Jeffrey Lansford

11/13/2020

Concurrency Lab

For this lab we wanted to do concurrency with threads on a NxN matrix to get the max, min, and the average of the matrix and time how long it took to calculate them. We made N threads to sub calculate the max, min, and a partial average for each row and do final calculation in the main parent thread.

```
//
// // from http://www.letmeknows.com/2017/04/24/wait-for-threads-to-finish-
java/ //
// This is a very small set up to get people started on using threads
//
//
//
//
//  Adopted by Shaun Cooper
//  last updated November 2020
//
//  We need static variable pointers in the main class so that
//  we can share these values with the threads.
//  the threads are address separate from us, so we need to share
//  pointers to the objects that we are sharing and updating

/**
 * Jeffrey Lansford
 * 11/13/2020
 * Concurrecy
 * This program get N (size) from command line input and creates a NxN matrix and
 calculates max, min, and average of the matrix using threads
 *
 */
import java.util.*;
import java.util.ArrayList;
import java.util.concurrent.TimeUnit;

public class MythreadTest {

    private static ArrayList<Thread> arrThreads = new ArrayList<Thread>();
```

```
    // we use static variables to help us connect the threads
    // to a common block
    public static int[][] A;

    // public varible to hold n^2
    public static float total_size;

    // arrays to allow threads to calcualte min. max, and average without a race
    // condition
    public static int min[];
    public static int max[];
    public static float average[];

    // main entry point for the process

    public static void main(String[] args) {
        try {
            // get size from command line input
            int size = Integer.parseInt(args[0]);
            // create the arrays from input
            A = new int[size][size];
            min = new int[size];
            max = new int[size];
            average = new float[size];

            // do n^2 for averge calculations
            total_size = size * (float) size;

            // fill array with random values
            Random rnd = new Random();
            for (int i = 0; i < A.length; i++) {
                for (int j = 0; j < A[i].length; j++) {
                    A[i][j] = (int) ((Math.pow(2, 32 - size) - Math.pow(2, 31 - s
ize)) * rnd.nextFloat()
                            + Math.pow(2, 31 - size));
                }
            }

            // start timer
            long startTime = System.nanoTime();

            // create N threads to work on each row
            for (int i = 0; i < size; i++) {
                Thread T1 = new Thread(new ThreadTest(i));
                T1.start(); // standard thread start
```

```
                arrThreads.add(T1);
            }

            // wait for each thread to complete
            for (int i = 0; i < arrThreads.size(); i++) {
                arrThreads.get(i).join();
            }

            // all the threads are done
            // do final calculations
            // set base values to first element in arrays
            int max_Main = max[0];
            int min_Main = min[0];

            // find max and min in arrays that was calculated from the threads
            for (int i = 0; i < size; i++) {
                max_Main = Math.max(max_Main, max[i]);
                min_Main = Math.min(min_Main, min[i]);
            }

            // gets average of the whole matrix from adding the partial computed
averages
            // from threads
            float total_average = 0;
            for (int i = 0; i < size; i++) {
                total_average += average[i];
            }

            // get end time
            long endTime = System.nanoTime();

            // calculate time elapsed for threads andmain to calculate max, min,
and average
            long timeElapsed = endTime - startTime;

            // show output of time Elapsed, Max, Min, and Average
            System.out.println("Execution time in nanoseconds: " + timeElapsed);
            System.out.println("Execution time in milliseconds: " + timeElapsed /
 1000000);
            System.out.printf("Max: %d\nMin: %d\nAverage: %f\n", max_Main, min_Ma
in, total_average);

            // This for loop will not stop execution of any thread,
            // only it will come out when all thread are executed
            System.out.println("Main thread exiting ");
```

```
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

// each thread should access its row based on "ind"
// and leave results I would suggest in a static array that you need
// to create in MythreadTest
// threads to calculate max, min and partail average per row
class ThreadTest implements Runnable {
    private int i;

    ThreadTest(int ind) {
        i = ind;
    }

    public void run() {
        try {
            // set base max and min
            MythreadTest.max[i] = MythreadTest.A[i][0];
            MythreadTest.min[i] = MythreadTest.A[i][0];

            // get max and min for row, and average of row with sum of i / n^2
            for (int j = 0; j < MythreadTest.A[i].length; j++) {
                MythreadTest.max[i] = Math.max(MythreadTest.A[i][j], MythreadTest
.max[i]);
                MythreadTest.min[i] = Math.min(MythreadTest.A[i][j], MythreadTest
.min[i]);
                MythreadTest.average[i] += MythreadTest.A[i][j] / MythreadTest.to
tal_size;

            }

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

I used a Makefile to do the test runs.

```
# Makefile for running program 5 times with inputs 2, 4, 8, 16
run: compile
```

```
    @for times in 1 2 3 4 5;                \
    do                                      \
        echo "---------------------------------------Run $$times----------------
------------------------";\
        for number in 2 4 8 16;             \
        do                                  \
            echo "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~N: $$number~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~";   \
            java MythreadTest $$number;     \
        done                                \
    done
compile:
    javac MythreadTest.java
```

javac MythreadTest.java

----------Run 1----------

~~~~~~N: 2~~~~~~

Execution time in nanoseconds: 1795928

Execution time in milliseconds: 1

Max: 920882720

Min: 551838208

Average: 700758784.000000

Main thread exiting

~~~~~~N: 4~~~~~~

Execution time in nanoseconds: 974073

Execution time in milliseconds: 0

Max: 263304808

Min: 149306824

Average: 198602752.000000

Main thread exiting

~~~~~~N: 8~~~~~~

Execution time in nanoseconds: 1035084

Execution time in milliseconds: 1
~~~~~~

Max: 16749400

Min: 8445952

Average: 12241174.000000

Main thread exiting

~~~~~~N: 16~~~~~~

Execution time in nanoseconds: 1626093

Execution time in milliseconds: 1

Max: 65483

Min: 32777

Average: 49405.175781

Main thread exiting

--Run 2--

~~~~~~N: 2~~~~~~

Execution time in nanoseconds: 652631

Execution time in milliseconds: 0

Max: 1009998848

Min: 750116352

Average: 937321280.000000

Main thread exiting

~~~~~~N: 4~~~~~~

Execution time in nanoseconds: 767637

Execution time in milliseconds: 0

Max: 265805040

Min: 134809456

Average: 202353248.000000

Main thread exiting

~~~~~~N: 8~~~~~~

Execution time in nanoseconds: 1193956

Execution time in milliseconds: 1

Max: 16677716

Min: 8532868

Average: 12630896.000000

Main thread exiting

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~N: 16~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Execution time in nanoseconds: 1645465

Execution time in milliseconds: 1

Max: 65429

Min: 32934

Average: 49437.015625

Main thread exiting

--Run 3--

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~N: 2~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Execution time in nanoseconds: 378355

Execution time in milliseconds: 0

Max: 894436288

Min: 546198784

Average: 732499968.000000

Main thread exiting

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~N: 4~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Execution time in nanoseconds: 725574

Execution time in milliseconds: 0

Max: 261644048

Min: 137990928

Average: 209834480.000000

Main thread exiting

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~N: 8~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Execution time in nanoseconds: 1005584

Execution time in milliseconds: 1

Max: 16642078

Min: 8406229

Average: 12293746.000000

Main thread exiting

~~~~~~N: 16~~~~~~

Execution time in nanoseconds: 1666428

Execution time in milliseconds: 1

Max: 65348

Min: 32775

Average: 48995.691406

Main thread exiting

--Run 4--

~~~~~~N: 2~~~~~~

Execution time in nanoseconds: 557455

Execution time in milliseconds: 0

Max: 1040987712

Min: 686319264

Average: 853902464.000000

Main thread exiting

~~~~~~N: 4~~~~~~

Execution time in nanoseconds: 804092

Execution time in milliseconds: 0

Max: 268089840

Min: 135819680

Average: 216495168.000000

Main thread exiting

~~~~~~N: 8~~~~~~

Execution time in nanoseconds: 1286410

Execution time in milliseconds: 1

Max: 16745686

Min: 8413849

Average: 12364130.000000

Main thread exiting

~~~~~~N: 16~~~~~~

Execution time in nanoseconds: 1665436

Execution time in milliseconds: 1

Max: 64925

Min: 32860

Average: 48801.234375

Main thread exiting

--Run 5--

~~~~~~N: 2~~~~~~

Execution time in nanoseconds: 538151

Execution time in milliseconds: 0

Max: 952963232

Min: 550368096

Average: 786030208.000000

Main thread exiting

~~~~~~N: 4~~~~~~

Execution time in nanoseconds: 778896

Execution time in milliseconds: 0

Max: 261057960

Min: 135436632

Average: 211427936.000000

Main thread exiting

~~~~~~N: 8~~~~~~

Execution time in nanoseconds: 1233067

Execution time in milliseconds: 1

Max: 16547613

Min: 8394359

Average: 12017372.000000

Main thread exiting

~~~~~~N: 16~~~~~~

Execution time in nanoseconds: 1387094

Execution time in milliseconds: 1

Max: 65408

Min: 32927

Average: 48396.296875

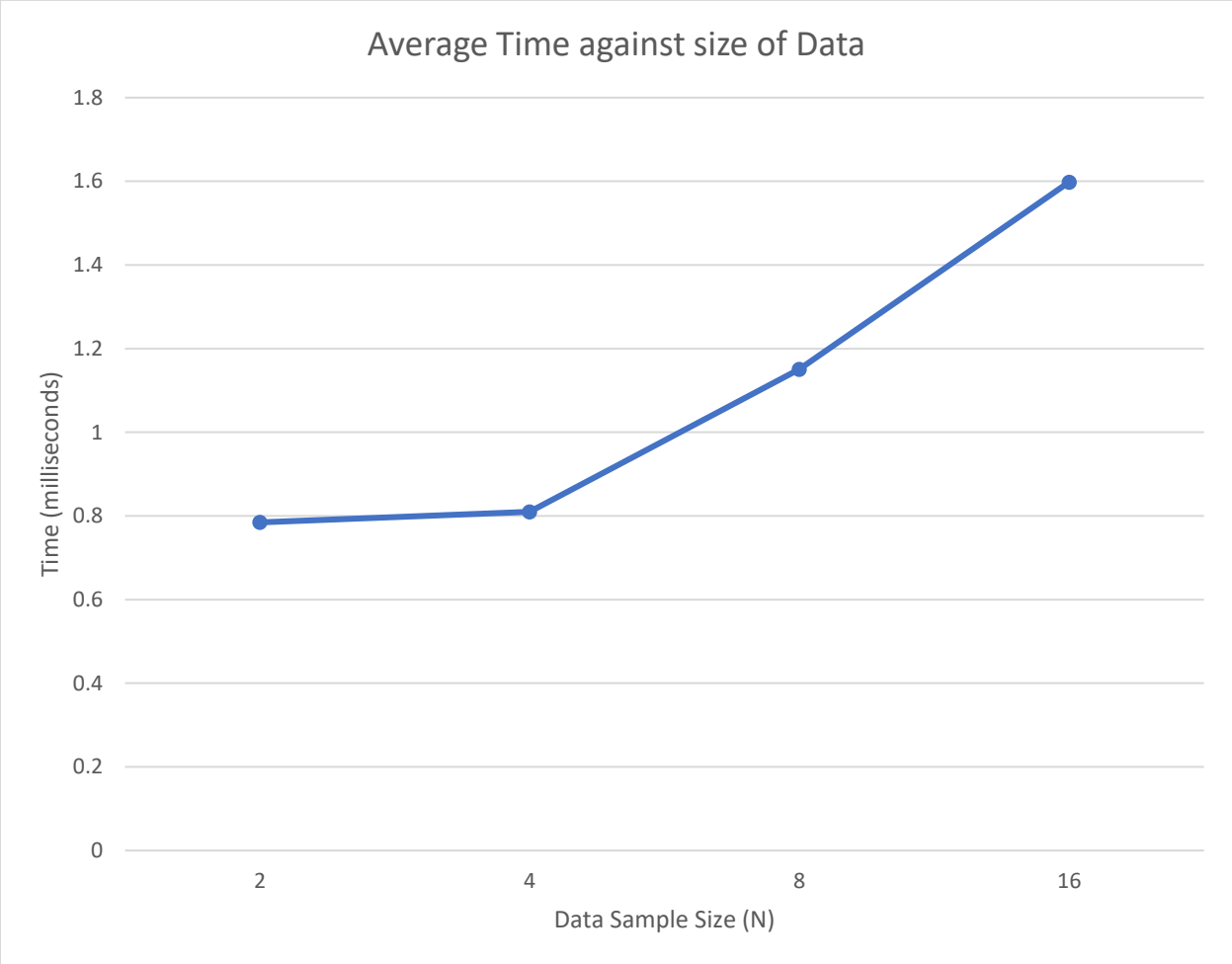
Main thread exiting

	Sample Size (N)	Time (nanoseconds)	Time (microsecond)	Time (milliseconds)
Test Run 1	2	1795928	1795.928	1.795928
	4	974073	974.073	0.974073
	8	1035084	1035.084	1.035084
	16	1626093	1626.093	1.626093
Test Run 2	2	652631	652.631	0.652631
	4	767637	767.637	0.767637
	8	1193956	1193.956	1.193956
	16	1645465	1645.465	1.645465
Test Run 3	2	378355	378.355	0.378355
	4	725574	725.574	0.725574
	8	1005584	1005.584	1.005584
	16	1666428	1666.428	1.666428
Test Run 4	2	557455	557.455	0.557455
	4	804092	804.092	0.804092
	8	1286410	1286.41	1.28641
	16	1665436	1665.436	1.665436
Test Run 5	2	538151	538.151	0.538151
~~~~~~

	4	778896	778.896	0.778896
	8	1233067	1233.067	1.233067
	16	1387094	1387.094	1.387094

Average	2	784504	784.504	0.784504
	4	810054.4	810.0544	0.8100544
	8	1150820.2	1150.8202	1.1508202
	16	1598103.2	1598.1032	1.5981032

Standard Deviation	2	573926.6188	573.9266188	0.573926619
	4	95974.82881	95.97482881	0.095974829
	8	123994.2025	123.9942025	0.123994202
	16	119114.898	119.114898	0.119114898



Looking at the data, we can see that the runtime of going through an NxN Matrix still has a O(n^2). Even though are doing threads to speed up computations, it is still, at least in this data, as n^2. There is some discrepancy in the first data point of N = 2. It seems that maybe when the threads get first created, that the OS puts the threads in a wait status for a little bit until running them, since I use a makefile for running the tests, it could have let the threads run with out blocking them.