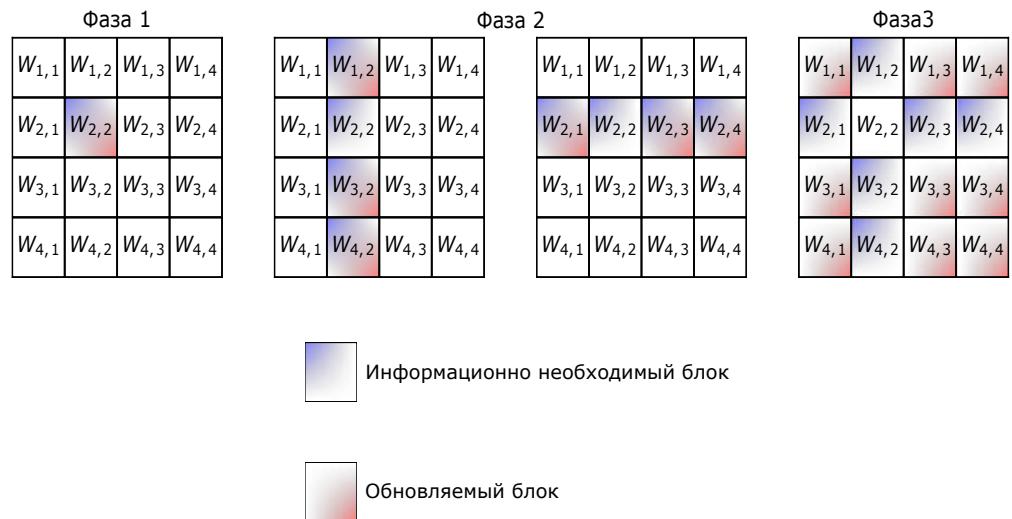


Рисунок 1.1 — Фазы блочного алгоритма Флойда для 2-й итерации

Следует отметить, что в случае, если  $p$  не кратно  $b$ , на краях матрицы появятся неравномерные прямоугольные блоки, тем не менее — они обрабатываются аналогично основным.

Для параллельных вычислений будет использована собственная реализация пул потоков. В простейшем случае пул состоит из фиксированного числа рабочих потоков. Когда у программы появляется какая-то работа, она вызывает функцию, которая помещает эту работу в очередь. Рабочий поток забирает работу из очереди, выполняет указанную в ней задачу, после чего проверяет, есть ли в очереди другие работы. [4, с. 383]

Для работы пула потоков необходима система сообщений между потоками (в частности - для оповещения потоков о появлении новой задачи, а также об опустошении очереди задач). Для осуществления этого используется переменная условия. Для корректной работы с переменной условия также используется мьютекс.

## **Вывод**

В данной части были приведены ключевые определения, связанные с орграфом, и способ его представления, а также определения примитивов, необходимых для организацией параллельной работы программы. Помимо этого было приведено задание на лабораторную работу и описаны основы последовательного и параллельного алгоритмов, необходимых для его выполнения.

## 2 Конструкторская часть

В этой части будут представлены схемы разработанных алгоритмов.

### 2.1 Схемы алгоритмов

#### 2.1.1 Последовательного алгоритм

На рисунке 2.1 приведена схема последовательного алгоритма по поиску пар вершин, находящихся на расстоянии меньше заданного.

На рисунке 2.2 приведена схема алгоритма создания матрицы минимальных расстояний из матрицы смежности.

На рисунке 2.3 приведена схема алгоритма выбора пар вершин на расстоянии меньше заданного из матрицы минимальных расстояний.

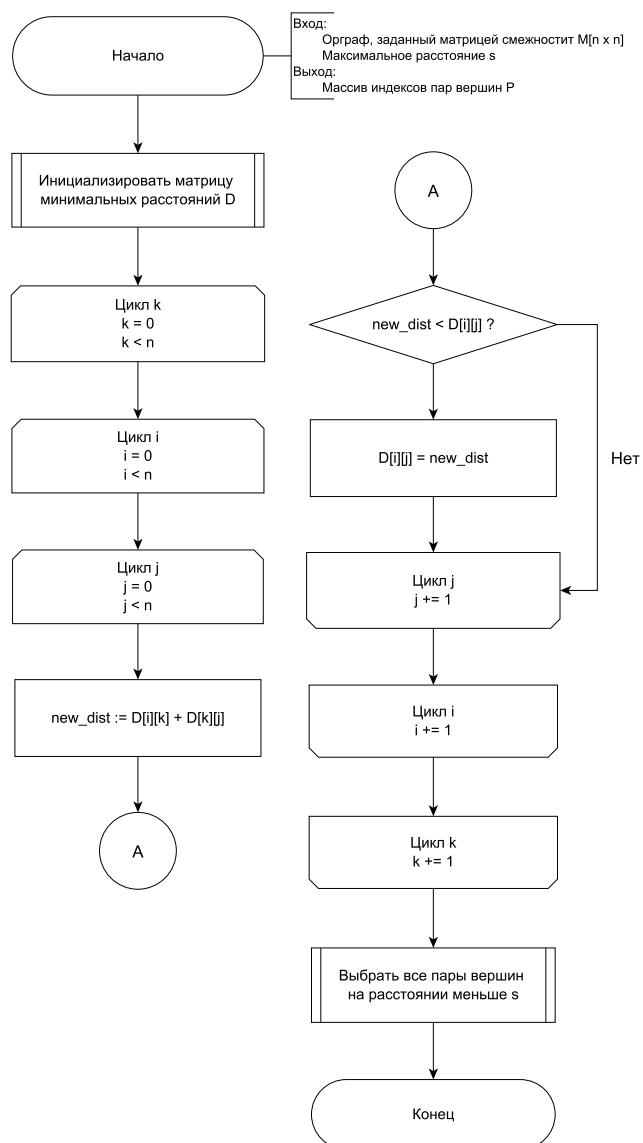


Рисунок 2.1 — Схема последовательного алгоритма

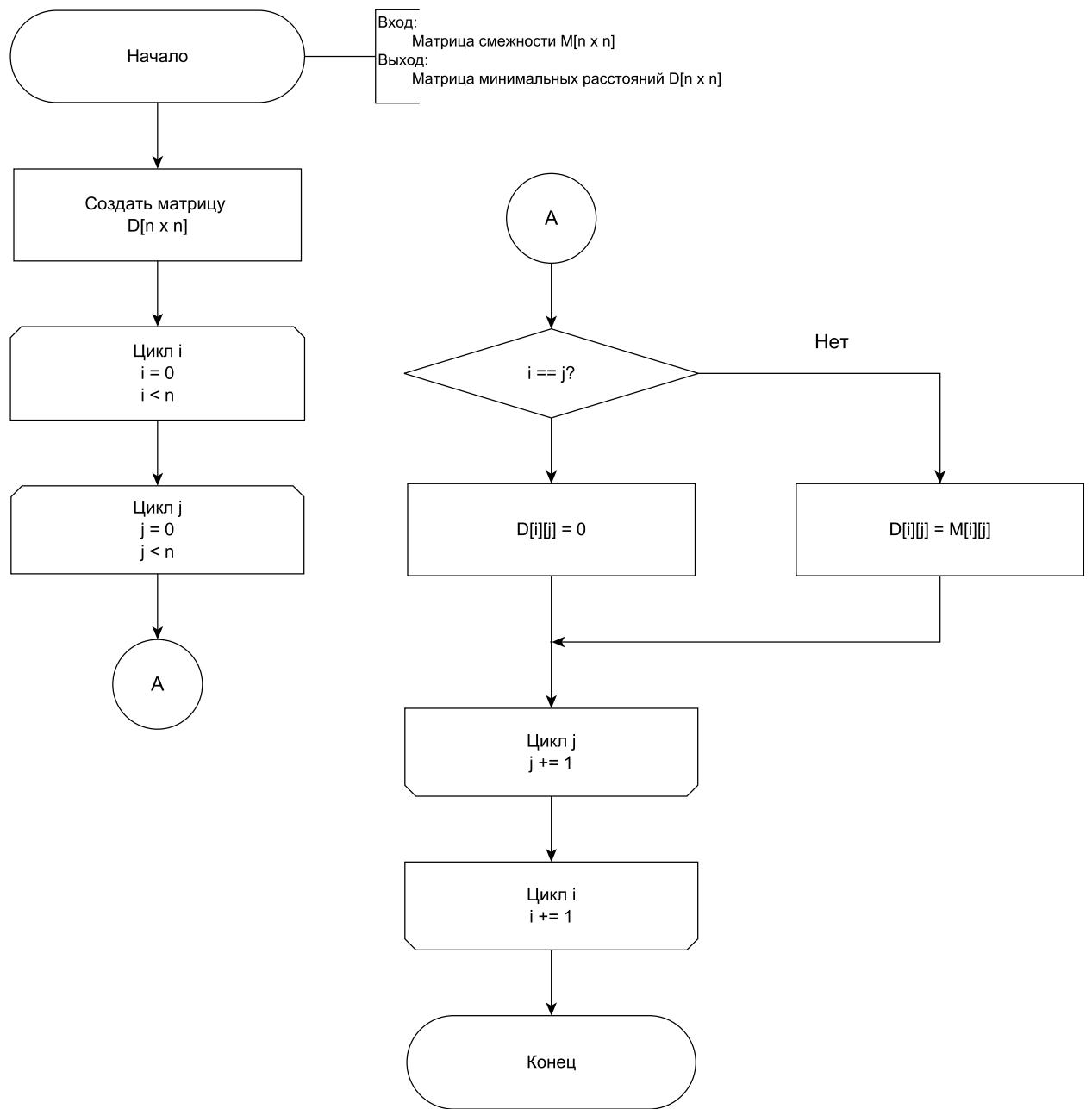


Рисунок 2.2 — Схема алгоритма инициализации матрицы кратчайших расстояний

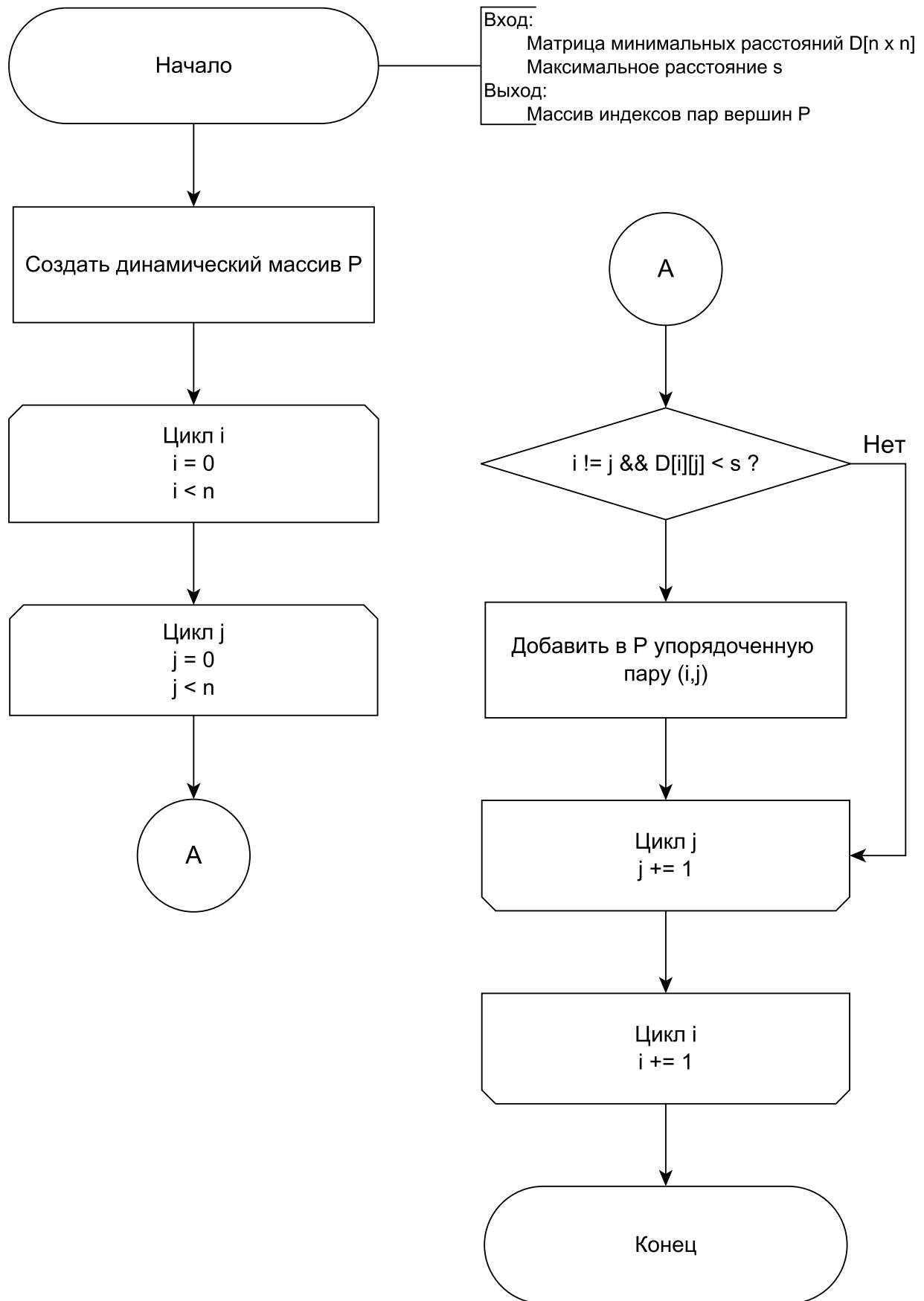


Рисунок 2.3 — Схема алгоритма выбора пар на расстоянии меньше заданного

## 2.1.2 Параллельный алгоритм

На рисунке 2.4 приведена схема алгоритма главного потока.

На рисунке 2.5 приведена схема алгоритма рабочего потока, отвечающая за обработку одного блока матрицы кратчайших расстояний.

На рисунке 2.6 приведены схемы алгоритмов методов пула потоков, необходимых для реализации параллельного алгоритма.

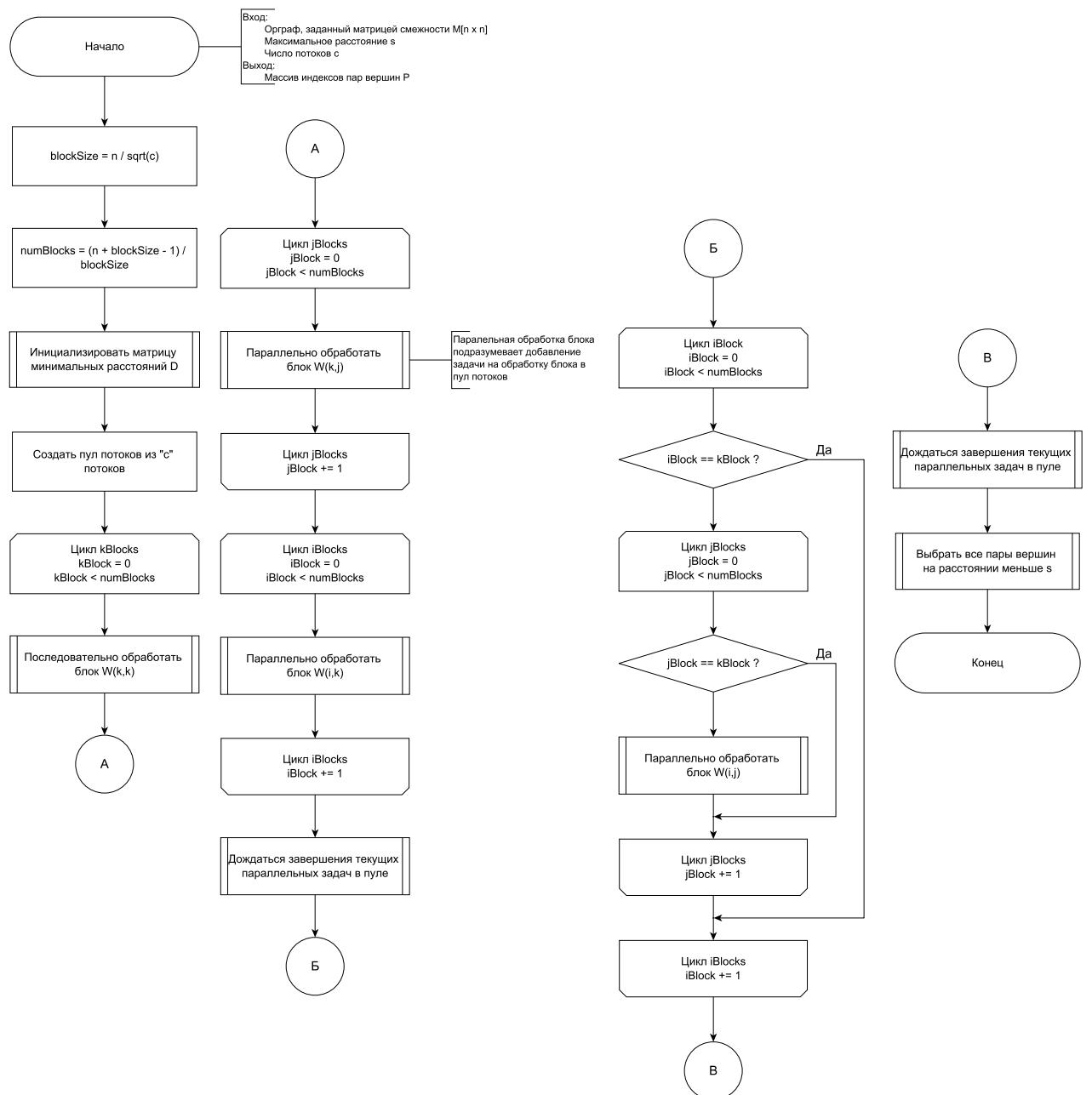


Рисунок 2.4 — Главный поток параллельного алгоритма

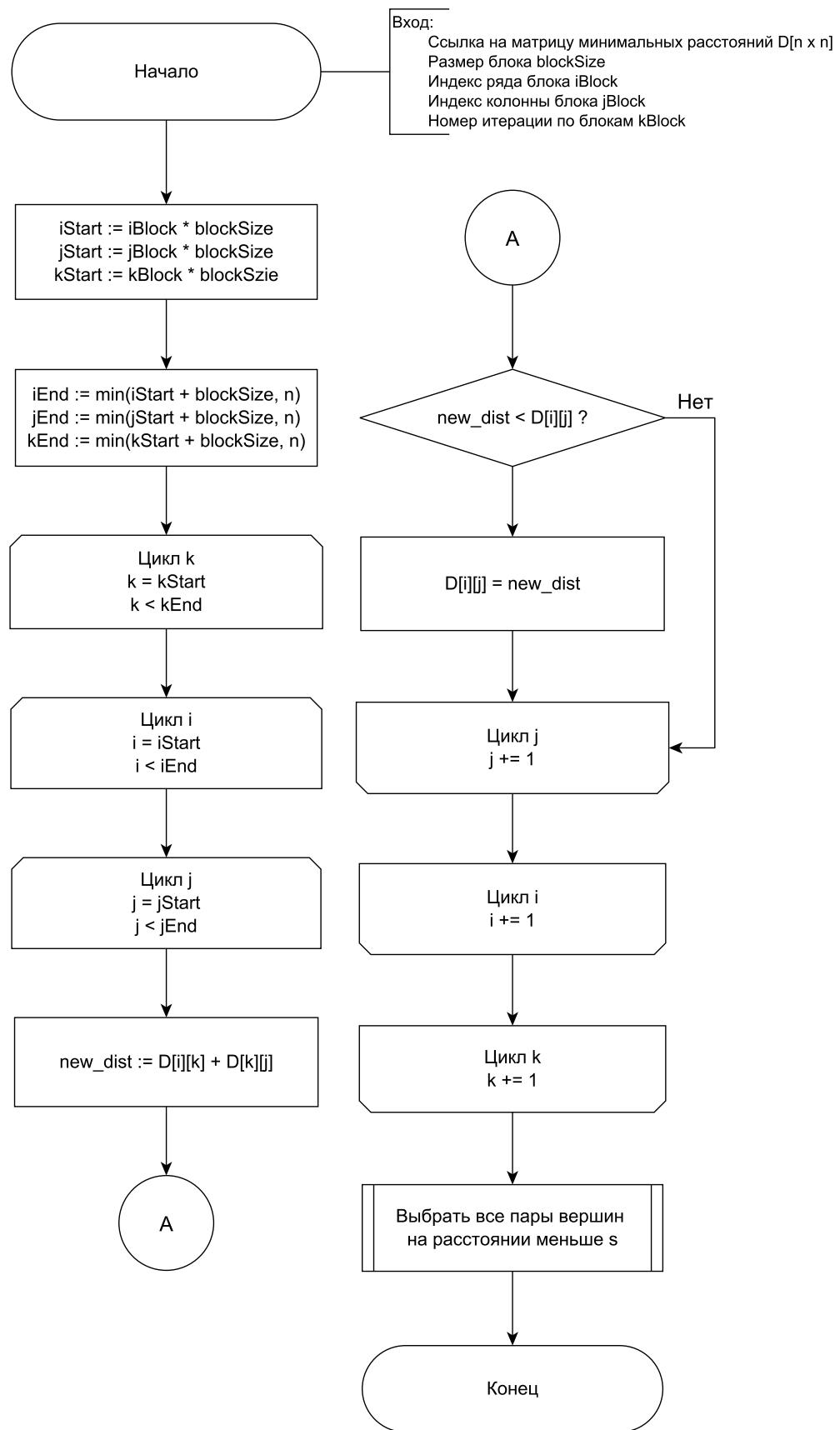


Рисунок 2.5 — Рабочий поток параллельного алгоритма — алгоритм обработки одного блока матрицы минимальных расстояний

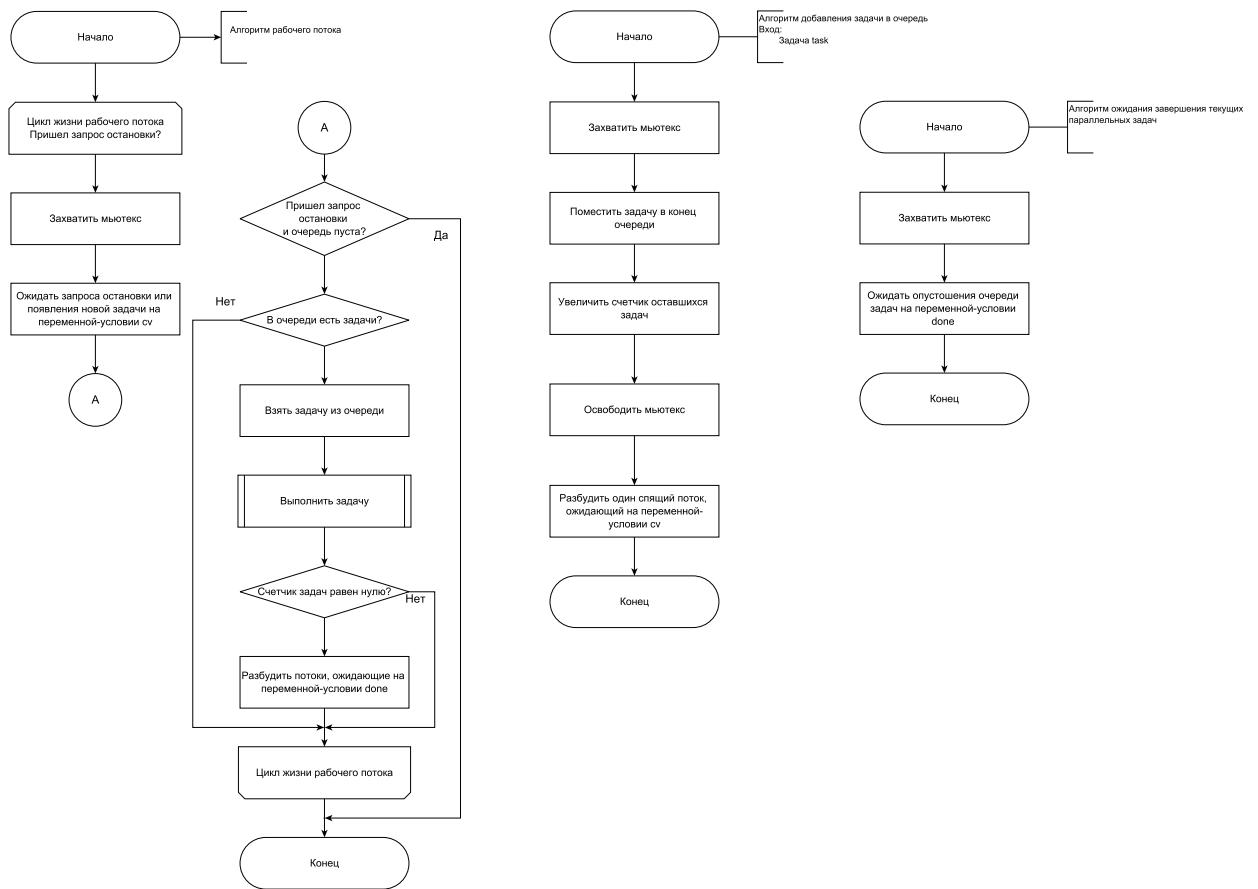


Рисунок 2.6 — Схемы алгоритмов методов пула потоков

## Вывод

В данной части были приведены схемы последовательного и параллельного алгоритмов, а также схемы алгоритмов методов пула потоков, необходимые для реализации параллельного алгоритма.

### 3 Технологическая часть

В данном разделе будут перечислены средства реализации, листинги кода и функциональные тесты.

#### 3.1 Средства реализации

Требования к языку программирования:

- статическая типизация;
- гарантированное время выполнения.

В качестве языка программирования, соответствующего указанным требованиям, для выполнения данной лабораторной работы был выбран C++ [3].

Для замеров реального времени используется *high\_resolution\_clock* [3, с. 1016] библиотеки *chrono* [3, с. 1009–1018].

Для создания автоматически присоединяемых потоков с возможностью отправки в них запроса остановки используется *jthread* [5, с. 244].

Для создания мьютекса используется *mutex* библиотеки *mutex* [3, с. 1220].

Для создания блокировки используются *lock\_guard::lock()* библиотеки *mutex* [3, с. 1220].

Для создания переменных условий используется *condition\_variable* из одноимённой библиотеки [3, с. 1231].

#### 3.2 Реализация алгоритмов

В листинге 3.1 приведена реализация последовательного алгоритма поиска пар вершин графа, расстояние между которыми меньше заданной действительной величины.

В листинге 3.2 приведена реализация главного потока параллельного алгоритма поиска пар вершин графа, расстояние между которыми меньше заданной действительной величины.

В листинге 3.3 приведена реализация рабочего потока параллельного алгоритма поиска пар вершин графа, расстояние между которыми меньше заданной действительной величины — алгоритма обработки одного блока матрицы минимальных расстояний.

В листинге 3.4 приведены реализации методов пула потоков, необходимых для работы параллельного алгоритма.

```
vector<Pair>
find_pairs_within_range__sequential(const vector<vector<uint>> &adj ,
                                     double max_distance) {
    size_t n = adj.size();
    double INF = numeric_limits<double>::infinity();
    vector<vector<double>> dist(n, vector<double>(n, INF));
```

```

for (size_t i = 0; i < n; ++i) {
    dist[i][i] = 0;
    for (size_t j = 0; j < n; ++j) {
        if (adj[i][j] == numeric_limits<uint>::infinity())
            continue;
        dist[i][j] = static_cast<double>(adj[i][j]);
    }
}
for (size_t k = 0; k < n; ++k) {
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < n; ++j) {
            double new_dist = dist[i][k] + dist[k][j];
            if (new_dist < dist[i][j]) {
                dist[i][j] = new_dist;
            }
        }
    }
}
vector<Pair> pairs;
for (size_t i = 0; i < n; ++i) {
    for (size_t j = 0; j < n; ++j) {
        if (i != j && dist[i][j] <= max_distance)
            pairs.emplace_back(i, j);
    }
}
return pairs;
}

```

Листинг 3.1 — Реализация последовательного алгоритма

```

vector<Pair> find_pairs_within_range_parallel(vector<vector<uint>> &
                                              adj,
                                              double max_dist,
                                              size_t thread_count) {
    size_t n = adj.size();
    int blockSize = static_cast<double>(n) / sqrt(thread_count);
    blockSize = max(blockSize, 1);
    int numBlocks = (static_cast<int>(n) + blockSize - 1) / blockSize;
    const double INF = numeric_limits<double>::infinity();
    vector<vector<double>> dist(n, vector<double>(n, INF));
    for (size_t i = 0; i < n; ++i) {
        dist[i][i] = 0;
    }
    for (size_t k = 0; k < n; ++k) {
        for (size_t i = 0; i < n; ++i) {
            for (size_t j = 0; j < n; ++j) {
                double new_dist = dist[i][k] + dist[k][j];
                if (new_dist < dist[i][j]) {
                    dist[i][j] = new_dist;
                }
            }
        }
    }
}

```

```

        for (size_t j = 0; j < n; ++j) {
            if (adj[i][j] == numeric_limits<uint>::infinity())
                continue;
            dist[i][j] = static_cast<double>(adj[i][j]);
        }
    }

ThreadPool pool(thread_count);

for (int kBlock = 0; kBlock < numBlocks; ++kBlock) {
    floyd_marshall_block(dist, blockSize, kBlock, kBlock, kBlock);

    for (int jBlock = 0; jBlock < numBlocks; ++jBlock) {
        if (jBlock != kBlock) {
            pool.enqueue([&, kBlock, jBlock, n, blockSize]() {
                floyd_marshall_block(dist, blockSize, kBlock, jBlock, kBlock)
                ;
            });
        }
    }
}

for (int iBlock = 0; iBlock < numBlocks; ++iBlock) {
    if (iBlock != kBlock) {
        pool.enqueue([&, kBlock, iBlock, n, blockSize]() {
            floyd_marshall_block(dist, blockSize, iBlock, kBlock, kBlock)
            ;
        });
    }
}
pool.wait_done();
for (int iBlock = 0; iBlock < numBlocks; ++iBlock) {
    if (iBlock == kBlock)
        continue;
    for (int jBlock = 0; jBlock < numBlocks; ++jBlock) {
        if (jBlock == kBlock)
            continue;
        pool.enqueue([&, iBlock, jBlock, kBlock, n, blockSize]() {
            floyd_marshall_block(dist, blockSize, iBlock, jBlock, kBlock)
            ;
        });
    }
}

```

```

    }

    pool.wait_done();

}

vector<Pair> result;
for (size_t i = 0; i < n; ++i) {
    for (size_t j = 0; j < n; ++j) {
        if (i != j && dist[i][j] <= max_dist) {
            result.emplace_back(i, j);
        }
    }
}

return result;
}

```

Листинг 3.2 — Реализация главного потока параллельного алгоритма

```

void floyd_marshall_block(vector<vector<double>> &dist, int blockSize,
                           int blockRow, int blockCol, int kBlock) {
    int n = dist.size();
    int rowStart = blockRow * blockSize;
    int colStart = blockCol * blockSize;
    int kStart = kBlock * blockSize;

    int rowEnd = min(rowStart + blockSize, n);
    int colEnd = min(colStart + blockSize, n);
    int kEnd = min(kStart + blockSize, n);

    for (int k = kStart; k < kEnd; ++k) {
        for (int i = rowStart; i < rowEnd; ++i) {
            for (int j = colStart; j < colEnd; ++j) {
                auto new_dist = dist[i][k] + dist[k][j];
                if (new_dist < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}

```

Листинг 3.3 — Реализация рабочего потока параллельного алгоритма

```

void ThreadPool::enqueue(function<void()> task) {

```

```

    {
        unique_lock lock(m);
        tasks.emplace(move(task));
        ++pending_tasks;
    }
    cv.notify_one();
}

void ThreadPool::wait_done() {
    unique_lock lock(m);
    done_cv.wait(lock, [this]() { return pending_tasks == 0; });
}

void ThreadPool::worker_thread(stop_token stoken) {
    while (!stoken.stop_requested()) {
        function<void()> task;
        {
            unique_lock lock(m);
            cv.wait(lock, stoken, [this, &stoken] {
                return stoken.stop_requested() || !tasks.empty();
            });

            if (stoken.stop_requested() && tasks.empty())
                break;

            if (!tasks.empty()) {
                task = move(tasks.front());
                tasks.pop();
            } else {
                continue;
            }
        }
        task();
    }
    unique_lock lock(m);
    --pending_tasks;
    if (pending_tasks == 0) {
        done_cv.notify_all();
    }
}
}

```

{}

Листинг 3.4 — Реализации методов пула потоков

### 3.3 Функциональные тесты

Тестирование проведено по методологии чёрного ящика. Функциональные тесты приведены в таблице 3.1

Таблица 3.1 — Функциональные тесты для реализаций алгоритмов поиска пар вершин на расстоянии меньшем, чем данная на вход действительная величина

№	Описание	Массив граней	Веса	Порог	Обычный алгоритм	Параллельный алгоритм
1	Несвязанный граф	()	()	1	[ ]	[ ]
2	Связанный граф без искомых пар вершин	((1, 2), (2, 3), (3, 1))	(3, 4, 5)	2	[ ]	[ ]
3	Связанный граф с имеющимися искомыми парами	((1, 2), (2, 3), (3, 4), (1, 3))	(2, 1, 3, 4)	3	[[1, 2], [2, 3], [1, 3]]	[[1, 2], [2, 3], [1, 3]]

Все тесты пройдены успешно.

### Вывод

В данном разделе были перечислены использованные технические средства, приведены и протестираны реализации последовательного и параллельного алгоритмов поиска пар вершин на расстоянии меньше заданной величины. В результате тестирования корректность реализаций была подтверждена.

## 4 Исследовательская часть

### 4.1 Характеристики ЭВМ

В листинге 4.1 приведены характеристики ЭВМ и системы, использованных для проведения замеров времени выполнения реализаций алгоритмов.

```
systeminfo.exe:

OS name:                               Microsoft Windows 11 Home for
                                         single language
OS version:                            10.0.26100 N/D build 26100
OS Configuration:                      Isolated work station

OS Build Type:                          Multiprocessor Free
System Model:                           ASUS Zenbook 14
                                         UX3405MA_UX3405MA
System Type:                            x64-based PC

Processors:                             Number of processors - 1.
[01]: Intel64 Family 6 Model 170 Stepping 4 GenuineIntel ~1400 MHz

BIOS Version:                           American Megatrends
                                         International, LLC. UX3405MA.308, 23.07.2024

Total Physical Memory:                 15,741 MB
Virtual Memory: Max Size:             22,653 MB

wmic cpu:

Name                                    Intel(R) Core(TM) Ultra 7 155H
NumberOfCores                         16
NumberOfEnabledCore                   16
NumberOfLogicalProcessors            22
```

Листинг 4.1 — Характеристика ЭВМ — частичный вывод команд *systeminfo.exe* и *wmic cpu*

### 4.2 Время выполнения алгоритмов

Время работы каждой реализации считалось как среднее арифметическое из 100 повторений. На вход подавались граф с числом вершин от 10 до 2000, для параллельного

алгоритма — число потоков от  $2^1$  до  $2^7$ .

В таблицах 4.1–4.3 приведены результаты замеров времени выполнения последовательного алгоритма и параллельного алгоритма с различным числом потоков при различном количестве вершин во входном графе. Число потоков равное 0 соответствует времени выполнения последовательной реализации алгоритма.

Таблица 4.1 — Замер времени выполнения реализаций, часть 1

Размер графа	Число потоков		
	0	1	2
10	$2.308\,82 \times 10^{-5}$	0.000 130 843	0.000 129 115
50	0.000 892 585	0.001 004 13	0.001 287 85
100	0.006 935 08	0.007 562 51	0.007 889 14
300	0.169 208	0.181 418	0.150 983
500	0.835 674	0.851 258	0.679 207
1000	6.722 19	6.791 04	5.509 54
2000	52.3328	55.4725	43.5272

Таблица 4.2 — Замер времени выполнения реализаций, часть 2

Размер графа	Число потоков		
	4	8	16
10	0.000 218 367	0.000 593 067	0.001 075 33
50	0.001 376 32	0.000 886 386	0.001 266 67
100	0.008 455 87	0.004 932 97	0.004 115 63
300	0.143 859	0.075 577 3	0.052 523 4
500	0.668 956	0.351 325	0.220 384
1000	5.361 54	2.856 11	1.919 15
2000	41.5779	23.4642	16.3222

Таблица 4.3 — Замер времени выполнения реализаций, часть 3

Размер графа	Число потоков		
	32	64	128
10	0.005 966 04	0.006 881 62	0.008 511 11
50	0.002 816 14	0.005 526 92	0.014 393
100	0.003 926 5	0.006 561	0.016 791 9
300	0.040 772 7	0.037 503 5	0.037 576 2
500	0.162 833	0.146 575	0.139 394
1000	1.417 74	1.222 45	1.201 76
2000	12.0204	11.3218	8.181 24

**Замечание:** вопреки заданию к лабораторной работе, замеры времени выполнения для графов с числом вершин, равным 3000, не были произведены. При попытке их проведения, процессор ЭВМ перегревался, что приводило к аварийному завершению работы.

Из-за опасности для ЭВМ, а также из-за низкой степени доверия к результатам, полученным в подобных условиях, после 2-х безуспешных попыток было принято решение не проводить замеры на графах размером более 2000 вершин.

На рисунке 4.1 приведён графики времён выполнения реализаций от размера графа.

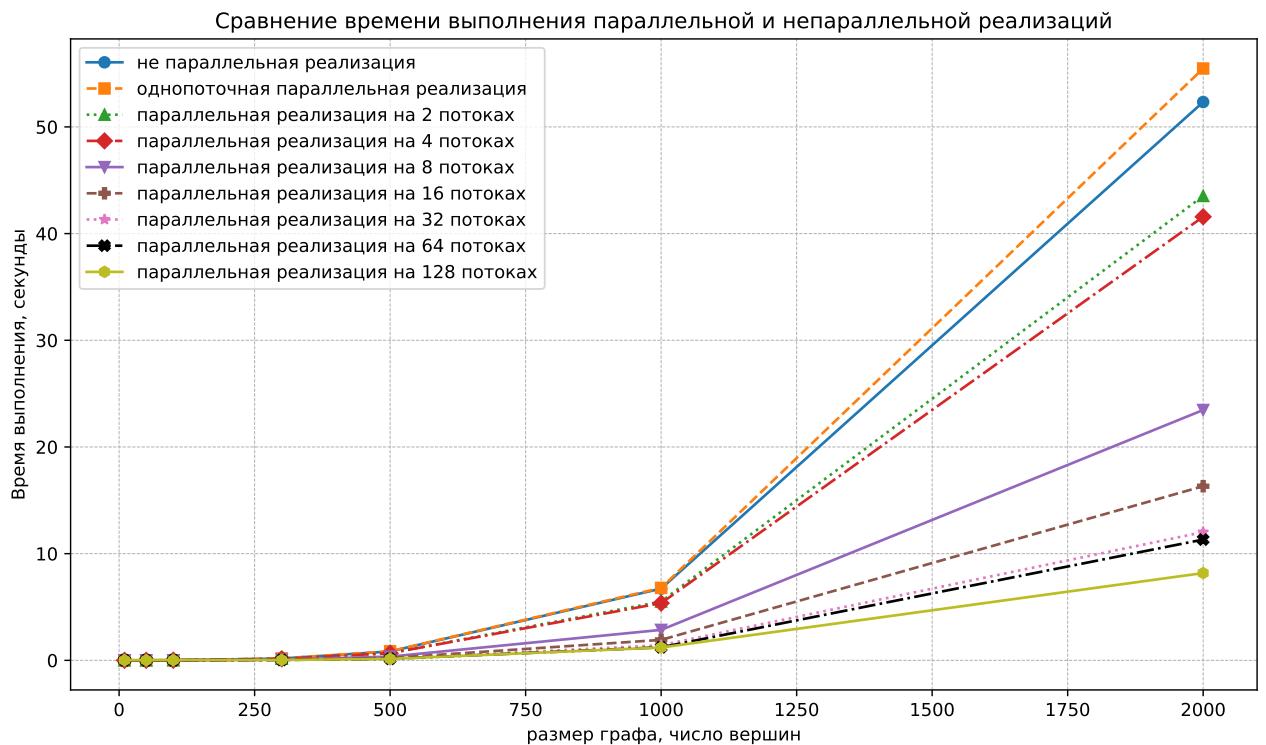


Рисунок 4.1 — Сравнение реализаций последовательного алгоритма и параллельного алгоритма с разным числом потоков

## Вывод

В данной части были проведены исследования зависимости реального времени выполнения реализаций последовательного алгоритма и параллельного алгоритма с разным числом потоков от числа вершин в графе. По результатам замеров, представленных в таблицах 4.1–4.3 установлено следующее:

- при числе вершин в графе  $\leq 50$ , реализация последовательного алгоритма выполняется быстрее реализаций параллельного с любым числом потоков. Это связано с накладными расходами на создание и завершение потоков, а также с дополнительными вычислениями внутри потоков, возникающими из-за особенностей блочного алгоритма;
- при числе вершин = 100, быстрее всех выполняется параллельная реализация с 32-я потоками;
- при числе вершин = 300, быстрее всех выполняется параллельная реализация с 64-я потоками;
- при числе вершин  $\geq 300$ , быстрее всех выполняется параллельная реализация с

максимальным числом потоков (равным 128).

А также:

- 1) при числе вершин равном 100, реализация последовательного алгоритма выполняется быстрее реализации параллельного с числом потоков в только промежутке 1–4.
- 2) при большем числе вершин во входном графе, последовательная реализация всегда выигрывает только у параллельной реализации с одним потоком, т.к. при одном рабочем потоке алгоритм идентичен последовательному, но добавляет накладные расходы на создание потока.

# ЗАКЛЮЧЕНИЕ

Все поставленные задачи, а именно:

- 1) описание последовательного алгоритма решения задачи согласно персональному варианту;
- 2) разработка параллельной версии алгоритма;
- 3) реализация обеих версий алгоритма;
- 4) сравнительный анализ зависимостей времени решения задачи от размерности входа для реализации последовательного алгоритма и для реализации модифицированного алгоритма, запущенного с единственным рабочим потоком;
- 5) сравнительный анализ зависимостей времени решения задачи от размерности входа для реализации модифицированного алгоритма при  $k$  рабочих потоках,  $k$  принимает значения  $1, 2, \dots, 8 \cdot q$ , где  $q$  — количество логических ядер процессора ЭВМ;
- 6) формулировка рекомендаций о выборе  $k$  для решения задачи на ЭВМ.

Были выполнены.

Цель лабораторной работы, заключающаяся в разработке и сравнительном анализе последовательного и параллельного алгоритмов, была достигнута.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ю.Б. Буркатовчкая. Теория графов. Часть 1 : учебное пособие. Томкс: Изд-во Томского политехнического университета, 2014. 200 — С.
2. Ю. Вахалия. UNIX изнутри. СПб.: Питер, 2003. 844 — С.
3. Stroustrup B. The C++ Programming Language. Fourth Edition. 2013. 1366 — Р.
4. Уильямс Энтони. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. Пер. с англ. Слинкин А. А. М.: ДМК Пресс, 2012. 672 — С.
5. Б. Страуструп. Тур по C++. Третье издание. Пер. с англ. Вячеслав К. 2022. 314 — С.