PROGRAMMING ASSIGNMENT 2

SYNCHRONIZATION AND COORDINATION USING THREAD POOLS

Version 1.1

DUE DATE: Wednesday, March 6th, 2024 @ 5:00 pm

1 Objective

The objective of this assignment is to get you to be comfortable with threads and synchronization mechanisms. Another objective of this assignment is to introduce the role that data structures and locking mechanisms play in designing concurrent programs.

2 Grading

This assignment will account for **15 points** towards your cumulative course grade. The components of this assignment, and the points breakdown is listed in the remainder of the text. This assignment is to be done individually. The lowest score that you can get for this assignment is 0. The deductions will not result in a negative score.

3 Setting

In this assignment you will be designing a thread pool that manages a set of matrix multiplications. The matrix multiplications will be expressed as a set of tasks that are managed by a thread pool; a given task is performed by a single thread within the pool.

Given a set of 4 input matrices, you will be computing two intermediate matrices *en route* to computing the final product matrix. Here is a compact representation for the goals of this assignment.

- There are 4 input matrices: **A**, **B**, **C**, and **D**
- You will be computing 2 intermediate matrices: **X** and **Y**. where **X**=**AB** and **Y**=**CD**
- The final product matrix **Z** is the product of the two intermediate matrices. **Z**=**XY**

4 Some Basics on Matrix Multiplications

We have included some text (source: Wikipedia) to describe the basics of matrix multiplication. Most of you are probably incredibly familiar with this concept. The text has been included to ensure completeness of the assignment. Feel free to skip this section if you'd like.

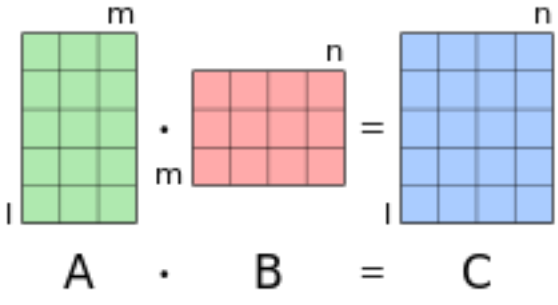

Figure 1: Basic matrix multiplication. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The result matrix has the number of rows of the first and the number of columns of the second matrix.

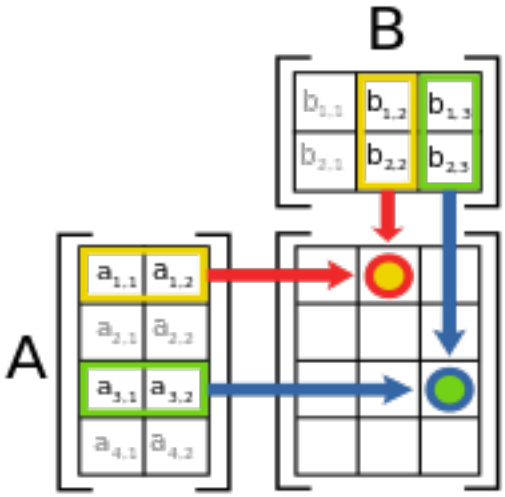

Figure 2: Graphical depiction graphical depiction of how the cells of the product matrix are computed from the original matrices: A and B.

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix, the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

$$\mathbf{C} = \begin{bmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{bmatrix}$$

5 Components & Details

As part of this assignment, you will be implementing two components – a thread pool and task queue – that work in concert with each other to accomplish the objective of computing the products.

5.1 Thread Pool

Broadly a thread pool encapsulates a fixed set of threads. Threads within the thread pool are initialized exactly once. The individual threads then remain in the run state (i.e. they never exit their run() method) till such a time that the process is ready to terminate. For the purposes of this assignment, the number of threads within the thread pool can be anywhere between 1 to 15.

5.2 The Task Queue

Computing the value of each cell within the product matrix should be expressed as a separate *task*. You have the freedom to encode how this task should be expressed. However, you must have a representation of the task that can be unambiguously interpreted by any thread within the pool when it decides to perform the task.

The task queue is backed by a data structure of your choosing. A key requirement is that your task queue must include synchronization and/or locking mechanisms that facilitate concurrent calculation of the product matrices. Incorrect synchronization primitives will result in data corruption (that manifest as incorrect results) or stalls (programs may take an inordinately long time to wrap up the computations). Prolonged execution times are also representative of the case where your lock scope is far too long.

In particular, you are required to design a synchronization scheme so that the worker thread initiates a task only when it is ready. The notion of readiness is based either on when the initialization completes and when the necessary row & column is ready.

One measure of the effectiveness of your synchronization schemes is how well your programs scale as the number of threads increase. Note that as you increase the number of threads, beyond a certain number the execution times will actually increase as context-switching overheads start to dominate.

5.3 Initialization of the Input Matrices A, B, C, and D

The input matrices are initialized using a random number generator, which has been given a starting seed from the command line. Pass the seed into an instance of the java.util.Random class. The input matrices should be initialized in sequence – A, then B, then C, and then D using the same Random object. Within each input matrix, the elements must be initialized from the top row (left to right) on to the bottom rows. The elements in your input matrices are signed integers, so the input matrices can have a mix of positive and negative values. You should bound this random integer to be within the range of [-1000, 1000]. This initialization scheme ensures that multiple, successive runs of your program with the same random seed will result in the same initializations.

For the purposes of this assignment, we are dealing with square matrices (with equal number of rows and columns) that can have anywhere from 300 to 3000 rows.

5.4 Correctness Verification

Here's a quick way to check whether you have correctness errors. A summation of all the elements of the final matrix, Z should result in the same number if you are executing your thread pool with different threads (1 or 10) with the same random seed.

6 Parameters and Program Execution

Please call the primary program that triggers program execution: **`MatrixThreads`** in the package **`csx55.threads`**.

Here are the arguments that will be specified during program execution

`java csx55.threads.MatrixThreads thread-pool-size matrix-dimension seed`

e.g. **`java csx55.threads.MatrixThreads 8 3000 31459`**

`thread-pool-size`	This parameter refers to the thread pool size and represents the number of threads that will be created upon start up. Once started, the threads must never exit their `run()` method till such time that the entire computation has been completed.
`matrix-dimension`	Note that in this assignment we are working only with square matrices i.e., matrices where the number of rows and columns are identical. This is why you only need to specify one value for the matrix dimension. For example, if the matrix-dimension is 1000, each of your input matrices **A**, **B**, **C**, and **D** have a dimensionality of 1000 x 1000 with a million elements.
`seed`	The seed refers to the seed for your random number generator. The random number generator is used to initialize the elements in your input matrices: **A**, **B**, **C**, and then **D**. Each matrix is initialized in row-major format with the first row being initialized left-to-right (first column to the last column). Initializations proceed from the top row to the bottom row.

The output of your matrix multiplications should indicate only the following progress elements:

Example Output:
Dimensionality of the square matrices is: 3000
The thread pool size has been initialized to: 8

Sum of the elements in input matrix A = 3409964
Sum of the elements in input matrix B = 3799344
Sum of the elements in input matrix C = 4095260
Sum of the elements in input matrix D = 626540

Calculation of matrix X (product of A and B) complete – sum of the elements in X is: -37432324759
Time to compute matrix X: 3.440 s

Calculation of matrix Y (product of C and D) complete – sum of the elements in Y is: -79329110607
Time to compute matrix Y: 3.735 s

Calculation of matrix Z (product of X and Y) complete – sum of the elements in Z is: -3449983994057

Time to compute matrix Z is: 3.723 s

Cumulative time to compute matrixes X, Y, and Z using a thread pool of size = <size> is : 10.898 s

7 Points distribution:

2 points	Initialization of the matrices deterministically using the random number generator. Multiple runs of the program with the same random seed should produce the set of initializations for the matrices.
1 point	Setting up of the thread pool where threads are created only once.
1 point	For print the required (and only the required) diagnostic elements for verifying program execution.
3 points	Correctness of the results: 1 point each for getting the products of AB and CD correct. 2 points for getting the product matrix Z correct.
3 points	Correct, non-stalling execution of program in a multi-threaded environment
3 points	Program executes correctly with thread pools of different sizes for the same seed.
2 points	Program executes faster with 10 threads than it does for 1 thread.

8 Third-party libraries and restrictions:

The assignment must be implemented using the core packages in Java. However, you cannot use thread pool implementations that are available in the Java language library. You are not allowed to use *any* external jar files. You can discuss the project with your peers at the architectural level, but the project implementation is an individual effort.

9 Milestones:

You have 4 weeks to complete this assignment. The weekly milestones below correspond to what you should be able to complete at the end of every week.

MILESTONE 1 [WEEK 1]: You should be able create your thread pool. Initialization of the input matrices A, B, C, and D based on the random number generator is complete.

MILESTONE 2 [WEEK 2]: Calculation of the matrices with a thread pool size of 1 is complete. This includes registering tasks in the task-queue and ensuring that each task is performed at least once. Concurrent bugs will manifest themselves in two ways: program stalls and incorrect results.

MILESTONE 3 [WEEK 3]: You should be able to contrast the results from thread pools of different sizes. When you have concurrent program execution, your product matrices would be completed at different times and the outputs will reflect that. For example, several times you will be seeing product matrix Y complete before X. You should also be seeing some differences in program execution times when you size your thread pool differently.

MILESTONE 4 [WEEK 4]: Iron out any wrinkles that may preclude you from getting the correct (outputs at all times.

10 What to Submit

Use using **CANVAS** to submit a single .tar file that contains:

- all the Java files related to the assignment (please document your code)
- the `build.gradle` file you use to build your assignment
- a README.txt file containing a manifest of your files and any information you feel the GTA needs to grade your program.

Filename Convention: You may call your support classes anything you like. All classes should reside in a package called **`csx55.threads`**. The archive file should be named as <FirstName>-<LastName>-HW2.tar. For example, if you are Cameron Doe then the tar file should be named Cameron-Doe-HW2.tar.

11 Change History

Version	Date	Change
1.0	2/7/2024	First public release of the assignment.
1.1	2/13/2024	Fixed a typo in the package name in section 10.