Федеральное государственное бюджетное образовательное учреждение высшего образования «**Московский государственный** **технический** **университет** **имени**

Н. Э. **Баумана**

Факультет информатики и систем управления

Кафедра ИУ9 «Теоретическая информатика и компьютерные технологии»

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа №1  
«Pаспараллеливание алгоритма вычисления произведения двух матриц»

по курсу: «Разработка параллельных и распределенных программ»

Выполнила: Студентка группы ИУ9-51Б

Александрова О.С

Проверил: Царев А. С.

Задание:

Изображение выглядит как текст, снимок экрана, Шрифт, черно-белый

Автоматически созданное описание

Характеристики устройства, на котором производились вычисления

Название модели: MacBook Air

Идентификатор модели: MacBookAir10,1

Чип: Apple M1

Общее количество ядер: 8 (4 производительности и 4 эффективности)

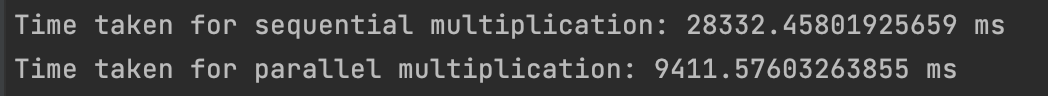
Память: 8 ГБ

Ход работы (листинг текста программы)

import sys  
import multiprocessing  
import numpy as np  
import time  
  
def mat\_mult(matIn1, matIn2):  
  
 n, m = matIn1.shape  
 \_, r = matIn2.shape  
  
 matOut = np.zeros((n, r), dtype=int)  
 for i in range(n):  
 for j in range(r):  
 for k in range(m):  
 matOut[i][j] = matOut[i][j] + matIn1[i][k] \* matIn2[k][j]  
  
 return matOut  
  
  
def mat\_mult\_parallel(matIn1, matIn2, sharedMemArr, lastI):  
  
  
 n, m = matIn1.shape  
 \_, r = matIn2.shape  
 # print((n,m,r))  
 for i in range(n):  
 # print(i,j,k)  
 for j in range(r):  
 sumMat = 0  
 for k in range(m):  
 sumMat = sumMat + int(matIn1[i][k] \* matIn2[k][j])  
  
 # print(lastI)  
 sharedMemArr[lastI \* r + i \* r + j] = sumMat  
  
def from1D\_to\_2D\_arr(arr1D, desired\_shape):  
  
 m, n = desired\_shape  
 ind = 0  
 # print(m,n)  
 arr2D = np.zeros((m, n), dtype=int)  
  
 for i in range(m):  
 for j in range(n):  
 arr2D[i][j] = arr1D[ind]  
 ind = ind + 1  
 return arr2D  
  
  
def runSequentialMatMul(matIn1, matIn2):  
 timeStart = time.time()  
 matOut = mat\_mult(matIn1, mat\_trp(matIn2))  
 timeEnd = time.time()  
  
 timeForExecSeq = (timeEnd - timeStart) \* 1000  
 print("Time taken for sequential multiplication:", timeForExecSeq, "ms")  
 return timeForExecSeq, matOut  
  
  
def print\_message\_about\_default(numProcessors, lowerLimit, upperLimit, n, m):  
 print("Values in arguments: ")  
 print("numProcessors = %d" % numProcessors)  
 print("lowerLimit = %d, upperLimit = %d" % (lowerLimit, upperLimit))  
 print("n = %d, m = %d" % (n, m))  
  
def get\_division(tot\_rows, numProcessors):  
  
 division = []  
 resLast, res = 0, 0  
 division.append(res)  
  
 while numProcessors != 0:  
 if tot\_rows >= numProcessors:  
 resLast = resLast + res  
 res = tot\_rows // numProcessors  
  
 division.append(resLast + res)  
 tot\_rows = tot\_rows - res  
 numProcessors = numProcessors - 1  
  
 else:  
 division = np.linspace(0, tot\_rows, num=tot\_rows, dtype=int)  
 break  
  
 return division  
  
def mat\_trp(matIn):  
 n,m = matIn.shape  
 matOut = np.zeros((m,n),dtype=int)  
  
 for i in range(n):  
 for j in range(m):  
 matOut[j][i] = matIn[i][j]  
  
 return matOut  
  
def runParallelMatMul(matIn1, matIn2, sharedMemArr, numProcessors):  
  
 procArr = []  
  
 tot\_rows, \_ = matIn1.shape  
 division = get\_division(tot\_rows, numProcessors)  
 timeStart = time.time()  
  
 for i in range(len(division) - 1):  
 matIn1\_slice = matIn1[division[i]: division[i + 1], :]  
 matIn2\_trp = mat\_trp(matIn2)  
  
 p = multiprocessing.Process(target=mat\_mult\_parallel,  
 args=(matIn1\_slice, matIn2\_trp, sharedMemArr, division[i]))  
 procArr.append(p)  
  
 if len(division) == 1:  
 sharedMemArr = mat\_mult(matIn1, mat\_trp(matIn2))  
  
 for p in procArr:  
 p.start()  
  
 for p in procArr:  
 p.join()  
  
 timeEnd = time.time()  
  
 timeForExecPar = (timeEnd - timeStart) \* 1000  
  
 n = int(len(sharedMemArr) \*\* 0.5)  
 desired\_shape = (n, n)  
  
 matOut = from1D\_to\_2D\_arr(sharedMemArr, desired\_shape)  
  
 print("Time taken for parallel multiplication:", timeForExecPar, "ms")  
  
 return timeForExecPar, matOut  
  
  
def main(argv):  
 # numProcessors = multiprocessing.cpu\_count()  
 # numProcessors, lowerLimit, upperLimit, n, m = get\_args(argv)  
 numProcessors = int(sys.argv[1])  
 lowerLimit = int(sys.argv[2])  
 upperLimit = int(sys.argv[3])  
 n = int(sys.argv[4])  
 m = n  
 print\_message\_about\_default(numProcessors, lowerLimit, upperLimit, n, m)  
  
 matIn1 = np.random.randint(low=lowerLimit, high=upperLimit, size=(n, m))  
 matIn2 = np.random.randint(low=lowerLimit, high=upperLimit, size=(n, m))  
  
 print("First Matrix \n", matIn1)  
 print("Second Matrix \n", matIn2)  
  
 timeForExecSeq, matOutSeq = runSequentialMatMul(matIn1, matIn2)  
 sharedMemArr = multiprocessing.Array('i', n \* n)  
 timeForExecPar, matOutPar = runParallelMatMul(matIn1, matIn2, sharedMemArr, numProcessors)  
  
 print("\n" , "Serial Multiplication: ")  
 # print(matOutSeq)  
 print("\n" , "Parallel Multiplication: ")  
 # print(matOutPar)  
 print("\n", "Hence, with {0} cores, parallel algorithm runs faster {1}".format(numProcessors, float(  
 timeForExecSeq - timeForExecPar)))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 argv = sys.argv[1:]  
 main(argv)

Времена работы программы

Вычисление матрицы после перемножения матриц 300 на 300



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Число потоков | 2 | 3 | 4 | 8 |
| Время работы,ms | 9925.521850585938 | 9411.57603263855 | 7950.6189823150635 | 6936.492919921875 |

Изображение выглядит как График, линия, диаграмма, текст

Автоматически созданное описание

Вывод

Были изучены и реализованы способы распараллеливания алгоритма вычисления произведения двух матриц. По полученным данным можно сделать вывод: при большом объеме вычислений распараллеливания показывает время, которое в 3, 4 раза меньше, чем при стандартном решении.