SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Lab 1: Java Threads Practice

Due: 4:00 PM, Tuesday, September 17, 2024

In this lab, we will work together to parallelize a program using Java Threads.

You will work individually on this lab and submit your own final products, but this will really be a collaborative class effort.

Learning goals:

1. To practice using Java Threads.
2. To experiment with a few approaches to parallelizing a problem.

The code for this lab will be graded as a practice program.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `javathreads-lab-yourgitname`, for this lab. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

There are only a few quick questions to answer, so we will answer those in your repository's `README.md` file rather than starting a shared document.

An Example Computation

In your repository, you should find a Java program that will perform a computation related to Goldbach's Conjecture. The conjecture has to do with writing any even number greater than 2 as the sum of 2 prime numbers.

This code has two command-line parameters related to multithreading, but they have no effect. For now, just specify any positive integer as the first and any word as the second. Remaining command-line parameters are the numbers for which the pair of primes is to be computed.

This program reports timings about the solutions it completed.

Question 1: Which variables are used to compute the timings, and what does each do? (2 points)

Question 2: Run the program, trying out many groups of even numbers as parameters. Some numbers will take longer to process than others. What causes a number to take longer? (1 point)

Question 3: Find a group of numbers that takes between 1 and 5 seconds each to process. (1 point)

Adding Threads

Based on the class example

`https://github.com/SienaCSISParallelProcessing/JavaThreadsIntro`

we will first add basic threading capabilities to the `Goldbach.java` program in your respository.

Like in the class example, we will want each thread to have a unique thread ID in the range from 0 to the `numThreads` - 1. The `WorkerThread` class from the example is included at the bottom of `Goldbach.java`.

Before having the program do any useful work, let's set up the threads (3 points):

- create the `numThreads` threads (as `WorkerThread` instances),
- have the `run` method of each call the function `doThreadWork`, a stub of which is also provided,
- have the original thread `join` with each of the threads to make sure we wait for them to complete,
- add a printout to `doThreadWork` that just reports its thread ID, and
- comment out the loop in `main` that computes the answers.

Compile and run and make sure you get messages from the appropriate number of threads, as specified by the first command-line parameter. Don't worry about the garbage output you'll also see at the end.

Computation by the Threads

Next, we restore the computation to the program. We'll start by putting the original computation loop into `doThreadWork`.

Question 4: In this case, which thread will compute which results? (2 points)

This is not good, so we will modify the loop so each value in the `vals` array will be computed by the thread such that the index into the array mod the number of threads is equal to the thread ID. (5 points)

Compile and run this version of the program. Run with 12 numbers large enough to take about 1 second each, and for numbers of threads ranging from 1 to 6.

Question 5: Which numbers in `vals` are computed by each thread for each number of threads, and is this decomposition of the work the same every time you run for the same number of values with the same number of threads? (1 point)

Question 6: Describe and briefly explain the timings you see. (2 points)

Alternate Decompositions

Next, we will add other decomposition options. The second command-line parameter, stored as the `mode` variable, will select which one.

We will have three options. The one we have so far could be described as an interleaved decomposition, so we'll use a `mode` of `"interleaved"` for this. The second is a block decomposition, which we'll select by providing a `mode` of `"block"`, and the third is a bag of tasks decomposition, when `mode` is specified as `"bag"`.

Add code in `main` to check that the `mode` variable is set to one of these values, printing an error message and exiting if not. Modify `doThreadWork` to have a conditional structure that will execute your existing code for the interleaved decomposision, and blocks where you can add code for the other two modes.

For the block mode, each thread will be assigned a contiguous block of indices. Each thread should get the same number of indices, except that any that cannot be divided evenly should be distributed one per thread, starting with thread 0. The first block will consist of the lowest numbered indices and be assigned to thread 0, the next block to 1, etc.

Question 7: What indices would be computed by each thread if we have 12 numbers and 4 threads? 12 numbers and 5 threads? (1 point)

Write code in the `doThreadWork` method for block mode such that each thread independently computes its own range of indices, and print those out to make sure all indices are assigned to the appropriate thread. Once you are sure that's the case, implement code to perform those computations. (5 points)

Compile and run this version of the program. Run with 12 numbers large enough to take about 1 second each, and for numbers of threads ranging from 1 to 6.

Question 8: Describe and briefly explain the timings you see. (2 points)

Finally, we will add the *bag of tasks* decomposition approach.

We will work together to develop the code in `doThreadWork` and other parts of the program to support this. (5 points)

Compile and run this version of the program. Run with 12 numbers large enough to take about 1 second each, and for numbers of threads ranging from 1 to 6.

Question 9: Describe and briefly explain the timings you see. (2 points)

Question 10: Why did we need to use the `synchronized` keyword to get a correct implementation? (2 points)

Question 11: Give a specific example of what could go wrong if we were to omit the `synchronized` keyword. (2 points)

Question 12: Discuss the relative advantages and disadvantages of the three decomposition ap-

proaches we have implemented for this program, specifically thinking about this problem and different sets of numbers that might be presented to the program. (4 points)

Submission

Commit and push!

Grading

This assignment will be graded out of 40 points.

Feature	Value	Score
Question 1	2	
Question 2	1	
Question 3	1	
Basic thread code	3	
Question 4	2	
Interleaved decomposition code	5	
Question 5	1	
Question 6	2	
Question 7	1	
Block decomposition code	5	
Question 8	2	
Bag of tasks decomposition code	5	
Question 9	2	
Question 10	2	
Question 11	2	
Question 12	4	
Total	40	