学 院 智能与计算

专 业 软件工程

班 级 三班

学 号 3018216144

姓 名 王文君

一、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。

二、设计思路

按照行分计算矩阵，A矩阵的每一行和B矩阵的每一列的相乘和加和，都交给一个线程来计算

并行时主线程需要等到全部的子线程计算完成后得到的结果，采用CountDownLatch类作为计数工具。

countDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行。

是通过一个计数器来实现的，计数器的初始值是线程的数量。每当一个线程执行完毕后，计数器的值就-1，当计数器的值为0时，表示所有线程都执行完毕，然后在闭锁上等待的线程就可以恢复工作了。

三、源代码

```
package Matrix;

public class MatrixGenerator{
    // 表示行和列
    private int mRow, mColumn;
    private double [][]Data;

    // 构造方法
    public MatrixGenerator(int row, int column) {
        mRow = row;
        mColumn = column;
        Data= new double[mRow ][mColumn];
    }


    public double[][] getData() {
        return Data;
    }

    private double random() {
        double random1 = Math.random();
        double random2 = random1 * 20; // 0.0 - 20.0
        return  random2;
    }

    // 创建矩阵
    public void createMatrix() {

        for (int i = 0; i < mRow; i++) {
              for (int j = 0; j < mColumn; j++)
                    {
                        Data[i][j]=random();
                    }
        }
    }

    public double getMatrix(int row, int col) {

        return Data[row ][col] ;

    }
```

```
    public int getmRow() {
        return mRow;
    }


    public int getmColumn() {
        return mColumn;
    }

    public void setMatrix(int row , int col, double value) {

           this.Data[row][col] = value;

     }
    public void print()
    {
        for (int i = 0; i < mRow; i++) {
          for (int j = 0; j < mColumn; j++)
                {
                    System.out.print(Data[i][j]+" ");
                }
           System.out.println();
      }
    }


}

package Matrix;

import java.util.Arrays;
import java.util.concurrent.CountDownLatch;

public class multiply extends Thread{

    private MatrixGenerator m1,m2,m;
    private int gap,index;
    private CountDownLatch countDownLatch;

    multiply(MatrixGenerator m1,MatrixGenerator m2,MatrixGenerator m,int gap,int
index, CountDownLatch countDownLatch)
    {
        this.m1=m1;
        this.m2=m2;
        this.gap=gap;
        this.index=index;
        this.m=m;
        this.countDownLatch=countDownLatch;
    }

    public void run() {
            double temp=0;
            for (int i = index * gap; i < (index + 1) * gap; i++)
            {
                for(int k = 0;k<m2.getmColumn();k++) {
                    for(int j = 0;j<m1.getmColumn();j++)
                    {
                        temp += m1.getMatrix(i, j) * m2.getMatrix(j, k);
```

```
                    }

                            m.setMatrix(i, k, temp);
                            temp = 0;

                }
            }
            countDownLatch.countDown();
        }


    public static MatrixGenerator multiply_serial(MatrixGenerator
m1,MatrixGenerator m2)
    {
        MatrixGenerator m3=new MatrixGenerator(m1.getmRow(),m2.getmColumn());
        double temp=0;

        for(int i = 0;i<m3.getmRow();i++) {

            for(int k = 0;k<m2.getmColumn();k++) {

                for(int j = 0;j<m1.getmColumn();j++)

                    {

                    temp += m1.getMatrix(i, j) * m2.getMatrix(j, k);
//                    System.out.println(m1.getMatrix(i, j)+"*"+m2.getMatrix(j,
k));


                    }

                m3.setMatrix(i, k, temp);

                temp = 0;

            }
        }

        return m3;

    }

    public static boolean isEqual(double[][] a, double b[][]) {
        if (a.length != b.length) {
        return false;
        }
        else {
            for (int i = 0; i < a.length; i++) {
                    if (a[i].length != b[i].length) {
                        return false;
                    } else {
                        if (!Arrays.equals(a[i], b[i]))
                            return false;
                        }
            }
        }
        return true;
    }
```

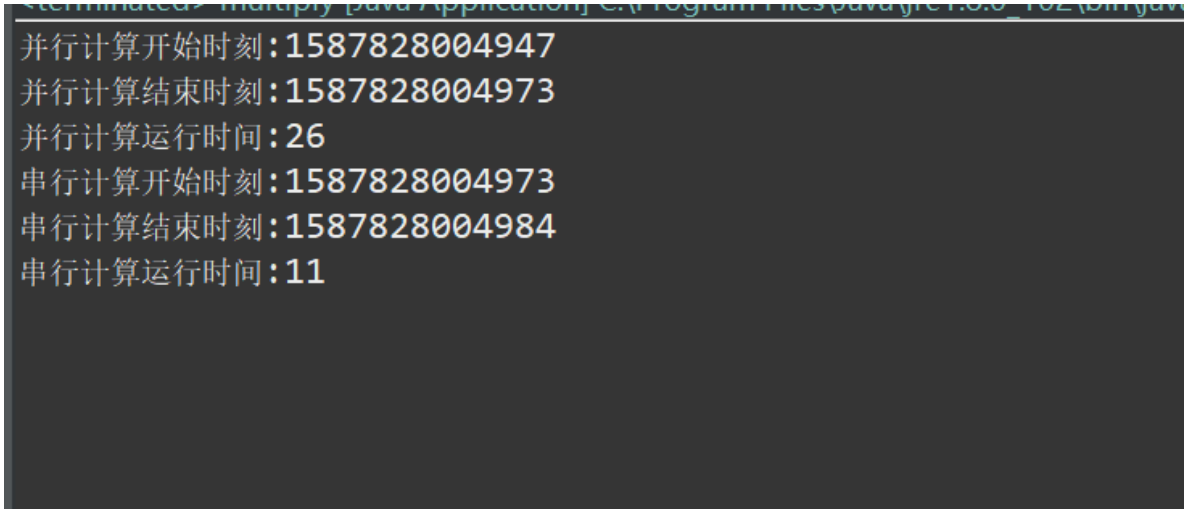
```
    public static void main(String[] args) throws InterruptedException {
        // TODO Auto-generated method stub
        long startTime;
        long endTime;
        MatrixGenerator m1=new MatrixGenerator(300,10);
        m1.createMatrix();
        MatrixGenerator m2=new MatrixGenerator(10,100);
        m2.createMatrix();
//      m1.print();
//      System.out.println();
//      m2.print();
//      System.out.println();

        //并行
        MatrixGenerator m3=new MatrixGenerator(m1.getmRow(),m2.getmColumn());
        int threadNum=3;
        CountDownLatch countDownLatch = new CountDownLatch(threadNum);
        startTime = System.currentTimeMillis();
        int gap=m1.getmRow()/threadNum;
        for (int i = 0; i < threadNum; i++) {
            multiply ct = new multiply(m1, m2, m3, gap, i,countDownLatch);
            ct.start();
        }
        ////调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行
        countDownLatch.await();
        endTime = System.currentTimeMillis();
        System.out.println("并行计算开始时刻:" + (startTime));
        System.out.println("并行计算结束时刻:" + (endTime));
        System.out.println("并行计算运行时间:" + (endTime - startTime));
//      串行
        startTime = System.currentTimeMillis();
        MatrixGenerator m4=multiply_serial(m1,m2);
        endTime = System.currentTimeMillis();
        System.out.println("串行计算开始时刻:" + (startTime));
        System.out.println("串行计算结束时刻:" + (endTime));
        System.out.println("串行计算运行时间:" + (endTime - startTime));
        assert(isEqual(m4.getData(),m3.getData()));

    }

}
```

四、运行结果

A:100 *100

B :100 *100

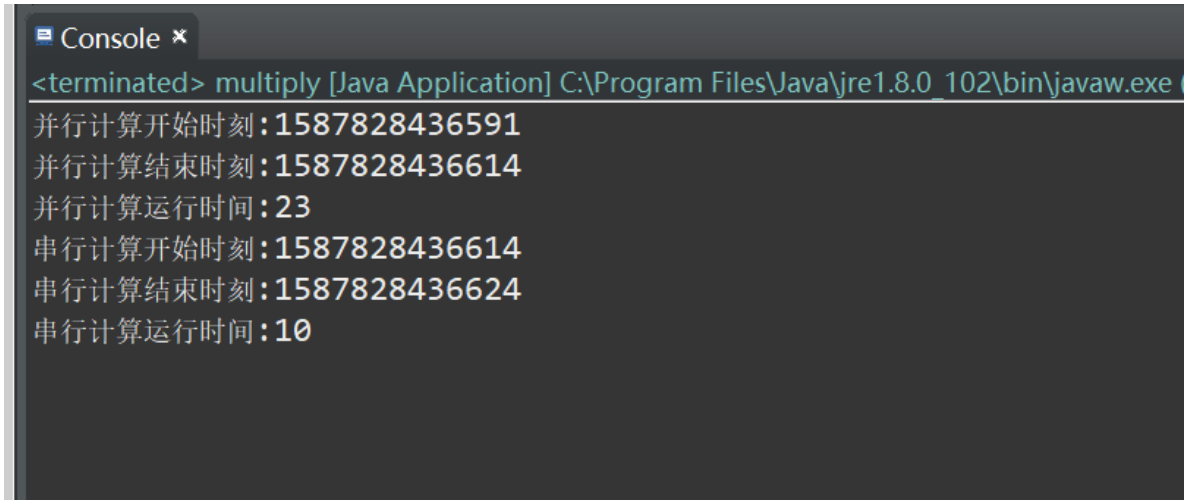
10个线程



A:100 *100

B :100 *100

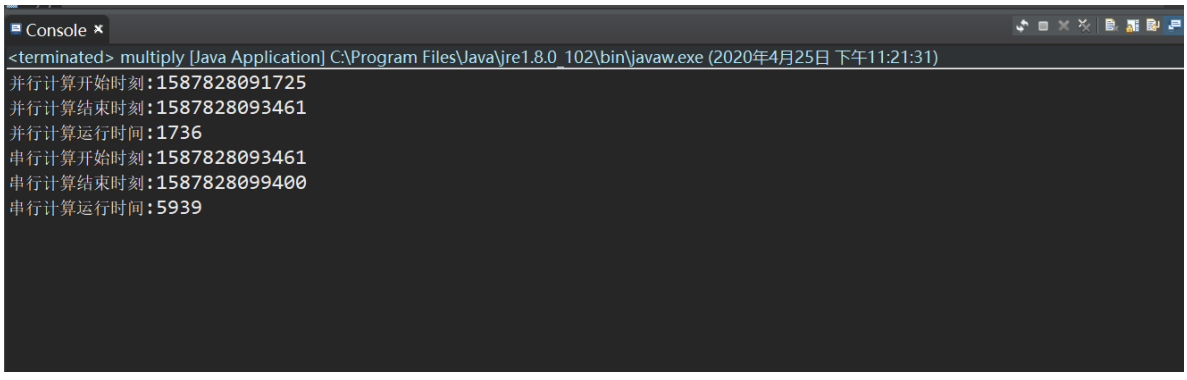
5个线程



A:1000 *1000

B :1000 *1000

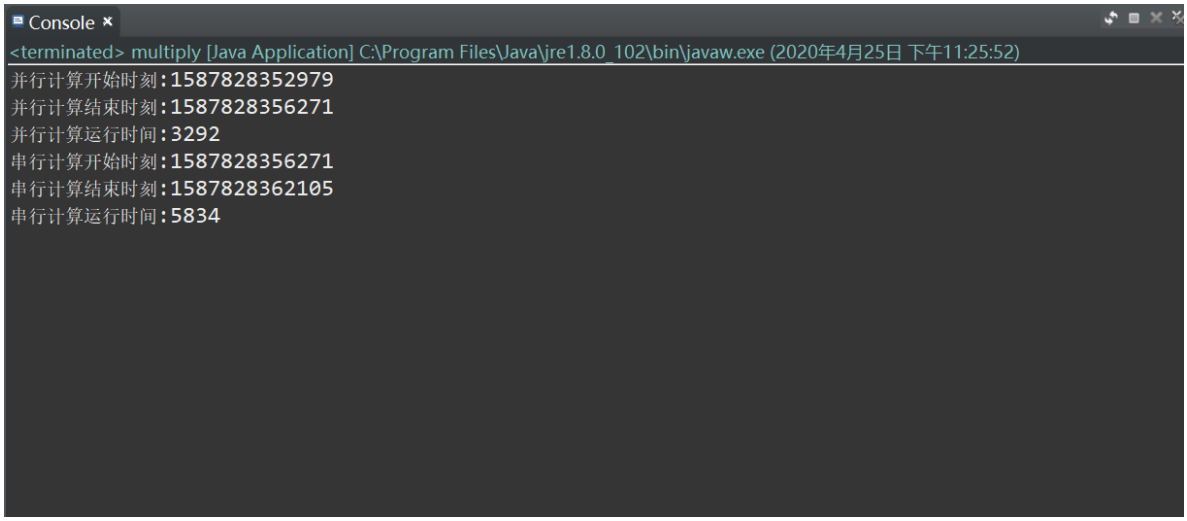
10个线程



A:1000 *1000

B :1000 *1000

5个线程

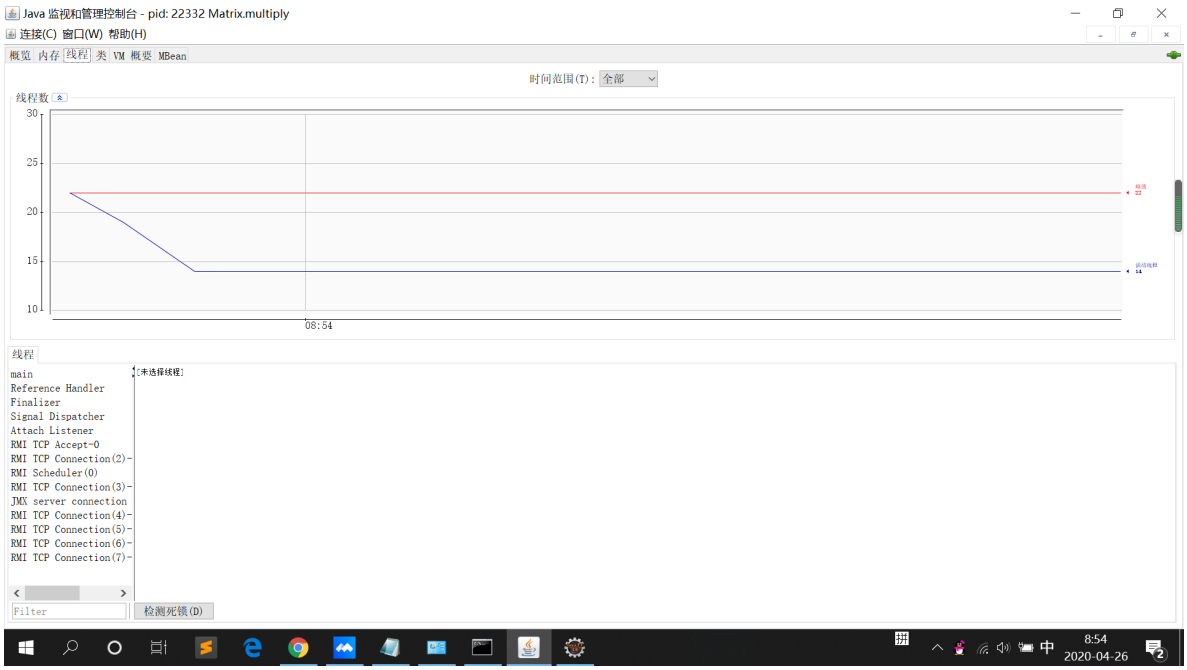


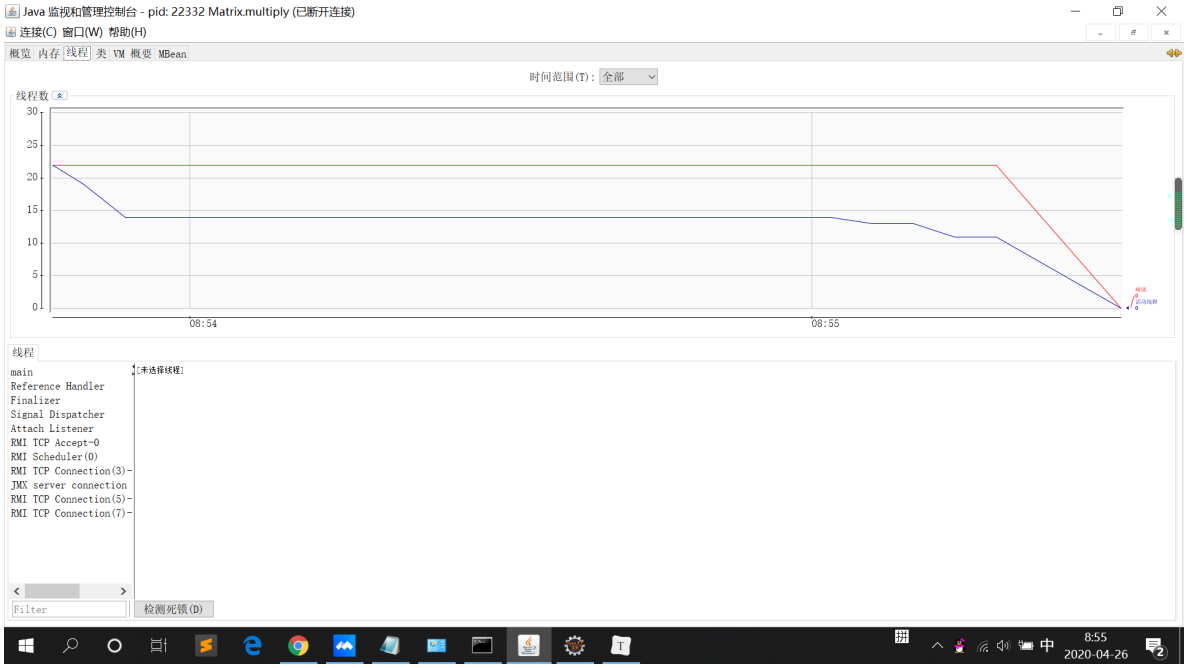
A:2000 *2000

B :2000 *2000

10个线程

jconsole





分析

在测试维度较小时，并行时间大于串行时间，线程数量对并行时间影响不大，猜想创建线程和上下切换耗费时间大于直接计算了

在测试维度较大时，并行时间少于串行时间，线程越多，并行计算速度越快