Государственное образовательное учреждение высшего образования

«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина»

09.03.02 410008

**«Параллельные вычислительные системы»**

Семестр 7

**ОТЧЁТ**

по Лабораторной работе №7

**«Многопоточность в Python на примере задачи кластеризации»**



Преподаватель: Кашин И.В.

Студент : Шардаков В.А.

Группа : Фт-410008

Дата : 03.12.2024

Екатеринбург 2024

# Задачи

Кластеризация является важной задачей в области обработки и анализа данных, позволяющей выявлять структуры внутри наборов информации, которые могут быть визуально и статистически обоснованными. В данной лабораторной работе рассмотрен алгоритм кластеризации на основе генерации точек с использованием функции make\_blobs, которая создает кластеры в двумерном пространстве. Графическое представление полученных кластеров позволяет наглядно оценить эффективность работы алгоритма. Наша задача заключается в реализации алгоритма, который находит центры кластеров, повторно пересчитывает их и на основе этих центров делит точки на группы:

Основными задачами данной лабораторной работы являются:

1. Реализация алгоритма кластеризации, который:
   * Определяет центры кластеров, основываясь на плотности точек вокруг каждого потенциального центра.
   * Итерируется, пересчитывая центры кластеров и присваивая точки к ближайшим центрам, до достижения сходимости.
2. Оптимизация процесса кластеризации путем использования параллельных вычислений, что позволяет ускорить работу алгоритма за счет распределения нагрузки на многоядерные процессоры.
3. Визуализация результатов кластеризации с помощью графиков, позволяющих сравнить исходные данные и обнаруженные кластеры, а также исследовать зависимость времени выполнения от количества задействованных процессов.

# Ход работы

## Метод решения задачи

Для реализации задачи кластеризации была выбрана последовательность шагов, нацеленных на точное определение центров кластеров и эффективное распределение вычислительных ресурсов:

1. **Генерация точек**: используя библиотеку sklearn, создаются точки с помощью функции make\_blobs, которые образуют естественные кластеры в двумерной плоскости.
2. **Определение центров кластеров**: алгоритм сначала находит потенциальные центры кластеров, исследуя среднее количество точек в окрестности каждой точки. Для этого рассчитываются веса точек на основе расстояния до потенциального центра.
3. **Итеративное перераспределение**: на каждом шаге алгоритма точки присваиваются к ближайшим центрам кластеров, пересчитываются новые центры на основе текущего распределения, и процесс повторяется до тех пор, пока изменения в центре не станут минимальными.
4. **Параллельные вычисления**: для ускорения процесса используется модуль multiprocessing, который позволяет разделить обработку данных по процессам, значительно сокращая общее время выполнения.
5. **Визуализация результатов**: Полученные кластеры визуализируются с использованием matplotlib, что позволяет увидеть распределение точек и их принадлежность к кластерам, а также оценить влияние количества процессов на время вычислений

## Решение

***Результат работы программы (ПРИЛОЖЕНИЕ А)***

*Исходный график (3000 точек):*

*Изображение выглядит как снимок экрана, Красочность

Автоматически созданное описание*

*График после работы программы (3000 точек):*

*Изображение выглядит как снимок экрана, Красочность

Автоматически созданное описание*

*График зависимости времени от количества потоков:*

*Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание*

# Выводы

Эффективность параллельной сортировки:

Реализованный алгоритм параллельной сортировки продемонстрировал значительное сокращение времени выполнения при увеличении числа рабочих процессов. Это подтверждает эффективность использования многопоточности для решения задач, связанных с обработкой больших объемов данных.

Зависимость времени выполнения от количества потоков:

Графическое представление времени выполнения показало, что с увеличением числа потоков время сортировки уменьшается. Однако наблюдались ограничения по снижению времени выполнения, достигаемым путем дальнейшего увеличения процесса. Это связано с накладными расходами на управление потоками и объединение результатов.

Идеальное время выполнения:

Расчет идеального времени выполнения, основанный на теоретическом распределении нагрузки, подтвердил, что многопоточность может использоваться для оптимизации производительности, хотя реальное время выполнения почти никогда не достигало идеального, из-за накладных расходов.

Рекомендации по использованию:

Для систем с многопроцессорными архитектурами параллельные алгоритмы, такие как реализованная сортировка, могут заметно повысить производительность, особенно для задач, связанных с большим объемом данных. Однако важно тщательно анализировать пределы эффективности, чтобы избежать избыточных затрат на создание и управление потоками.

Перспективы дальнейшего изучения:

Дальнейшее изучение может сосредоточиться на сравнении различных алгоритмов сортировки в параллельной среде, а также исследовании альтернативных методов распределения нагрузки. Также возможно изучение более сложных структур данных и алгоритмов, которые могли бы дополнительно улучшить производительность.

Практическое применение:

Подобная реализация алгоритмов параллельной сортировки имеет широкие практические применения в реальных задачах, таких как обработка больших наборов данных в научных исследованиях, финансовом анализе и других областях, где требуется быстрая и эффективная сортировка.

# ПРИЛОЖЕНИЕ А

import matplotlib.pyplot as plt  
import multiprocessing as mp  
import numpy as np  
from PIL import Image, ImageOps  
import time  
from scipy.special import rel\_entr  
from skimage.metrics import structural\_similarity  
  
  
# KL-дивергенция для сравнения распределений  
def is\_similar(img\_matrix, agent\_matrix):  
 if np.all(img\_matrix == 0) or np.all(agent\_matrix == 0):  
 # являются ли обе матрицы нулевыми  
 return np.all(img\_matrix == 0) and np.all(agent\_matrix == 0)  
 else:  
 # малое смещение  
 c = 1e-12  
 P = img\_matrix / np.sum(img\_matrix) + c  
 Q = agent\_matrix / np.sum(agent\_matrix) + c  
 return np.sum(rel\_entr(P, Q))  
  
  
# структурное сходство (SSIM)  
def similarity(image1, image2):  
 similarity\_index, \_ = structural\_similarity(image1, image2, full=True)  
 return similarity\_index  
  
  
# приведение матрицы к целым числам от 0 до 255  
def scaling(agent\_matrix):  
 if np.all(agent\_matrix == 0):  
 return agent\_matrix  
  
 min\_val = np.min(agent\_matrix)  
 max\_val = np.max(agent\_matrix)  
 scaled\_agent\_matrix = np.round(255 \* (agent\_matrix - min\_val) / (max\_val - min\_val))  
 return scaled\_agent\_matrix.astype(np.uint8)  
  
  
# получение соседних координат  
def get\_neighbor\_coords(x, y, max\_x, max\_y):  
 x = int(x)  
 y = int(y)  
 neighbor\_coords = np.array([(x + dx, y + dy)  
 for dx in (-1, 0, 1)  
 for dy in (-1, 0, 1)  
 if (dx != 0 or dy != 0) and x + dx >= 0 and y + dy >= 0 and x + dx < max\_x and y + dy < max\_y])  
 return neighbor\_coords  
  
  
# получение следующих оптимальных координат для движения агента (тех при которых разница распределений минимальна)  
def get\_next\_step\_coords(x, y, img\_matrix, agent\_matrix):  
 neighbor\_coords = get\_neighbor\_coords(x, y, max\_x=img\_matrix.shape[0], max\_y=img\_matrix.shape[1])  
  
 # массив распределений в соседних точках оригинальной матрицы  
 img\_distributions = img\_matrix[neighbor\_coords[:, 0], neighbor\_coords[:, 1]] / np.sum(img\_matrix)  
  
 # массив потенциальных распределений в соседних точках матрицы для агентов  
 agent\_potential\_distributions = (1 + agent\_matrix[neighbor\_coords[:, 0], neighbor\_coords[:, 1]]) / (  
 1 + np.sum(agent\_matrix))  
  
 dist\_diff = np.abs(img\_distributions - agent\_potential\_distributions)  
  
 return neighbor\_coords[np.argmin(dist\_diff)]  
  
  
# обновление матрицы числа посещений агентами соответствующих ячеек  
def update\_agent\_matrix(agent\_coords, img\_matrix, agent\_matrix, num\_iter):  
 new\_agent\_matrix = np.copy(agent\_matrix)  
  
 # назначенные агентам индексы  
 agent\_idxs = np.arange(agent\_coords.shape[0])  
  
 # установка агентов в начальные координаты  
 if np.all(new\_agent\_matrix == 0):  
 new\_agent\_matrix[agent\_coords[:, 0], agent\_coords[:, 1]] = 1  
  
 # число итераций перед усреднением матрицы агентов между процессами  
 for \_ in range(num\_iter):  
 # перемешивание индексов агентов  
 shuffled\_agent\_idxs = np.random.permutation(agent\_idxs)  
  
 for agent\_idx in shuffled\_agent\_idxs:  
 # поиск новой координаты в зависимости от распределения  
 new\_x, new\_y = get\_next\_step\_coords(\*agent\_coords[agent\_idx], img\_matrix, new\_agent\_matrix)  
  
 if new\_agent\_matrix[new\_x, new\_y] != 255:  
 # изменение количества посещений агентами позиции  
 new\_agent\_matrix[new\_x, new\_y] += 1  
  
 # обновление текущих координат каждого агента  
 agent\_coords[agent\_idx] = new\_x, new\_y  
  
 return new\_agent\_matrix, agent\_coords  
  
  
# воосстановление исходного изображения с помощью агентов  
def image\_reconstruction(img\_matrix, agent\_num, num\_proc, ssim\_accuracy=None):  
 # число итераций перед усреднением значений м ежду процессами  
 num\_iter = 4000  
  
 agent\_matrix = np.zeros\_like(img\_matrix, dtype=np.uint8)  
 new\_agent\_matrix = agent\_matrix  
  
 # сходство между изображениями  
 ssim = similarity(img\_matrix, agent\_matrix)  
 new\_ssim = ssim  
  
 start\_time = time.time()  
  
 if num\_proc == 1:  
 while new\_ssim >= ssim:  
 ssim = new\_ssim  
 agent\_matrix = new\_agent\_matrix  
  
 if ssim\_accuracy and ssim >= ssim\_accuracy:  
 break  
  
 # случайные начальные координаты  
 agent\_coords = np.random.randint(low=0, high=128, size=(agent\_num, 2))  
 new\_agent\_matrix, agent\_coords = update\_agent\_matrix(agent\_coords, img\_matrix, agent\_matrix, num\_iter)  
 new\_ssim = similarity(img\_matrix, new\_agent\_matrix)  
  
 else:  
 # массив для хранения случайных начальных координат агентов всех процессов (при первом и последующих запусках)  
 all\_agent\_coords = np.empty((num\_proc, agent\_num, 2), np.uint32)  
  
 with mp.Pool(processes=num\_proc) as pool:  
 while new\_ssim >= ssim:  
 ssim = new\_ssim  
 agent\_matrix = new\_agent\_matrix  
  
 if ssim\_accuracy and ssim >= ssim\_accuracy:  
 break  
  
 # генерация начальных координат агентов  
 for i in range(num\_proc):  
 all\_agent\_coords[i, :, 0] = np.random.randint(0, img\_matrix.shape[0], size=agent\_num)  
 all\_agent\_coords[i, :, 1] = np.random.randint(0, img\_matrix.shape[1], size=agent\_num)  
  
 # аргументы целевой функции  
 args = [(all\_agent\_coords[n], img\_matrix, agent\_matrix, num\_iter) for n in range(num\_proc)]  
  
 results = pool.starmap\_async(update\_agent\_matrix, args).get()  
 result\_agent\_matrix\_list = np.array([res[0] for res in results])  
  
 # попадание агентов в те же координаты для соответствующих процессов  
 agent\_coords = np.array([res[1] for res in results])  
  
 # усреднение матрицы  
 new\_agent\_matrix = scaling(np.sum(result\_agent\_matrix\_list, axis=0) / num\_proc)  
 new\_ssim = similarity(img\_matrix, new\_agent\_matrix)  
  
 # закрытие пула  
 pool.close()  
 # ожидание завершения процессов  
 pool.join()  
  
 end\_time = time.time()  
  
 return agent\_matrix, ssim, end\_time - start\_time  
  
  
def plot\_time\_dependence(nums\_processes, times):  
 plt.title('Зависимость времени вычислений\nот количества выделенных процессов')  
 plt.plot(nums\_processes, times, label='Изменение времени по результатам эксперимента')  
  
 # график при начальном времени (1 процесс) и постепенном разбиении на подпроцессы (эталон: гипербола)  
 plt.plot(nums\_processes, [times[0] / i for i in nums\_processes], label='Эталонное изменение времени')  
 plt.ylabel('Время выполнения в секундах')  
 plt.xlabel('Количество выделенных процессов')  
 plt.legend()  
 plt.show()  
  
  
def show\_images(images, titles):  
 num\_images = len(images)  
  
 # размер сетки  
 cols = int(np.ceil(np.sqrt(num\_images)))  
 rows = int(np.ceil(num\_images / cols))  
  
 # параметры отображения  
 \_, axes = plt.subplots(rows, cols, figsize=(15, 15))  
 for i, ax in enumerate(axes.flat):  
 if i < num\_images:  
 ax.imshow(images[i], cmap='gray')  
 ax.set\_title(titles[i])  
 ax.axis('off')  
  
 plt.show()  
  
  
def main():  
 # загрузка изображения  
 original\_img = Image.open('image.jpg')  
 gray\_img = ImageOps.grayscale(original\_img)  
  
 img\_matrix = np.asarray(gray\_img, dtype=np.uint8)  
  
 # список разного числа используемых процессов (зависит от числа доступных ЦПУ)  
 nums\_processes = [i + 1 for i in range(mp.cpu\_count())]  
  
 # список измерений времени для разного количества используемых процессов  
 times = []  
  
 # число агентов  
 agent\_num = int(img\_matrix.size \* 0.01)  
  
 images = [img\_matrix]  
 titles = ['Оригинал']  
  
 # искомая степень сходства изображений  
 ssim\_accuracy = None  
  
 # эксперимент для разного числа процессов  
 for num\_proc in nums\_processes:  
 if num\_proc == 1:  
 agent\_matrix, ssim, time = image\_reconstruction(img\_matrix, agent\_num, num\_proc)  
 ssim\_accuracy = ssim  
 else:  
 agent\_matrix, ssim, time = image\_reconstruction(img\_matrix, agent\_num, num\_proc, ssim\_accuracy)  
  
 images.append(agent\_matrix)  
 titles.append(f'CPU: {num\_proc}, схожесть: {ssim \* 100:.1f}%')  
 times.append(time)  
  
 show\_images(images, titles)  
 plot\_time\_dependence(nums\_processes, times)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()