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 Куликов Н.В.

Группа ИУ7-53Б

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

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель: разработка и сравнительный анализ последовательного и параллельного алгоритмов кластеризации вершин графа по алгоритму DBSCAN.

Для достижения поставленной цели необходимо было выполнить следующие задачи:

1) описать последовательный алгоритм решения задачи;
2) разработать параллельную версию алгоритма;
3) реализовать обе версии алгоритма;
4) выполнить сравнительный анализ зависимостей времени решения задач от размерности входа для реализации последовательного алгоритма и для реализации модифицированного алгоритма, запущенной с единственным вспомогательным (рабочим) потоком;
5) выполнить сравнительный анализ зависимостей времени решения задач от размерности входа для реализации модифицированного алгоритма при К вспомогательных (рабочих) потоках, К принимает значения 1, 2, 4,..., 8·q, где (q – количество логических ядер процессора);
6) сформулировать рекомендацию о выборе К для решения задачи.

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 Вывод

В аналитической части были рассмотрены понятия параллелизма и 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.

2.2.1 Последовательный алгоритм DBSCAN

На рисунках 2.1 – 2.3 представлена схема последовательного алгоритма DBSCAN.

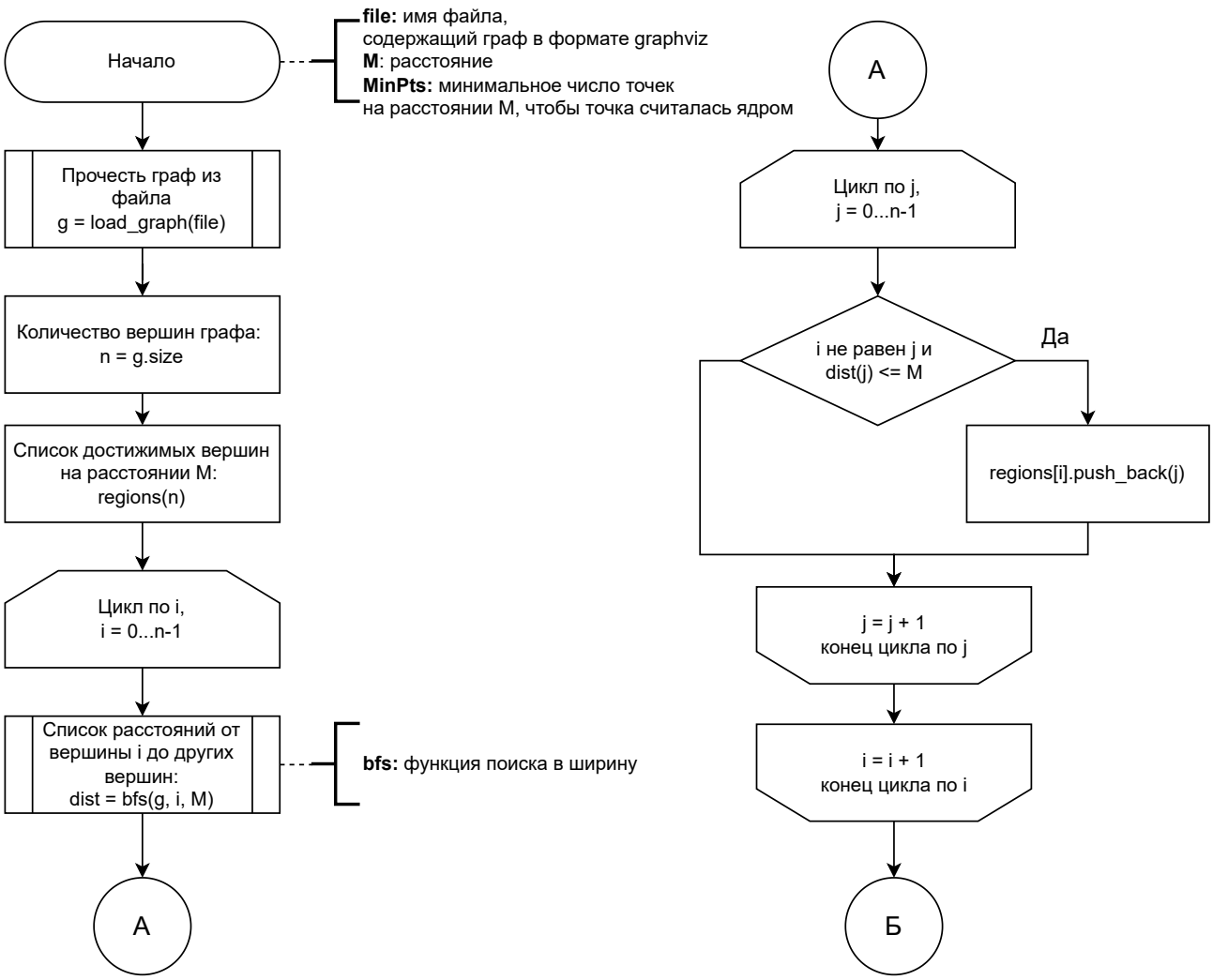

Рисунок 2.1 — Схема последовательного алгоритма DBSCAN (часть 1)

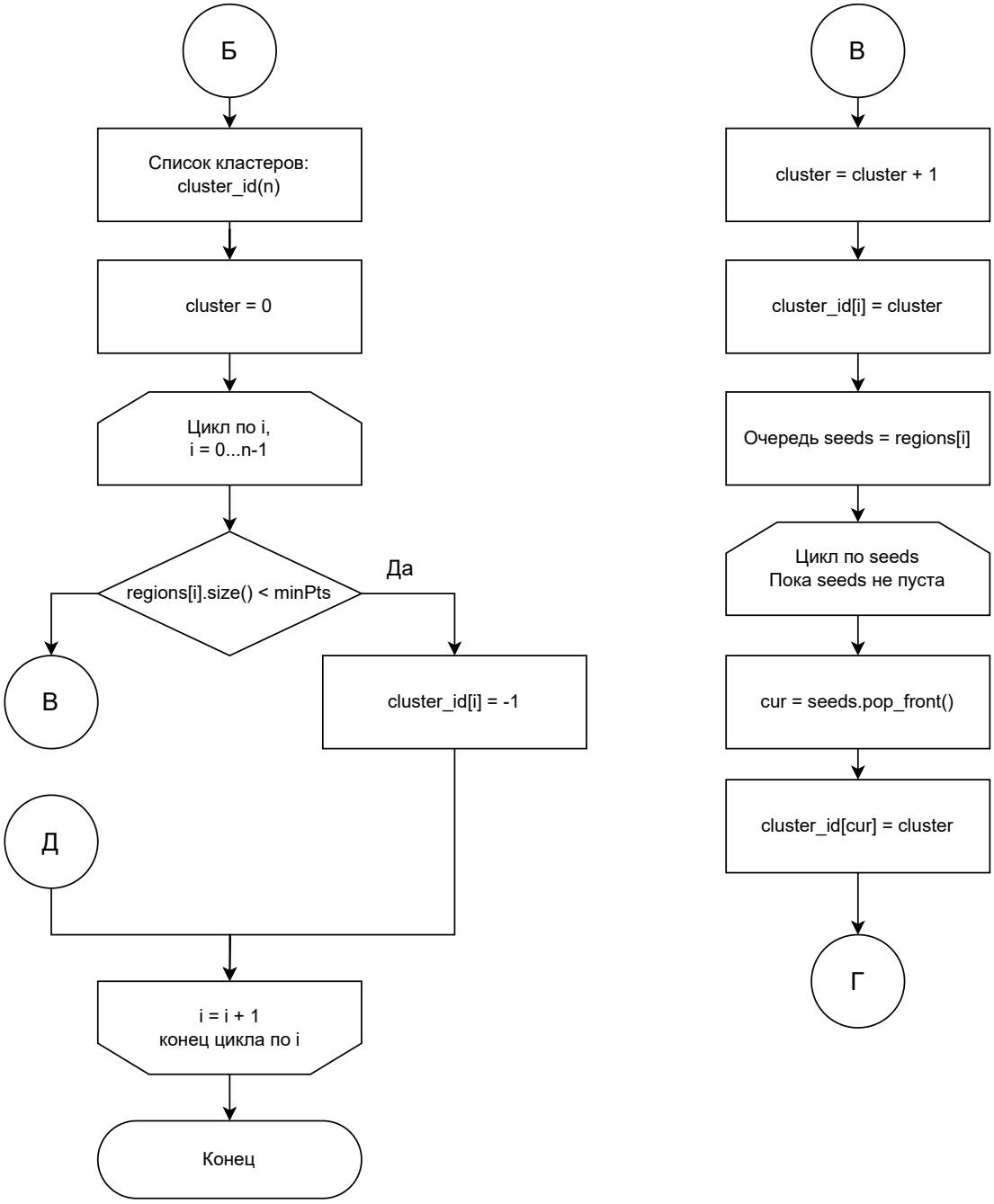

Рисунок 2.2 — Схема последовательного алгоритма DBSCAN (часть 2)

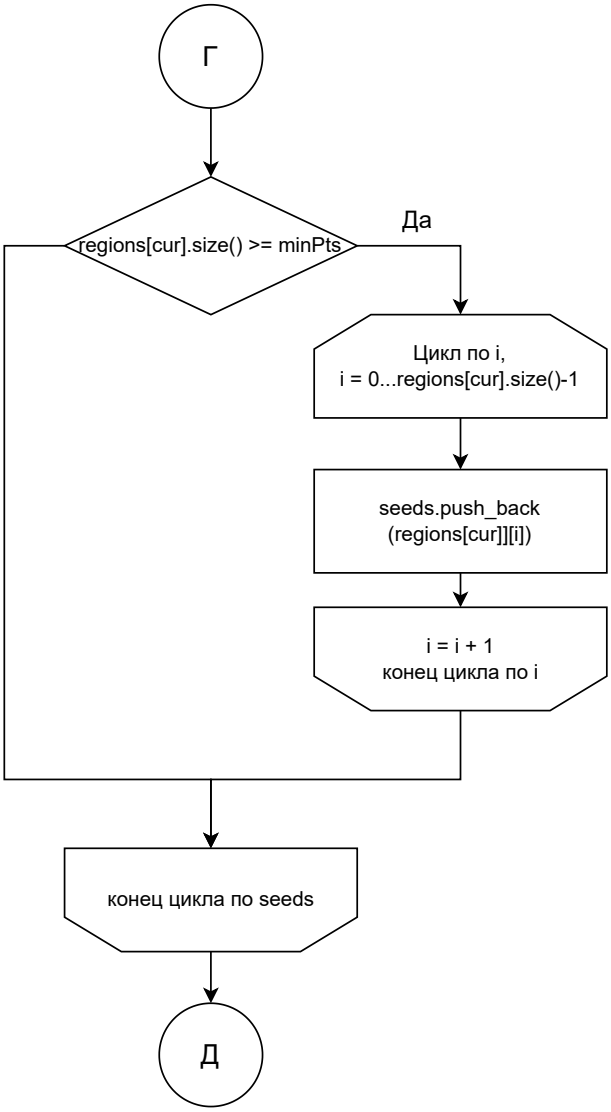

Рисунок 2.3 — Схема последовательного алгоритма DBSCAN (часть 3)

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 параллельного алгоритма DBSCAN.

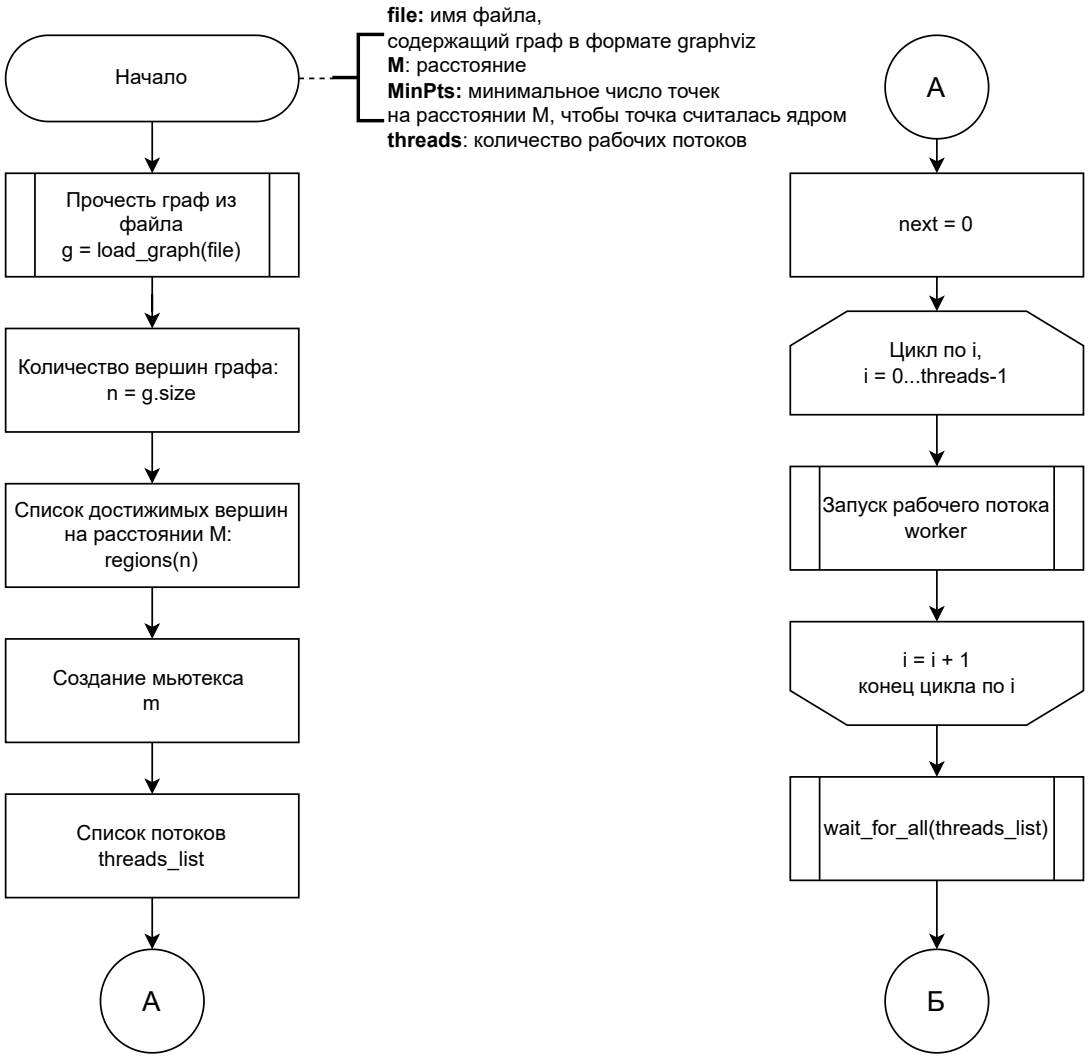

Рисунок 2.4 — Схема параллельного алгоритма DBSCAN (часть 1)

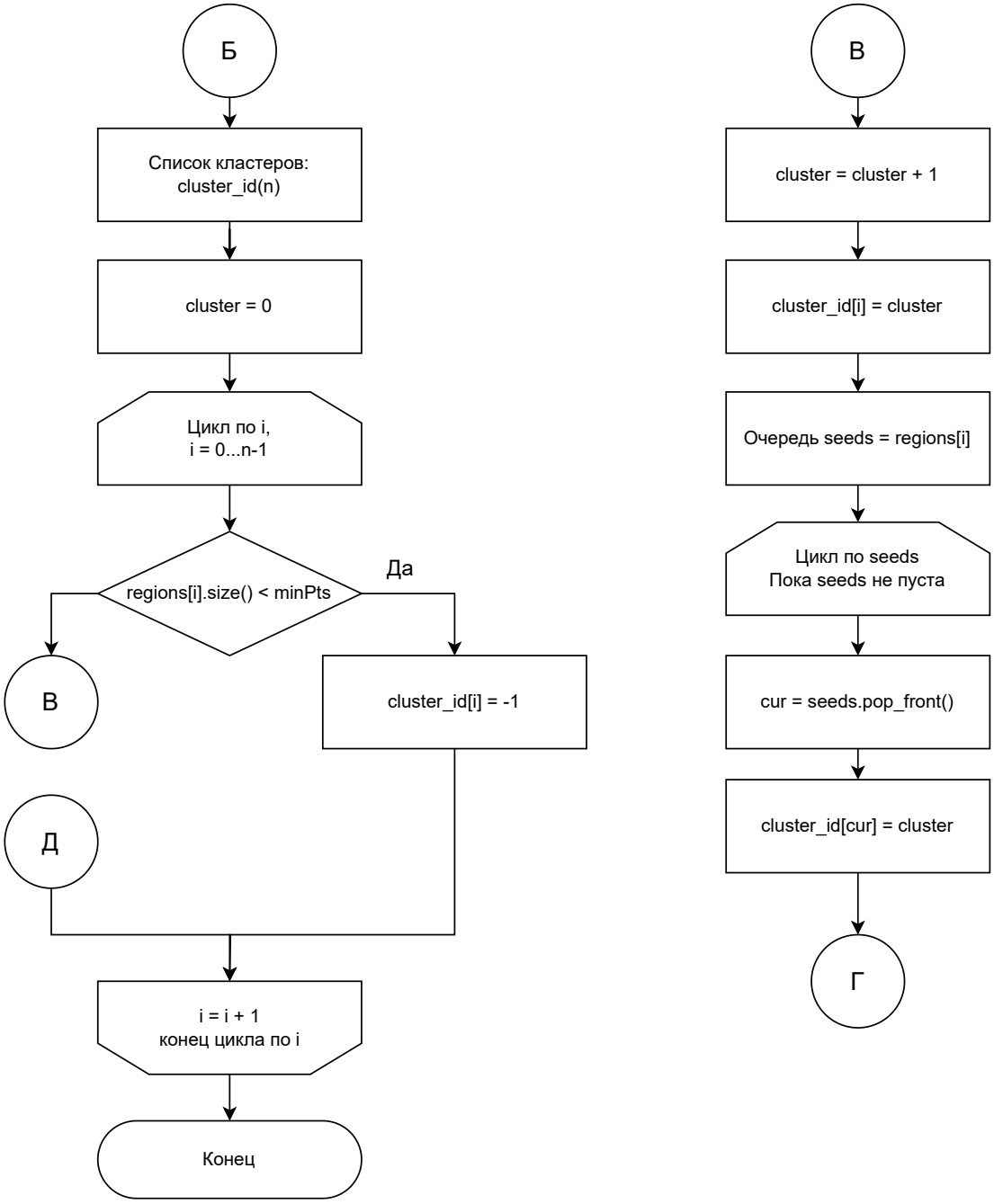

Рисунок 2.5 — Схема параллельного алгоритма DBSCAN (часть 2)

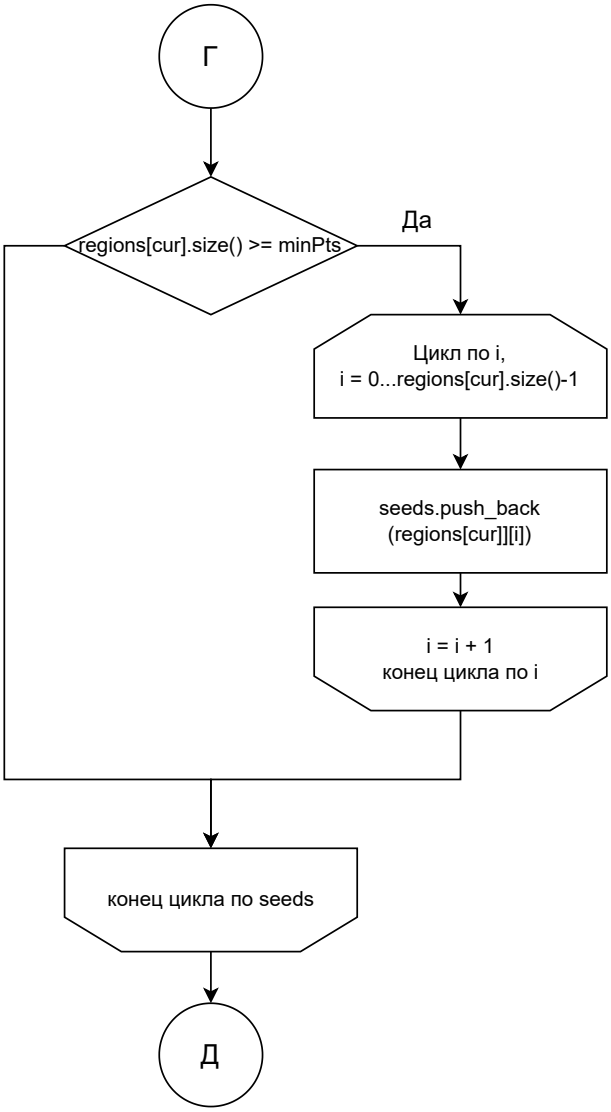

Рисунок 2.6 — Схема параллельного алгоритма DBSCAN (часть 3)

2.2.3 Рабочий поток

На рисунке 2.7 представлена схема алгоритма работы рабочего потока.

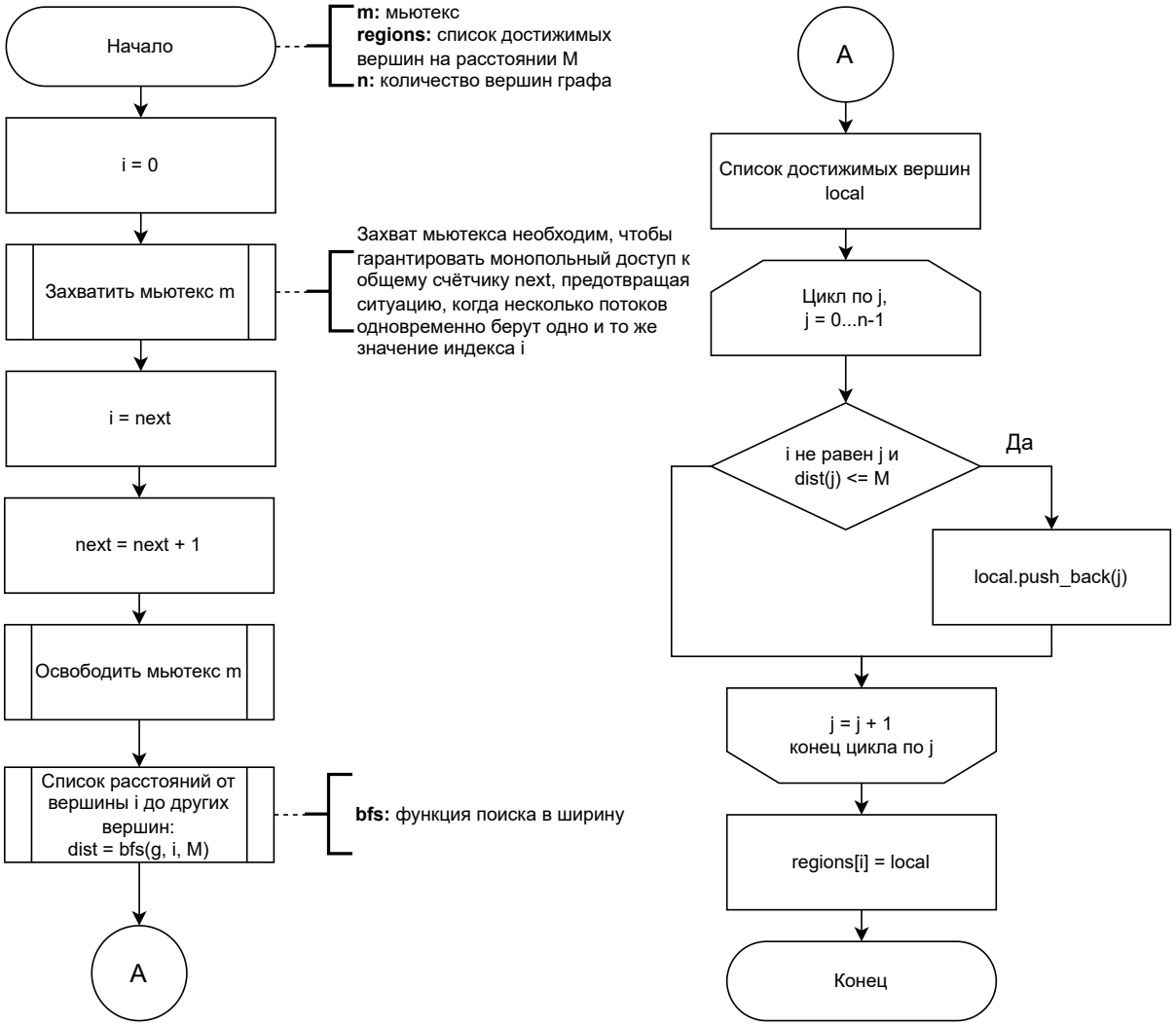

Рисунок 2.7 — Схема алгоритма работы рабочего потока

2.3 Вывод

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

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 процессорного времени.

Для замера процессорного времени использовалась функция clock() из модуля ctime [5].

Для создания нативных потоков использовался класс threads [6].

Был использован шаблон класса lock_guard для работы с мьютексом и функция захвата/освобождения мьютекса guard() [6].

3.2 Реализация алгоритмов

В листингах 3.1 – 3.3 показаны реализации алгоритмов: 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, file name, directed);
}
```

Листинг 3.1 — Реализация последовательного алгоритма DBSCAN

```
void dbscan_parallel(const string &filename, int M, int mi nPts, 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, file name, 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

3.3 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 Вывод

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

4 Исследовательская часть

4.1 Технические характеристики ЭВМ

Замеры процессорного 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 процессорного времени ноутбук был подключён к электропитанию, сторонними приложениями нагружен не был.

4.2 Замеры процессорного времени

4.2.1 Последовательная и параллельная реализация

Были проведены замеры процессорного времени выполнения кластеризации для реализации последовательного алгоритма DBSCAN и реализации параллельного алгоритма DBSCAN, запущенной с единственным рабочим потоком. Результаты замеров времени представлены в таблице 4.1.

Таблица 4.1 — Результаты замеров времени выполнения реализаций последовательного и параллельного алгоритмов DBSCAN

Размер графа	Последовательный DBSCAN, мс	Параллельный DBSCAN, мс
500	8.731	8.975
1000	18.728	18.995
1500	33.253	34.168
2000	49.231	49.968
2500	67.042	69.009
3000	95.621	99.627
3500	131.152	136.551
4000	175.648	179.641

На рисунке 4.1 изображены графики зависимостей времени работы реализаций алгоритмов от размеров графа.

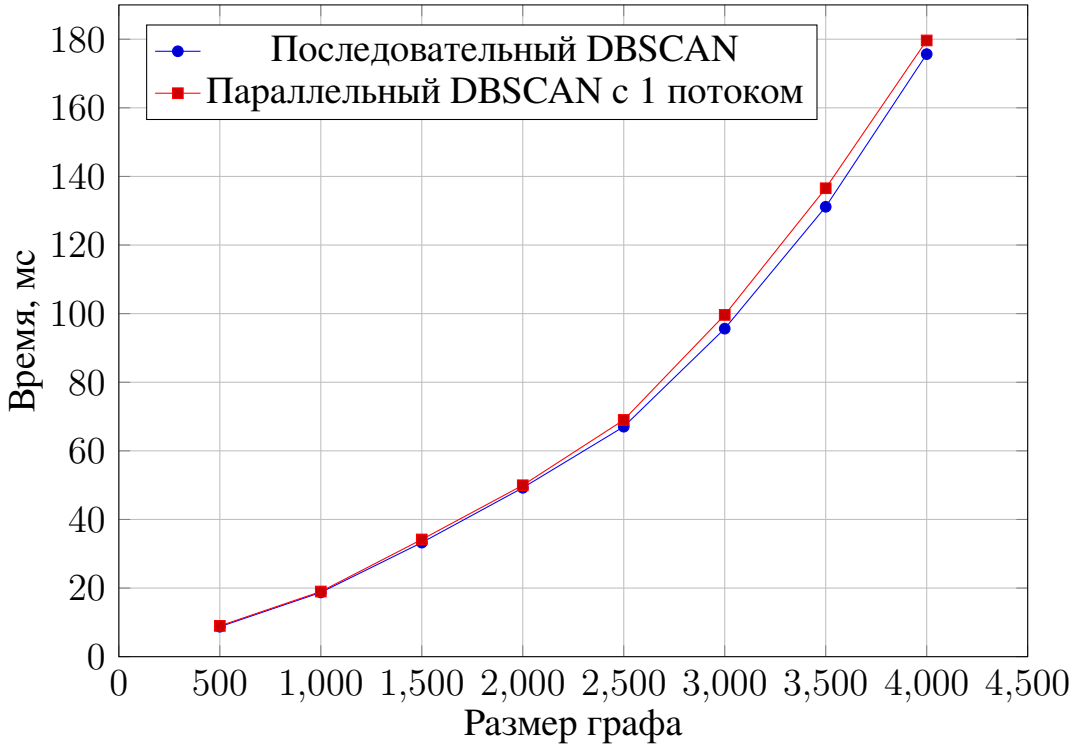

Рисунок 4.1 — Зависимости времени выполнения от размера графа для реализаций последовательного и параллельного алгоритмов DBSCAN

В среднем реализация параллельного алгоритма DBSCAN, запущенная с единственным рабочим потоком, выполняет задачу на 4 % медленнее, чем реализация последовательного алгоритма. Это связано с тем, что часть времени выполнения параллельной реализации уходит на диспетчеризацию потоков, включающую в себя создание и запуск рабочего потока.

4.2.2 Параллельная реализация с множеством потоков

Были проведены замеры процессорного времени выполнения кластеризации для реализации параллельного алгоритма DBSCAN при K рабочих потоках, K принимает значения 1, 2, 4,..., 8·q, где (q – количество логических ядер процессора). Результаты замеров времени представлены в таблице 4.2.

Таблица 4.2 — Время выполнения параллельного алгоритма DBSCAN при разном количестве потоков, мс

Размер графа	1 поток	2 потока	4 потока	8 потоков	16 потоков	32 потока	64 потока	96 потоков
500	8.853	7.565	7.526	7.857	8.606	9.465	11.962	13.967
1000	19.611	15.184	14.498	15.435	16.267	18.331	19.576	21.015
1500	33.394	24.134	22.412	23.101	23.213	25.324	27.263	29.197
2000	47.317	35.248	29.509	29.969	30.383	32.424	34.748	37.949
2500	64.558	47.337	39.621	38.381	38.301	40.166	41.531	43.827
3000	84.906	57.963	46.076	45.704	45.545	47.784	49.498	57.148
3500	108.202	72.448	56.259	53.636	53.571	54.989	57.863	65.497
4000	128.613	89.643	66.763	63.163	62.758	62.351	66.858	75.308

На рисунке 4.2 изображены графики зависимостей времени работы реализации параллельного алгоритма DBSCAN от размеров графа с разным количеством рабочих потоков.

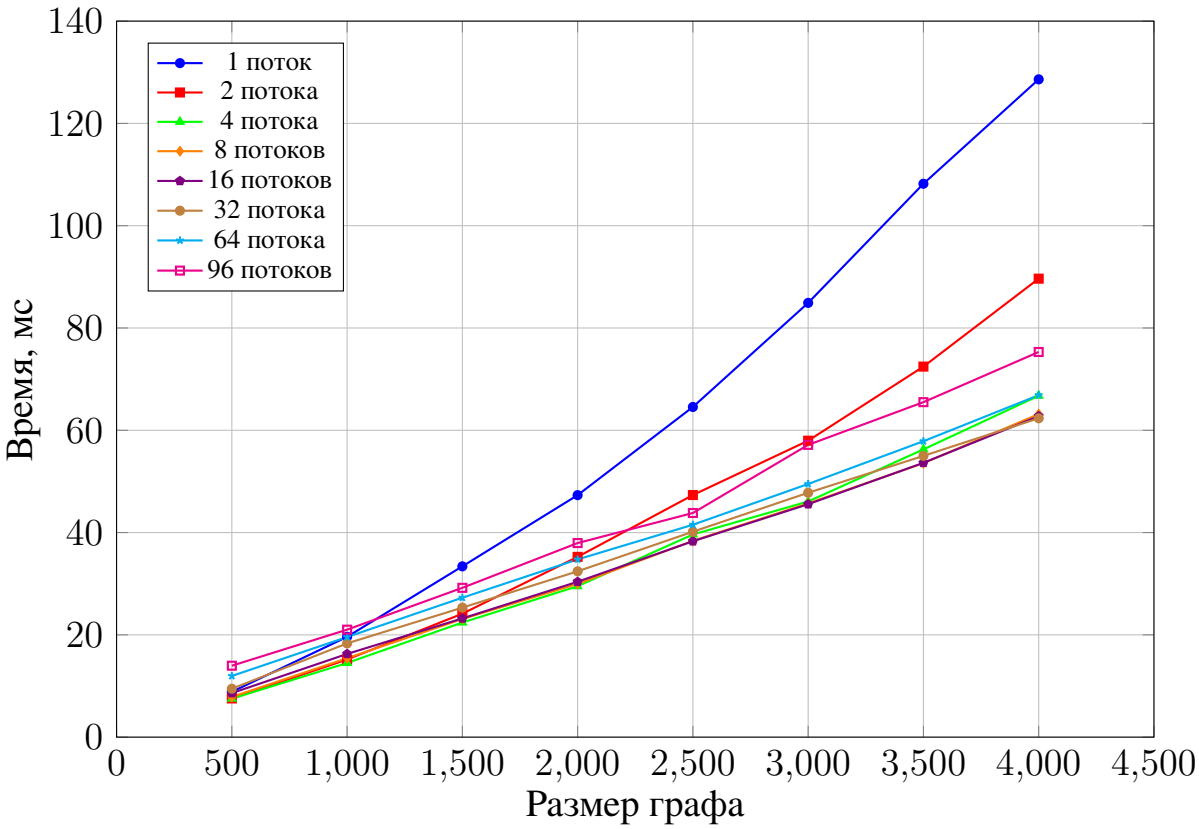

Рисунок 4.2 — Зависимости времени выполнения от размера графа для разного количества потоков

В результате анализа результатов замеров времени выполнения реализации параллельного алгоритма DBSCAN был сделан вывод, что не всегда программа выполняет задачу быстрее, чем больше рабочих потоков, например, при 96 потоках реализация алгоритма выполняет задачу в 1,8 раз медленнее, чем при 4 потоках и размере графа в 500 вершин. Это связано с тем, что кроме полезных вычислений много времени уходит на диспетчеризацию потоков.

Рекомендация по выбору количества рабочих потоков для решения задачи кластеризации графа:

— 4 потока при размерах графа от 500 до 2000 вершин;
— 16 потоков при размерах графа от 2500 до 3500 вершин;
— 32 потока при размерах графа от 4000 вершин.

4.3 Вывод

В данном разделе были описаны технические характеристики машины, на которой проводились замеры времени. Продемонстрированы результаты замеров процессорного времени, был проведён сравнительный анализ времени работы реализаций последовательного и параллельного алгоритмов DBSCAN. Была сформулирована рекомендация о выборе количества рабочих потоков для решения задачи.

ЗАКЛЮЧЕНИЕ

В результате лабораторной работы были разработаны последовательный и параллельный версии алгоритма кластеризации графа DBSCAN. Был проведён сравнительный анализ алгоритмов.

Выполнены следующие задачи:

1) описан последовательный алгоритм решения задачи;
2) разработана параллельная версия алгоритма;
3) реализованы обе версии алгоритма;
4) выполнен сравнительный анализ зависимостей времени решения задач от размерности входа для реализации последовательного алгоритма и для реализации модифицированного алгоритма, запущенной с единственным вспомогательным (рабочим) потоком;
5) выполнен сравнительный анализ зависимостей времени решения задач от размерности входа для реализации модифицированного алгоритма при K вспомогательных (рабочих) потоках, K принимает значения 1, 2, 4,..., 8·q, где (q – количество логических ядер процессора);
6) сформулирована рекомендация о выборе K для решения задачи.

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

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