**JAVA编程进阶上机报告**

****

第4次上机作业

**学 院\_智能与计算学部\_**

**专 业\_\_ 软件工程\_\_\_\_\_**

**姓 名\_\_\_ 李济孚\_\_ \_ \_**

**学 号\_\_ 3018216260\_\_\_**

**年 级\_\_\_\_ 2018\_\_\_\_\_\_**

**班 级\_\_\_\_\_\_ 5\_\_\_\_\_\_\_\_**

# 一、实验要求

## 一、实验目的

## 运用线程并发计算

## 二、实验要求

## • 编写矩阵随机生成类 MatrixGenerator 类，随机生成任意大小的矩阵，矩阵单元使用 double 存储。

## • 使用串行方式实现矩阵乘法。

## • 使用多线程方式实现矩阵乘法。

## • 比较串行和并行两种方式使用的时间，利用第三次使用中使用过的 jvm状态查看命令，分析产生时间差异的原因是什么。

## 三、源代码

**MyThread类:**

public class MyThread extends Thread{

int line1;

int row1;

int row2;

double[][] matrix1;

double[][] matrix2;

double[][] result2;

int line2;

public MyThread(int line1, int row1, int row2, double[][] matrix1, double[][] matrix2, double[][] result2, int lines\_Per\_Thread)

{

this.line1 = line1;

this.row1 = row1;

this.row2 = row2;

this.matrix1 = matrix1;

this.matrix2 = matrix2;

this.result2 = result2;

this.line2 = line1 + lines\_Per\_Thread;

}

@Override

public void run() {

// System.out.println(Thread.currentThread().getName() + "开始的时间" + from);

// from = System.currentTimeMillis();

// for(int i = 0; i < line1; i++)

// {

// if(i == line1/2)

// from = System.nanoTime();

for(; line1 < line2; line1++)

{

result2[line1] = new double[row2];

for(int j = 0; j < row2; j++)

{

result2[line1][j] = 0.0;

for(int k = 0; k < row1; k++)

result2[line1][j] += matrix1[line1][k] \* matrix2[k][j];

}

}

// }

// long end = System.currentTimeMillis();

// System.out.println(Thread.currentThread().getName() + "结束的时间" + end);

// long gap2 = end - from;

// System.out.println(Thread.currentThread().getName() + "运行时间" + gap2);

}

}

**MatrixGenerator类:**

import java.math.BigDecimal;

import java.math.RoundingMode;

import java.util.Random;

import java.util.concurrent.atomic.AtomicInteger;

public class MatrixGenerator {

private double[][] matrix1;

private double[][] matrix2;

private double[][] result1;

private double[][] result2;

private long time\_LBL;

private AtomicInteger count = new AtomicInteger(0);

private int row1;

private int line1;

private int row2;

private MyThread[] thread;

public MatrixGenerator(int numberMax, int Maximum)

{

Random random = new Random();

row1 = random.nextInt(Maximum)+1;

line1 = random.nextInt(Maximum)+1;

row2 = random.nextInt(Maximum)+1;

matrix1 = new double[line1][row1];

matrix2 = new double[row1][row2];

for(int i = 0; i < line1; i++)

{

for(int j = 0; j < row1; j++)

{

BigDecimal b = BigDecimal.valueOf(random.nextDouble() \* numberMax);

matrix1[i][j] = b.setScale(2, RoundingMode.HALF\_UP).doubleValue();

}

}

for(int i = 0; i < row1; i++)

{

for(int j = 0; j < row2; j++)

{

BigDecimal b = BigDecimal.valueOf(random.nextDouble() \* numberMax);

matrix2[i][j] = b.setScale(2, RoundingMode.HALF\_UP).doubleValue();

}

}

}

public void printMatrix1() {

System.out.println("Matrix\_ONE [" + line1 + "][" + row1 + "] ->");

for(int i = 0; i < line1; i++)

{

for (int j = 0; j < row1; j++)

{

System.out.print(matrix1[i][j] + " ");

}

System.out.println();

}

}

public void printMatrix2()

{

System.out.println("Matrix\_TWO [" + row1 + "][" + row2 + "] ->");

for(int i = 0; i < row1; i++)

{

for (int j = 0; j < row2; j++)

{

System.out.print(matrix2[i][j] + " ");

}

System.out.println();

}

}

public void printResultMatirx1()

{

System.out.println("Matrix\_RESULT [" + line1 + "][" + row2 + "] ->");

for(int i = 0; i < line1; i++)

{

for (int j = 0; j < row2; j++)

{

System.out.print(new BigDecimal(result1[i][j]).setScale(2, RoundingMode.HALF\_UP) + " ");

}

System.out.println();

}

}

public void printResultMatirx2()

{

while(true)

{

if(count.get() == 2)

{

System.out.println("已经有" + count + "个线程完成计算");

System.out.println("Matrix\_RESULT [" + line1 + "][" + row2 + "] ->");

for(int i = 0; i < line1; i++)

{

for (int j = 0; j < row2; j++)

{

System.out.print(new BigDecimal(result2[i][j]).setScale(2, RoundingMode.HALF\_UP) + " ");

}

System.out.println();

}

break;

}

}

}

public double[][] calculate\_LBL()

{

result1 = new double[line1][row2];

long from = System.nanoTime();

for(int i = 0; i < line1; i++)

{

for(int j = 0; j < row2; j++)

{

result1[i][j] = 0.0;

for(int k = 0; k < row1; k++)

result1[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

time\_LBL = System.nanoTime() - from;

return result1;

}

public void calculate\_BTH(int Lines\_Per\_Thread) throws InterruptedException {

assert !(Lines\_Per\_Thread==0);

result2 = new double[line1][row2];

int number\_of\_threads = line1/Lines\_Per\_Thread;

thread = new MyThread[number\_of\_threads];

int from = 0;

for(int i = 0; i < number\_of\_threads; i++, from+=Lines\_Per\_Thread) {

if(i == number\_of\_threads-1)

thread[i] = new MyThread(from, row1, row2, matrix1, matrix2, result2, line1-from);

else

thread[i] = new MyThread(from, row1, row2, matrix1, matrix2, result2, Lines\_Per\_Thread);

}

long time = System.nanoTime();

for(int i = 0; i < number\_of\_threads; i++)

{

thread[i].start();

}

for(int i = 0; i < number\_of\_threads; i++)

{

thread[i].join();

// System.out.println("线程活跃数量"+Thread.activeCount());

}

long time2 = System.nanoTime()-time;

System.out.println("Parallel calculating(" + Lines\_Per\_Thread + " lines/thread): " + time2 + "ns");

}

public long getTime\_LBL() {

return time\_LBL;

}

public int getLine1() {

return line1;

}

public int getRow1() {

return row1;

}

public int getRow2() {

return row2;

}

public boolean is\_correct()

{

for(int i = 0 ; i< line1; i++)

{

for(int j = 0; j < row2; j++)

{

if(result1[i][j] != result2[i][j])

return false;

}

}

return true;

}

}

**Main类：**

public class Main {

public static void main(String[] args) throws InterruptedException {

MatrixGenerator matrixGenerator = new MatrixGenerator(1000, 10);

System.out.println("Matrix\_ONE: LINE-> " + matrixGenerator.getLine1() +" | " + "ROW-> " + matrixGenerator.getRow1());

System.out.println("Matrix\_TWO: LINE-> " + matrixGenerator.getRow1() +" | " + "ROW-> " + matrixGenerator.getRow2());

// matrixGenerator.printMatrix1();

// matrixGenerator.printMatrix2();

System.out.println("\r\n" + "----------------Start Calculating--------------" + "\r\n");

matrixGenerator.calculate\_LBL();

// matrixGenerator.printResultMatirx1();

System.out.println("Serial calculating: " + matrixGenerator.getTime\_LBL() + "ns" + "\r\n");

// matrixGenerator.calculate\_LBL();

// matrixGenerator.printResultMatirx1();

// System.out.println("串行计算时间: " + matrixGenerator.getTime\_LBL() + "纳秒" + "\r\n");

matrixGenerator.calculate\_BTH(1);

matrixGenerator.calculate\_BTH(4);

System.out.println("\r\n" + "----------------Result Testing--------------" + "\r\n");

System.out.println("VALIDITY：" + matrixGenerator.is\_correct());

// matrixGenerator.printResultMatirx2();

}

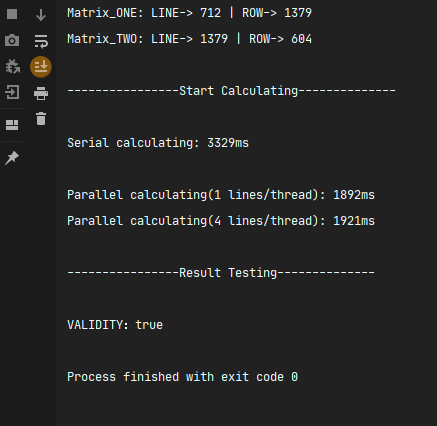
}

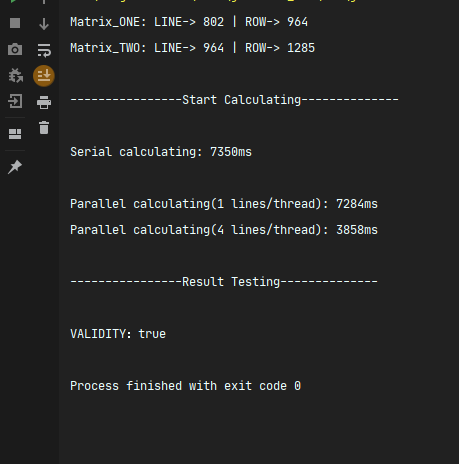
## 三、运行结果

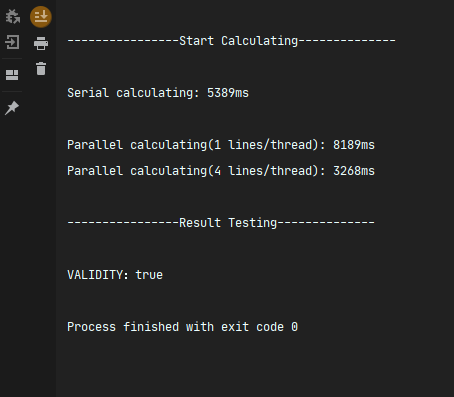
首先通过随机数生成两个行和列都不小于600的矩阵，并且矩阵的元素范围是0到1000且保留两位小数。

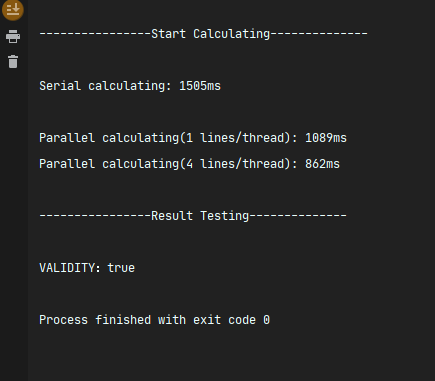
然后开始比较串行计算和并行计算。在并行计算的设计中，计算两次。一次是每1行矩阵相乘都使用一个线程来计算，另外一次是每4行矩阵相乘使用一个线程来计算。

为了尽量消除偶然性，进行四次运行。







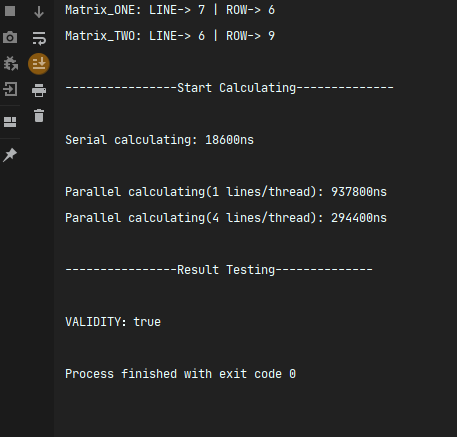


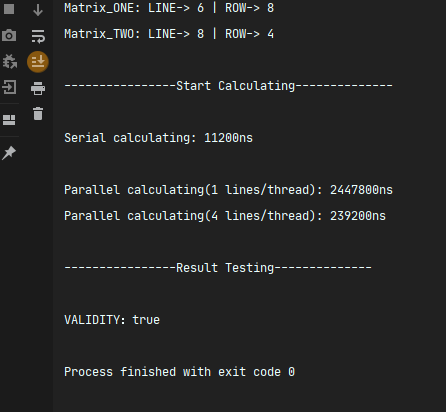
由上图可以看出，在实际运行的时候，1行1线程大概率会比穿行运算所用时间短，而4行1线程则会明显加快整体运行速度（在更多的测试中，有时候4行1线程与1行1线程相当）。

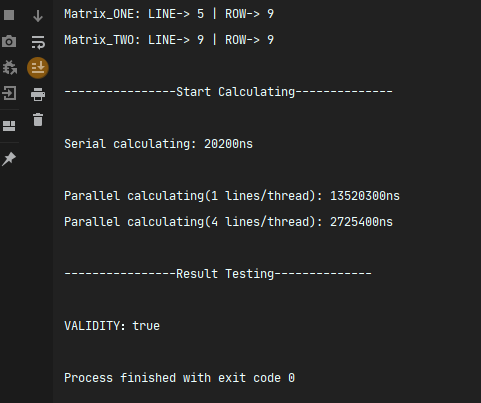
这是在矩阵比较庞大的情况，下面是矩阵较小的情况。

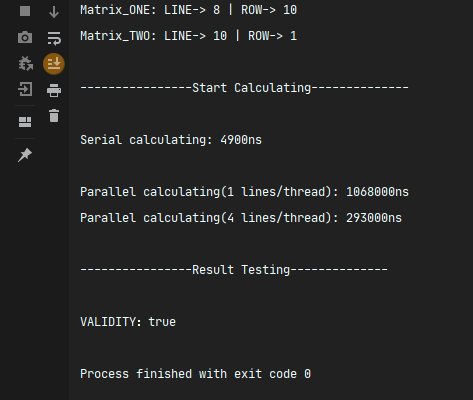
在矩阵较小的情况下，一样实行4次运行。矩阵的行列数均被限制在10以内。

为了得到更高的时间精确度，将ms级别换成ns级别。









结果如上图所示。可以得出结论，在矩阵较小时，串行计算的效率远大于并行计算的效率，并且在并行计算中，1个线程负责越多的计算整体效率越高