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 № 5
по дисциплине «Анализ алгоритмов»

Тема Конвейерная обработка данных

Студент Куликов Н.В.

Группа ИУ7-53Б

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

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель: разработка и реализация конвейерной обработки набора заявок на кластеризацию вершин графа по алгоритму DBSCAN.

Для достижения поставленной цели необходимо было выполнить следующие задачи:

1) описать последовательный алгоритм DBSCAN;
2) разработать параллельную версию алгоритма;
3) реализовать обе версии алгоритма;
4) реализовать последовательную обработку N заявок;
5) разработать параллельный конвейер обработки заявок, включающий три обслуживающих устройства, взаимодействующих через очереди;
6) реализовать конвейер;
7) провести замеры времени обработки $N \in \{25, 50, 75, 100, 125\}$ заявок для последовательной и конвейерной реализаций.

1 Аналитическая часть

1.1 Параллелизм

Параллелизм — это одновременное выполнение двух или более операций. В контексте компьютеров, имеется в виду, что одна и та же система выполняет несколько независимых операций параллельно, а не последовательно [1].

1.1.1 Поток

Поток можно понимать как любой автономный последовательный (линейный) набор команд процессора [2].

Источником этого линейного кода для потока могут служить:

— бинарный исполняемый файл, на основе которого системой или вызовом группы $spawn()$ запускается новый процесс и создаётся его главный поток;

— дубликат кода главного потока процесса родителя при клонировании процессов вызовом $fork()$ (тоже относительно главного потока);

— участок кода, оформленный функцией специального типа $(void * ()$ $(void*))$, так называемой функцией потока.

1.1.2 Многопоточность

Многопоточность подразумевает разбиение задач на параллельные потоки выполнения в рамках одного процесса с помощью пользовательских потоков [3].

Потоки могут существовать только внутри процессов; не бывает такого потока, у которого не было бы владельца. Каждый процесс содержит по меньшей мере один поток, обычно называемый главным или основным [3].

1.1.3 Примитивы синхронизации

Семафор — объект, над которым можно провести две атомарные операции: инкремент и декремент внутреннего счётчика при условии, что внутренний счётчик не может принимать значение меньше нуля. Если некий поток пытается уменьшить на единицу значение внутреннего счётчика семафора, значение

которого уже равно нулю, то этот поток блокируется до тех пор, пока внутренний счётчик семафора не примет значение, равное 1 или больше (посредством воздействия на него других потоков). Разблокированный поток сможет осуществить декремент нового значения [2].

Мьютекс (от mutual exclusion – взаимное исключение) — это один из базовых примитивов синхронизации потоков. Этот элемент реализуется на уровне ядра системы и имеет широкий набор атрибутов и настроек. Назначение мьютекса — защита участка кода от совместного выполнения несколькими потоками. Такой участок кода называют критической секцией, и обычно он является областью модификации общих переменных или обращения к разделяемому ресурсу [2].

Принцип работы мьютекса заключается в следующем: при обращении потока к функции блокировки (захвата) проверяется, захвачен ли уже мьютекс, и если да, то вызвавший поток блокируется до освобождения критической секции. Если же нет, то объект мьютекс запоминает, какой поток его захватил (то есть владельца) и устанавливает признак, что он захвачен [2].

1.2 Алгоритм DBSCAN

Алгоритм DBSCAN (Density Based Spatial Clustering of Applications with Noise), плотностный алгоритм для кластеризации пространственных данных с присутствием шума, был предложен как решение проблемы разбиения (изначально пространственных) данных на кластеры произвольной формы. Большинство алгоритмов, производящих плоское разбиение, создают кластеры по форме близкие к сферическим, так как минимизируют расстояние документов до центра кластера [4].

Идея, положенная в основу алгоритма, заключается в том, что внутри каждого кластера наблюдается типичная плотность точек (объектов), которая заметно выше, чем плотность снаружи кластера, а также плотность в областях с шумом ниже плотности любого из кластеров. Ещё точнее, что для каждой точки кластера её соседство заданного радиуса должно содержать не менее некоторого числа точек, это число точек задаётся пороговым значением.

Определение 1. Eps соседство точки, обозначаемое как $N_{eps}(p)$, опреде-

ляется как множество документов, находящихся от точки p на расстояния не более Eps: $N_{eps}(p) = q \in D|dist(p,q) \leqslant Eps$. Поиска точек, чьё $N_{eps}(p)$ содержит хотя бы минимальное число точек $MinPt$ не достаточно, так как точки бывают двух видов: ядровые и граничные.

Определение 2. Точка p непосредственно плотно достижима из точки q (при заданных Eps и $MinPt$), если $p \in N_{eps}(p)$ и $|N_{eps}(p)| \geqslant MinPt$.

Определение 3. Точка p плотно достижима из точки q, если существует последовательность точек $q = p_1, p_2, ..., p_n = p : p_{i+1}$ непосредственно плотно достижимы из p_i. Это отношение транзитивно, но не симметрично в общем случае, однако симметрично для двух ядровых точек.

Определение 4. Точка p плотно связана с точкой q, если существует точка o: p и q плотно достижимы из o (при заданных Eps и $MinPt$).

Определение 5. Кластер C_j(при заданных Eps и $MinPt$) – это не пустое подмножество документов, удовлетворяющее следующих условиям:

1) $\forall p, q$: если $p \in C_j$ и p плотно достижима из q (при заданных Eps и $MinPt$), то $q \in C_j$;
2) $\forall p, q \in C_j$: p плотно связана с q (при заданных Eps и $MinPt$).

Итак, кластер – это множество плотно связанных точек. В каждом кластере содержится хотя бы $MinPt$ документов.

Шум – это подмножество документов, которые не принадлежат ни одному кластеру.

Алгоритм DBSCAN для заданных значений параметров Eps и $MinPt$ исследует кластер следующим образом:

1) выбирает случайную точку, являющуюся ядровой, в качестве затравки;
2) помещает в кластер саму затравку;
3) помещает в кластер все точки, плотно достижимые из затравки [4].

1.3 Конвейерная обработка данных

Конвейер данных — это система, которая осуществляет извлечение, преобразование и загрузку данных из различных источников в пункт назначения для дальнейшего использования [5].

Параллельная обработка данных объединяет два вида обработки данных: параллельную и конвейерную. В отличие от последовательной обработки, при

которой все операции выполняются строго одна после другой, параллелизм и конвейерность предполагают возможность совмещения во времени выполнения операций целиком или каких-либо их частей. Устройства, выполняющие операции, работают независимо друг от друга при параллельной обработке. В случае конвейерной обработки все ступени конвейера работают одновременно, но обработка данных на разных ступенях идёт с перекрытием во времени, когда результаты промежуточной обработки последовательно передаются от одной ступени конвейера к другой [5].

1.4 Вывод

В аналитической части были рассмотрены понятия параллелизма и конвейера данных, рассмотрен алгоритм DBSCAN.

2 Конструкторская часть

2.1 Функциональные требования

Выполнить кластеризацию вершин графа по числу соседей, находящихся на расстоянии 1 ребра, 2 рёбер, ..., M рёбер (M — входной параметр). Алгоритм DBSCAN.

Входные данные:

— файл, содержащий граф в формате graphviz;

— расстояние M;

— минимальное число точек на расстоянии M, чтобы точка считалась ядром.

Выходные данные:

— кластеризованный граф;

— количество кластеров;

— количество вершин в каждом кластере.

2.2 Разработка алгоритмов

В данном разделе представлены схемы алгоритмов работы: последовательного и параллельного DBSCAN, рабочего потока, конвейера обработки данных.

2.2.1 Последовательный алгоритм DBSCAN

На рисунках 2.1 – 2.3 представлена схема работы последовательного алгоритма DBSCAN.

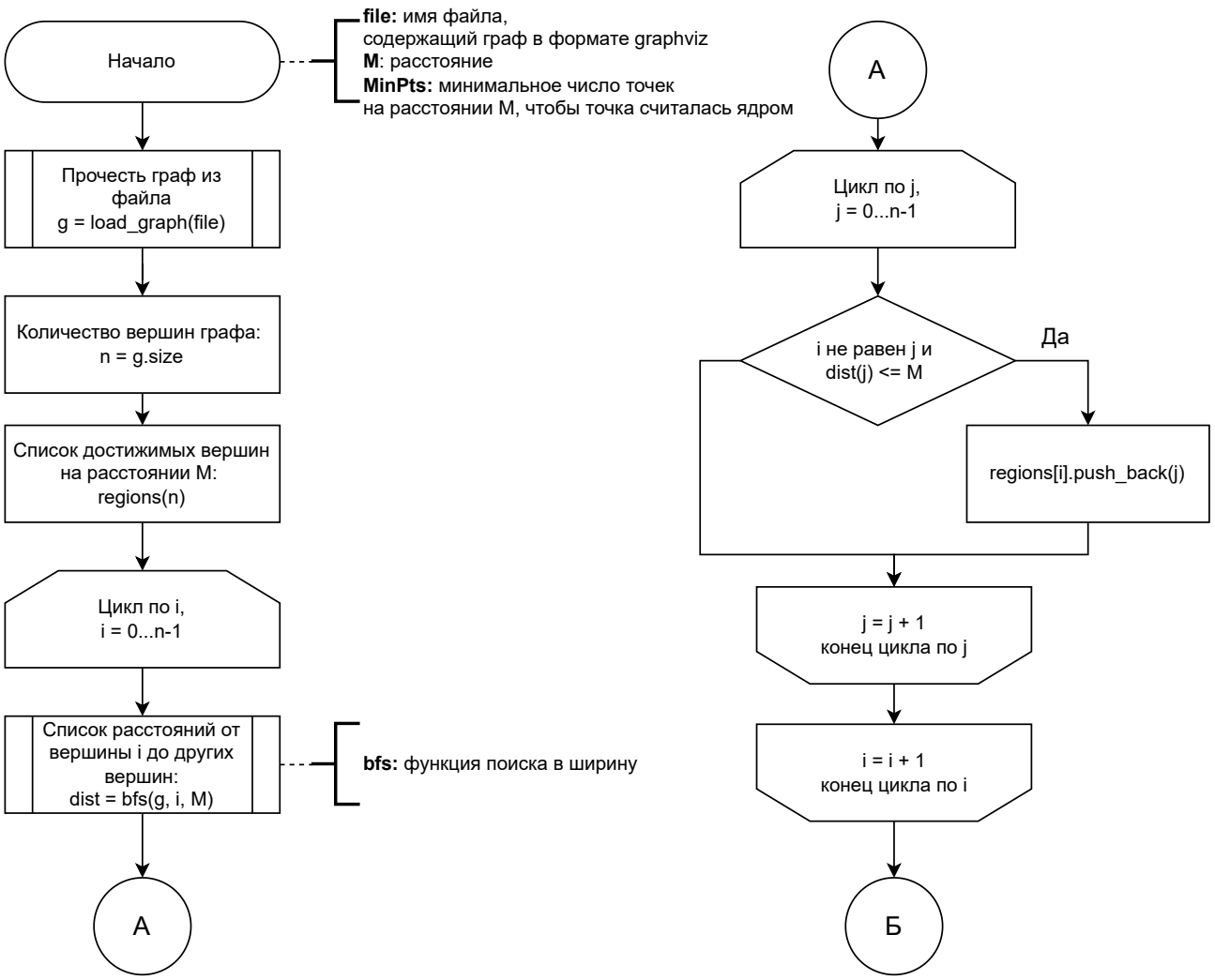

Рисунок 2.1 — Схема работы последовательного алгоритма DBSCAN (часть 1)

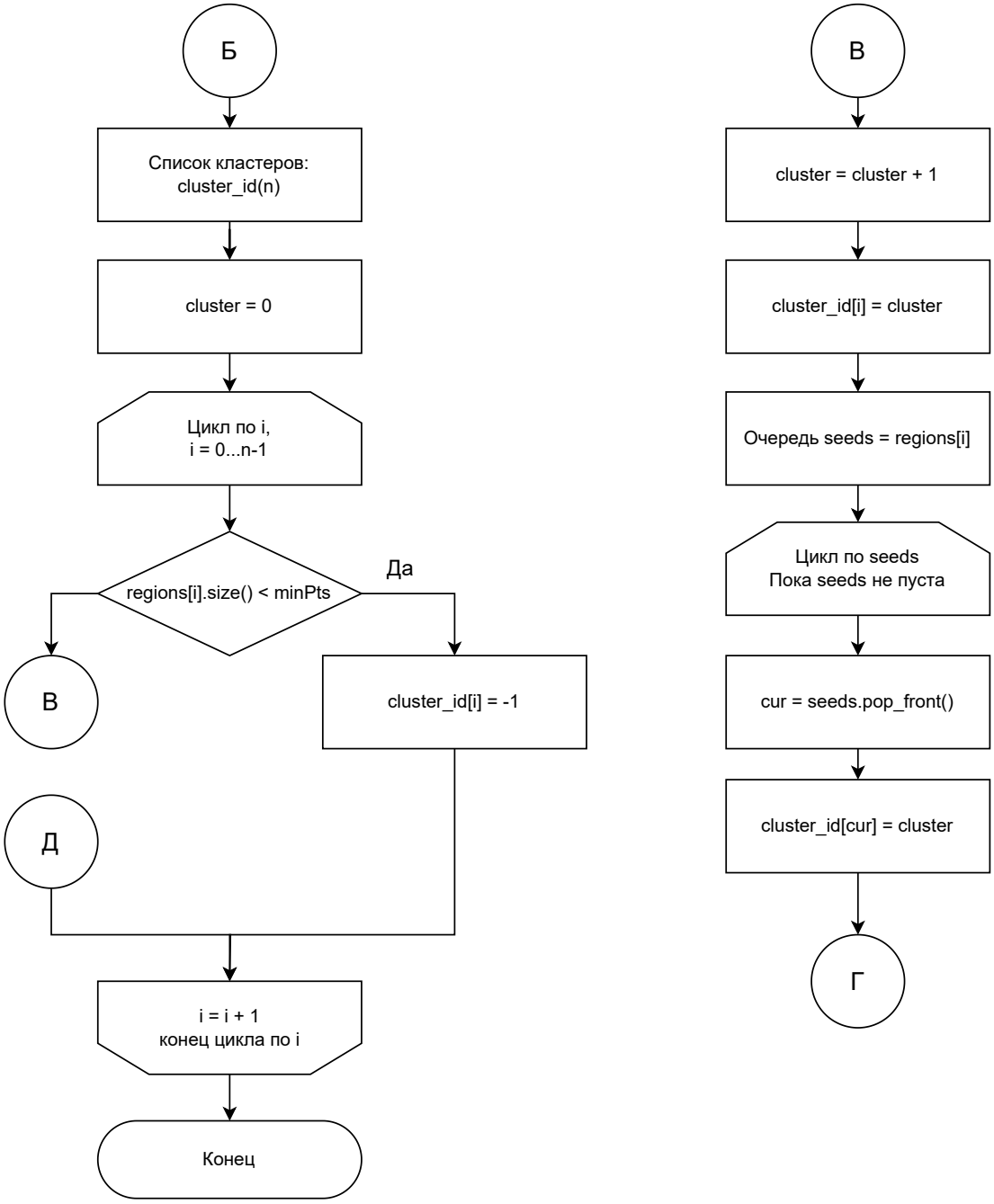

Рисунок 2.2 — Схема работы последовательного алгоритма DBSCAN (часть 2)

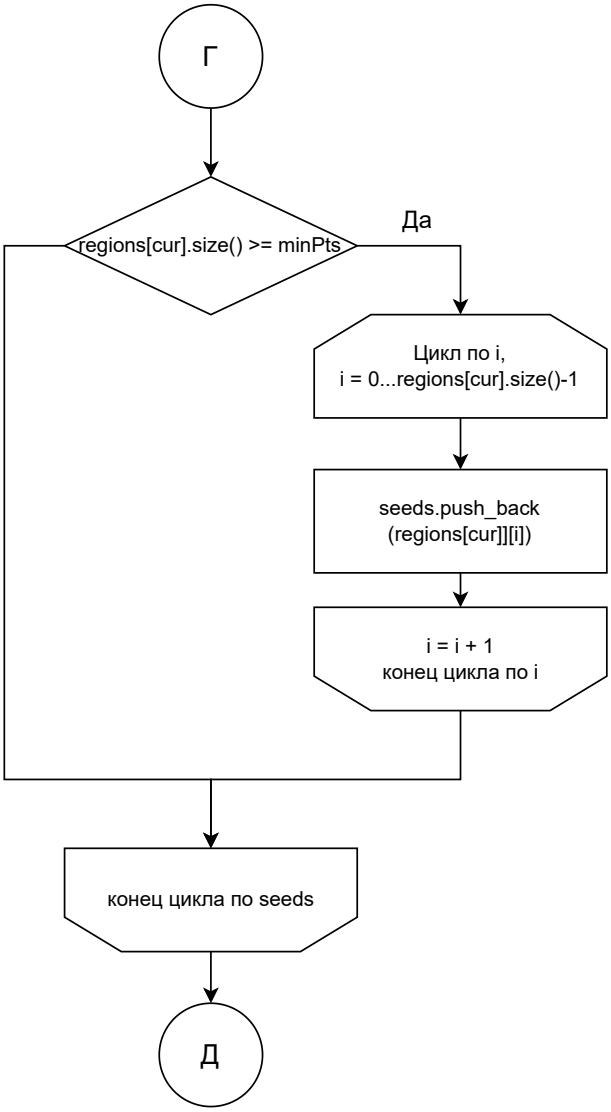

Рисунок 2.3 — Схема работы последовательного алгоритма DBSCAN (часть 3)

2.2.2 Параллельный алгоритм DBSCAN

Модифицированный алгоритм делает параллельно операцию поиска расстояний до достижимых вершин (находящихся не дальше расстояния M).

На рисунках 2.4 – 2.6 представлена схема работы параллельного алгоритма DBSCAN.

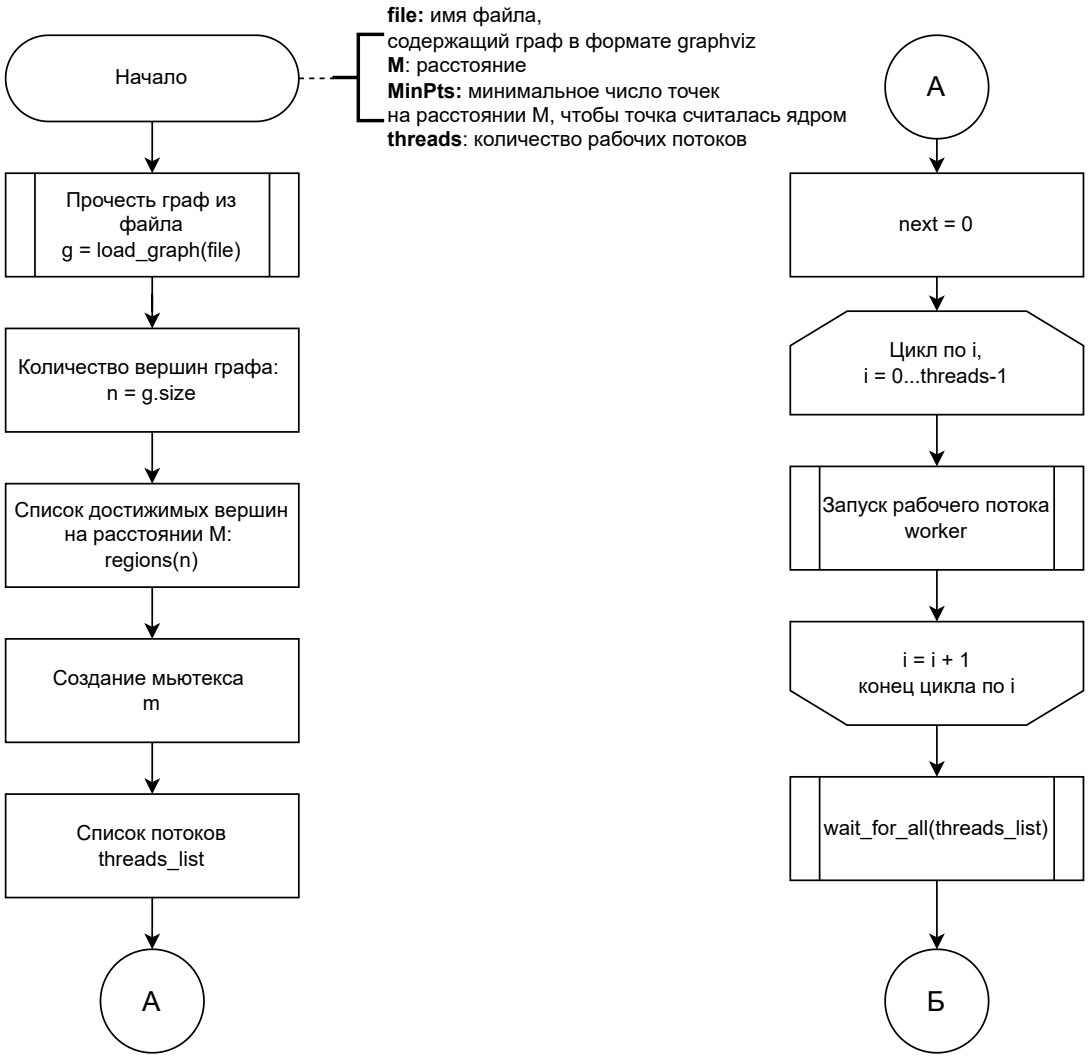

Рисунок 2.4 — Схема работы параллельного алгоритма DBSCAN (часть 1)

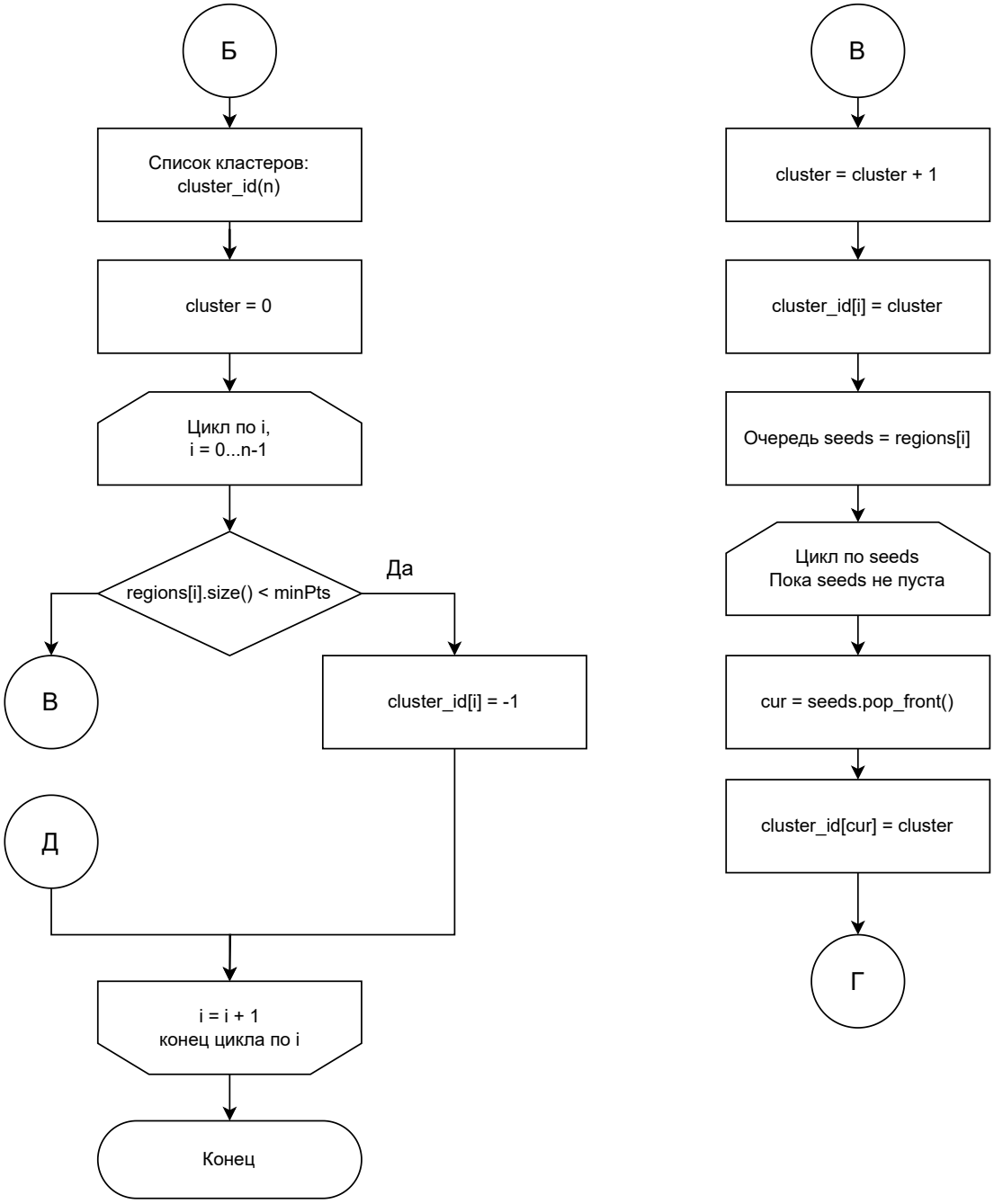

Рисунок 2.5 — Схема работы параллельного алгоритма DBSCAN (часть 2)

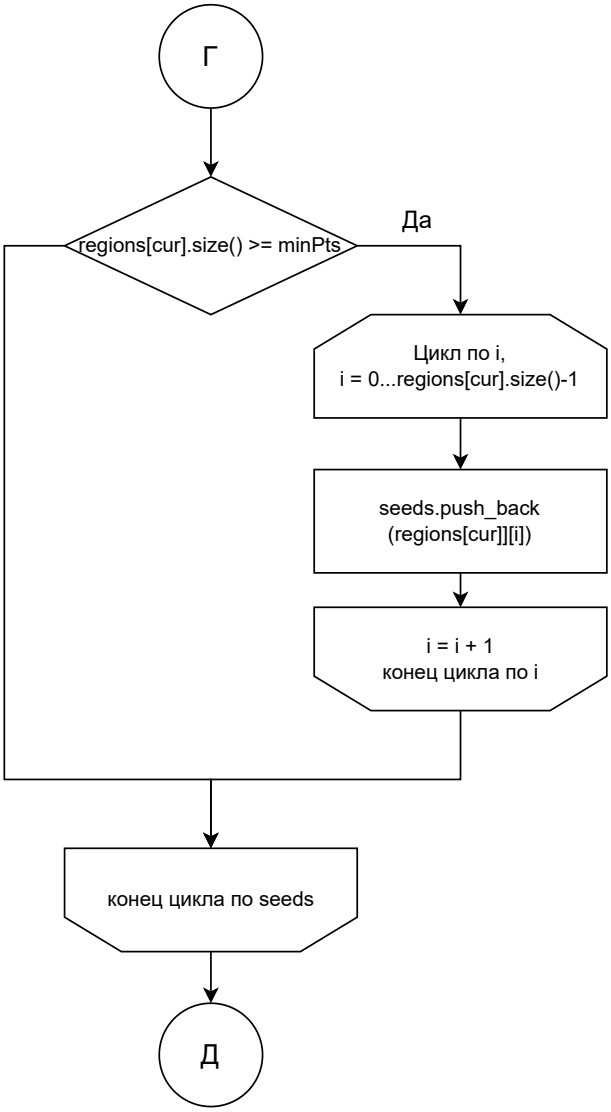

Рисунок 2.6 — Схема работы параллельного алгоритма DBSCAN (часть 3)

2.2.3 Рабочий поток

На рисунке 2.7 представлена схема алгоритма работы рабочего потока.

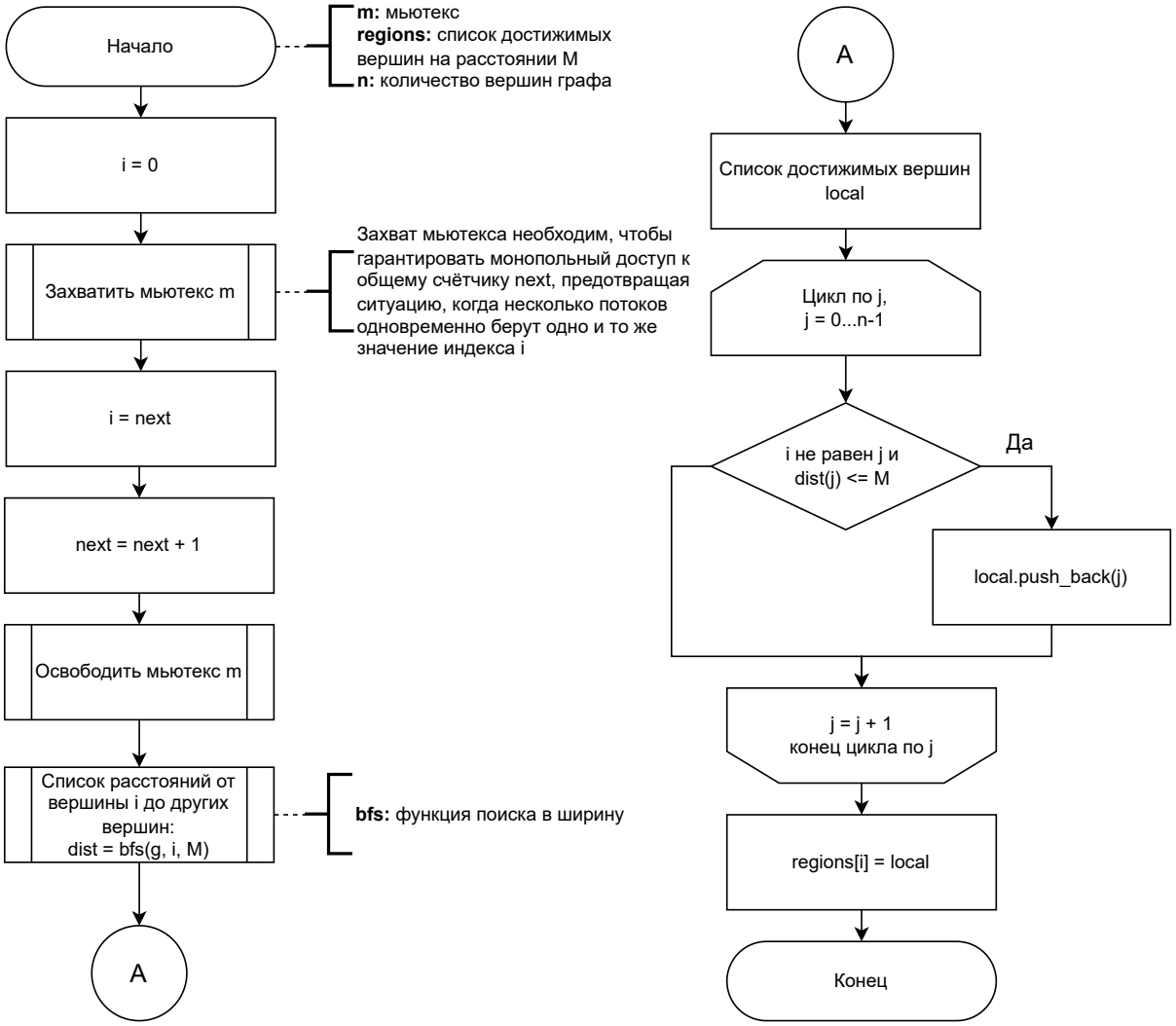

Рисунок 2.7 — Схема алгоритма работы рабочего потока

2.2.4 Конвейер

На рисунке 2.8 представлена схема работы конвейера.

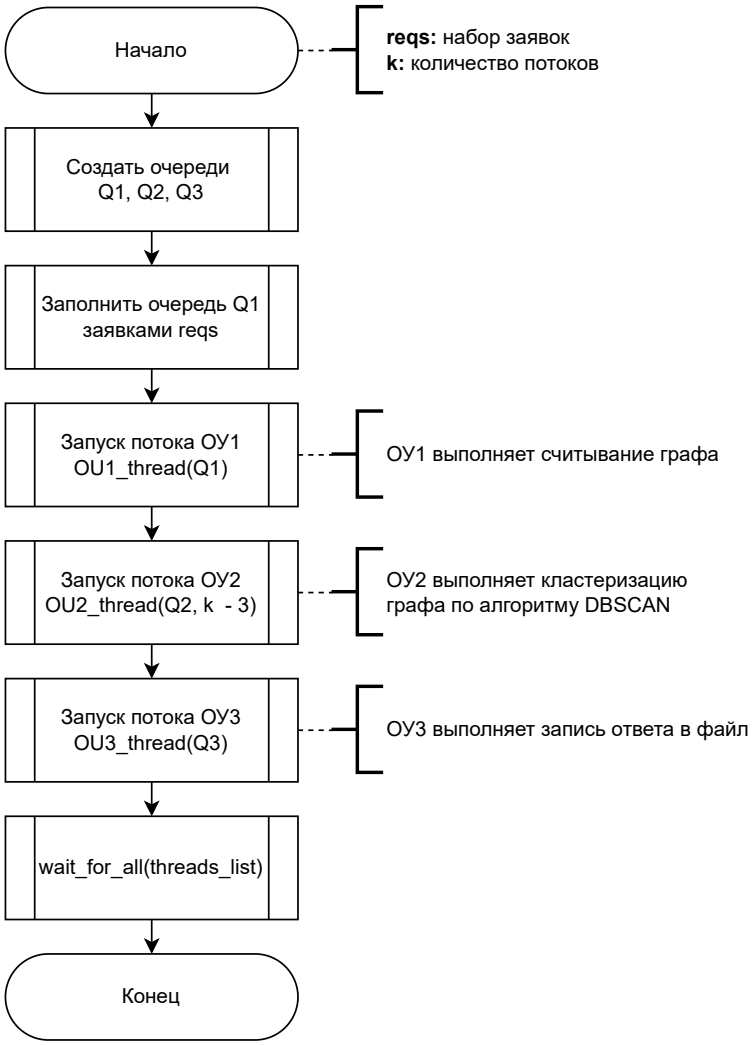

Рисунок 2.8 — Схема работы конвейера

2.3 Вывод

В данном разделе были описаны функциональные требования к программе, построены схемы работы алгоритмов: последовательного DBSCAN, параллельного алгоритма DBSCAN, рабочего потока, конвейера обработки данных.

3 Технологическая часть

3.1 Средства реализации

Для реализации алгоритмов был выбран язык C++. Выбор обусловлен тем, что C++ статически типизированный язык программирования, в нём нет сборщика мусора, имеется стандартная библиотека для замера времени.

Для замера времени использовалась функция clock() из модуля ctime [6].

Для создания нативных потоков использовался класс threads [7].

Был использован шаблон класса lock_guard для работы с мьютексом и функция захвата/освобождения мьютекса guard() [7].

3.2 Реализация алгоритмов

3.2.1 DBSCAN

В листингах 3.1 – 3.3 показаны реализации алгоритмов DBSCAN: рекурсивного и нерекурсивного.

```
void dbscan(const string &filename, int M, int minPts, bool directed,
           bool verbose) {
  graph_t g = load_graph(filename, directed, verbose);
  int n = g.adj.size();

  vector<vector<int>> regions(n);
  for (int i = 0; i < n; i++) {
    auto dist = bfs_distances(g, i, M);
    for (int j = 0; j < n; j++)
      if (i != j && dist[j] != -1 && dist[j] <= M)
        regions[i].push_back(j);
  }

  vector<int> cluster_id(n, 0);
  clustering(g, minPts, regions, cluster_id, verbose, filename, directed);
}
```

Листинг 3.1 — Реализация последовательного алгоритма DBSCAN

```
void dbscan_parallel(const string &filename, int M, int minPts, int threads,
                     bool directed, bool verbose) {
  graph_t g = load_graph(filename, directed, verbose);
  int n = g.adj.size();

  vector<vector<int>> regions(n);

  mutex m;
  int next = 0;
  auto worker = [&]() {
    while (true) {
      int i;
      {
        lock_guard<mutex> guard(m);
        if (next >= n)
          return;
        i = next++;
      }
      auto dist = bfs_distances(g, i, M);
      vector<int> local;
      for (int j = 0; j < n; j++)
        if (i != j && dist[j] != -1 && dist[j] <= M)
          local.push_back(j);
      regions[i].swap(local);
    }
  };

  vector<thread> thrds;
  for (int i = 0; i < threads; i++)
    thrds.emplace_back(worker);
  for (auto &th : thrds)
    th.join();

  vector<int> cluster_id(n, 0);
  clustering(g, minPts, regions, cluster_id, verbose, filename, directed);
}
```

Листинг 3.2 — Реализация параллельного алгоритма DBSCAN

```
static void clustering(const graph_t &g, int minPts,
                       const vector<vector<int>> &regions,
                       vector<int> &cluster_id, bool verbose,
                       const string &filename, bool directed) {
  int n = g.adj.size();
  int cluster = 0;

  for (int i = 0; i < n; i++) {
    if (cluster_id[i] != 0)
      continue;
    if (regions[i].size() < minPts) {
      cluster_id[i] = -1;
      continue;
    }

    cluster++;
    cluster_id[i] = cluster;
    deque<int> seeds(regions[i].begin(), regions[i].end());

    while (!seeds.empty()) {
      int cur = seeds.front();
      seeds.pop_front();
      if (cluster_id[cur] == -1)
        cluster_id[cur] = cluster;
      if (cluster_id[cur] != 0)
        continue;
      cluster_id[cur] = cluster;
      if (regions[cur].size() >= minPts)
        for (int nb : regions[cur])
          seeds.push_back(nb);
    }
  }
```

Листинг 3.3 — Функция кластеризации

3.2.2 Конвейер

В листинге 3.4 показана реализация конвейера и обслуживающих устройств.

```
static long long run_pipeline(std::vector<Request*>& reqs, int k) {
  int threads = k - 3;
  if (threads < 0) threads = 0;

  BlockingQueue<Request*> Q1, Q2, Q3;
  for (Request* r : reqs) Q1.push(r);
  Q1.close();

  auto ou1 = [&]() {
    Request* r;
    while (Q1.pop(r)) {
      r->t_start[0] = get_nanotime();
      r->g = load_graph(r->filename, r->directed, false);
      r->t_end[0] = get_nanotime();
      Q2.push(r);
    }
    Q2.close();
  };

  auto ou2 = [&]() {
    Request* r;
    while (Q2.pop(r)) {
      r->t_start[1] = get_nanotime();
      r->json_result = dbscan_to_json(r->g, r->M, r->minPts, threads);
      r->t_end[1] = get_nanotime();
      Q3.push(r);
    }
    Q3.close();
  };

  auto ou3 = [&]() {
    Request* r;
    while (Q3.pop(r)) {
      r->t_start[2] = get_nanotime();
      write_answer_file(*r);
      r->t_end[2] = get_nanotime();
    }
```

```
  };
  long long t0 = get_nanotime();
  std::thread t1(ou1);
  std::thread t2(ou2);
  std::thread t3(ou3);
  t1.join();
  t2.join();
  t3.join();
  long long t1_end = get_nanotime();
  return t1_end - t0;
}
```

Листинг 3.4 — Реализация конвейера

В листинге 3.5 показана реализация блокирующей очереди.

```
template <typename T>
struct BlockingQueue {
  std::deque<T> q;
  std::mutex m;
  std::condition_variable cv;
  bool closed = false;

  void push(const T& v) {
    {
      std::lock_guard<std::mutex> g(m);
      q.push_back(v);
    }
    cv.notify_one();
  }

  bool pop(T& out) {
    std::unique_lock<std::mutex> lk(m);

    while (q.empty() && !closed) {
      cv.wait(lk);
    }

    if (q.empty() && closed)
      return false;

    out = q.front();
    q.pop_front();
    return true;
  }

  void close() {
    {
      std::lock_guard<std::mutex> g(m);
      closed = true;
    }
    cv.notify_all();
  }
};
```

Листинг 3.5 — Реализация блокирующей очереди

3.3 Функциональные тесты

В таблице 3.1 представлены результаты функционального тестирования реализаций: последовательного алгоритма DBSCAN и параллельного. Каждая реализация каждого алгоритма прошла тесты успешно.

Таблица 3.1 — Результаты функционального тестирования реализаций алгоритма DBSCAN

Граф	**M**	**minPts**	**Ожидаемый результат**	**Фактический результат**
1->2; 2->3; 3->4; 5->6; 6->7; 8;	2	2	clusters: [[1,2,3,4], [5,6,7]] noise: [8]	clusters: [[1,2,3,4], [5,6,7]] noise: [8]
Пусто	1	3	clusters: [] noise: []	clusters: [] noise: []
1; 2; 3;	2	2	clusters: [] noise: [1,2,3]	clusters: [] noise: [1,2,3]
1->2; 2->3; 3->4; 4->5;	3	1	clusters: [[1,2,3,4,5]] noise: []	clusters: [[1,2,3,4,5]] noise: []

3.4 Пример сформированного конвейером лога

В листинге 3.6 представлен пример сформированного конвейером лога при трёх входящих заявках.

```
0.000us START req=1 ou=1
85.700us END req=1 ou=1
88.900us START req=2 ou=1
94.100us START req=1 ou=2
227.900us END req=2 ou=1
228.800us START req=3 ou=1
253.200us END req=1 ou=2
257.100us START req=2 ou=2
261.900us START req=1 ou=3
314.200us END req=3 ou=1
419.100us END req=2 ou=2
419.700us START req=3 ou=2
492.400us END req=1 ou=3
493.100us START req=2 ou=3
553.100us END req=3 ou=2
624.800us END req=2 ou=3
625.600us START req=3 ou=3
729.000us END req=3 ou=3
```

Листинг 3.6 — Пример сформированного конвейером лога

3.5 Вывод

В этом разделе были описаны средства реализации алгоритмов. Также были продемонстрированы листинги реализаций конвейера и последовательного и параллельного алгоритмов DBSCAN. Приведены результаты функционального тестирования и пример сформированного конвейером лога.

4 Исследовательская часть

4.1 Технические характеристики ЭВМ

Замеры времени проводились на ноутбуке ACER Predator со следующими техническими характеристиками:

— процессор Intel(R) Core(TM) i7-10750H с тактовой частотой 2.60ГГц;

— ОЗУ 16 ГБ;

— ОС Windows 10 Pro 64 разрядная;

— 12 логических ядер.

Во время замеров времени ноутбук был подключён к электропитанию, сторонними приложениями нагружен не был.

4.2 Замеры времени

Были проведены замеры времени обработки набора заявок для двух реализаций: последовательной и конвейерной параллельной (три обслуживающих устройства ОУ1 — ОУ3, работающих в отдельных потоках и обменивающихся заявками через очереди, при этом ОУ2 использует вспомогательные рабочие потоки).

Измерения проводились для $N \in \{25, 50, 75, 100, 125\}$ заявок при $k = 7$ потоках.

В таблице 4.1 представлены результаты замеров времени обработки заявок последовательно и с помощью конвейера.

Таблица 4.1 — Результаты замеров времени обработки заявок последовательно и с помощью конвейера

Количество заявок	**Последовательно, мс**	**Конвейер, мс**
25	1387.000	554.004
50	2820.000	1110.000
75	4199.000	2155.000
100	5543.000	2276.000
125	7623.000	2815.000

На рисунке 4.1 изображены графики зависимостей времени последовательной и конвейерной обработки набора заявок от их количества.

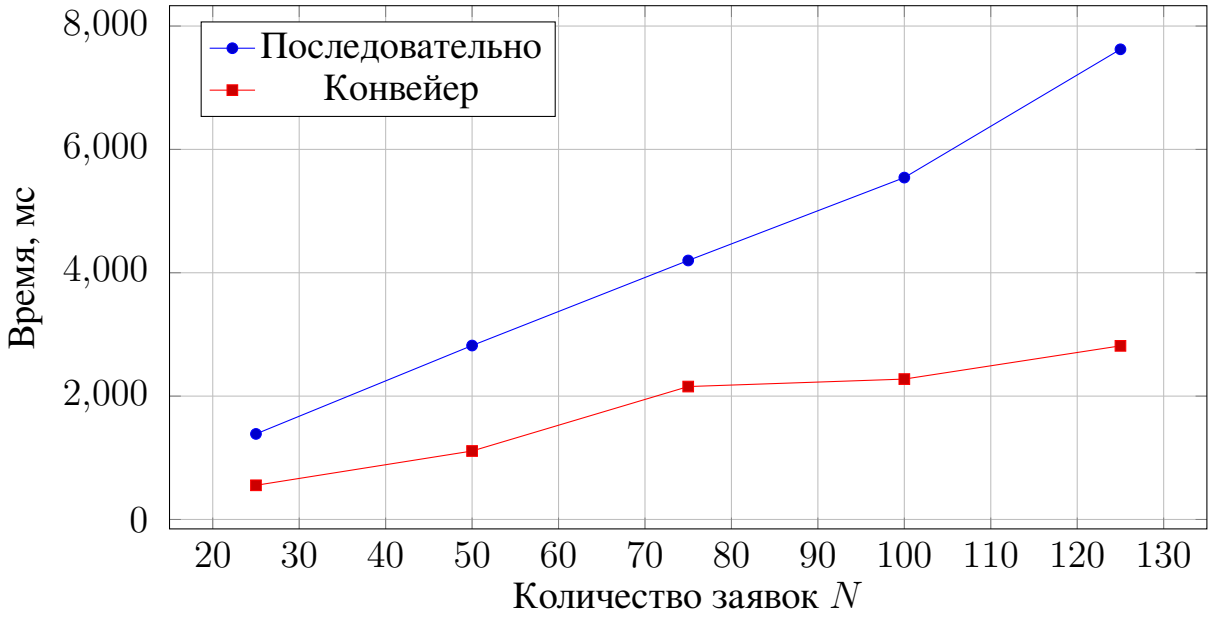

Рисунок 4.1 — Зависимости времени последовательной и конвейерной обработки набора заявок от их количества

В среднем конвейер обрабатывал заявки в 2,4 раза быстрее, чем при последовательной обработке.

4.3 Вывод

В данном разделе были описаны технические характеристики машины, на которой проводились замеры времени. Продемонстрированы результаты замеров времени обработки набора заявок для двух реализаций: последовательной и конвейерной.

ЗАКЛЮЧЕНИЕ

В результате лабораторной работы была разработана и реализована конвейерная обработка набора заявок на кластеризацию вершин графа по алгоритму DBSCAN.

Выполнены следующие задачи:

1) описан последовательный алгоритм DBSCAN;
2) разработана параллельная версия алгоритма;
3) реализованы обе версии алгоритма;
4) реализована последовательная обработка N заявок;
5) разработан параллельный конвейер обработки заявок, включающий три обслуживающих устройства, взаимодействующих через очереди;
6) реализован конвейер;
7) проведены замеры времени обработки $N \in \{25, 50, 75, 100, 125\}$ заявок для последовательной и конвейерной реализаций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Уильямс Э. Параллельное программирование на C++ в действии. — Москва: Изд-во ДМК, 2012. — С. 24.

2. Цилюрик О., Горошко Е. QNX/UNIX Анатомия параллелизма. — Москва: Изд-во Символ, 2006. — С. 47, 152, 162.

3. Камран А. Экстремальный Си. — Санкт-Петербург: Изд-во ПИТЕР, 2021. — С. 440–443.

4. Большакова Е.И., Клышинский Э.С., Ландэ Д.В. Автоматическая обработка текстов на естественном языке и компьютерная лингвистика. — Москва: МИЭМ, 2011. — С. 197–199.

5. Харенслак Б., Руйтер Д. Apache Airflow и конвейеры обработки данных. — Москва: ДМК Пресс, 2021. — С. 27 – 33.

6. ISO/IEC 9899:1999. 2007. — С. 7.23.2.1.

7. ISO/IEC JTC1 SC22 WG21 N4860. 2018. — С. 30.3.2, 30.4.4.1.