Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Курсовая работа

Дисциплина: Операционные системы

Тема: Работа с потоками в ОС Linux. Перемножение матриц.

Выполнил студент гр. 3530901/90203 _____ Т.Ю. Янтимиров

(подпись)

Преподаватель _____ З.В Куляшова

(подпись)

“1” октября 2021 г.

Санкт-Петербург

2021

Содержание

Введение

Для улучшения производительности приложения часто подходят к такому подходу, как многопоточность. Суть этого подхода в том, чтобы “заставить” наше приложение работать асинхронно, то есть, одновременно выполнять несколько задач. Рассмотрим этот подход в задаче перемножения матриц.

Теоретическая информация

Задача перемножения матриц – весьма затратна по времени. Асимптотическая сложность алгоритма перемножения матриц – $O(n^3)$. Сам алгоритм:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

m – количество строк матрицы A и количество столбцов матрицы B

a_{ik} – элементы первой матрицы

b_{kj} – элементы второй матрицы

c_{ij} – элементы результирующей матрицы

Если с небольшими матрицами размера 3x3, 10x10 машина работает без каких-либо проблем, то с огромными матрицами 1000x1000, 2000x2000 у ЭВМ возникают трудности. Для ускорения работы алгоритма – реализуем многопоточное перемножение.

Реализация многопоточности

Разработаем многопоточное приложение на языке Kotlin. Для упрощения приложения, исследуем только умножение квадратных матриц. Таким образом, нам не надо будет постоянно заботиться о правильных размерах матриц, при этом, кардинальной разницы в перфомансе не будет.

Сама многопоточность приложения будет заключаться в том, что мы будем умножать каждую строку в матрице в отдельном потоке.

Исследование умножения матрицы в 1 и 10 потоках

Для начала, оценим сколько времени тратится для умножения в простом однопоточном приложении:

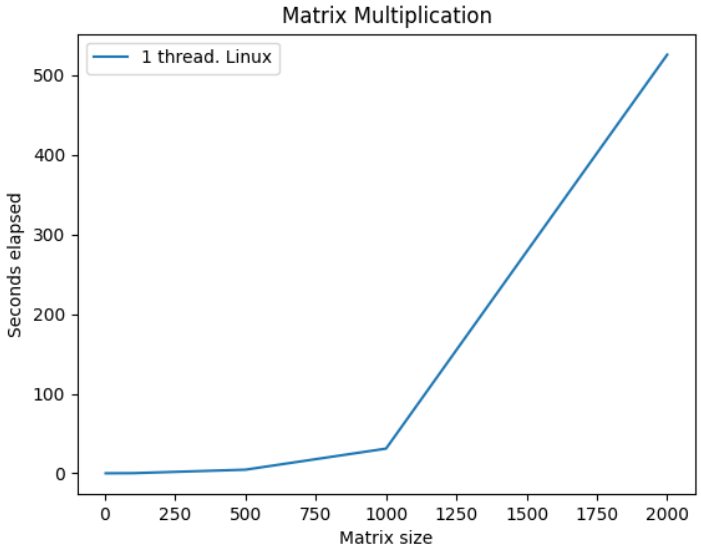

Рис. 1. График работы приложения с 1 потоком

Можно заметить, что начиная с размерности матрицы 1000x1000 – время начинает сильно увеличиваться. Для ускорения работы, распараллелим приложение на 10 потоков.

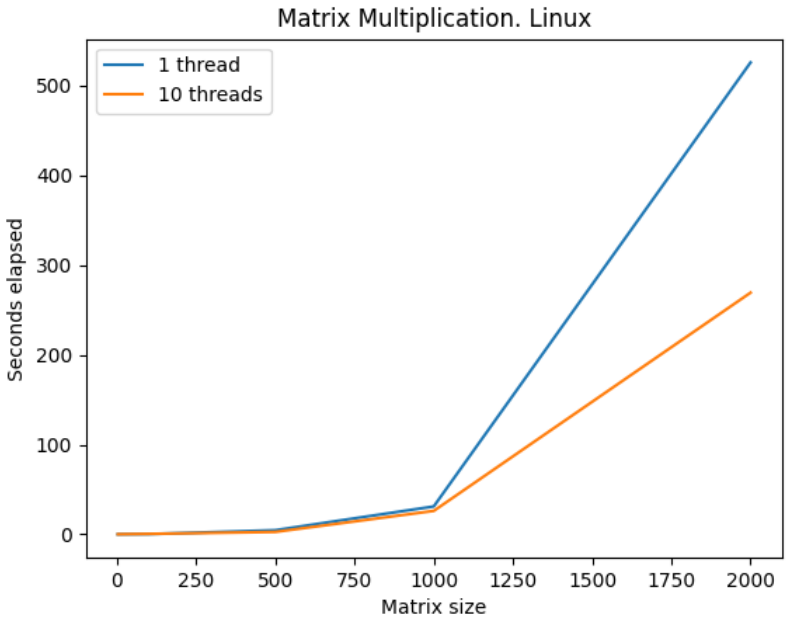

Рис. 2. График работы приложения с 1 и 10 потоками

По графику можно увидеть, что приложение значительно ускорилось. Матрица размером 2000x2000 умножилась быстрее почти в 2 раза. Однако, “брать наугад” число потоков, как мы сделали сейчас – плохая идея.

Исследование умножения матрицы на оптимальном количестве потоков

Модернизируем наше приложение тем, что количество потоков будет не какой-то константой, а количеством ядер в процессоре, которые доступны Java Virtual Machine.

В языках Java/Kotlin можно получить количество доступных ядер с помощью конструкции:

```
Runtime.getRuntime().getAvialiableProcess()
```

В нашем случае – доступно 2 ядра, соответственно, будем распараллеливать приложение на 2 потока.

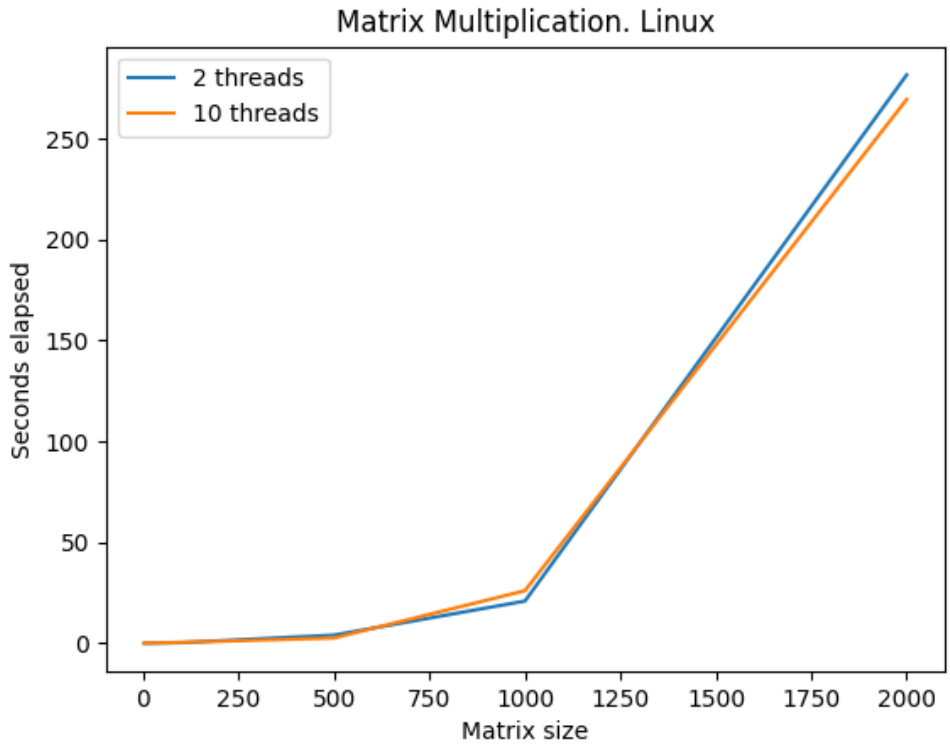

Рис. 3. График работы приложения с 2 и 10 потоками

Можно заметить, что время работы приложения с 2 и 10 потоками очень близко. Следовательно, нет смысла тратить ресурсы машины на создание большего количества потоков

Исследование умножения матрицы на случайном количестве потоков

Рассмотрим матрицу размером 1100x1100, так как именно с этих значений, начинаются сильные различия в работе приложений с разным количеством потоков. Распараллелим приложение на 2(количество ядер в процессоре), 10, 20, 50, 100 потоков и посмотрим на время работы приложения

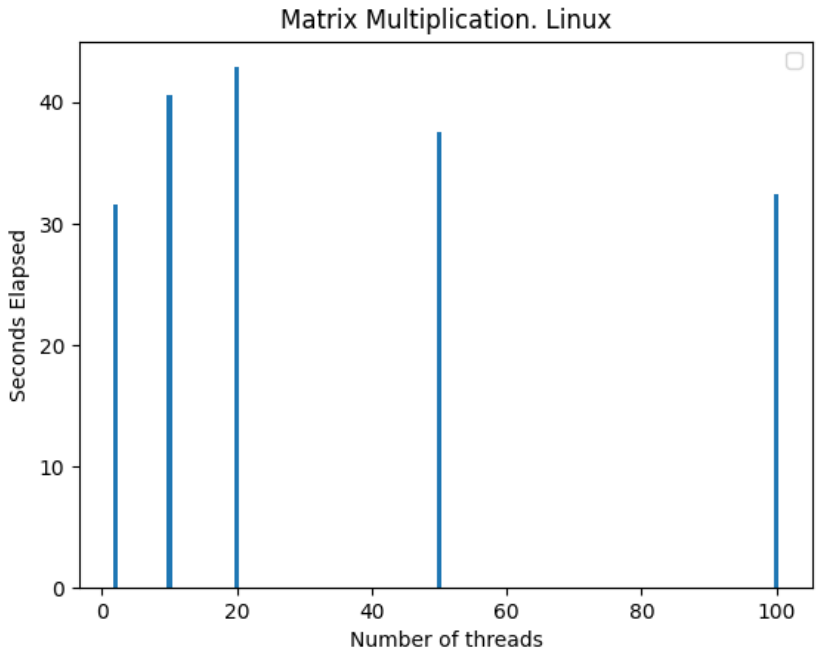

Рис. 4. Время работы приложения в зависимости от количества потоков

По рисунку видно, что быстрее всего отработало приложение, распараллеленное на 2 потока. Следовательно, ускорение работы приложения зависит не просто от количества потоков, а от их оптимального количества.

Сравнение работы многопоточных приложений в Windows и Linux VM

Рассмотрим разницу во времени работы многопоточных приложений в Windows и виртуальной машине Linux.

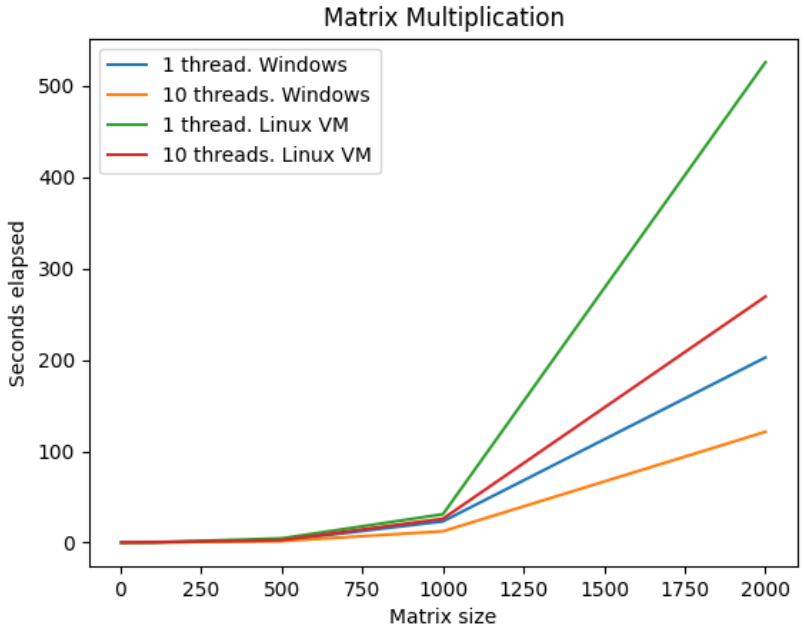

Рис. 5. Разница во времени работы в разных ОС

В связи с тем, что Linux, в нашем случае, именно виртуальная машина, то время работы в этой ОС гораздо больше, чем в “главной” ОС – Windows. Следовательно, тяжелые вычисления лучше делать на “родной” для ЭВМ операционной системе.

Сравнение работы потоков и корутин

В языке Kotlin есть еще один способ асинхронно выполнять код – корутины. По своей концепции, это что-то вроде легковесного потока. “Под капотом” у корутин тот же самый Thread, но работает он по-другому.

Рассмотрим 4 случая:

- Количество потоков == количеству строк в матрице

- Количество корутин == количеству строк в матрице

- Один поток

- Оптимальное количество потоков (равное количеству ядер)

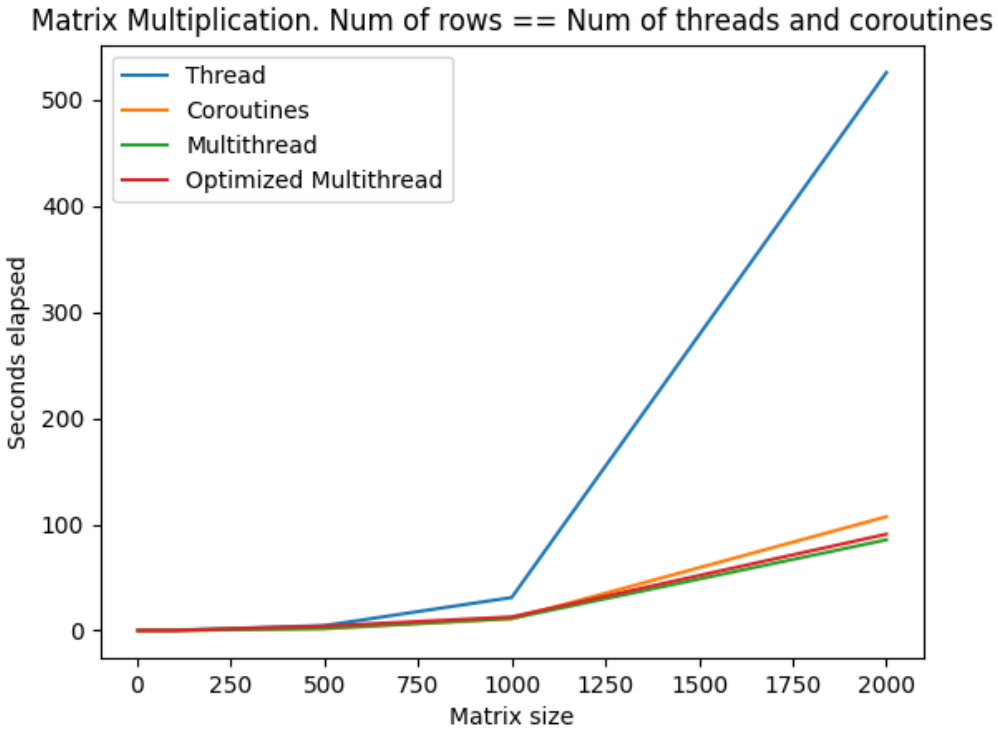

Рис. 6. Разница в работе разных подходов

Можно в очередной раз убедиться, что самое лучшее решение – распараллелить на количество потоков, равное количеству ядер.

Вывод

В ходе данной работы было разработано и исследовано простое многопоточное приложение, реализующее умножение матриц. Было выяснено, что большое количество потоков не гарантирует наилучшую производительность приложения. Напротив, в нашем случае, минимально-многопоточное приложение с двумя потоками оказалось самым производительным. То есть, самое оптимальное число потоков – количество ядер в процессоре (в нашем случае – 2).

Также сравнили работу корутин и потоков. При больших вычислениях потоки работают лучше.

По графикам видно, что умножение матриц до размеров 1000x1000 работает приблизительно одинаково как в одном потоке, так и на нескольких. Следовательно, умножение относительно небольших размеров матриц можно реализовать на одном потоке без каких-либо потерь.

Приложение

Листинг 1.

```
package matrix

import kotlin.random.Random

class MatrixBuilder(override val height: Int, override val width: Int) :
Matrix {
    fun buildMatrix(matrixNumRange: Int): Array<Array<Int>> {
        val matrix = Array(height) { Array(width) { 0 } }
        for (i in 0 until width) {
            for (j in 0 until height) {
                if (matrixNumRange == 0) {
                    matrix[i][j] = 0
                } else {
                    matrix[i][j] = Random.nextInt(matrixNumRange)
                }
            }
        }
        return matrix
    }
}
```

Листинг 2.

```
package matrix


interface Matrix {
    val height: Int
    val width: Int
}
```

Листинг 3.

```
package multiplication


import matrix.MatrixBuilder

class MatrixMultiplier {
    fun multiply(matrixA: Array<Array<Int>>,matrixB: Array<Array<Int>>):
Array<Array<Int>> {
        val sizeA = matrixA.size
        val sizeB = matrixB.size
        val result = MatrixBuilder(matrixA.size, sizeB).buildMatrix(0)

        for (i in 0 until sizeA) {
            for (j in 0 until sizeB) {
                for(k in 0 until sizeB ){
                    result[i][j] += matrixA[i][k] * matrixB[k][j]
```

```
                }

            }
        }
        return result
    }
}
```

Листинг 4.

```
class RowMultiplierWorker(
    private val result: Array<Array<Int>>,
    private val matrixA: Array<Array<Int>>,
    private val matrixB: Array<Array<Int>>,
    private val row: Int
) : Runnable {
    override fun run() {
        for (i in 0 until matrixB[0].size) {
            result[row][i] = 0
            for (j in 0 until matrixA[row].size) {
                result[row][i] += matrixA[row][j] * matrixB[j][i]
            }
        }
    }
}
```

Листинг 5.

```
object ParallelThreadsCreator {
    fun multiply(
        matrixA: Array<Array<Int>>,
        matrixB: Array<Array<Int>>,
        result: Array<Array<Int>>,
        numThreads:Int
    ): Array<Array<Int>> {
        val threads = mutableListOf<Thread>()
        val rows1 = matrixA.size
        for (i in 0 until rows1) {
            val task = RowMultiplierWorker(result, matrixA, matrixB, i)
            val thread = Thread(task)
            thread.start()
            threads.add(thread)

            if (threads.size % numThreads == 0) {
                waitForThreads(threads)
            }
        }
        return result
    }

    private fun waitForThreads(threads: MutableList<Thread>) {
        for (thread in threads) {
            try {
                thread.join()
            } catch (e: InterruptedException) {
                e.printStackTrace()
            }
        }
        threads.clear()
    }
}
```

Листинг 6.

```
import matrix.MatrixBuilder
import multiplication.MatrixMultiplier
import multiplication.multiplyConcurrently
import java.util.Date

suspend fun main(args: Array<String>) {
    val start = Date().time
    val matrixNumRange = 10
    val matrixSize = args[1].toInt()
    val matrixA = MatrixBuilder(matrixSize,
matrixSize).buildMatrix(matrixNumRange)
    val matrixB = MatrixBuilder(matrixSize,
matrixSize).buildMatrix(matrixNumRange)
    val res = MatrixBuilder(matrixSize, matrixSize).buildMatrix(0)

    when (args[0]) {
        "-m" -> {
            ParallelThreadsCreator.multiply(matrixA, matrixB, res,
args[2].toInt())
            println("Mode: Multi thread")
        }
        "-t" -> {
            MatrixMultiplier().multiply(matrixA, matrixB)
            println("Mode: One thread")
        }
        "-c" -> {
            multiplyConcurrently(matrixA, matrixB, res)
            println("Mode: Coroutines")
        }

        else -> println("Unknown operation")
    }

    val end = Date().time
    val diff = (end - start)
    println("Elapsed milliseconds: $diff")
    println("Num of threads: ${args[2]}")
    println("Size of matrix: $matrixSize")

}
```

Листинг 7.

```
package multiplication

import kotlinx.coroutines.*

fun multiplyRows(
    result: Array<Array<Int>>,
    matrixA: Array<Array<Int>>,
    matrixB: Array<Array<Int>>,
    row: Int
): Array<Array<Int>> {
    for (i in 0 until matrixB[0].size) {
        result[row][i] = 0
        for (j in 0 until matrixA[row].size) {
            result[row][i] += matrixA[row][j] * matrixB[j][i]
        }
    }
    return result
}

suspend fun multiplyConcurrently(
    matrixA: Array<Array<Int>>,
    matrixB: Array<Array<Int>>,
    result: Array<Array<Int>>,
) = coroutineScope {
    val rows = matrixA.size
    for (i in 0 until rows) {
        launch {
            multiplyRows(result, matrixA, matrixB, i)
        }

    }
}
```