# 实验二 多线程编程

姓名： 高歌 学号： 2030416018 专业： 软件工程

**一、实验目的**

1. 掌握进程和线程的概念、区别和联系。

2. 理解多线程编程的特点。

3. 掌握常用用户级线程库（如JAVA线程库、Pthread）中多线程编程基础。

**二、实验内容**

使用用户级线程库编写多线程程序，要求实现16×16、64×64、128×128、512×512或更高阶矩阵的相乘运算，并统计计算时间。最后进行总结比较，说明线程数目与运算时间的关系。线程库可采用JAVA线程库或Pthread线程库。

**三、实验平台**

在此说明你所使用的实验环境，包括操作系统版本、线程库、CPU的个数等内容。

计算机型号：Lenovo Y7000

CPU参数：Intel i5-10300H，4核心8线程

内存容量：16G

操作系统版本：Windows 11 家庭版 22H2

线程库：Java标准线程库（使用的Java版本为Java 17）

**四、实验步骤**

分别说明采用什么方法实现多线程计算各个矩阵运算。

大致思路：矩阵乘法的一般思路是对两个矩阵的每一行和每一列的对应元素依次相乘，时间复杂度为O(n³)，如下面的Java代码所示。

for (int i = 0; i < matrix1.length; i++) {

    int sum = 0;

    for (int j = 0; j < matrix2[0].length; j++) {

        for (int k = 0; k < matrix1[0].length; k++) {

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

        }

        result[i][j] = sum;

    }

}

为了简单起见，使用分块的快速矩阵乘法不做考虑。

在本次实验中，考虑将行列对应元素依次相乘的部分放到线程中执行，因此构建类RowColMultiplier，并实现Runnable接口以实现多线程计算。

public static class RowColMultiplier implements Runnable {

    private final double[][] matrix1;

    private final int row;

    private final double[][] matrix2;

    private final int column;

    private final double[][] result;

public RowColMultiplier(double[][] result,

double[][] matrix1,

int row,

double[][] matrix2,

int column) {

        this.matrix1 = matrix1;

        this.row = row;

        this.matrix2 = matrix2;

        this.column = column;

        this.result = result;

    }

    @Override

    public void run() {

        double sum = 0;

        for (int i = 0; i < matrix1[row].length; i++) {

            sum += matrix1[row][i] \* matrix2[i][column];

        }

        result[row][column] = sum;

    }

}

然后创建类ParallelMultiplier，其构造器接受一个参数nThreads，用以指定同时使用多少个线程进行运算；其还有一个实例方法multiply，接受两个矩阵并返回它们相乘的结果。

public static class ParallelMultiplier {

    private final int nThreads;

    private final Thread[] threads;

    public ParallelMultiplier(int nThreads) {

        this.nThreads = nThreads;

        threads = new Thread[nThreads];

    }

    public double[][] multiply(double[][] matrix1, double[][] matrix2) {

        if (matrix1[0].length != matrix2.length) {

            throw new IllegalArgumentException(

"Matrix1 column count must be equal to matrix2 row count");

        }

        double[][] result = Matrices.zeroes(

matrix1.length, matrix2[0].length);

        int n = 0;

        for (int i = 0; i < matrix1.length; i++) {

            for (int j = 0; j < matrix2[0].length; j++) {

                threads[n] = new Thread(

new RowColMultiplier(result, matrix1, i, matrix2, j));

                threads[n].start();

                n = (n + 1) % nThreads;

                if (n == 0) {

                    waitForThreads(threads);

                }

            }

        }

        return result;

    }

    private void waitForThreads(Thread[] threads) {

        for (Thread thread : threads) {

            try {

                thread.join();

            } catch (InterruptedException e) {

                e.printStackTrace();

*// restore interrupted status*

                Thread.currentThread().interrupt();

            }

        }

    }

}

当ParallelMultiplier初始化时，会根据nThreads的值创建对应大小的线程数组。multiply方法一次创建nThreads个RolColMultiplier线程并运行，然后等待这些线程全部结束（调用Thread.join()）后创建下一批线程并运行，直到矩阵的各行各列组合全部计算完毕后返回结果。

为了方便起见，定义工具类Matrices，用于生成矩阵。

public static class Matrices {

    private static final Random random = new Random();

    private Matrices() {

        throw new UnsupportedOperationException("Utility class");

    }

    public static double[][] read(int size, String path) {

        return read(size, size, path);

    }

    public static double[][] read(int row, int col, String path) {

        double[][] result = new double[row][col];

        try (Scanner scanner = new Scanner(new File(path))) {

            for (int i = 0; i < row; i++) {

                for (int j = 0; j < col; j++) {

                    result[i][j] = scanner.nextDouble();

                }

            }

        } catch (FileNotFoundException e) {

            e.printStackTrace();

        }

        return result;

    }

    public static double[][] random(int size) {

        return random(size, size);

    }

    public static double[][] random(int row, int col) {

        double[][] result = new double[row][col];

        for (int i = 0; i < row; i++) {

            for (int j = 0; j < col; j++) {

                result[i][j] = random.nextDouble();

            }

        }

        return result;

    }

    public static double[][] zeroes(int size) {

        return zeroes(size, size);

    }

    public static double[][] zeroes(int row, int col) {

        return full(row, col, 0);

    }

    public static double[][] full(int size, double value) {

        return full(size, size, value);

    }

    public static double[][] full(int row, int col, double value) {

        double[][] result = new double[row][col];

        for (int i = 0; i < row; i++) {

            for (int j = 0; j < col; j++) {

                result[i][j] = value;

            }

        }

        return result;

    }

}

然后在main方法中编写相关逻辑：每次生成两个大小为MATRIX\_SIZE的随机矩阵matrix1和matrix2，然后创建nThreads为THREAD\_COUNT的ParallelMultiplier，调用multiply方法计算两个矩阵并记录用时。重复REPEAT\_TIMES次，统计平均用时。

public static void main(String[] args) {

    long totalTime = 0;

    for (int count = 0; count < REPEAT\_TIMES; count++) {

        double[][] matrix1 = Matrices.random(MATRIX\_SIZE);

        double[][] matrix2 = Matrices.random(MATRIX\_SIZE);

        ParallelMultiplier parallelMultiplier =

        new ParallelMultiplier(THREAD\_COUNT);

        long start = System.currentTimeMillis();

        double[][] result = parallelMultiplier.multiply(matrix1, matrix2);

        long end = System.currentTimeMillis();

        totalTime += (end - start);

*// print result when all threads are finished*

        if (count == REPEAT\_TIMES - 1) {

            for (int i = 0; i < result.length; i++) {

                for (int j = 0; j < result[i].length; j++) {

                    System.out.printf("%.2f ", result[i][j]);

                    if (j == 5) {

                        System.out.print("...");

                        break;

                    }

                }

                System.out.println();

                if (i == 5) {

                    System.out.println("...");

                    break;

                }

            }

        }

    }

System.out.println("Average time spent: " +

totalTime / 1000.0 / REPEAT\_TIMES + "s");

}

全部代码如下所示。

import java.util.Random;

public class MatrixCalculate {

    private static final int MATRIX\_SIZE = 128;

    private static final int THREAD\_COUNT = 4;

    private static final int REPEAT\_TIMES = 10;

    public static class Matrices {

        private static final Random random = new Random();

        private Matrices() {

            throw new UnsupportedOperationException("Utility class");

        }

        public static double[][] random(int size) {

            return random(size, size);

        }

        public static double[][] random(int row, int col) {

            double[][] result = new double[row][col];

            for (int i = 0; i < row; i++) {

                for (int j = 0; j < col; j++) {

                    result[i][j] = random.nextDouble();

                }

            }

            return result;

        }

        public static double[][] zeroes(int size) {

            return zeroes(size, size);

        }

        public static double[][] zeroes(int row, int col) {

            return full(row, col, 0);

        }

        public static double[][] full(int size, double value) {

            return full(size, size, value);

        }

        public static double[][] full(int row, int col, double value) {

            double[][] result = new double[row][col];

            for (int i = 0; i < row; i++) {

                for (int j = 0; j < col; j++) {

                    result[i][j] = value;

                }

            }

            return result;

        }

    }

    public static class ParallelMultiplier {

        private static class RowColMultiplier implements Runnable {

            private final double[][] matrix1;

            private final int row;

            private final double[][] matrix2;

            private final int column;

            private final double[][] result;

            public RowColMultiplier(double[][] result,

                        double[][] matrix1,

                        int row,

                        double[][] matrix2,

                        int column) {

                this.matrix1 = matrix1;

                this.row = row;

                this.matrix2 = matrix2;

                this.column = column;

                this.result = result;

            }

            @Override

            public void run() {

                double sum = 0;

                for (int i = 0; i < matrix1[row].length; i++) {

                    sum += matrix1[row][i] \* matrix2[i][column];

                }

                result[row][column] = sum;

            }

        }

        private final int nThreads;

        private final Thread[] threads;

        public ParallelMultiplier(int nThreads) {

            this.nThreads = nThreads;

            threads = new Thread[nThreads];

        }

        public double[][] multiply(double[][] matrix1, double[][] matrix2) {

            if (matrix1[0].length != matrix2.length) {

                throw new IllegalArgumentException(

                "Matrix1 column count must be equal to matrix2 row count");

            }

            double[][] result = Matrices.zeroes(matrix1.length, matrix2[0].length);

            int n = 0;

            for (int i = 0; i < matrix1.length; i++) {

                for (int j = 0; j < matrix2[0].length; j++) {

                    threads[n] = new Thread(new RowColMultiplier(

                    result, matrix1, i, matrix2, j));

                    threads[n].start();

                    n = (n + 1) % nThreads;

                    if (n == 0) {

                        waitForThreads(threads);

                    }

                }

            }

            return result;

        }

        private void waitForThreads(Thread[] threads) {

            for (Thread thread : threads) {

                try {

                    thread.join();

                } catch (InterruptedException e) {

                    e.printStackTrace();

*// restore interrupted status*

                    Thread.currentThread().interrupt();

                }

            }

        }

    }

    public static void main(String[] args) {

        long totalTime = 0;

        for (int count = 0; count < REPEAT\_TIMES; count++) {

            double[][] matrix1 = Matrices.random(MATRIX\_SIZE);

            double[][] matrix2 = Matrices.random(MATRIX\_SIZE);

            ParallelMultiplier parallelMultiplier =

            new ParallelMultiplier(THREAD\_COUNT);

            long start = System.currentTimeMillis();

            double[][] result = parallelMultiplier.multiply(matrix1, matrix2);

            long end = System.currentTimeMillis();

            totalTime += (end - start);

*// print result when all threads are finished*

            if (count == REPEAT\_TIMES - 1) {

                for (int i = 0; i < result.length; i++) {

                    for (int j = 0; j < result[i].length; j++) {

                        System.out.printf("%.2f ", result[i][j]);

                        if (j == 5) {

                            System.out.print("...");

                            break;

                        }

                    }

                    System.out.println();

                    if (i == 5) {

                        System.out.println("...");

                        break;

                    }

                }

            }

        }

        System.out.println("Average time spent: " +

        totalTime / 1000.0 / REPEAT\_TIMES + "s");

    }

}

**四、实验结果**

在此报告各矩阵运算所花费的时间

16×16的矩阵乘法计算1000次取平均值，64×64计算100次，128×128计算10次，512×512计算1次，1024×1024计算1次。相乘的两个矩阵中的每一元素均为随机生成的浮点数。为了避免缓存造成的误差，测试时每次均使用生成的随机数据，不使用提供的数据。

下为运算时间。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 线程数 | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| 16×16 | 0.0317s | 0.0226s | 0.0184s | 0.0173s | 0.0167s | 0.0162s | 0.0160s | 0.0159s |
| 64×64 | 0.5336s | 0.3703s | 0.2993s | 0.2791s | 0.2658s | 0.2592s | 0.2557s | 0.2573s |
| 128×128 | 2.0946s | 1.4496s | 1.1839s | 1.1153s | 1.0656s | 1.0361s | 1.0285s | 1.0201s |
| 512×512 | 34.670s | 24.017s | 19.447s | 17.895s | 17.155s | 16.764s | 16.552s | 16.394s |
| 1024×1024 | 153.33s | 102.69s | 82.909s | 75.769s | 70.433s | 68.951s | 68.883s | 68.124s |

**五、实验总结与思考**

（1）总结各阶矩阵运算的时间。

各阶矩阵运算所需时间（以线程数8为准）的比值大致呈规模倍数²的关系，如T(64×64)/T(16×16)=4²，T(128×128)/T(64×64)=2²。

该结果与理论结果不符（因为在本次实验的实现中矩阵运算的时间复杂度应仍是O(n³)），推测可能是由于线程创建与销毁的开销太大，以至于在测试使用的矩阵规模下远远超过了计算矩阵对应行列相乘所需要的时间（经测试确实如此，单线程矩阵计算的耗时约为多线程计算耗时的1/500）。

按这种猜测来看，由于创建和销毁进程为主要耗时，因此可将在测试使用的矩阵规模下任意规模的对应行列计算用时视为相同，这样一来矩阵运算的所需时间可视为O(n²)，与实际结果相吻合。

（2）总结说明线程数目与运算时间之间的关系。

观察到在上述结果中，当线程数小于等于8时，运算时间随着线程数的增加相对显著减少，而当线程数大于8时，运算时间随线程数增加减少不显著，并出现了运算时间反而增加的情况。

出现这种情况的原因可能是当线程数小于或等于CPU线程数时，大致为一个系统线程对应一个用户线程，此时线程数的增长能够利用更多硬件资源；而当线程数多于CPU线程数时，一个系统线程对应一个或多个用户线程，此时线程数的增长无法利用更多硬件资源。在本次实验的运行环境中，计算机的CPU线程数为8，运算结果基本符合预期。