Operating Systems and Concurrency

Week 2 lab exercise

Introduction

For this exercise you will create a program that computes a value by parallelising a task using threads.

The task itself is the simple computation of a mean value of a list of numbers. This will be achieved by splitting the input data into *slices* and then creating a thread to compute the mean for each slice. Finally, the results from all threads will be combined to produce an overall mean for the list of numbers. Only slices of equal length will be dealt with.

This task is an example of an *embarrassingly parallel* task, i.e. it can be divided up and computed easily in parallel. There are a lot of other task that fit this description, for example a lot of computer graphics tasks (which is why GPUs have been capable of parallelism for a long time). The inherent characteristic of such as task is that they involve performing the same operation on members of a large data set. In this case, we will be taking a large list of numbers, dividing it into slices, and then performing the same operation on each of the slices concurrently.

The final part of this exercise is to attempt to test the extent of the parallelisation of the task by the operating system. This involves gathering timing data for a number of runs of the task for increasing numbers of threads to try to experimentally ascertain the number of threads that minimises the overall task run time. The outcome of these experiments will depend very much on the hardware and operating system you use to test this, hence there are no 'correct' results.

If possible, this task is best performed on a multicore computer (dual core and above). If you can't access one of those, don't worry. You can still see the results of this program and we will discuss the results in a tutorial anyway.

Task description

You will create a program that outputs the mean of a large list of numbers. We want the numbers to be large so you should use the Java class `BigInteger` to represent them. You should store the numbers themselves (as type `BigInteger`) in a regular Java array[1]. The maximum size of the list of numbers your program can handle will also depend on the underlying system and JVM settings. You may need to experiment with that (and you don't need to fiddle with JVM settings, etc.). Start with smaller sizes as you get your program to work, and then increase the size incrementally. If you make the size too large, you are likely to get an 'out of heap memory' error. That's fine. Just reduce the size again if that happens. You may be able to get away with the size declared below:

```
private static int INPUT_LIST_SIZE = 15_000_000;
```

[1] Don't forget, the array will actually hold the references to the numbers. The `BigInteger`s themselves will be stored on the heap.

You will need to slice the input array and pass each slice to a separate thread to work on. Thus, the number of slices and number of threads are equal. If you have four slices, you will have four threads. Each thread will compute the mean of the list of numbers it is given. Your program does not need to cope with variable size slices: all slices will have the same length. Of course, this puts some constraints on the input list size and the number of threads.

Each thread will make available the mean value of its slice of the input numbers when it is complete. The main thread will then assemble the results of each thread and compute the overall mean. The main thread can wait for each thread to complete by using the `join()` method. Be aware that all of the threads should be started before calling `join()` on any one of them.

You may decide to approach this overall task your own way. However, below it is split into sub-tasks just in case you would prefer to approach it that way.

Task 1

For the first task, create a class that represents the data. This class should have a `BigInteger` array to hold the numbers. The class should also be able to generate the elements of that array. The easiest way to do this is to iterate through the array setting each element to the value of a loop index, starting at zero. That is, the value of each element is simply the value of its index. This is essentially just a test data set. There is no need to try and create a more complex data set.

The most important task of this class is to create the data slices. The class requires a method that receives an integer specifying how many slices of the data are required and which returns an array of `BigInteger` arrays containing the slices.

For example, if the input array was:

{0,1,2,3,4,5,6,7,8,9}

And the number of slices requested was 5, the slices, hence the return value of the method, would be as follows:

{{0,1}{2,3}{4,5}{6,7}{8,9}}

As stated above, note that for this task you can assume that the input array can always be sliced into equal length slices. This is important for the simplification of the computation of the mean.

Task 2

For the next task, you will create the class that actually does the work of each thread. For this, create a class that implements the `Runnable` interface.

Add a constructor to the class that receives a slice of the data (as created by the class in Task 1). This will, therefore, be an array of `BigInteger`s.

The `run()` method of this class will compute the mean of the numbers in that slice. It can't *return* this value since we can't change the signature of the `run()` method and that has a `void` return value and receives no parameters. Therefore, the `run()` method should store the result in an instance variable for

later retrieval.

The class will, therefore, require a method that returns the result. Of course, the result of calling such a method depends on the timing of the calls to it (it will return zero if we call it too early), so we will make sure that each thread has actually finished in a later task. This method doesn't need to worry about that part of it. It should just return the result when asked for it.

Task 3

For the next task, you will create a program that computes the mean of a list of numbers using the classes created above.

This program needs to:

1. Set up the input data using the class from Task 1.
2. Create the data slices using the class from Task 1.
3. Create a thread for each data slice using the `Runnable` class from Task 2.
4. Start all of the threads.
5. Wait for all of the threads to complete.
6. Gather the results of all of the threads.
7. Compute the mean of the means returned[2].
8. Output the mean to the console.

Remember, all of the threads need to be started and only then should the main thread wait. This can be achieved using the method `join()`: the main thread calls `join()` on all of the threads.

Task 4 [Extension]

This task involves adapting your program from Task 3 so that it can run multiple times, gathering run time data. The purpose of this is to try and ascertain what, if any, run time speed up is achieved by using different numbers of threads.

The basic idea is as follows. Your program will start by selecting a number of slices (threads) to use and then will run that set up 10 times, recording the run time of each run. It will then compute a mean run time from the 10 individual run times. Once this is done, the program will select a different number of slices and run that 10 times, and compute the mean run time. The program should do this for the following numbers of slices: {1,2,4,8,16,32}.

In order to determine the run time of each run you can use `System.nanoTime()`. You can convert the units to milliseconds to make the times more readable. You should record the time using `System.nanoTime()` just before you *start* all of the threads. You should then record the time again when all of the threads have finished (that is, when the main thread is no longer blocking on any `join()` call).

[2]Since the slices are of equal size, this is simply (the sum of the means÷number of slices).

Note that we include a run with only one slice. From the perspective of our program, this is the same as it being single-threaded. This gives us a *baseline* run time to compare with all of the other runs.

When your program has finished running you should have a mean run time for each of the numbers of slices (1, 2, 4, etc.) which you can print to the console.

What result would you expect? The results you get will only give an indication rather than conclusive proof of anything. The results will also depend very largely on the system you are running the program on. Different operating systems and different numbers of cores will affect the results.

What you *might* expect is that there will be some kind of time advantage to running this program with multiple threads rather than with just one, i.e. the run times should be lower if it is threaded. You may find, however, that threads do give a speed up but with diminishing returns, i.e. the run times will plateau and will not be improved with ever more and more threads.

[Ian's note: I ran this program on a dual core MacBook Pro and my results converged on a 'minimum' run time within 4 threads. There was a big reduction in run time between one and two threads though. I then tested it on a newer MacBook Pro with an Apple M1 CPU (which has '8' cores (4 'high-performance' + 4 'high-efficiency')). The overall run times were lower but the same pattern occurred as on the older machine.]