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»

Рубежный контроль №1

По предмету: «Анализ алгоритмов»

Алгоритмы параллельного и асинхронного умножения матриц

Студент: Гасанзаде М.А.,
Группа: ИУ7-56Б

Москва, 2019 г.

Оглавление

Введение

Параллельные вычисления - способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно).

Асинхронные вычисления – это те вычисления, которые возникают независимо от основного потока выполнения программы. Асинхронные вычисления — это действия обработки ввода/вывода, выполненные в неблокирующем режиме, позволяющие продолжить обработку других задач, не ожидая завершения передачи.

Целью данной лабораторной работы является изучение данных алгоритмов и оценка этих алгоритмов по затратам времени и памяти.

1. Аналитическая часть

В данном разделе будут представлены описания алгоритмов умножения матриц.

1.1 Описание алгоритмов

Произведение двух матриц – это матрица с элементами, равными сумме произведений соответствующих элементов строк первой матрицы-сомножителя и элементов столбцов второй матрицы-сомножителя.

$$C_{m\times k} = A_{m\times n}\cdot B_{n\times k} \Leftrightarrow \left\|c_{it}\right\|_{m\times k} = \left\|\sum_{j=1}^{n} a_{ij}b_{jt}\right\|_{m\times k} \Leftrightarrow$$

$$\Leftrightarrow \left\|c_{it}\right\|_{m\times k} = \left\|a_{i1}b_{1t} + a_{i2}b_{2t} + \ldots + a_{in}b_{nt}\right\|_{m\times k}$$

Где, **m** – число строк первой матрицы-сомножителя и матрицы-произведения, **n** – число столбцов первой матрицы-сомножителя и число строк второй матрицы-сомножителя, **k** – число столбцов второй матрицы-сомножителя и матрицы-произведения, **m*n** – размерность первой матрицы-сомножителя, **n*k** – размерность второй матрицы-сомножителя, **m*k** – размерность матрицы-произведения, $\mathbf{a_{ij}}$ – элемент матрицы **A**, лежащий на пересечении **i**-ой строки и **j**-ого столбца матрицы, $\mathbf{b_{jt}}$ – элемент матрицы **B**, лежащий на пересечении **j**-ой строки и **t**-ого столбца матрицы, $\mathbf{c_{it}}$ – элемент матрицы **C**, лежащий на пересечении **i**-ой строки и **t**-ого столбца матрицы.

1.2 Вывод:

В данном разделе была рассмотрена стандартная формула умножения матриц.

2 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и примеры тестирования.

2.1 Требования к программному обеспечению

На вход подаются матрицы строки, на выходе необходимо получить матрицу и 3 результата, выдаваемых матричными реализациями обоих алгоритмов и рекурсивной реализации алгоритма Дамерау-Левенштейна. Требуется замерить время работы каждой реализации.

2.2 Средства реализации

В качестве языка программирования был выбран Python в связи с его широким функционалом и огромнейшим набором библиотек, а также из-за привычного для меня синтаксиса. Среда разработки - стандартная IDLE Python. Время работы программы замеряется с помощью библиотеки time[1] и logging[2]. На *листинге 1* представлен код использования библиотеки logging.

```
import logging

logging.basicConfig(format="[%(thread)-5d]%(asctime)s: %(message)s")
logger = logging.getLogger('async')
logger.setLevel(logging.INFO)


logger.info("Completed in {} seconds".format(time.time() -
start_time))
```

Листинг 1. Использование logging

2.3 Листинг кода

Листинг кода был представлен на *листингах 2, 3, 4, 5.*

```
def random_matrix(n, m):
    return [[randint(0, 100) for i in range(m)] for j in range(n)]


matrix_a = random_matrix(3, 3)
matrix_b = random_matrix(3, 3)
```

Листинг 2. Генерация рандомной матрицы.

```
def multi(A, B):
    if len(B) != len(A[0]):
        print("Different dimension of the matrics")
        return

    n = len(A)
    m = len(A[0])
    t = len(B[0])

    answer = [[0 for i in range(t)] for j in range(n)]
    for i in range(n):
        for j in range(m):
            for k in range(t):
                answer[i][k] += A[i][j] * B[j][k]
    return answer

async def multi_async(A, B):
    tmp = tuple(zip(*B))

    results = await asyncio.gather(*[get_new_elem(row, tmp) for row in
A])
    return results
```

Листинг 3. Умножение матрицы при асинхронной реализации.

```
def multi_matrices(first, second, third):
    res = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    for i in range(len(first)):
        for j in range(len(first)):
            res[i][j] = first[i][j] + second[i][j] + third[i][j]

    return res
```

Листинг 4. Функция умножения матрицы при параллельной реализации.

```
class Thread1(threading.Thread):
    @profile
    def __init__(self, id, name):
        threading.Thread.__init__(self)
        self.id = id
        self.name = name
    @profile
    def run(self):
        print ("Starting " + self.name + "\n")

        vector_3x1a = [matrix_a[i][0] for i in range(len(matrix_a))]
        vector_1x3b = [matrix_b[0][j] for j in range(len(matrix_b[0]))]

        matrix_c_aux1[0][0] = vector_3x1a[0] * vector_1x3b[0]
        matrix_c_aux1[0][1] = vector_3x1a[0] * vector_1x3b[1]
        matrix_c_aux1[0][2] = vector_3x1a[0] * vector_1x3b[2]

        matrix_c_aux1[1][0] = vector_3x1a[1] * vector_1x3b[0]
        matrix_c_aux1[1][1] = vector_3x1a[1] * vector_1x3b[1]
        matrix_c_aux1[1][2] = vector_3x1a[1] * vector_1x3b[2]

        matrix_c_aux1[2][0] = vector_3x1a[2] * vector_1x3b[0]
        matrix_c_aux1[2][1] = vector_3x1a[2] * vector_1x3b[1]
        matrix_c_aux1[2][2] = vector_3x1a[2] * vector_1x3b[2]

        print ("End " + self.name + "\n")
```

Листинг 5. Реализация потока в параллельной реализации

2.4 Описание тестирования

Так как для максимального быстродействия в параллельном умножении были созданы 3 потока для каждой матрицы, мы будем рассматривать рандомные матрицы 3x3. Таблицы с результатами были представлены на *рис.* 1, 2.

Рисунок 1 — Данные тестов при генерации значений матрицы от 0 до 100

Async	**Parallel**
0.0019991397857666016	**0.028941869735717773**
0.001999378204345703	**0.03298020362854004**
0.0009829998016357422	**0.07339262962341309**
0.000982522964477539	**0.07525110244750977**
0.0009829998016357422	**0.0809319019317627**

Рисунок 2 – Данные тестов при генерации значений матрицы от 0 до 1000

Async **Время в секундах**	**Parallel** **Время в секундах**
0.0019989013671875	**0.025501251220703125**
0.0009992122650146484	**0.03381609916687012**
0.00099945068359375	**0.0721273422241211**
0.0019986629486083984	**0.07727336883544922**
0.001979351043701172	**0.07005810737609863**
0.0009834766387939453	**0.06937527656555176**
0.0019989013671875	**0.06998252868652344**

Все тесты пройдены успешно.

2.5 Вывод

В данном разделе мы рассмотрели листинг кода, а также убедились в безошибочной работе программы.

3. Экспериментальная часть.

В данном разделе будут рассмотрены примеры работ программы.

Память была замерена с помощью библиотеки memory_profiler[4] в 32х битном Python IDLE.

3.1 Пример работы программы

На рисунках 3, 4 приведены изображения внешнего вида интерфейса программы во время его работы

```
Starting Thread 1
Starting Thread 2
Starting Thread 3



End Thread 1
End Thread 2
End Thread 3



[3188 ]2019-12-13 03:46:21,346: Completed in 0.026981353759765625 seconds
Execution Time ---> 30
[[6376, 12199, 6208], [9926, 15155, 10706], [4991, 14301, 4397]]
```

Рисунок 4, пример работы программы параллельного умножения.

```
================== RESTART: C:\Users\pbhac\Desktop\multi.py ==============
[5252 ]2019-12-13 03:51:29,804: Completed in 0.002001047134399414 seconds
Execution Time ---> 37
[[8328, 7664, 5700], [7702, 6997, 5317], [8367, 8075, 7425]]
>>>
```

Рисунок 5, пример работы программы асинхронного умножения.

3.2 Сравнительный анализ алгоритмов

На примере малоразмерных матриц было показано, что алгоритм параллельного умножении матриц, сильно проигрывает по времени и затрачиваемой памяти, асинхронному алгоритму.

3.3 Сравнительный анализ на материале экспериментальных данных

Алгоритмы были протестированы по скорости работы и используемой памяти. На рисунках 1 и 2.

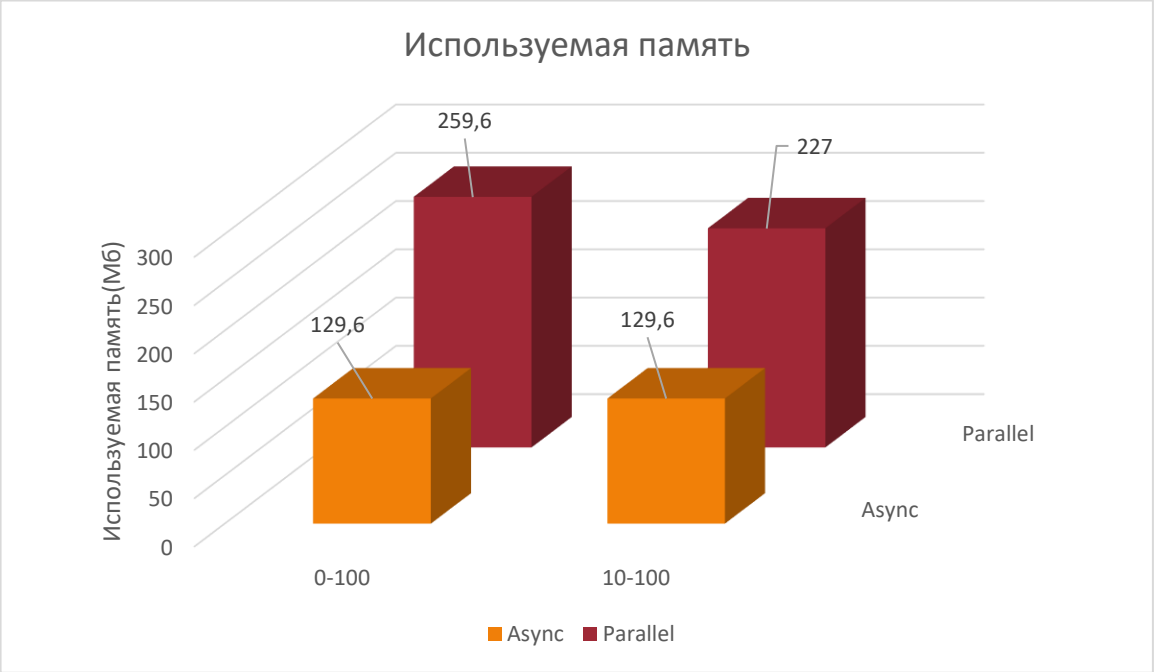

Рисунок 1- Использование памяти.

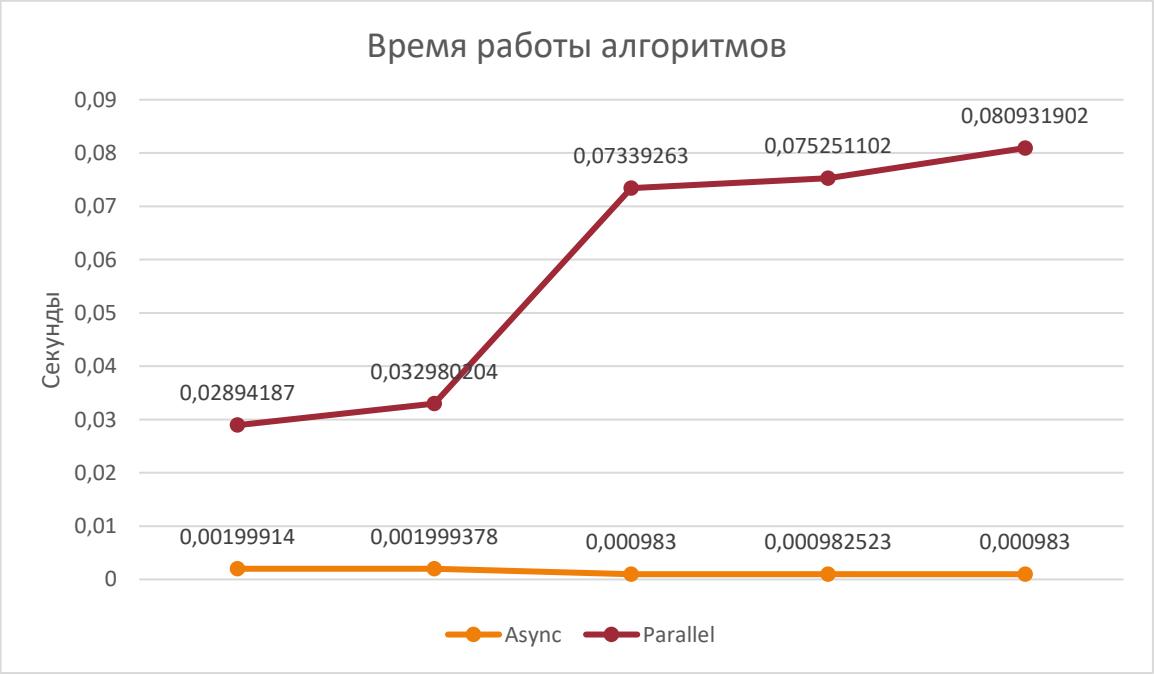

Рисунок 2 – Замер времени

3.4 Вывод

В данном разделе был представлен эксперимент по замеру времени и памяти выполнения каждого алгоритма. По итогам замеров алгоритм параллельного умножения матрицы показал себя хуже чем асинхронный во всех замерах.

Заключение

В данной работе были реализованы и протестированы два современных методов умножения матриц. Проведён сравнительный анализ обоих реализаций.

Асинхронная реализация показала себя намного лучше из-за особенностей работы Python3, и малоразмерности матриц.[3]

Параллельная реализация эффективна при применении к большим матрицам.

Список литературы

1. time — Time access and conversions // Python URL: https://docs.python.org/3/library/time.html
2. logging – https://docs.python.org/3/library/logging.html
3. Использование асинхронных вычислений: http://ikit.sfu-kras.ru/files/ikit/03_Asinhronnye_spiski.pdf
4. Замер памяти в Python3: URL: https://pypi.org/project/memory-profiler/
5. Параллельные алгоритмы умножения матриц: Shortulr.wizardmh