Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики
Кафедра програмування

Звіт
до лабораторної роботи №2
“Розпаралелення множення матриць”

Підготував:
студент групи ПМІ-31
Процьків Назарій

Львів 2023

Завдання

Напишіть програми обчислення множення двох матриць (послідовний та паралельний алгоритми). Порахуйте час роботи кожної з програм, обчисліть прискорення та ефективність роботи паралельного алгоритму.

В матрицях розмірності (n,m) робіть змінними, щоб легко змінювати величину матриці.

Кількість потоків k - також змінна величина. Програма повинна показувати час при послідовному способі виконання програми, а також при розпаралеленні на k потоків.

Зверніть увагу на випадки, коли розмірність матриці не кратна кількості потоків!!!

Хід роботи

Я виконав це завдання двома мовами програмування: C# і Python.

C#

До класу Матриці з першої лабораторної роботи дописав три методи: SimpleMultiplication, ThreadsMultiplication і ThreadMultParallel.

```
public class Matrix
{
    22 references
    public uint Height { get; set; }
    31 references
    public uint Width { get; set; }
    19 references
    public int[,] Mtrx { get; set; }

    8 references
    public Matrix(uint height, uint width)...
    0 references
    public Matrix(Matrix other)...
    2 references
    public void Generate()...
    0 references
    public Matrix SimpleAddition(Matrix other)...
    0 references
    public Matrix ThreadsAddition(Matrix other, uint amount)...
    2 references
    public Matrix SimpleMultiplication(Matrix other)...
    0 references
    public Matrix ThreadsMultiplication(Matrix other, uint amount)...
    2 references
    public Matrix ThreadMultParallel(Matrix other, uint amount)...
    public override string ToString()...
}
```

SimpleMultiplication – функція для множення матриць послідовно.

ThreadsMultiplication – функція написана вручну для множення матриць паралельно.

ThreadMultParallel – функція написана за допомогою Parallel.For для множення матриць паралельно.

Main(string[] args):

Створення малих матриць для перевірки правильності множення, перевірив вручну. Результат справа.

```
public static void Main(string[] args)
{
    uint rows = 2, columns = 2, threads = 2;

    var matrix1 = new Matrix(rows, columns);
    matrix1.Generate();
    var matrix2 = new Matrix(rows, columns);
    matrix2.Generate();

    Console.WriteLine($"Matrix 1: \n{matrix1}");
    Console.WriteLine($"Matrix 2: \n{matrix2}");

    var resultMatrix = new Matrix(rows, columns);
    resultMatrix = matrix1.SimpleMultiplication(matrix2);
    Console.WriteLine($"Simple multiplication: \n{resultMatrix}");

    resultMatrix = matrix1.ThreadMultParallel(matrix2, threads);
    Console.WriteLine($"Threads multiplication: \n{resultMatrix}");
```

```
D:\Study\Паралельні та розподілені обчислення\MatrixOperationsC#\M
Matrix 1:
961 18
457 148

Matrix 2:
550 325
812 691

Simple multiplication:
543166 324763
371526 250793

Threads multiplication:
543166 324763
371526 250793
```

Створення великих матриць для визначення прискорення та ефективності:

```
rows = 1000;
columns = 1000;
Console.WriteLine($"Dimensions:\nrows: {rows}\ncolumns: {columns}");

matrix1 = new Matrix(rows, columns);
matrix2 = new Matrix(rows, columns);

var clock = System.Diagnostics.Stopwatch.StartNew();
resultMatrix = matrix1.SimpleMultiplication(matrix2);
clock.Stop();

var singleThreadTime = clock.Elapsed;
Console.WriteLine($"Single thread: {singleThreadTime}");

var threadsNum = new uint[] { 2, 3, 4, 5, 6, 7, 8, 9, 10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };

foreach (var i in threadsNum)
{
    clock = System.Diagnostics.Stopwatch.StartNew();
    resultMatrix = matrix1.ThreadMultParallel(matrix2, i);
    clock.Stop();

    var acceleration = Math.Round(singleThreadTime / clock.Elapsed, 4);
    var efficiency = Math.Round(acceleration / i, 5);
    Console.WriteLine($"Threads: {i}\tTime: {clock.Elapsed}\tAcceleration: {acceleration}\tEfficiency: {efficiency}");
}
```

Результат з використанням розпаралелення вручну:

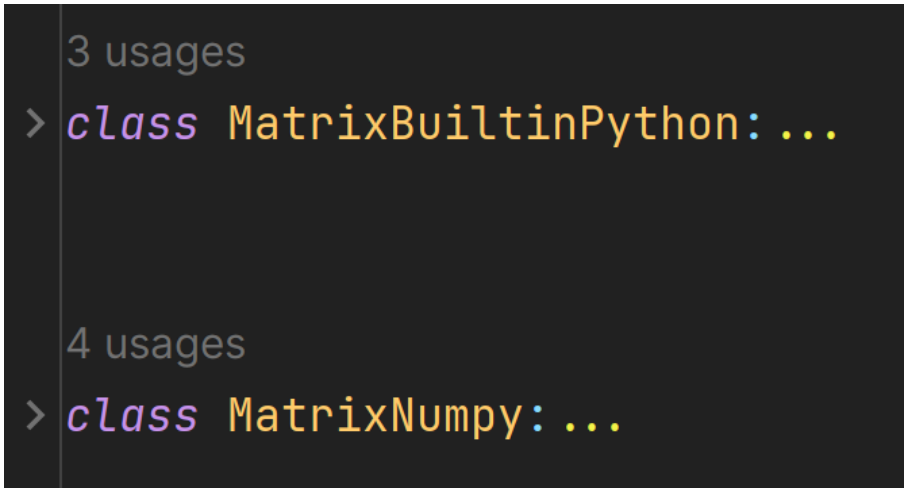
```
Dimensions:
rows: 1000
columns: 1000
Single thread: 00:00:09.1244682
Threads: 2      Time: 00:00:05.2018319  Acceleration: 1,7541    Efficiency: 0,87705
Threads: 3      Time: 00:00:03.8793051  Acceleration: 2,3521    Efficiency: 0,78403
Threads: 4      Time: 00:00:03.3692117  Acceleration: 2,7082    Efficiency: 0,67705
Threads: 5      Time: 00:00:03.5415435  Acceleration: 2,5764    Efficiency: 0,51528
Threads: 6      Time: 00:00:03.5484433  Acceleration: 2,5714    Efficiency: 0,42857
Threads: 7      Time: 00:00:03.6808167  Acceleration: 2,4789    Efficiency: 0,35413
Threads: 8      Time: 00:00:03.6630437  Acceleration: 2,491     Efficiency: 0,31138
Threads: 9      Time: 00:00:04.1298144  Acceleration: 2,2094    Efficiency: 0,24549
Threads: 10     Time: 00:00:04.3812275  Acceleration: 2,0826    Efficiency: 0,20826
Threads: 100    Time: 00:00:05.8530169  Acceleration: 1,5589    Efficiency: 0,01559
Threads: 200    Time: 00:00:07.4827112  Acceleration: 1,2194    Efficiency: 0,0061
Threads: 300    Time: 00:00:08.9309964  Acceleration: 1,0217    Efficiency: 0,00341
Threads: 400    Time: 00:00:10.9437741  Acceleration: 0,8338    Efficiency: 0,00208
Threads: 500    Time: 00:00:11.8604087  Acceleration: 0,7693    Efficiency: 0,00154
Threads: 600    Time: 00:00:14.4107602  Acceleration: 0,6332    Efficiency: 0,00106
Threads: 700    Time: 00:00:16.3780618  Acceleration: 0,5571    Efficiency: 0,0008
Threads: 800    Time: 00:00:17.1757871  Acceleration: 0,5312    Efficiency: 0,00066
Threads: 900    Time: 00:00:19.1679559  Acceleration: 0,476     Efficiency: 0,00053
Threads: 1000   Time: 00:00:20.0851098  Acceleration: 0,4543    Efficiency: 0,00045
```

Результат з використанням розпаралелення Parallel.For:

```
Dimensions:
rows: 1000
columns: 1000
Single thread: 00:00:09.0122525
Threads: 2      Time: 00:00:05.1002040  Acceleration: 1,767     Efficiency: 0,8835
Threads: 3      Time: 00:00:03.7336736  Acceleration: 2,4138    Efficiency: 0,8046
Threads: 4      Time: 00:00:03.4758094  Acceleration: 2,5929    Efficiency: 0,64823
Threads: 5      Time: 00:00:03.4542720  Acceleration: 2,609     Efficiency: 0,5218
Threads: 6      Time: 00:00:03.4534240  Acceleration: 2,6097    Efficiency: 0,43495
Threads: 7      Time: 00:00:03.7351604  Acceleration: 2,4128    Efficiency: 0,34469
Threads: 8      Time: 00:00:03.7738970  Acceleration: 2,388     Efficiency: 0,2985
Threads: 9      Time: 00:00:03.9722196  Acceleration: 2,2688    Efficiency: 0,25209
Threads: 10     Time: 00:00:04.1612016  Acceleration: 2,1658    Efficiency: 0,21658
Threads: 100    Time: 00:00:04.2555283  Acceleration: 2,1178    Efficiency: 0,02118
Threads: 200    Time: 00:00:04.2114424  Acceleration: 2,1399    Efficiency: 0,0107
Threads: 300    Time: 00:00:04.2245908  Acceleration: 2,1333    Efficiency: 0,00711
Threads: 400    Time: 00:00:04.5661327  Acceleration: 1,9737    Efficiency: 0,00493
Threads: 500    Time: 00:00:04.2759172  Acceleration: 2,1077    Efficiency: 0,00422
Threads: 600    Time: 00:00:04.6539803  Acceleration: 1,9365    Efficiency: 0,00323
Threads: 700    Time: 00:00:04.4387278  Acceleration: 2,0304    Efficiency: 0,0029
Threads: 800    Time: 00:00:04.2835405  Acceleration: 2,1039    Efficiency: 0,00263
Threads: 900    Time: 00:00:04.3449446  Acceleration: 2,0742    Efficiency: 0,0023
Threads: 1000   Time: 00:00:04.2178192  Acceleration: 2,1367    Efficiency: 0,00214
```

Python

Мовою Python в мене є два класи: MatrixBuiltinPython та MatrixNumpy



Різниця між ними полягає у тому, що поле, яке відповідає за матрицю в класі MatrixBuiltinPython використовує вбудований тип ***list*** у Python, а в класі MatrixNumpy тип numpy.ndarray.

Вбудований Пайтон (Експеримент №1)

Це клас MatrixBuiltinPython:

```
class MatrixBuiltinPython:
    def __init__(self, dimensions):...

    def __str__(self):...

    2 usages
    def generate(self):...

    1 usage
    def simple_python_multiplication(self, other):...

    1 usage
    def parallel_matrix_multiplication(self, matrix2, num_threads):...

    1 usage
    @staticmethod
    def multiply_row(args):...
```

В ньому ініціалізація об’єкта проходить зі створенням розмірів і самої матриці за допомогою списка списків:

```
class MatrixBuiltinPython:
    def __init__(self, dimensions):
        self.width, self.height = dimensions
        self.matrix = [[0] * self.width for _ in range(self.height)]
```

Заповнення матриці випадковими числами проходить реалізовано методом generate(), він не потребує багато уваги.

Послідовне множення

Метод simple_python_multiplication:

```
def simple_python_multiplication(self, other):
    if self.width ≠ other.height:
        raise ValueError("Dimensions do not match")

    result = MatrixBuiltinPython((self.height, other.width))

    for i in range(self.height):
        for j in range(other.width):
            res = 0
            for k in range(self.width):
                res += self.matrix[i][k] * other.matrix[k][j]
            result.matrix[i][j] = res

    return result
```

В цьому методі проходить множення двох матриць послідовно. Спершу, як завжди, перевірка на валідність розмірів, далі створення результуючої матриці і цикл ***for*** для запису в кожен її елемент обрахованого числа.

Паралельне множення

Перший експеримент полягав у тому, щоб розпаралелити множення матриці використовуючи вбудований пайтон. Для початку, потрібно пояснити допоміжну функцію multiply_row_python, яка буде дуже потрібна при розпаралеленні множення:

```
@staticmethod
def multiply_row(args):
    row_index, matrix1, matrix2, result = args
    row = matrix1.matrix[row_index]
    num_cols = len(matrix2.matrix[0])
    result_row = [0] * num_cols

    for i in range(num_cols):
        for j in range(len(row)):
            result_row[i] += row[j] * matrix2.matrix[j][i]

    result[row_index] = result_row
```

Вона приймає аргумент ***args***, розділяє його на:

- номер рядка – row_index
- першу матрицю – matrix1
- другу матрицю – matrix2
- результуючу матрицю, в яку буде записаний результат – result

Цикл ***for*** множить рядок першої матриці на колонку другої матриці і присвоює результат в результуючу матрицю (по рядках).

Тепер про головну функцію розпаралелення parallel_matrix_multiplication:

```
def parallel_matrix_multiplication(self, matrix2, num_threads):
    if len(self.matrix[0]) != len(matrix2.matrix):
        raise ValueError("Dimensions do not match")

    num_rows = len(self.matrix)
    num_cols = len(matrix2.matrix[0])

    result = [[0] * num_cols for _ in range(num_rows)]

    with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as executor:
        executor.map(MatrixBuiltinPython.multiply_row,
                     [(i, self, matrix2, result) for i in range(num_rows)])

    return result
```

Ця функція приймає другу матрицю, як параметр та число потоків. Спершу, як завжди, проходить перевірка валідності розмірів матриць. Далі створення результуючої матриці з потрібними розмірами.

ThreadPoolExecutor – клас похідний від ***Executor***. Pool перекладається як басейн. Тобто це середовище існування/виконання потоків, які всі разом будуть виконані Executor-ом. Цей клас треба імпортувати з вбудованого пакету concurrent.futures. Робота з цим пакетом під капотом автоматично відключає ***Global Interpreter Lock*** – замок, який стоїть на інтерпретаторі Python для того, щоб не було багатопоточності.

Функція map() зв’язує кожен потік з кількома функціями multiply_row, а в якості аргументів передає елементи з ***list comprehension***, який складається з об’єктів типу ***tuple***. Їхня кількість буде рівномірно розподілена між кожним потоком. Саму функцію multiply_row було описано вище, вона з поточного класу.

Таким чином завдання складається з множення рядка першої матриці на колонку другої. Ці завдання ми розподілили з допомогою ***ThreadPoolExecutor*** на стільки потоків скільки потрібно.

Результат

Створив малі матриці, заповнив випадковими числами та вивів на екран. Перевірив результат множення вручну.

```python
if __name__ == "__main__":
    dimensions, threads = (3, 3), 2

    m1 = MatrixBuiltinPython(dimensions)
    m2 = MatrixBuiltinPython(dimensions)

    m1.generate()
    m2.generate()

    print(f"Matrix 1:\n{m1}")
    print(f"Matrix 2:\n{m2}")

    result = m1.simple_python_multiplication(m2)
    print(f"Simple Python multiplication:\n{result}")

    result = m1.parallel_matrix_multiplication(m2, threads)
    print(f"Parallel Python multiplication:")
    for i in result:
        print(i)
```

Output 1:

```
Matrix 1:
553 976 442
73 413 264
161 695 638

Matrix 2:
33 565 473
146 427 386
829 499 656
```

```
Simple Python multiplication:
527163 949755 928257
281563 349332 367131
635685 706092 762951

Parallel Python multiplication:
[527163, 949755, 928257]
[281563, 349332, 367131]
[635685, 706092, 762951]
```

Як бачимо, множення послідовно і паралельно виводить однаковий результат.

Приклад більших матриць.

Пайтон достатньо повільна мова, тому йому вистачило матриць з розмірами 200 на 200, щоб трохи повисіти, або це я такий програміст.

```
if __name__ == "__main__":
    dimensions, threads = (200, 200), 2

    m1 = MatrixBuiltinPython(dimensions)
    m2 = MatrixBuiltinPython(dimensions)

    m1.generate()
    m2.generate()

    start_ = time.time()
    result = m1.simple_python_multiplication(m2)
    end_ = time.time()

    print(f"Dimensions: {dimensions}")
    single_thread_time = end_ - start_
    print(f"Single thread time: {single_thread_time:.4f}s")

    threads = [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
    for i in threads:
        current_time = []
        for _ in range(3):
            start_ = time.time()
            m_threads = m1.parallel_matrix_multiplication(m2, i)
            end_ = time.time()

            current_time.append(end_ - start_)

        current_time = sum(current_time) / 3
        acceleration = single_thread_time / current_time
        efficiency = acceleration / i

        print(f"Threads: {i} \t"
              f"Time: {current_time:.4f}s \t"
              f"Acceleration: {acceleration:.4f} \t"
              f"Efficiency: {efficiency:.4f}")
```

Для кожної кількості потоків я проводжу експеримент тричі і беру середнє значення.

Результат:

```
Dimensions: (200, 200)
Single thread time: 0.8758s
Threads: 2  Time: 0.7700s   Acceleration: 1.1374    Efficiency: 0.5687
Threads: 3  Time: 0.7504s   Acceleration: 1.1670    Efficiency: 0.3890
Threads: 4  Time: 0.7958s   Acceleration: 1.1005    Efficiency: 0.2751
Threads: 5  Time: 0.8643s   Acceleration: 1.0133    Efficiency: 0.2027
Threads: 6  Time: 0.8734s   Acceleration: 1.0027    Efficiency: 0.1671
Threads: 7  Time: 0.9176s   Acceleration: 0.9544    Efficiency: 0.1363
Threads: 8  Time: 1.0469s   Acceleration: 0.8365    Efficiency: 0.1046
Threads: 9  Time: 0.8628s   Acceleration: 1.0151    Efficiency: 0.1128
Threads: 10     Time: 0.8424s   Acceleration: 1.0396    Efficiency: 0.1040
```

Показники прискорення та ефективності не надто вражають.

Але погляньмо, що буде далі.

NumPy (Експеримент №2)

Це клас MatrixNumpy:

```python
class MatrixNumpy:
    def __init__(self, dimensions):...

    def __str__(self):...

    def generate(self):...

    def simple_python_multiplication(self, other):...

    def numpy_multiplication(self, other):...

    1 usage
    @staticmethod
    def multiply_row_numpy(args):...

    def threads_numpy_multiplication(self, other, num_threads):...
```

Він відрізняється від попереднього класу тим, що при поле об’єкта матриці це тепер numpy.ndarray.

```
class MatrixNumpy:
    def __init__(self, dimensions):
        self.width, self.height = dimensions
        self.matrix = np.zeros((self.height, self.width), dtype=int)
```

Послідовне множення

Послідовне множення в цьому класі проходить так само.

Функція simple_python_multiplication – ідентична до цієї ж, в попередньому класі.

Але я вирішив дописати щось нове, тому використав NumPy.

Функція numpy_multiplication – множить дві матриці. Код в один рядок.

```
def numpy_multiplication(self, other):
    return np.matmul(self.matrix, other.matrix)
```

За таке пайтону лайк.

Паралельне множення

Функція multiply_row_numpy – так само як і в попередньому класі це допоміжна функція, яка множить рядок першої матриці на колонку другої, але цього разу використана вбудована в NumPy функція dot.

```
@staticmethod
def multiply_row_numpy(args):
    row, main, other, result = args
    result.matrix[row, :] = np.dot(main.matrix[row, :], other.matrix)
```

Функція threads_numpy_multiplication:

```
def threads_numpy_multiplication(self, other, num_threads):
    if len(self.matrix[0]) != len(other.matrix):
        raise ValueError("Dimensions does not match")

    num_rows = len(self.matrix)
    num_cols = len(other.matrix[0])

    result = MatrixNumpy((len(m1.matrix), len(other.matrix[0])))

    with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as executor:
        executor.map(MatrixNumpy.multiply_row_numpy,
                     [(i, self, other, result) for i in range(num_rows)])

    return result
```

Код функції аналогічний до коду цієї самої функції в попередньому класі, але тепер в функцію ***map*** передано функцію, яка написана на NumPy.

Результат

Створення малих матриць, їхнє заповнення випадковими числами та виведення на екран. Перевірив результат множення вручну.

```
if __name__ == "__main__":
    dimensions, threads = (2, 2), 2

    m1 = MatrixNumpy(dimensions)
    m2 = MatrixNumpy(dimensions)

    m1.generate()
    m2.generate()

    print(f"Matrix 1:\n{m1}")
    print(f"Matrix 2:\n{m2}")

    result = m1.numpy_multiplication(m2)
    print(result)

    result = m1.threads_numpy_multiplication(m2, threads)
    print(result)
```

Output:

```
Matrix 1:
297 260
147 522

Matrix 2:
285 589
426 940
```

```
Numpy single thread multiplication:
[[195405 419333]
 [264267 577263]]

Numpy parallel multiplication:
195405 419333
264267 577263
```

Приклад більших матриць.

```python
if __name__ == "__main__":
    dimensions, threads = (250, 250), 2

    m1 = MatrixNumpy(dimensions)
    m2 = MatrixNumpy(dimensions)

    m1.generate()
    m2.generate()

    start_ = time.time()
    result = m1.simple_python_multiplication(m2)
    end_ = time.time()

    print(f"Dimensions: {dimensions}")
    single_thread_time = end_ - start_
    print(f"Single thread time: {single_thread_time:.4f}s")

    threads = [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
    threads = [2, 3, 4, 5, 6, 7, 8, 9, 10]

    for i in threads:
        current_time = []
        for _ in range(3):
            start_ = time.time()
            m_threads = m1.threads_numpy_multiplication(m2, i)
            end_ = time.time()

            current_time.append(end_ - start_)

        current_time = sum(current_time) / 3
        acceleration = single_thread_time / current_time
        efficiency = acceleration / i

        print(f"Threads: {i} \t"
              f"Time: {current_time:.4f}s \t"
              f"Acceleration: {acceleration:.4f} \t"
              f"Efficiency: {efficiency:.4f}")
```

Output:

```
Dimensions: (250, 250)
Single thread time: 4.4226s
Threads: 2  Time: 0.0084s   Acceleration: 524.7170  Efficiency: 262.3585
Threads: 3  Time: 0.0060s   Acceleration: 735.2763  Efficiency: 245.0921
Threads: 4  Time: 0.0060s   Acceleration: 737.3026  Efficiency: 184.3256
Threads: 5  Time: 0.0068s   Acceleration: 649.2223  Efficiency: 129.8445
Threads: 6  Time: 0.0075s   Acceleration: 591.9581  Efficiency: 98.6597
Threads: 7  Time: 0.0078s   Acceleration: 563.4411  Efficiency: 80.4916
Threads: 8  Time: 0.0091s   Acceleration: 487.7118  Efficiency: 60.9640
Threads: 9  Time: 0.0082s   Acceleration: 540.5687  Efficiency: 60.0632
Threads: 10     Time: 0.0083s   Acceleration: 531.6854  Efficiency: 53.1685
```

Показники прискорення та ефективності тепер є “набагато кращими”. Те, що ці показники в принципі є не нульові означає, що багатопоточність працює. Прискорення та ефективність тут нереально великі. Зважаючи на те, що ефективність не може бути такою великою, я можу зробити ***висновок***, що порівняння цих двох функцій є ***некоректним***, тому що одна з них (послідовне множення матриць) написана на чистому пайтоні, а інша (функція паралельного обрахунку) складається з двох функцій, одна з яких написана на нампаї, а інша вже на чистому пайтоні. Чистий пайтон не може змагатись з NumPy, бо він в основному написаний на C++. Але якщо порівняти написане паралельне множення з функцією numpy_multiplication, яка була написана в один рядок за допомогою NumPy, то показники будуть більш схожі до реальних:

Output 1:

```
Dimensions: (1000, 1000)
Single thread time: 1.5887s
Threads: 2  Time: 0.4101s   Acceleration: 3.8743    Efficiency: 1.9371
Threads: 3  Time: 0.2953s   Acceleration: 5.3808    Efficiency: 1.7936
Threads: 4  Time: 0.2435s   Acceleration: 6.5235    Efficiency: 1.6309
Threads: 5  Time: 0.2418s   Acceleration: 6.5709    Efficiency: 1.3142
Threads: 6  Time: 0.2578s   Acceleration: 6.1614    Efficiency: 1.0269
Threads: 7  Time: 0.2730s   Acceleration: 5.8196    Efficiency: 0.8314
Threads: 8  Time: 0.2949s   Acceleration: 5.3879    Efficiency: 0.6735
Threads: 9  Time: 0.2894s   Acceleration: 5.4889    Efficiency: 0.6099
Threads: 10     Time: 0.2916s   Acceleration: 5.4475    Efficiency: 0.5447
```

Output 2:

```
Dimensions: (1000, 1000)
Single thread time: 1.5421s
Threads: 2  Time: 0.4200s   Acceleration: 3.6716    Efficiency: 1.8358
Threads: 3  Time: 0.2937s   Acceleration: 5.2512    Efficiency: 1.7504
Threads: 4  Time: 0.2363s   Acceleration: 6.5251    Efficiency: 1.6313
Threads: 5  Time: 0.2537s   Acceleration: 6.0792    Efficiency: 1.2158
Threads: 6  Time: 0.2650s   Acceleration: 5.8192    Efficiency: 0.9699
Threads: 7  Time: 0.2733s   Acceleration: 5.6418    Efficiency: 0.8060
Threads: 8  Time: 0.2930s   Acceleration: 5.2631    Efficiency: 0.6579
Threads: 9  Time: 0.2907s   Acceleration: 5.3054    Efficiency: 0.5895
Threads: 10     Time: 0.3107s   Acceleration: 4.9638    Efficiency: 0.4964
```

Але все одно прискорення досягає значення 3.8-4, а ефективність 1.9-2. В цьому випадку також не можна вважати коректним це порівняння, бо функція послідовного множення використовує чистий нампай, а функція паралельного множення використовує наполовину нампай і наполовину пайтон, тому вони також є не в рівних умовах.

Однак моєю ціллю було дослідити паралелізм в пайтоні.

Після тонни досліджень і виконаної роботи можу сказати, що багатопоточність в пайтоні можлива, але мені вона забрала багато часу. Як-не-як це мова програмування глобального рівня, велика потужна машина, яка використовується у великих проектах для data science, тому без багатопоточності вона не змогла б існувати.