



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ _____ «09.03.04 Программная инженерия»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

Название: _____ Распараллеливание алгоритма DBSAN

Дисциплина: _____ Анализ алгоритмов

Студент	<u>ИУ7-56Б</u>	_____	<u>Ковель А.Д.</u>
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель		_____	<u>Волкова Л.Л.</u>
Преподаватель		_____	<u>Строганов Ю.В.</u>
		Подпись, дата	И. О. Фамилия

Москва, 2022 г.

Оглавление

Введение	3
1 Аналитическая часть	5
1.1 Плотностный алгоритм DBSCAN	5
1.2 Параллелизация плотностного алгоритма DBSCAN	5
2 Конструкторский часть	7
2.1 Разработка алгоритмов	7
2.1.1 Разработка простого DBSCAN	7
2.1.2 Разработка параллельного DBSCAN	9
3 Технологический часть	10
3.1 Требования к программе	10
3.2 Средства реализации	10
3.3 Средства замера времени	10
3.4 Реализации алгоритма	11
3.5 Тестовые данные	16
4 Исследовательская часть	17
4.1 Технические характеристики	17
4.2 Демонстрация работы программы	17
4.3 Время выполнения реализации алгоритмов	18
Заключение	20
Список использованных источников	21

Введение

Многопоточность [1] — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций. Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Цель лабораторной работы — исследование плотностного алгоритма DBSCAN.

Задачи данной лабораторной:

- описать понятие параллельных вычислений и плотностный алгоритм DBSCAN;
- реализовать последовательный и параллельный алгоритм DBSCAN;
- провести сравнительный анализ алгоритмов на основе экспериментальных данных, а именно по времени;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе представляется описание алгоритма DBSCAN и его параллелизация.

1.1 Плотностный алгоритм DBSCAN

Алгоритм DBSCAN [2] (Density Based Spatial Clustering of Applications with Noise), плотностный алгоритм для кластеризации пространственных данных с присутствием шума). Данный алгоритм является решением разбиения (изначально пространственных) данных на кластеры произвольной формы [3]. Большинство алгоритмов, производящих плоское разбиение, создают кластеры по форме близкие к окружности, так как минимизируют расстояние документов до центра кластера.

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.

1.2 Параллелизация плотностного алгоритма DBSCAN

В изначальном плотностном алгоритме DBSCAN на вход подается множество точек, в виде матрицы, которая поэлементно обрабатывается в циклах. Так как в алгоритме необходимо рассматривать точки и ее ближайших соседей, данный алгоритм возможно обрабатывать параллельно. Идея параллелизация состоит в том, чтобы обрабатывать не все множество, а некоторые его части разбитые по потокам.

Вывод

Плотностный алгоритм DBSCAN независимо вычисляет элементы входного множества, что дает возможность реализовать параллельный вариант алгоритма.

2 Конструкторский часть

В данном разделе представлены схемы алгоритмов DBSCAN и его модификации.

2.1 Разработка алгоритмов

2.1.1 Разработка простого DBSCAN

На рисунке 2.1 приведена схема плотностного алгоритма DBSCAN.

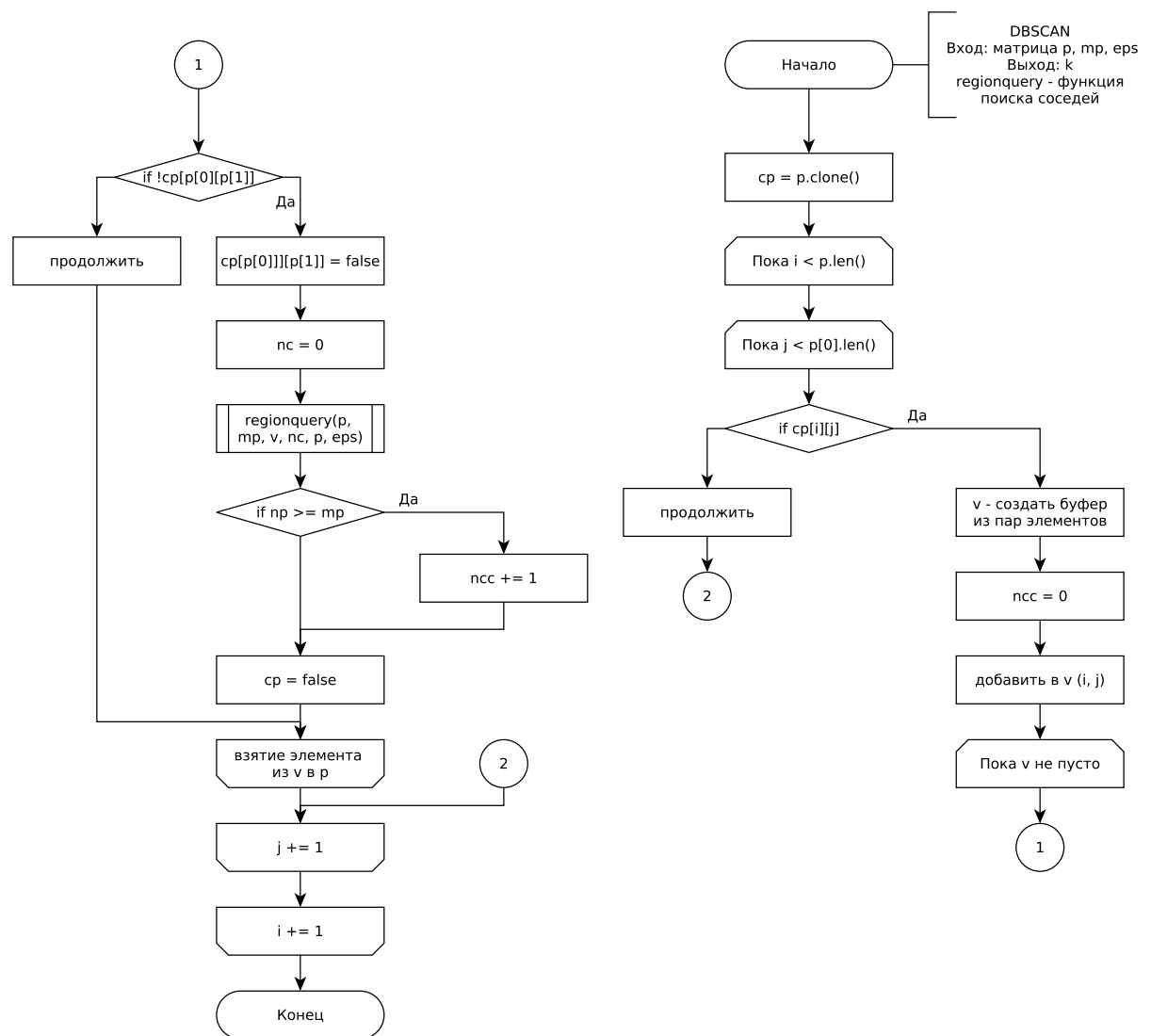


Рисунок 2.1 – Схема плотностного алгоритма DBSCAN

На рисунке 2.2 приведена схема функции поиска ближайшей соседний

точки в кластере.

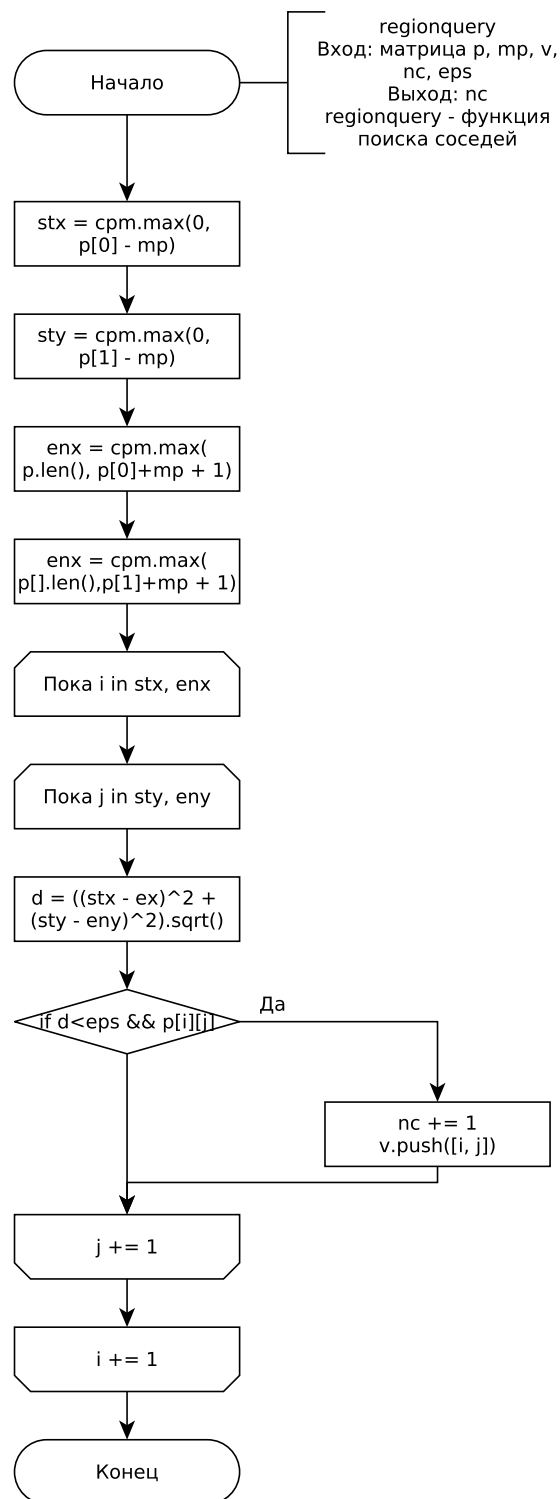


Рисунок 2.2 – Схема функции regionquery

2.1.2 Разработка параллельного DBSCAN

На рисунке 2.3 приведена схема параллельного алгоритма плотностного алгоритма DBSCAN.

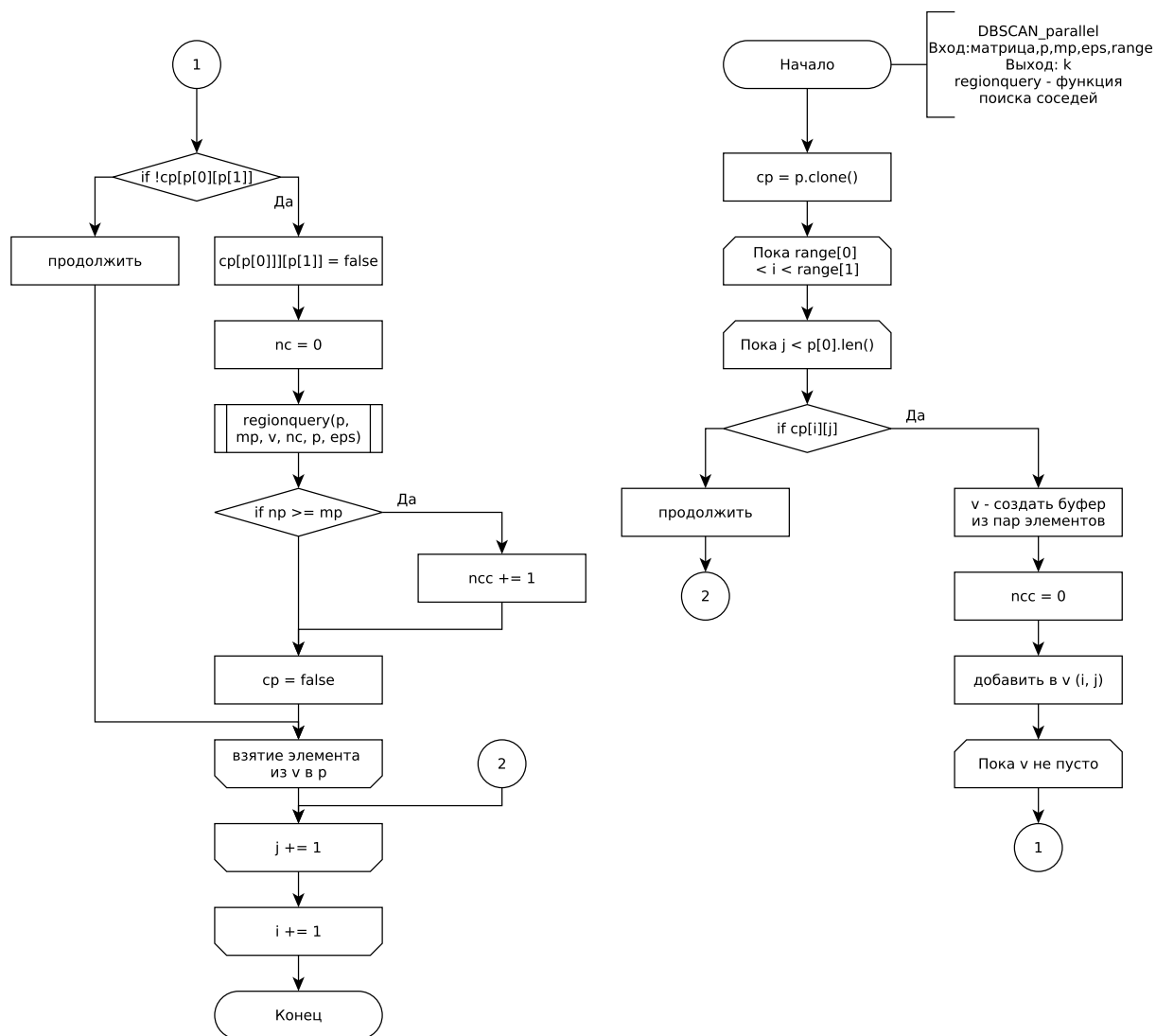


Рисунок 2.3 – Схема алгоритма параллельного DBSCAN

Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема алгоритма DBSCAN, а также схема параллельного DBSCAN.

3 Технологический часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинга кода.

3.1 Требования к программе

Программное обеспечение должно удовлетворять следующим требованиям:

- на вход подается имя файла;
- возвращается число кластеров;
- используется параллелизация программы;
- возможно измерение реального времени.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования Rust[4]. В данном языке есть все требующиеся инструменты для данной лабораторной работы. В качестве среды разработки была выбрана среда VS Code[5], запуск происходил через команду `cargo run`.

3.3 Средства замера времени

Замеры времени выполнения реализаций алгоритма проводиться при помощи функции `Instant::now()` [6] библиотеки `std::time`. Данная команда возвращает значения времени типа `int` в наносекундах.

Листинг 3.1 – Пример замера затраченного времени

```
1 pub fn run_tests(points: Vec<Vec<bool>>, min_ptx: usize, eps:
    f64, imgbuf: RgbImage) {
2     let n = 100;
3     let img_guard = imgbuf.clone();
4     for (algorithm, description) in
        MULTS_ARRAY.iter().zip(MULTS_DESCRIPTIONS.iter()) {
5         let time = Instant::now();
6         let mut result = 0;
7         for _ in 0..n {
8             let img_clone = img_guard.clone();
9             result = algorithm(&points, min_ptx, eps, img_clone);
10        }
11        let time = time.elapsed().as_nanos();
12    }
13 }
```

3.4 Реализации алгоритма

На листинге 3.2 приведена реализация простого алгоритма DBSCAN.

Листинг 3.2 – Реализация простого алгоритма DBSCAN

```

1 pub fn dbscan(points: &Vec<Vec<bool>>, min_ptx: usize, eps: f64,
2     mut imgbuf: RgbImage) -> u32 {
3     let mut cluster_count = 0;
4     let mut current_point = points.clone();
5     let mut current_color = get_random_color();
6     for i in 0..points.len() {
7         for j in 0..points[i].len() {
8             if current_point[i][j] {
9                 let mut v = Vec::<[usize; 2]>::new();
10                v.push([i, j]);
11                let mut neighbor_count_check = 0;
12                while !v.is_empty() {
13                    let p = v.pop().unwrap();
14                    if !current_point[p[0]][p[1]] {
15                        continue;
16                    }
17                    current_point[p[0]][p[1]] = false;
18
19                    let mut neighbor_count = 0;
20
21                    regionquery(points, min_ptx, &mut v, &mut
22                        neighbor_count, p, eps);
23
24                    if neighbor_count >= min_ptx {
25                        neighbor_count_check += 1;
26                        *imgbuf.get_pixel_mut(p[1] as u32, p[0] as u32) =
27                            image::Rgb(current_color);
28                    }
29                }
30                if neighbor_count_check > 0 {
31                    cluster_count += 1;
32                    current_color = get_random_color();
33                }
34                current_point[i][j] = false;
35            }
36        }
37    }
38    cluster_count
39 }

```

На листинге 3.3 приведена реализация функции поиска ближайших точек `regionquery`.

Листинг 3.3 – Реализация функции `regionquery`

```
1 fn regionquery(points: &Vec<Vec<bool>>, min_ptx: usize, v: &mut
  Vec::<[usize; 2]>, neighbor_count: &mut usize, p: [usize; 2],
  eps: f64) {
2   let start_x = cmp::max(0, p[0] - min_ptx);
3   let start_y = cmp::max(0, p[1] - min_ptx);
4   let end_x = cmp::min(points.len(), p[0] + min_ptx + 1);
5   let end_y = cmp::min(points[0].len(), p[1] + min_ptx + 1);
6   for _i in start_x..end_x {
7       //check how many neighbors in the distance
8       for _j in start_y..end_y {
9           let distance = get_eculid_distance(_i as i32, _j as i32,
              p[0] as i32, p[1] as i32);
10          if distance <= eps && points[_i][_j] {
11              *neighbor_count += 1;
12              v.push([_i, _j]);
13          }
14      }
15  }
16 }
```

На листинге 3.4 приведена реализация параллельного алгоритма DBSCAN.

Листинг 3.4 – Реализация параллельного алгоритма DBSCAN

```

1 pub fn dbscan_parallel(points: &Vec<Vec<bool>>>,
2 min_ptx: usize, eps: f64,
3 range: std::ops::Range<usize>,
4 guard_copy: Arc<Mutex<Vec<Vec<bool>>>>,
5 n: Arc<Mutex<u32>>,
6 mut imgbuf: RgbImage)
7 {
8     let mut cluster_count = n.lock().unwrap();
9     let mut current_point = guard_copy.lock().unwrap();
10    let mut current_color = get_random_color();
11    for i in range {
12        for j in 0..points[i].len() {
13            if current_point[i][j] {
14                let mut v = Vec::<[usize; 2]>::new();
15                v.push([i, j]);
16                let mut neighbor_count_check = 0;
17                while !v.is_empty() {
18                    let p = v.pop().unwrap();
19                    if !current_point[p[0]][p[1]] {
20                        continue;
21                    }
22                    current_point[p[0]][p[1]] = false;
23                    let mut neighbor_count = 0;
24                    regionquery(points, min_ptx, &mut v, &mut
                        neighbor_count, p, eps);
25                    if neighbor_count >= min_ptx {
26                        neighbor_count_check += 1;
27                        *imgbuf.get_pixel_mut(p[1] as u32, p[0] as u32) =
                            image::Rgb(current_color);
28                    }
29                }
30                if neighbor_count_check > 0 {
31                    *cluster_count += 1;
32                }
33                current_point[i][j] = false;
34            }
35        }
36    }
37 }
38 }

```

На листинге 3.5 приведена реализация функции параллелизации матрицы точек для алгоритма DBSCAN.

Листинг 3.5 – Реализация функции параллелизации матрицы точек

```
1 pub fn parrallize(points: &Vec<Vec<bool>>, min_ptx: usize, eps:
    f64, imgbuf: RgbImage, nofth: usize) -> u32 {
2
3     let counter = Arc::new(Mutex::new(0));
4     let current_point = Arc::new(Mutex::new(points.clone()));
5     let size = points.len() / (nofth + 1);
6
7     thread::scope(|s| {
8         let mut threads = Vec::with_capacity(nofth);
9
10        for i in 0..nofth {
11            let range = (i * size)..((i + 1) * size);
12            let guard_copy = current_point.clone();
13            let counter_copy = counter.clone();
14            let img_clone = imgbuf.clone();
15            threads.push(s.spawn(move |_| dbscan_parallel(points,
                min_ptx, eps, range, guard_copy, counter_copy,
                img_clone)));
16        }
17        let range = (size * nofth)..points.len();
18        let guard_\_timecopy = current_point.clone();
19        let counter_copy = counter.clone();
20        let img_clone = imgbuf.clone();
21        dbscan_parallel(points, min_ptx, eps, range, guard_copy,
            counter_copy, img_clone);
22        for th in threads {
23            th.join().unwrap();
24        }
25
26        let counter_copy = counter.clone();
27        let cluster_count = counter_copy.lock().unwrap();
28        *cluster_count
29    }).unwrap()
30 }
```

3.5 Тестовые данные

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы DBSCAN. Применена методология черного ящика. Тесты для всех алгоритмов пройдены *успешно*.

Таблица 3.1 – Функциональные тесты

Вход	Ожидаемый результат	Результат послед.	Результат паралл.
$()$	0	0	0
(0)	0	0	0
$(0 \ 1)$	1	1	1
(1)	1	1	1
$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	3	3	3
$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	2	2	2
$(1 \ 0 \ 0)$	1	1	1
$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	4	4	4

Вывод

Написано и протестировано программное обеспечение для решения поставленной задачи.

4 Исследовательская часть

4.1 Технические характеристики

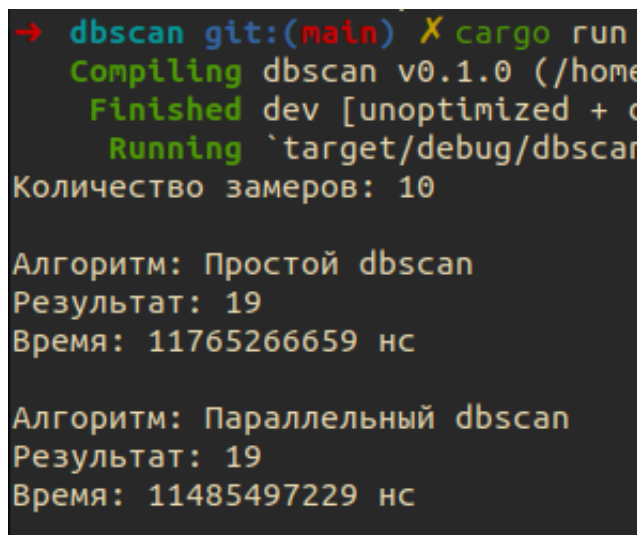
Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Pop!_OS 22.04 LTS [7] Linux [8];
- Оперативная память 16 Гбайт;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [9].

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений и системой тестирования.

4.2 Демонстрация работы программы

Результат работы программы, в которой выводится время работы алгоритма и количество найденных кластеров представлены на рисунке 4.1.



```
→ dbscan git:(main) X cargo run
Compiling dbscan v0.1.0 (/home
Finished dev [unoptimized + c
Running `target/debug/dbscan
Количество замеров: 10

Алгоритм: Простой dbscan
Результат: 19
Время: 11765266659 нс

Алгоритм: Параллельный dbscan
Результат: 19
Время: 11485497229 нс
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения реализации алгоритмов

Результаты замеров времени работы реализаций алгоритмов DBSCAN (в секундах) приведены в таблице 4.1. Сравнение проводилось между простым алгоритмом и параллельного алгоритма при исполнении на 8 потоках.

Таблица 4.1 – Результаты замеров реализаций алгоритмов DBSCAN

Размер	Простой DBSCAN	Параллельный DBSCAN
100	0.155203637	0.082057018
200	1.886091723	1.183958677
300	4.484127943	2.792452981
400	5.975877116	3.666312837
500	18.801009701	13.591029096

На графике 4.2 представлены времена работы параллельной и последовательной реализаций алгоритмов DBSCAN.

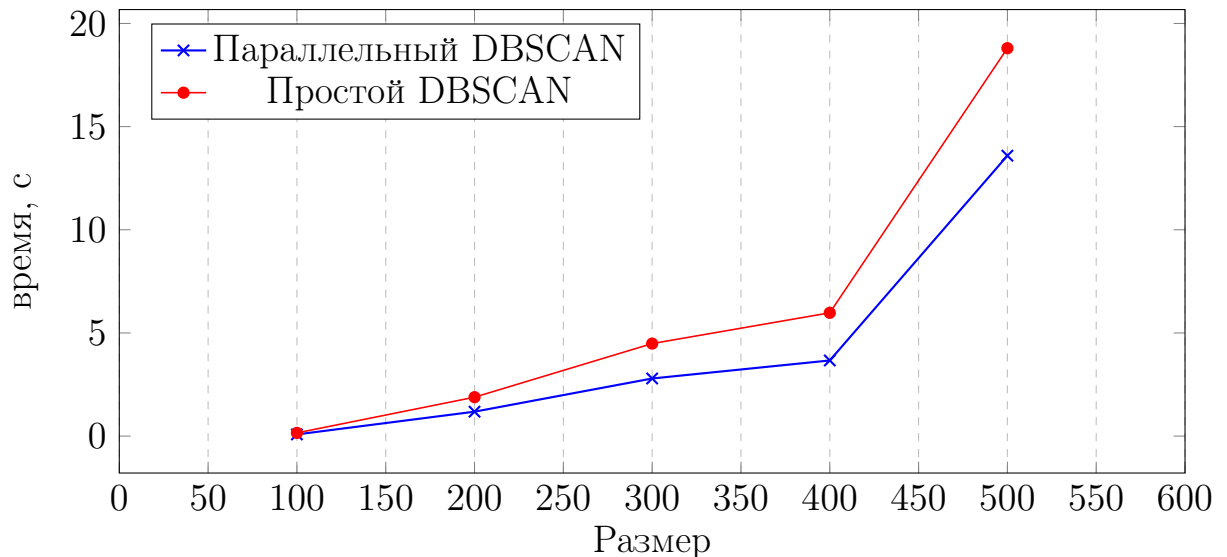


Рисунок 4.2 – Результаты работы реализации алгоритма DBSCAN.

На графике 4.3 представлена зависимость времени работы параллельного алгоритма DBSCAN от количество потоков на квадратном изображении размера 500 пикселей.

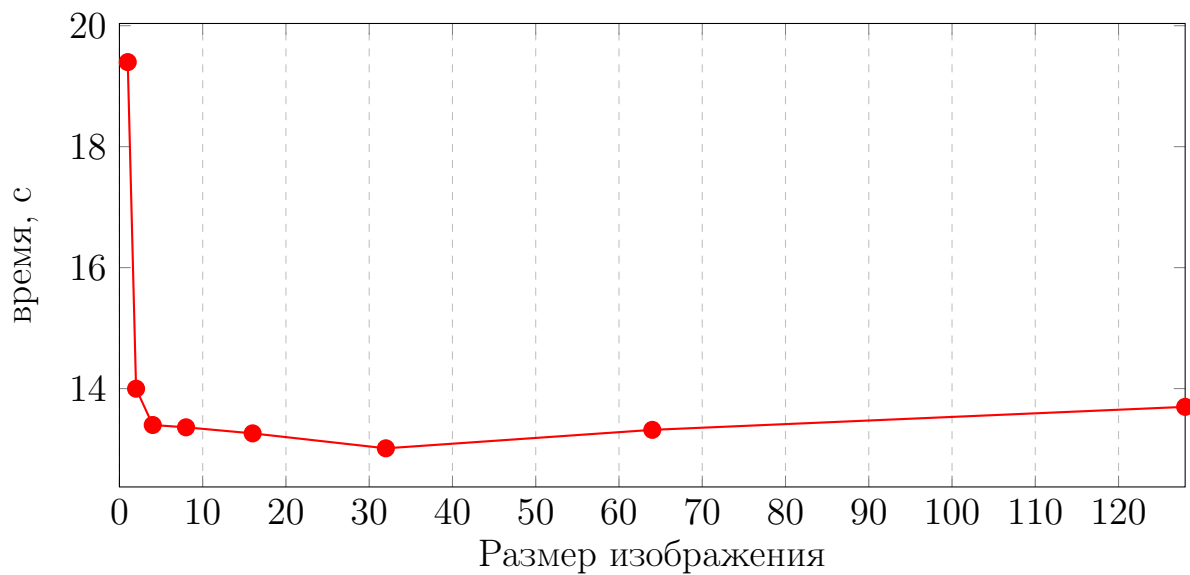


Рисунок 4.3 – Результаты работы реализации алгоритма DBSCAN на разном количестве потоков.

Вывод

В данном разделе были сравнены алгоритмы по времени. Параллельный алгоритм DBSCAN быстрее простого на 50 процентов (2.5 секунды) при размере 400 на 400. Наилучшее время параллельный алгоритм показывает на 32 потоках.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- были описаны понятия параллельных вычислений и плотностный алгоритм DBSCAN;
- был реализован последовательный и параллельный алгоритм DBSCAN;
- был сделан сравнительный анализ алгоритмов на основе экспериментальных данных;
- был подготовлен отчет о лабораторной работе.

Параллельный алгоритм DBSCAN быстрее простого на 50 процентов (2.5 секунды) при размере 400 на 400. Наилучшее время параллельный алгоритм показывает на 32 потоках.

Поставленная цель достигнута: алгоритм DBSCAN реализован, исследован и параллелизирован.

Список использованных источников

- [1] Vscode Документация[Электронный ресурс]. Режим доступа: <https://code.visualstudio.com/docs> (дата обращения: 24.09.2022).
- [2] Большакова Е.И. Клышинский Э.С. Ландэ Д.В. Носков А.А. Пескова О.В. Ягунова Е.В. Автоматическая обработка текстов на естественном языке и компьютерная лингвистика. – М.: МИЭМ, 2011. Т. 500. С. 197–200.
- [3] С.А. Иванов. Мировая система научной коммуникации как информационное пространство. – М.:, 2001., 2001. Т. 1. С. 1123–1126.
- [4] Rust Документация [Электронный ресурс]. Режим доступа: <https://www.rust-lang.org/learn> (дата обращения: 28.11.2022).
- [5] Vscode Документация[Электронный ресурс]. Режим доступа: <https://ru.bmstu.wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C> (дата обращения: 02.11.2022).
- [6] Функция `Instant::now()` модуля `std::time rust` [Электронный ресурс]. Режим доступа: <https://doc.rust-lang.org/std/time/index.html> (дата обращения: 28.09.2022).
- [7] Pop OS 22.04 LTS [Электронный ресурс]. Режим доступа: <https://pop.system76.com> (дата обращения: 04.09.2022).
- [8] Linux – Документация [Электронный ресурс]. Режим доступа: <https://docs.kernel.org> (дата обращения: 24.09.2022).
- [9] Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [Электронный ресурс]. Режим доступа: <https://www.amd.com/en/products/cpu/amd-ryzen-7-2700> (дата обращения: 04.09.2022).