Programming # 8 – Concurrency

**Code:**

//

// // 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

/\*

\* Updated by: Tony Maldonado

\* On date: November 15, 2020

\* Input: N, the size of the matrix

\* Output: Min, Max, Avg, and running time

\* Preconditions: N/A

\* Postconditions: N/A

\*/

import java.util.\*;

import java.math.\*;

public class Concurrency {

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 N = 0;

public static int[][] A;

// Create the 1D arrays to store the max, min, and avg

public static int[] Max;

public static int[] Min;

public static float[] Avg;

//main entry point for the process

public static void main(String[] args) {

try {

int localMin = 0;

int localMax = 0;

float localAvg = 0;

// Input from the user:

Scanner scan = new Scanner(System.in);

int size = scan.nextInt();

N = size;

// create the array from input

A = new int[size][size];

Min = new int[size];

Max = new int[size];

Avg = new float[size];

// Get the max and min range

int max = (int) (Math.pow(2, (32 - N)));

int min = (int) (Math.pow(2, (31 -N)));

int range = max - min;

// Now fill the array with the random values

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

for (int k = 0; k < A.length; k++) {

A[i][k] = (int)(range \* Math.random() + 1);

}

}

// Start the timer

long start = 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();

}

// Stop the timer

long end = System.nanoTime();

// Set the localMin to the first index of Min

localMin = Min[0];

// For loops to find the min, max, and avg

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

// Min

if (Min[i] < localMin) {

localMin = Min[i];

}

// Max

if (Max[i] > localMax) {

localMax = Max[i];

}

// Avg

localAvg = localAvg + Avg[i];

}

// Print how long the calculations took the complete

System.out.println("Time to calculate: " + (end - start) + " nanoseconds");

// Print out the min, max, and avg

System.out.println("Min: " + localMin + " Max: " + localMax + " Avg: " + localAvg);

// All the threads are done, do final calculations

System.out.println("Main Thread has N as value " + N);

//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

class ThreadTest implements Runnable {

private int i;

// Some local variables for min, max and avg

private int lMin = 0;

private int lMax = 0;

private float lAvg = 0;

ThreadTest(int ind) {

i = ind;

}

public void run() {

try {

lMin = Concurrency.A[i][0];

System.out.println("Thread is started " + i + " Array is " + Concurrency.A[i][0]);

for (int x = 0; x < Concurrency.N; x++) {

// Find the min

if (Concurrency.A[i][x] < lMin) {

lMin = Concurrency.A[i][x];

}

// Find the max

if (Concurrency.A[i][x] > lMax) {

lMax = Concurrency.A[i][x];

}

// Find the avg

lAvg = lAvg + (Concurrency.A[i][x] / (Concurrency.N \* Concurrency.N));

}

// Store the values in the global vars

Concurrency.Min[i] = lMin;

Concurrency.Max[i] = lMax;

Concurrency.Avg[i] = lAvg;

// Thread.sleep(1000);

System.out.println("Thread is exiting " + i);

}

catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

**Tables/Graphs:**





**Analysis:**

* Based on the graph, we can conclude that the runtime for the different matrix sizes increases linearly.
* Based on the outputs of min, max, and avg for each matrix size, we can also conclude that as N increases, the minimum, maximum, and average decrease because the range [ (2^(32-N)) and (2^(31-N)) ] of random integers that fill the arrays increases as N increases

**Output of Code:**

Text

Description automatically generated

Text

Description automatically generated

A close up of a newspaper

Description automatically generated

A picture containing text

Description automatically generated