**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**УНИВЕРСИТЕТ**

**«МЭИ»**

**Институт Информационных и Вычислительных технологий**

Кафедра Прикладной Математики и искусственного интеллекта

Лабораторная работа №4

**По курсу** «Параллельное программирование и параллельные системы»

**Выполнил:** Филиппов Е. И.

**Группа:** А-05м-23

**Преподаватель:** Задорин С.А.

Москва, 2024

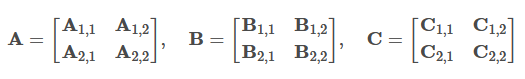
**Постановка задачи**

Реализовать алгоритм умножения матриц методом Штрассена с использованием языка программирования Haskell

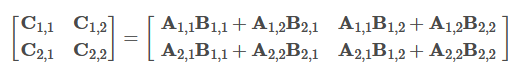
# Алгоритм

Алгоритм умножения матриц Штрассена - это первый алгоритм умножения матриц, который асимптотически работает лучше . Данный алгоритм имеет асимптотическую сложность равную .

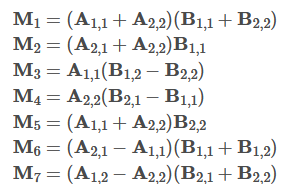
Пусть A и B — две (nxn)-матрицы, причём n — степень числа 2. Тогда можно разбить каждую матрицу A и B на четыре ((n/2)\*(n/2))-матрицы и через них выразить произведение матриц A и B:



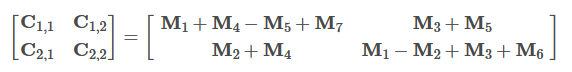
При стандартном подходе матрица C будет иметь такой вид:



Алгоритм Штрассена вводит новые элементы:



Элементы матрицы C будет вычисляться следующим образом:



Таким образом, количество умножений подматриц для приведенной выше формулы становится 7, а количество сложений подматриц для приведенной выше формулы становится 22. Сделано это с расчетом на то, что выполнение операции сложения обходится компьютеру намного дешевле, чем умножение.

Рекурсивный процесс будет продолжатся n раз до тех пор, пока размер матриц не станет достаточно малым, далее используется обычный метод умножения матриц. Это делают из-за того, что алгоритм Штрассена теряет эффективность по сравнению с обычным на малых матрицах в силу большего числа сложений.

# Параллельный алгоритм

При параллельном подходе предлагается вычислять каждый элемент в отдельном потоке. Алгоритм будет состоять из следующих этапов:

1. Задачи m1, m2, m3, m4, m5, m6, m7 выполняются параллельно и независимо друг от друга. Каждая из этих задач выполняет вычисление элементов . Параллелизм достигается с помощью операции «\*»
2. Затем происходит объединение результатов вычислений, которые также выполняется параллельно. Каждая из задач c11, c12, c21, c22 отвечает за соответствующую часть результирующей матрицы.

Таким образом, параллельное разбиение позволяет ускорить выполнение алгоритма за счет распараллеливания вычислений.

# Описание функций

* splitMatrix

Принимает матрицу в качестве входного параметра и разделяет ее на четыре квадранта: c11, c12, c21 и c22.

Квадранты получаются путем деления матрицы на четыре блока примерно равного размера.

Функция возвращает кортеж из четырех матриц, каждая из которых представляет один квадрант.

* combineMatrix

Принимает кортеж из четырех матриц в качестве входного параметра и объединяет их в одну матрицу.

Матрицы объединяются путем конкатенации их горизонтально и вертикально с помощью операторов <|> и <-> из пакета Data.Matrix.

Функция возвращает объединенную матрицу.

* writeMatrixToFile

Выводит матрицу в файл с заданным путем.

* strassen

Реализует алгоритм умножения матриц Стразена, который является рекурсивным подходом к умножению матриц.

Принимает две матрицы a и b в качестве входных параметров и возвращает их произведение.

Если матрицы маленькие (т.е. их размер меньше 32), то использует стандартный алгоритм умножения матриц (multStd).

В противном случае, разделяет матрицы на четыре квадранта с помощью splitMatrix, вычисляет произведения квадрантов с помощью рекурсивных вызовов strassen и объединяет результаты с помощью combineMatrix.

Функция использует параллелизм для вычисления произведений квадрантов одновременно с помощью функций rpar и rdeepseq из пакета Control.Parallel.Strategies.

# Конфигурация ПК

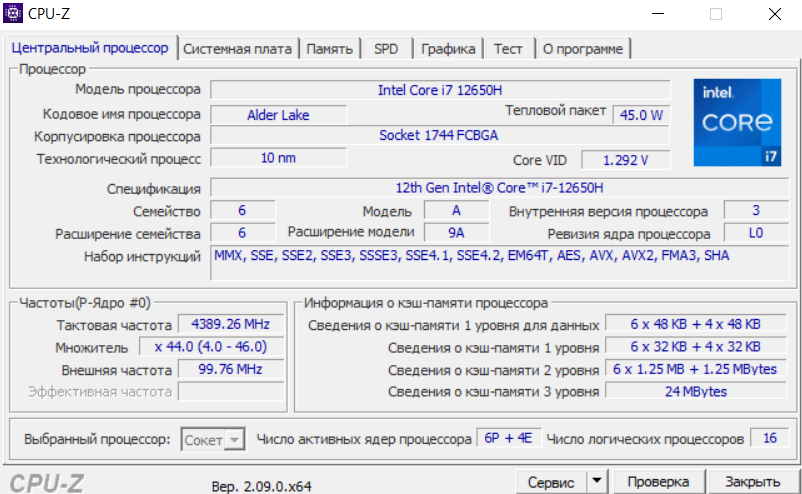


Рис. 1 Информация о процессоре.

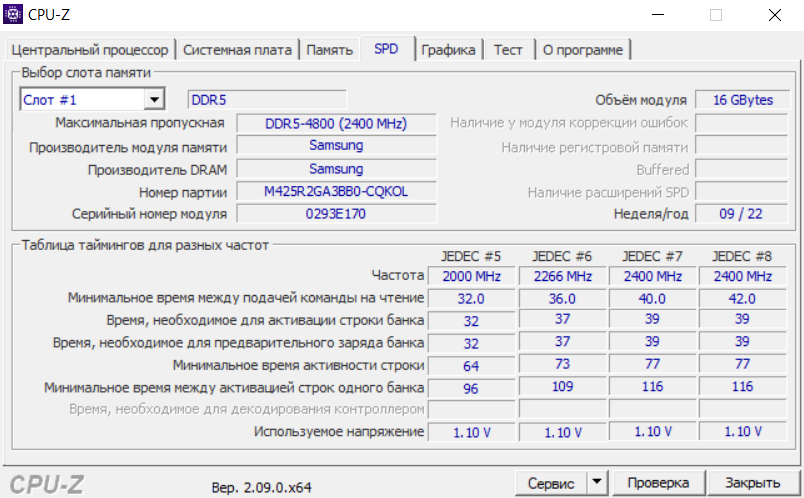
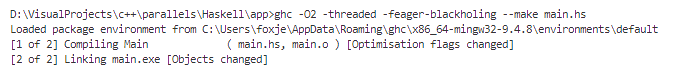


Рис. 2 Информация об оперативной памяти.

# Пример работы программы

*Сборка программы*



*Запуск программы*



*Содержимое файла*

**

# Тесты

При тестировании будем перемножать матрицы размером 512x512, 1024x1024, 2048x2048.

Таблица 1. Зависимость времени работы (в секундах) от числа потоков

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность матрицы  Число потоков | 512х512 | 1024х1024 | 2048х2048 |
| 1 | 1.382 | 10.444 | 76.212 |
| 2 | 0.945 | 6.876 | 49.425 |
| 3 | 0.758 | 5.792 | 43.661 |
| 4 | 0.701 | 5.442 | 41.252 |
| 5 | 0.712 | 5.547 | 42.360 |
| 6 | 0.729 | 5.643 | 44.758 |
| 8 | 0.809 | 6.297 | 45.757 |
| 12 | 1.25 | 8.271 | 61.366 |

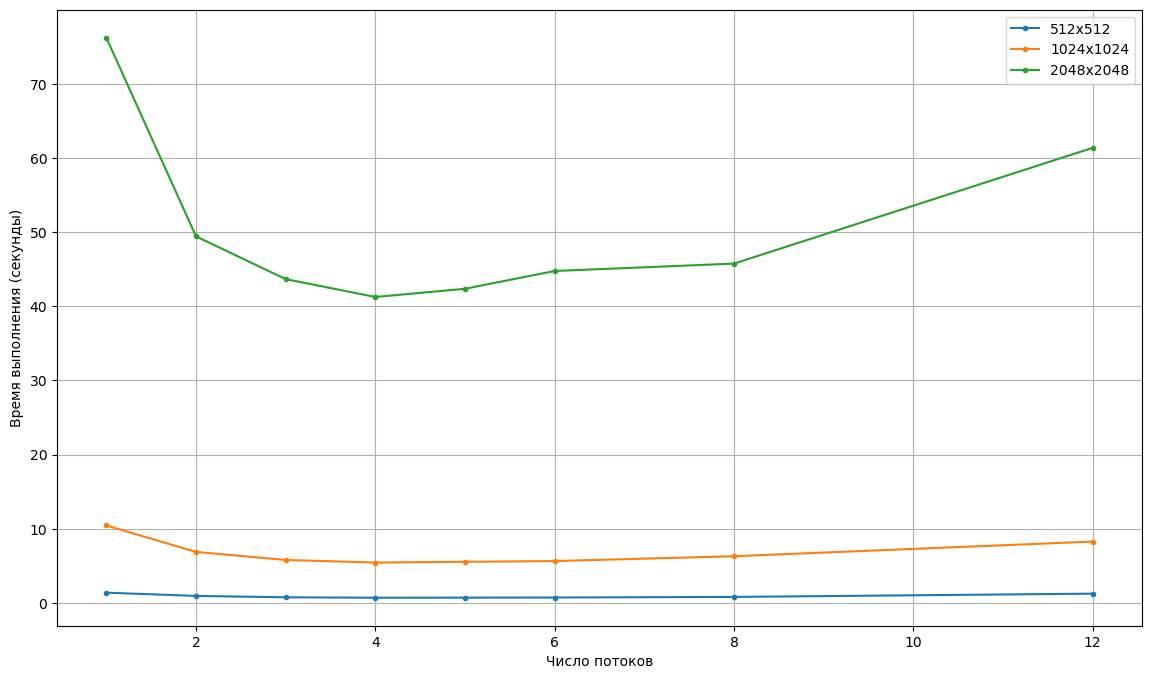


Рис. 3 График зависимости времени выполнения от числа потоков

Таблица 2. Зависимость коэффициента ускорения от числа потоков

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность матрицы  Число потоков | 512х512 | 1024х1024 | 2048х2048 |
| 1 | 1 | 1 | 1 |
| 2 | 1.462 | 1.518 | 1.541 |
| 3 | 1.823 | 1.803 | 1.745 |
| 4 | 1.971 | 1.919 | 1.847 |
| 8 | 1.941 | 1.882 | 1.799 |
| 10 | 1.895 | 1.850 | 1.702 |
| 12 | 1.708 | 1.658 | 1.665 |
| 16 | 1.105 | 1.262 | 1.241 |

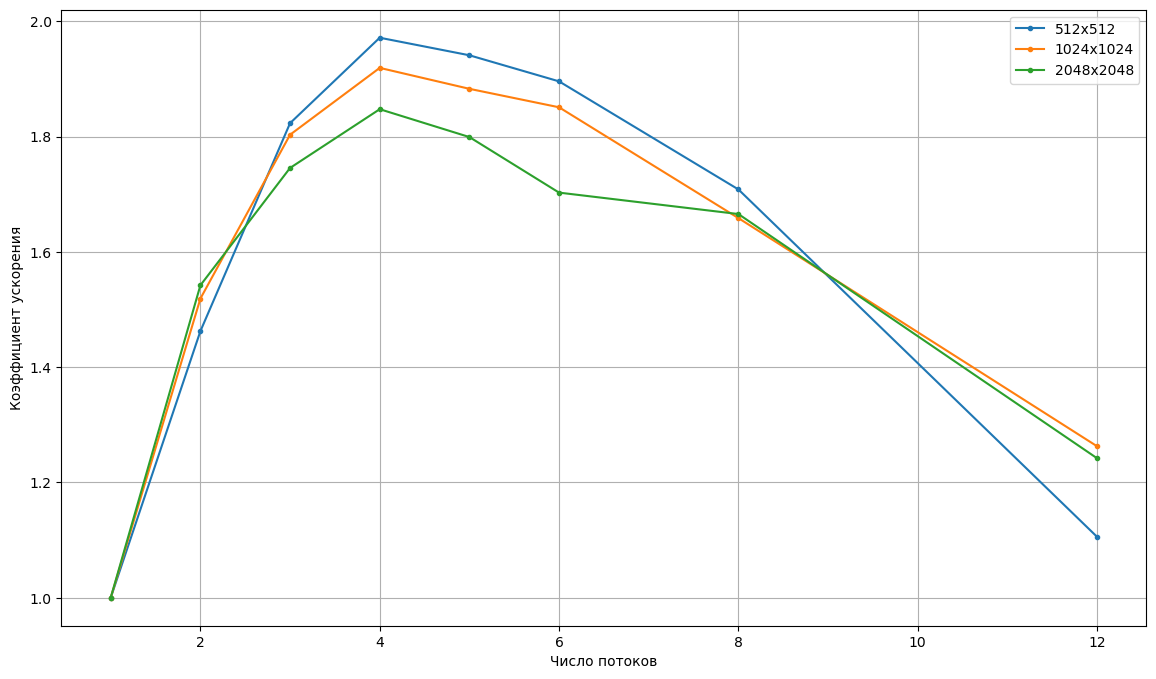


Рис. 4 График зависимости коэффициента ускорения от числа потоков

# Вывод

Грамотный подбор числа потоков позволяет нам достигать наилучшего соотношения выделяемых ресурсов к скорости вычисления. Время работы алгоритма уменьшается с увеличением числа потоков, достигая максимального ускорения при использовании 4-5 потоков для матриц размером 512x512 и 1024x1024, и 3-4 потоков для матриц размером 2048x2048. Дальнейшее увеличение числа потоков не приводит к значительному ускорению вычислений, а в некоторых случаях даже наблюдается ухудшение производительности. При параллельном решении задачи удалось ускорить выполнение алгоритма примерно в 2 раза, что слегка уступает показателям ускорения из предыдущих работ.

# Ссылки

https://github.com/FoxJefisto/lab4 - репозиторий с реализацией алгоритма

# Программный код

Haskell:

|  |
| --- |
| module Main where  import Data.Matrix  import Control.Parallel.Strategies  import System.Clock  import Control.DeepSeq  import System.Environment  splitMatrix :: Matrix Int -> (Matrix Int, Matrix Int, Matrix Int, Matrix Int)  splitMatrix mtrx = (c11, c12, c21, c22)      where          n = nrows mtrx          m = n `div` 2          (c11, c12, c21, c22) = splitBlocks m m mtrx  combineMatrix :: (Matrix Int, Matrix Int, Matrix Int, Matrix Int) -> Matrix Int  combineMatrix (c11, c12, c21, c22) = (c11 <|> c12) <-> (c21 <|> c22)  strassen :: Matrix Int -> Matrix Int -> Matrix Int  strassen a b      | n < 32 = multStd a b      | otherwise = combineMatrix (c11, c12, c21, c22)      where          n = nrows a          (a11, a12, a21, a22) = splitMatrix a          (b11, b12, b21, b22) = splitMatrix b          (c11, c12, c21, c22) = runEval $ do              p1 <- rpar $ strassen (a11 + a22) (b11 + b22)              p2 <- rpar $ strassen (a21 + a22) b11              p3 <- rpar $ strassen a11 (b12 - b22)              p4 <- rpar $ strassen a22 (b21 - b11)              p5 <- rpar $ strassen (a11 + a12) b22              p6 <- rpar $ strassen (a21 - a11) (b11 + b12)              p7 <- rpar $ strassen (a12 - a22) (b21 + b22)              rdeepseq p1              rdeepseq p2              rdeepseq p3              rdeepseq p4              rdeepseq p5              rdeepseq p6              rdeepseq p7              p11 <- rpar (p1 + p4 - p5 + p7)              p12 <- rpar (p3 + p5)              p21 <- rpar (p2 + p4)              p22 <- rpar (p1 - p2 + p3 + p6)              rdeepseq p11              rdeepseq p12              rdeepseq p21              rdeepseq p22              return (p11, p12, p21, p22)  writeMatrixToFile :: FilePath -> Matrix Int -> IO ()  writeMatrixToFile filePath matrix = writeFile filePath $ prettyMatrix matrix  main :: IO ()  main = do      args <- getArgs      let size = read (args !! 0) :: Int      timeStart <- getTime Monotonic      result <- return $!! (strassen (matrix size size $ \(i,j) -> 4) (matrix size size $ \(i,j) -> 4))      timeEnd <- getTime Monotonic      let timeRun = timeEnd - timeStart      let seconds = sec timeRun      let milliseconds = (nsec timeRun) `div` 1000000      print $ "Time run: " ++ show seconds ++ "." ++ show milliseconds      writeMatrixToFile "result.txt" result      return () |