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	ИУ7-56Б		Ковель А.Д.
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Волкова Л.Л.
Преподаватель			Строганов Ю.В.
		Подпись, дата	И. О. Фамилия

Москва, 2022 г.

Оглавление

Введение

Многопоточность — 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.

Многопоточность не следует путать ни с многозадачностью, ни с многопроцессорностью, несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность.

Цель лабораторной работы — реализация, исследование и параллелизация плотностного алгоритма DBSCAN.

Задачи данной лабораторной:

- изучить 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 будут представлены описание алгоритма DBSCAN и его параллелизация.

1.1 Плотностный алгоритм DBSCAN

Алгоритм DBSCAN [1] (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 [2]. Большинство алгоритмов, производящих плоское разбиение, создают кластеры по форме близкие к сферическим, 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 DBSCAB 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 раздел

В данном разделе представлены схемы алгоритмов DBSCAN и его модификации.

2.1 Разработка алгоритмов

На рисунке 2.1 приведена схема плотностного алгоритма DBSCAN.

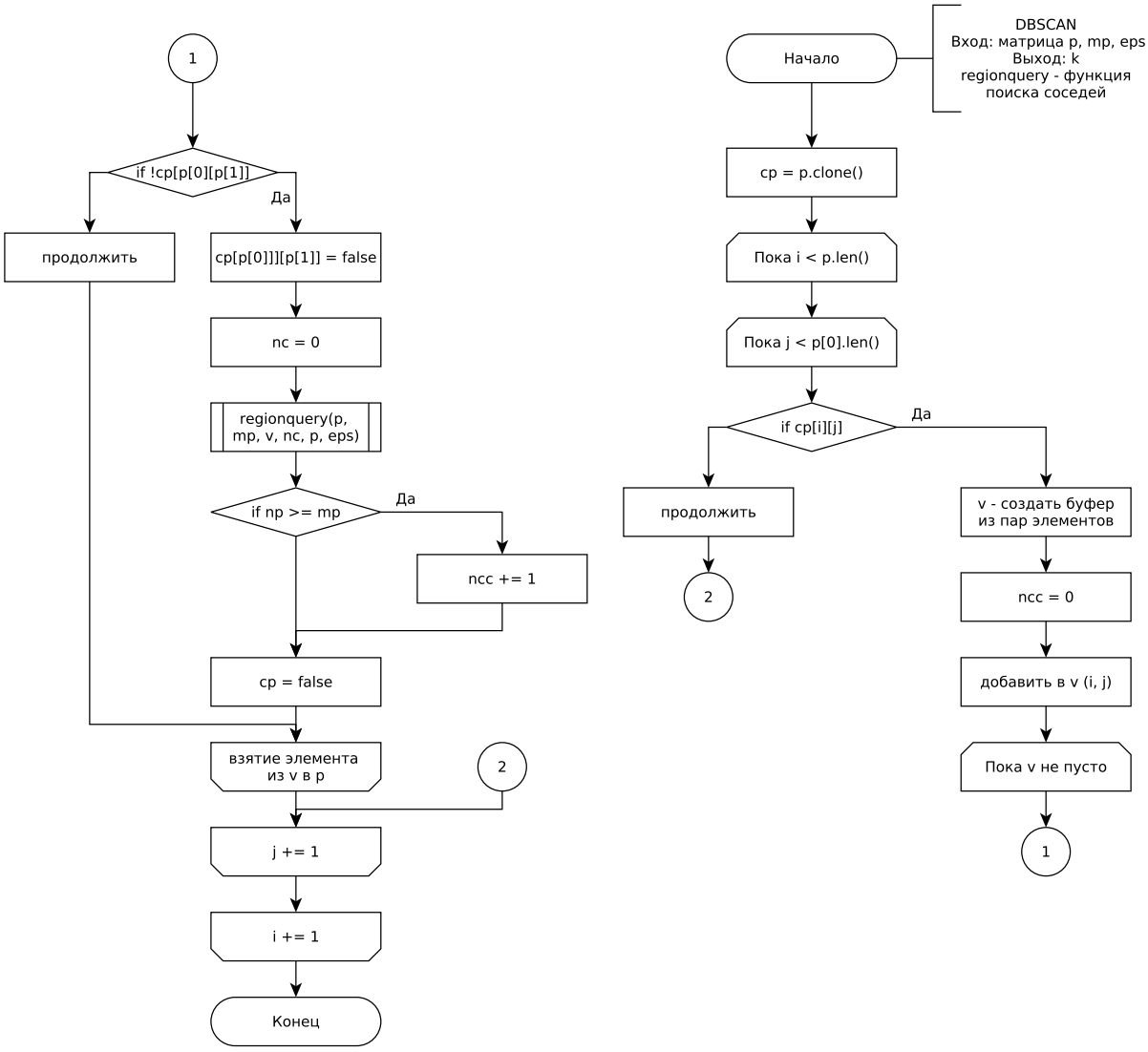

Рисунок 2.1 – Схема плотностного алгоритма DBSCAN

На рисунке 2.2 приведена схема функции поиска ближайшей соседний точки в кластере.

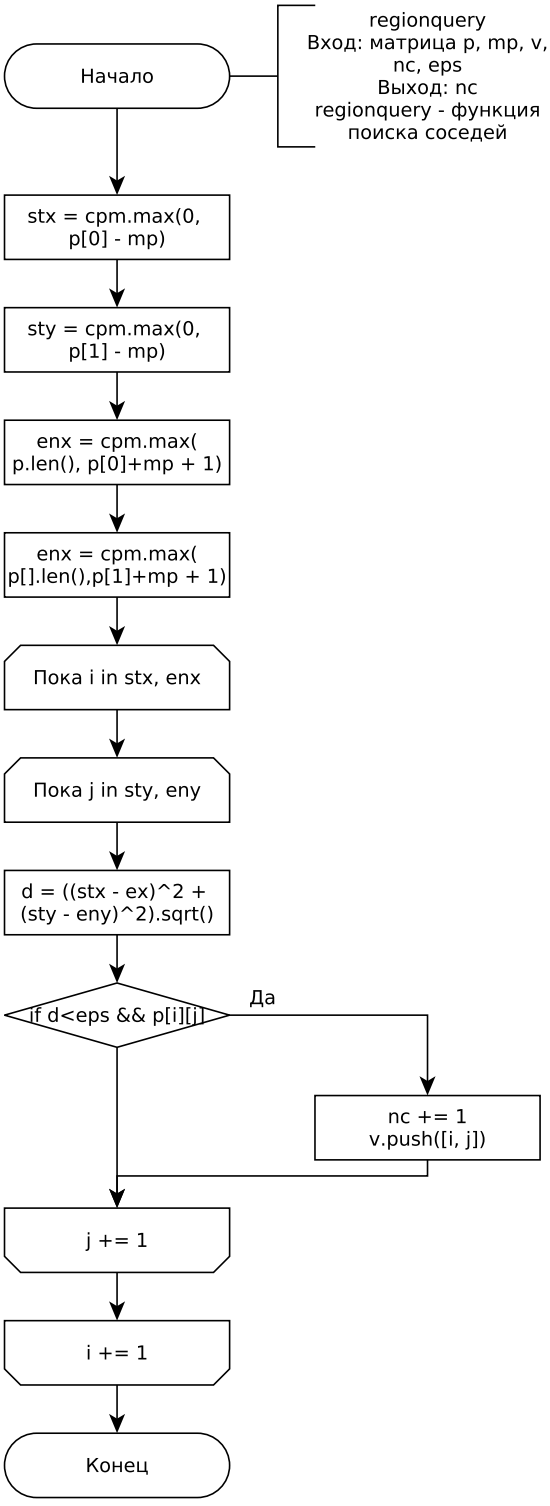

Рисунок 2.2 – Схема функции regionquery

2.1.1 Параллельная реализация алгоритма Винограда

Пусть размеры перемножаемых матриц непосредственно равны $M \times K$ и $K \times N$.

Рассмотрим необходимые для распараллеливания замечания:

- каждый из выделенных этапов может быть выполнен независимо от других;
- в следствие независимости этапов, каждый их них может быть выполнен в любой момент, в том числе и параллельно с другими;
- каждый из выделенных этапов содержит цикл в некотором промежутке, который может быть разбит на некоторое множество меньших промежутков, в сумме составляющих исходный;
- трудоёмкости первого и второго этапов - величины одного порядка и относятся M/N;
- трудоёмкость третьего этапа в N раз больше трудоёмкости первого этапа и в M раз больше трудоёмкости второго этапа, что, не позволяет распараллелить третий этап с первым и (или) вторым;
- четвертый этап требует обращения к матрице на каждой итерации цикла, что при распараллеливании приведёт к большому числу блокирований разделяемой памяти, и, в купе с затратами на порождение потоков, будет неэффективно.

На рисунке ?? представлена схема алгоритма функции, запускающая в требуемом количестве потоков функцию-аргумент, передавая ей равные по размеру промежутки из разбиения исходного. С помощью этой функции распараллеливаются этапы, описанные в рисунке ??. 170mmparallel-Схема распараллеливания произвольной функции, способной выполняться в некоторых независимых промежутках

Исходя из всего, описанного выше, можно сделать вывод, что третий этап следует параллелить независимо от других, в то время как первый и второй этапы можно сделать как параллельно (где каждому из этапов достается половина от общего числа потоков), так и последовательно (где каждому из этапов достается общее число потоков). Так же очевидно, что не следует параллелить четвертый этап.

На рисунке ?? представлена схема с параллельным выполнением первого и второго этапов, а на рисунке ?? представлена схема с последовательным выполнением первого и второго этапов.

170mmparallel1Схема с параллельным выполнением первого и второго этапов

Рисунок 2.3 – Схема с последовательным выполнением первого и второго этапов

Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема алгоритма Винограда, а так же после разделения алгоритма на этапы были предложены 2 схемы параллельного выполнения данных этапов.

3 Технологический раздел

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

3.1 Требования к программе

Программное обеспечение должно удовлетворять следующим требованиям:

- на вход подается массив целых чисел в диапозоне от 0 до 10000;
- возвращается отсортированный по возрастанию массив;
- в программе возможно измерение процессорного времени.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования Python[3].

В данном языке есть все требующиеся инструменты для данной лабораторной работы.

В качестве среды разработки была выбрана среда VS Code[4], запуск происходил через команду python main.py.

3.3 Средства замера времени

Замеры времени выполнения реализаций алгоритма будут проводиться при помощи функции process_time [5] библиотеки time. Данная команда возвращает значения процессорного времени типа int в наносекундах.

Замеры времени для каждой реализации алгоритма и для каждого комплекта входных данных проводились 100 раз.

Листинг 3.1 – Пример замера затраченного времени

```
  def test_simple_mult(A, B):
  # Start the stopwatch / counter
  t1_start = process_time()
  for i in range(N_TEST):
  simple_mult(A, M, B, N, M)
  # Stop the stopwatch / counter
  t1_stop = process_time()
```

3.4 Реализации алгоритма

На листинг 3.2 демонстрируется реализация алгоритма сортировки подсчетом.

Листинг 3.2 – Алгоритм сортировки подсчетомя

```
def counting_sort(alist, largest):
  c = [0]*(largest + 1)
  for i in range(len(alist)):
    c[alist[i]] += 1
  c[0] -= 1
  for i in range(1, largest + 1):
    c[i] += c[i - 1]
  result = [0]*len(alist)
  for x in reversed(alist):
    result[c[x]] = x
    c[x] -= 1
  return result
```

На листинге 3.3 демонстрируется реализация алгоритма сортировки бинарным деревом.

Листинг 3.3 – Алгоритм сортировки бинарным дерево

```
def binary_sort(alist):
  tree = TreeNode()
  for i in alist:
    tree.inpurt_in_tree(i)
  arr = []
  tree.pre_order(arr)
  return arr
```

На листинге 3.4 демонстрируется класс бинарного дерева.

Листинг 3.4 – Класс бинарного дерев

```
class TreeNode:
def __init__(self, value=None):
  self.value = value
  self.left = None
  self.right = None

def inpurt_in_tree(self, elem):
  if self.value is None:
    self.value = elem
    return

  if elem < self.value:
    if self.left is None:
      self.left = TreeNode(elem)
    else:
      self.left.inpurt_in_tree(elem)

  elif elem >= self.value:
    if self.right is None:
      self.right = TreeNode(elem)
    else:
      self.right.inpurt_in_tree(elem)

def pre_order(self, arr):
  if self.value:
    if self.left:
      self.left.pre_order(arr)
    if self.value:
      arr.append(self.value)

    if self.right:
      self.right.pre_order(arr)
```

На листинге 3.5 демонстрируется реализация алгоритма поразрядной сортировки.

Листинг 3.5 – Алгоритм поразрядной сортировки

```
1  def radixSort(array):
2      max_element = max(array)
3
4      place = 1
5      while max_element // place > 0:
6          countingSort(array, place)
7          place *= 10
8      return array
```

3.5 Тестовые данные

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

Таблица 3.1 – Функциональные тесты

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4, 5, 6, 7]$	$[1, 2, 3, 4, 5, 6, 7]$	$[1, 2, 3, 4, 5, 6, 7]$
$[7, 6, 5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5, 6, 7]$	$[1, 2, 3, 4, 5, 6, 7]$
$[9, 7, 5, 1, 4]$	$[1, 4, 5, 7, 9]$	$[1, 4, 5, 7, 9]$
$[69]$	$[69]$	$[69]$
$[]$	$[]$	$[]$

Вывод

Было 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 [6] Linux [7];
- Оперативная память 16 Гб;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [8].

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

4.2 Демонстрация работы программы

На рисунке **??** представлен результат работы программы, в которой выводится исходный массив сгенерированный программой и три отсортированных массива.

4.3 Процессорное время выполнения реализации алгоритмов

Результаты замеров времени работы реализаций алгоритмов сортировки на различных входных данных (в мс) приведены в таблицах 4.1, 4.2 и 4.3.

Таблица 4.1 – Результаты замеров реализаций сортировок, входными данными являлись отсортированные по возрастанию значений массивы.

Размер	Подсчетом	Поразрядная	Бинарным деревом
100	0.1662	0.0714	0.8650
200	0.5113	0.2058	3.3357
300	1.1026	0.3131	7.6464
400	2.0140	0.4364	13.6841
500	3.3046	0.5591	21.5524
600	5.0567	0.6798	31.3052
700	6.6944	0.7852	43.0406
800	8.5163	0.8766	56.4318

Таблица 4.2 – Результаты замеров реализаций сортировок, входными данными являлись отсортированные по убыванию значений массивы.

Размер	Подсчетом	Поразрядная	Бинарным деревом
100	0.1606	0.1048	0.7138
200	0.5005	0.2008	2.7633
300	1.0747	0.3110	6.3060
400	1.9383	0.4312	11.3831
500	3.1148	0.5427	18.0577
600	4.6409	0.6693	26.0260
700	6.7969	0.8317	36.7397
800	8.7922	0.9583	47.2628

Таблица 4.3 – Результаты замеров реализаций сортировок, входными данными являлись заполненные числами со случайными значениями массивы.

Размер	Подсчетом	Поразрядная	Бинарным деревом
100	0.2734	0.1043	0.1560
200	0.8321	0.2090	0.3756
300	1.6837	0.3142	0.6025
400	2.8938	0.4281	0.9785
500	4.4438	0.5419	1.1784
600	6.4153	0.6704	1.5523
700	8.6692	0.7678	1.9018
800	9.3752	0.8992	2.2986

4.4 Результаты выполнения реализаций алгоритмов

На графике **??** представлено время работы сортировок, входными данными являлись заполненные числами со случайными значениями массивы.

На графике **??** представлено время работы сортировок, входными данными являлись отсортированные по возрастанию значений массивы.

На графике **??** представлено время работы сортировок, входными данными являлись отсортированные по убыванию значений массивы.

Вывод

В данном разделе были сравнены алгоритмы по времени. Сортировка бинарным деревом на отсортированных массивах и сортировка подсчетом работает на случайном массиве дольше всех.

Самая быстрая сортировка поразрядная на любых данных.

Теоритические результаты оценки трудоемкости и полученные практическим образом результаты замеров совпадают.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы 3 алгоритма сортировки массива: бинарным деревом, поразрядная, подсчетом;
- был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- был сделан сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовлен отчет о лабораторной работе.

Сортировка бинарным деревом на отсортированных массивах и сортировка подсчетом работает на случайном массиве дольше всех. Самая быстрая сортировка поразрядная на любых данных.

Поставленная цель достигнута: алгоритмы умножения матриц изучены и проанализированы.

Список использованных источников

[1] 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.

[2] С.А. Иванов. Мировая система научной коммуникации kar информационное пространство. – М.:, 2001., 2001. Т. 1. С. 1123–1126.

[3] Python Документация[Электронный ресурс]. Режим доступа: `https://docs.python.org/3/` (дата обращения: 24.09.2022).

[4] Vscode Документация[Электронный ресурс]. Режим доступа: `https://code.visualstudio.com/docs` (дата обращения: 24.09.2022).

[5] Функция process_time модуля time python [Электронный ресурс]. Режим доступа: `https://docs-python.ru/standart-library/modul-time-python/funktsija-process-time-modulja-time/` (дата обращения: 04.09.2022).

[6] Pop OS 22.04 LTS [Электронный ресурс]. Режим доступа: `https://pop.system76.com` (дата обращения: 04.09.2022).

[7] Linux – Документация [Электронный ресурс]. Режим доступа: `https://docs.kernel.org` (дата обращения: 24.09.2022).

[8] 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).