Exercises week 5
Friday 28 September 2018

Goal of the exercises

The goal of this week's exercises is to make sure that you can use tasks and the Java executor framework; and that you can modify and extend a simple processing pipeline using a given blocking queue implementation as well as one from the Java class library. Note that all of Exercise 5.2 may be skipped; the other exercises are more interesting.

Do this first

Get and unpack this week's example code in zip file pcpp-week05.zip on the course homepage.

Exercise 5.1 Use the lecture's example in file TestCountPrimesTasks.java to count prime numbers using tasks and the executor framework instead of explicitly creating threads for each concurrent activity.

1. Using a CachedThreadPool as executor, measure the time consumption as a function of the number of tasks used to determine whether a given number is a prime. The Mark7 method is relevant.
2. Use a WorkStealingPool and repeat the experiments.
3. Use Excel or gnuplot or Google Docs online or some other charting package to make graphs of the execution time as a function of the number of tasks. Do this for the executors you have tried.
4. Reflect and comment on the results; are they plausible? How do they compare with the performance results obtained using explicit threads in last week's exercise? Is there any reasonable relation between the number of threads and the number of cores in the computer you ran the benchmarks on? Any surprises?
5. Java 8 has a class java.util.concurrent.atomic.LongAdder that is a kind of AtomicLong. It may perform much better than LongCounter and AtomicLong when many threads add to it often and its actual value is inspected only rarely. Also, it should perform well in case locking is especially slow on the hardware being used. Replace the use of LongCounter with LongAdder, and rerun the measurements.

Exercise 5.2 (Optional) Use the week 4 benchmarking techniques to measure the time to create, start, and finish a simple task. You may draw inspiration from TestTimeThreads.java used in Exercise 4.2.

For your experiments in questions 5.2.3 and 5.2.4, use a CachedThreadPool or a WorkStealingPool. Using a FixedThreadPool would create many threads up front and therefore allocate a huge amount of memory, and that would distort subsequent measurements.

NB: An Executor and the associated thread pool is a complicated and memory-consuming object, so *create only one Executor to run the measurements in each of those two exercise questions*. If you create a new Executor inside the IntToDoubleFunction then you will just measure the time spent creating and managing the Executors, and most likely your Java process will run out of memory.

To create a single separate executor for each of the exercise questions, use a code pattern like this:

```
{
  ExecutorService executor = Executors.newWorkStealingPool();
  Mark6("Task create submit cancel",
        ...
  );
  executor.shutdownNow();
}
```

This will also shut down the executor after performing the experiments.

1. Measure the time to run a for-loop that increments an AtomicInteger 1000 times.
2. Measure the time to create (but not submit) a Runnable task that increments an AtomicInteger 1000 times.

3. Measure the time to create and submit a Runnable task (that increments an AtomicInteger 1000 times) to get a Future, and then immediately cancel the Future. The cancellation is necessary, otherwise all the tasks submitted by benchmarking will accumulate on the executor, causing very large memory usage.

4. Measure the time to create and submit a Runnable task (that increments an AtomicInteger 1000 times) to get a Future, and then immediately wait for the Future to run by calling `get()` on it.

5. Reflect and comment on the results; are they plausible? How do they compare with the cost of creating, starting and completing threads in last week's Exercise 4.2? Any surprises?

Exercise 5.3 This exercise is about fetching a bunch of web pages. This activity is heavy on input-output and the latency (delay) involved in requests and responses sent over a network. In contrast to the previous exercises, fetching a webpage does not involve much computation; it is an input-output bound activity rather than a CPU-bound activity.

File TestDownload.java contains a declaration of a method `getPage(url, maxLines)` that fetches at most `maxLines` lines of the body of the webpage at `url`, or throws an exception.

1. First, run that code to fetch and print the first 10 lines of `www.wikipedia.org` to see that the code and net connection works.

2. Now write a sequential method `getPages(urls, maxLines)` that given an array `urls` of webpage URLs fetches up to `maxLines` lines of each of those webpages, and returns the result as a map from URL to the text retrieved. Concretely, the result may be an immutable collection of type Map<String,String>. You should use neither tasks nor threads at this point.

   File TestDownload.java contains such a list of URLs; you may remove unresponsive ones and add any others you fancy.

   Call `getPages` on a list of URLs to get at most 200 lines from each, and for each result print the URL and the number of characters in the corresponding body text — the latter is a sanity check to see that something was fetched at all.

3. Use the Timer class — **not** the Mark6 or Mark7 methods — for simple wall-clock measurement (described in the *Microbenchmarks* lecture note from week 4) to measure and print the time it takes to fetch these URLs sequentially. Do not include the time it takes to print the length of the webpages.

   Perform this measurement five times and report the numbers. Expect the times to vary a lot due to fluctuations in network traffic, webserver loads, and many other factors. In particular, the very first run may be slow because the DNS nameserver needs to map names to IP numbers.

   (In principle, you could use Mark6 or Mark7 from the *Microbenchmarks* note to more accurately measure the time to load webpages, but this is probably a bad idea. It would run the same web requests many times, and this might be regarded as a denial-of-service attack by the websites, which could then block requests from your network).

4. Now create a new parallel version `getPagesParallel` of the `getPages` method that creates a separate task (not thread) for each page to fetch. It should submit the tasks to an executor and then wait for all of them to complete, then return the result as a Map<String,String> as before.

   The advantage of this approach is that many webpage fetches can proceed in parallel, even on a single-core computer, because the webservers out there work in parallel. In the old sequential version, the first request will have to fully complete before the second request is even initiated, and so on; meanwhile the CPU and the network sits mostly idle, wasting much time.

   As executor, use a WorkStealingPool if you have Java 8, otherwise a CachedThreadPool. Make sure that the executor is allocated only once, for instance as a static field in the class; do not allocate a fresh executor for each call to `getPagesParallel`.

   Call it as in the previous question, measure the wall-clock time it takes to complete, and print that and the list of page lengths. Repeat the measurement five times and compare the results with the sequential version. Discuss results, in particular, why is fetching 23 webpages in parallel not 23 times faster than fetching them one by one?

Exercise 5.4 Consider the pipeline built in the lecture's file TestPipeline.java to gather and print webpage links.

1. Run it as is, to see that it works on your machine.

   (As you will notice, you need to manually terminate the program by pressing control-C or similar when it has not printed any new results for a while. This is because the "consumers" PageGetter, LinkScanner and LinkPrinter wait forever for more input arriving through their input queue. Fixing this deficiency in a general way seems to complicate the example considerably, so we will live with it).

2. If a webpage contains several occurrences of the same link to another webpage then LinkScanner will "produce" that link multiple times, and LinkPrinter will therefore print it multiple times; this is not desirable. Instead of changing LinkScanner or LinkPrinter, write a new class Uniquifier<T> that consumes items of type T and produces items of type T, but only once each. The class should implement Runnable and its constructor should take as argument an input queue and an output queue, each of type BlockingQueue<T>.

   The uniquifier's `run` method may maintain a HashSet<T> containing the items already seen. When an item received from the input queue is already in the hashset, it is ignored; otherwise it is added to the hashset and also sent to the output queue.

   Insert a Uniquifier<Link> stage, and suitable queues, between the LinkScanner and the LinkPrinter, so that a link `(from,to)` from one webpage to another is printed only once.

3. Change the implementation to submit the UrlProducer, PageGetter, LinkScanner, Uniquifier and LinkPrinter as tasks on an executor service rather create threads from them. If you have Java 8, use a WorkStealingPool, otherwise a CachedThreadPool, as executor service.

   The results should be the same as when using threads. Are they?

4. Now use a FixedThreadPool of size 6 to run the tasks. This should work as before.

5. Now use a FixedThreadPool of size 3 to run the tasks. This probably does not work. What so you observe? Can you explain why?

6. Probably the pipeline's most time-consuming stage is the PageGetter. To improve overall throughput, one may simply create two PageGetter objects (as threads or tasks), both taking input from the BlockingQueue<String> queue and sending output to the BlockingQueue<Webpage> queue.

   Implement this idea. You should get the same results as before, though possibly in a different order. Do you? Why?

7. (Optional) Implement your own bounded blocking queue class BoundedQueue<T>. It must implement our BlockingQueue<T> interface and be able to hold n elements, where n is given as a parameter when the BoundedQueue<T> object is created.

   The queue may use an ArrayList<T> as a cyclic buffer to hold the items. The queue constructor should create it with capacity n and add `null` to it n times; all subsequent use should be through indexing `items.set(i, x)` and `items.get(j)` where `i` and `j` are suitably computed indexes. (It would be more natural to store the items in a standard array `T[]` of size n but due to Java's weak generic types, this causes problems that have nothing to do with concurrency).

   As before, the `put` method should block when the buffer is full, and the `take` method should block when the buffer is empty.

   Explain why your bounded blocking queue works and why it is thread-safe.