Государственное образовательное учреждение высшего образования

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

09.03.02 310008

**«Параллельные вычислительные системы»**

Семестр 7

**ОТЧЁТ**

по Лабораторной работе №1

**«Многопоточность в Python на примере расчёта численного интеграла»**



Преподаватель: Кашин И.В.

Студент : Шардаков В.А.

Группа : Фт-410008

Дата : 20.09.2024

Екатеринбург 2024

# Задачи

В рамках данной работы было поставлено несколько ключевых задач:

1. Выполнить численное интегрирование функции методом левых прямоугольников на интервале от 5 до 10.
2. Реализовать решение с использованием библиотеки multiprocessing в языке Python для оптимизации вычислительного процесса.
3. Изучить влияние количества используемых ядер (от 1 до 12) на эффективность интегрирования, учитывая, что на исследуемом ноутбуке имеется 4 физических ядра и 8 потоков.

Функция:

Изображение выглядит как Шрифт, линия, число, символ

Автоматически созданное описание

# Ход работы

## Метод решения задачи

Метод левых прямоугольников является одним из основных методов численного интегрирования, который используется для приближенного вычисления определенного интеграла функции на заданном интервале. Суть данного метода заключается в том, что интеграл функции f(x) на интервале [a, b] приближается суммой площадей прямоугольников, основание которых параллельно оси абсцисс и высота определяется значением функции в левом конце каждого подинтервала.

Для реализации метода левых прямоугольников интервал [a, b] делится на (n) равных частей, каждая длина которых равна . Затем для каждого подинтервала  (где ,  ), высота прямоугольника определяется как .

Формула для вычисления интеграла может быть записана следующим образом:

Изображение выглядит как черный, темнота

Автоматически созданное описание

При этом I – это приближенное значение интеграла, n – количество подинтервалов, а  – значение функции в левом конце подинтервала.

## Метод параллелизации

## 2.2.1 Библиотека multiprocessing в Python

Библиотека multiprocessing является одним из центральных инструментов в Python для реализации параллелизма. Она позволяет создавать и управлять процессами, а не потоками, что дает возможность эффективно использовать многоядерные процессоры. Каждый процесс, запущенный с помощью этой библиотеки, является независимым экземпляром интерпретатора Python и имеет свою собственную память. Это позволяет избежать проблем, связанных с Global Interpreter Lock (GIL), который ограничивает выполнение кода Python в одном потоке.

Библиотека multiprocessing предоставляет различные механизмы для управления процессами, такие как:

* Process: Основной класс для создания новых процессов.
* Pool: позволяет создавать пул процессов для параллельного выполнения задач.
* Queue: используется для обмена данными между процессами.
* Value and Array: Объекты, которые позволяют передавать данные между процессами.

Применение multiprocessing дает возможность выполнять тяжелые вычисления параллельно, что существенно ускоряет обработку больших объемов данных. Это особенно важно для задач, требующих значительных вычислительных ресурсов, таких как численное интегрирование.

## 2.2.2 Ограничения GIL

Global Interpreter Lock (GIL) является механизмом, используемым в интерпретаторе CPython для обеспечения потокобезопасности. Он гарантирует, что только один поток может выполнять байт-код Python в любой момент времени. Это приводит к тому, что в многопоточных приложениях с использованием библиотеки threading, CPU-вычислений не происходит параллельно, поскольку все потоки, по сути, блокируются при выполнении основного потока.

Таким образом, несмотря на наличие нескольких потоков, задача выполняется только одним потоком, и потенциальные преимущества многоядерности остаются не реализованными. Это делает библиотеку threading менее подходящей для задач, требующих значительных вычислительных ресурсов.

## 2.2.3 Почему threading не подходит для данной задачи

Для численного интегрирования, как в нашем случае, требуется выполнение большого количества вычислений, что делает методы более подходящими для параллелизации на уровне процесса, а не потока.

Использование библиотеки threading может привести к следующим проблемам:

* Ограничение GIL: из-за GIL в многопоточных приложениях работа только одного потока ограничивается во времени. В результате параллелизм не дает желаемого эффекта, поскольку все вычисления осуществляются последовательно на одном ядре.
* Неполное использование ресурсов: на системах с многоядерными процессорами, использование потоков вместо процессов приводит к неэффективному использованию ресурсов, так как вычисления не распределяются между доступными ядрами.
* Сложность синхронизации: Исполнение в потоках также требует управления состоянием и синхронизацией между потоками, что может усложнить реализацию и увеличить вероятность ошибок.

В связи с вышеизложенным, использование библиотеки multiprocessing является более оптимальным выбором для решения задач численного интегрирования, позволяя эффективно использовать все доступные ядра процессора и предоставляя возможность обойти ограничения GIL.

## Решение

График функции:

Изображение выглядит как линия, диаграмма, текст, Параллельный

Автоматически созданное описание

Аналитическое значение интеграла на итервале [5, 10] = -23/200 + log(2) ≈ 0,578147  
Результат работы программы (ПРИЛОЖЕНИЕ А):  
При количестве интервалов n = 10^8:

**Результат**: 0.5781471643349402

**Идеальное время (с)**: [47.794997215270996, 23.897498607635498, 15.931665738423666, 11.948749303817749, 9.5589994430542, 7.965832869211833, 6.827856745038714, 5.9743746519088745, 5.310555246141222, 4.7794997215271, 4.344999746842817, 3.9829164346059165]

**Реальное время (с)**: [47.794997215270996, 25.19494366645813, 20.08841109275818, 17.835789918899536, 16.939514875411987, 16.920430898666382, 16.19223427772522, 16.018876552581787, 16.776580333709717, 16.626132488250732, 16.87700128555298, 16.95657968521118]

Изображение выглядит как текст, линия, диаграмма, График

Автоматически созданное описание

При количестве интервалов n = 10^7:

**Результат**: 0.5781470183099268

**Идеальное время**: [5.580317497253418, 2.790158748626709, 1.860105832417806, 1.3950793743133545, 1.1160634994506835, 0.930052916208903, 0.7971882138933454, 0.6975396871566772, 0.620035277472602, 0.5580317497253418, 0.5073015906594016, 0.4650264581044515]

**Реальное время**: [5.580317497253418, 3.0616888999938965, 2.4927682876586914, 2.161433696746826, 2.1681013107299805, 2.1345527172088623, 2.207980155944824, 2.320739984512329, 2.4953813552856445, 3.050311803817749, 3.466960906982422, 3.6373913288116455]

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

# Выводы

Результаты, полученные в ходе эксперимента, подтверждают, что метод параллельного интегрирования, реализованный с помощью библиотеки multiprocessing, эффективно снижает время выполнения задачи путем распределения нагрузки между процессами.

Численный интеграл был посчитан правильно, с минимальной погрешностью, так как количество интервалов очень большое.

Тем не менее, наблюдается, что при использовании более 4–5 процессов на ноутбуке с 4 физическими ядрами и 8 потоками время выполнения стабилизируется. Это подразумевает, что дальнейшее увеличение количества процессов не приводит к значительному увеличению производительности, что может быть связано с архитектурными ограничениями системы.

Таким образом, проведенное исследование подчеркивает важность оптимизации численных алгоритмов с учётом реальных аппаратных возможностей для достижения наилучших результатов.

# ПРИЛОЖЕНИЕ А

import matplotlib.pyplot as plt  
import time  
import multiprocessing  
  
  
# Определяем функцию  
def f(x):  
 return (1 / x) - (1 / x \*\* 2) - (1 / x \*\* 3)  
  
  
# Мы должны интегрировать f(x) от a до b  
def integrate\_f(a, b, n):  
 dx = (b - a) / n  
 integral = 0.0  
 for i in range(n):  
 x = a + i \* dx  
 integral += f(x)  
 integral \*= dx  
 return integral  
  
  
# Функция для работы в отдельном процессе  
def process\_integrate\_f(args):  
 a, b, n = args  
 return integrate\_f(a, b, n)  
  
  
# Многопроцессорная интеграция  
def parallel\_integrate\_f(a, b, n, num\_processes):  
 dx = (b - a) / n  
 chunk\_size = n // num\_processes  
 args = []  
 for i in range(num\_processes):  
 start\_a = a + i \* chunk\_size \* dx # Начало интервала для текущего процесса  
 end\_b = a + (i + 1) \* chunk\_size \* dx # Конец интервала для текущего процесса  
 args.append((start\_a, end\_b, chunk\_size))  
  
 with multiprocessing.Pool(processes=num\_processes) as pool:  
 results = pool.map(process\_integrate\_f, args)  
  
 return sum(results)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Основные параметры  
 a = 5  
 b = 10  
 n = 100000000  
 max\_processes = 12 # Максимальное количество процессов  
  
 # Сбор данных о времени выполнения  
 time\_ideal = []  
 time\_real = []  
  
 for num\_processes in range(1, max\_processes + 1):  
 start = time.time()  
 result = parallel\_integrate\_f(a, b, n, num\_processes)  
 end = time.time()  
  
 time\_real.append(end - start)  
 if num\_processes == 1:  
 time\_ideal.append(end - start) # Сохраняем время для одного процесса  
 else:  
 time\_ideal.append(time\_real[0] / num\_processes) # t/num\_processes, где t - время для 1 процесса  
  
 print("Результат:", result)  
 print("Идеальное время:", time\_ideal)  
 print("Реальное время:", time\_real)  
  
 # Построение графиков  
 plt.plot(range(1, max\_processes + 1), time\_ideal, label='Идеальная скорость', linestyle='--')  
 plt.plot(range(1, max\_processes + 1), time\_real, label='Реальная скорость', marker='o')  
 plt.xlabel('Количество процессов')  
 plt.ylabel('Время выполнения (с)')  
 plt.title('Сравнение скорости интеграции с использованием процессов')  
 plt.legend()  
 plt.grid()  
 plt.show()