Java.concurrent.util

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Java.util.concurrent

- We have seen all the “low level” abstractions for writing concurrent applications
- We have seen other abstractions provided by the java.util.concurrent package
- This package is intended to make life easier when writing concurrent applications
  - developed and peer-reviewed by concurrency experts a lot smarter than us
- There are a LOT of things in it, with MANY options
- Plenty of on-line documentation, tutorials, examples

What’s in the Package?

- What we’ve already seen/mentioned:
  - Locks
  - Condition Variables
  - CyclicBarrier
  - Semaphores
  - Atomic variables
- What here about to see
  - ExecutorService and Future
  - Concurrent/Blocking collections
- A bunch of other things you can do check out on your own
  - CountDownLatch (very barrier-like)
  - Phaser (generalization of barriers)
  - Exchanger (threads “meet” and exchange information)
  - etc..

The Thread Pool Concept

- When writing concurrent applications, one often ends up spawning off many short-lived "worker" threads that do some useful tasks, throughout program execution
- Doing this by hand has several drawbacks
  - It requires code to be written
    - We now know how to do it, but we're lazy?
  - One may want to control the maximum number of threads that are running simultaneously to avoid overload
    - e.g., have as many running threads as cores
  - Creating threads is a bit expensive, and it may be a better idea to keep threads "around" and reuse them
    - Still more code we don't want to write
- What we really need is a "thread pool"

Pools with ExecutorService

- A Thread Pool is a (possibly fixed) set of threads
- Example:
  - I have a pool that contains 3 threads
  - I keep giving things to do to the pool
  - Up to 3 threads can be running at a given time
  - Extra things to do are queued and will be started later when previous threads have completed
- java.concurrent.util provides just the right thing here: ExecutorService
- Let's see the code…

ExecutorService Interface

```
public class Task implements Runnable {
  private String message;
  private int iterations;

  public Task(String s, int n) {
    message = s; iterations = n;
  }

  public void run() {
    for (int i=0; i < iterations; i++) {
      System.out.println(message);
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) { }
    }
  }
}
```

```
import java.util.concurrent.*;
. . .
ExecutorService pool;
pool =
     Executors.newFixedThreadPool(3);

pool.execute(new Task(“three”,3));
pool.execute(new Task(“two”,2));
pool.execute(new Task(“five”,5));
pool.execute(new Task(“six”,6));
pool.execute(new Task(“one”,1));

pool.shutdown();
```

ExecutorService Interface

- The shutdown() method prevents new tasks from being submitted, but running and submitted tasks run to completion
- The shutdownNow() method prevents new tasks from being submitted, but (attempts) to let only currently running tasks finish
- The isTerminated() method returns true is there is no pending task
- It is possible to create thread pools that can grow, and tons of other bells and whistles that you can discover in the on-line documentation

Callable: more than Runnable

- What if our "task" abstraction is one in which a task returns something (i.e., some output)?
- The concept of "a thread that returns something" is provided by Java: Callable

```
public class Task implements Callable<SomeObject> {

  public SomeObject call() {
     // Do some work
    return new SomeObject(…);
  }
}
```

ExecutorService and Callable

- We can use an ExecutorService to manage the execution of Callables
- In that case one uses the ExecutorService.submit() method
- The call returns immediately with a Future
- A Future is an object that represents the result of an asynchronous computation
- One can then do various things on the Future:
  - check if it's done, wait for it to be done, wait for it to be done but with a time-out, etc.
- Let's see an example…

Executor, Callable, Future

```
public class Task implements Callable<String> {
    public String call() {
        return new String("stuff");
    }
}
```

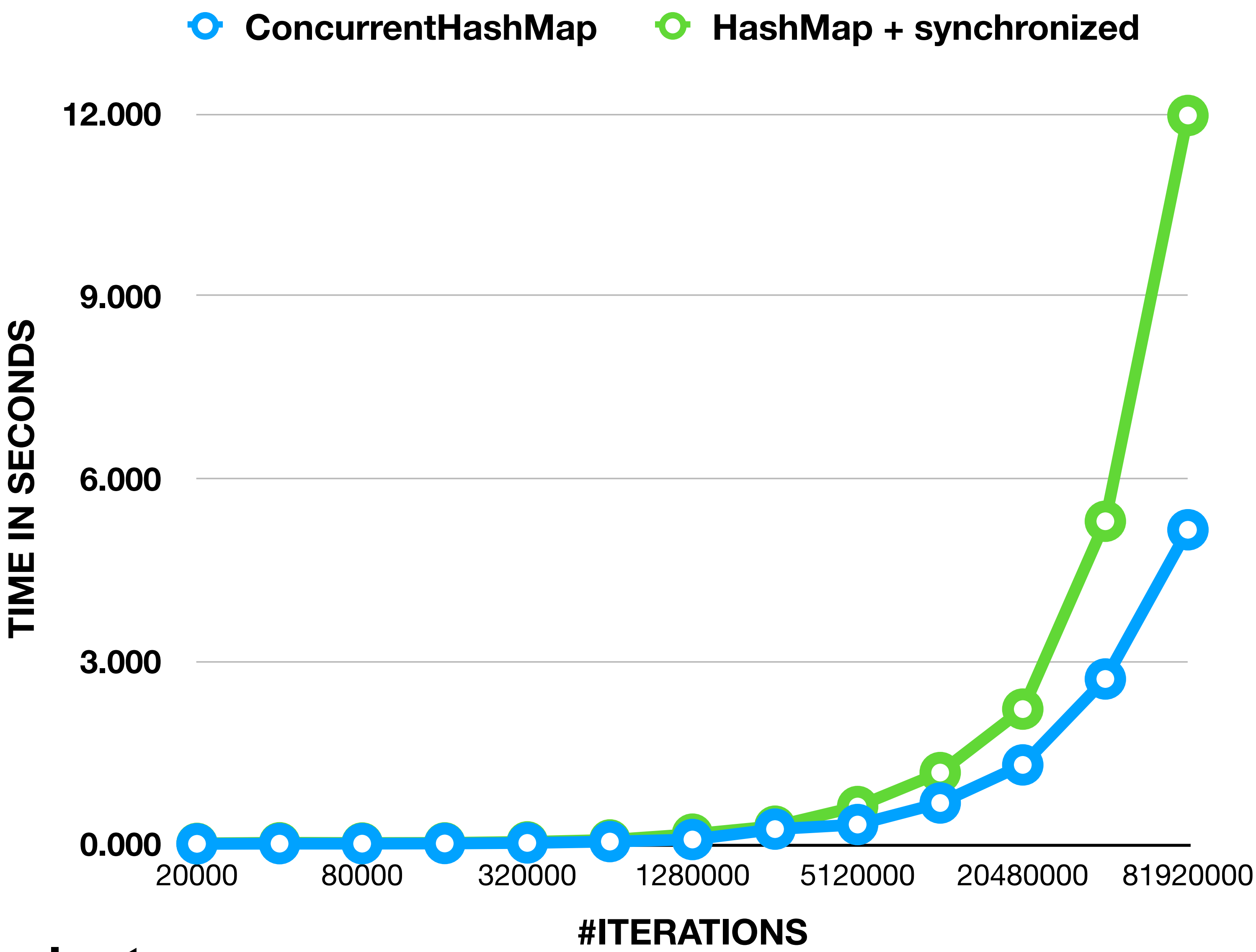
```
ExecutorService pool = Executors.newFixedThreadPool(3);

Future<String> result1 = pool.submit(new Task());

try {
   result1.get(10, TimeUnits.SECONDS);
} catch (InterruptedException | ExecutionException | TimeoutException ignore)  {}

pool.shutdown();
```

Concurrent/Blocking Collections

- Java provides thread-safe collection data structures
- Example: Queues
  - ConcurrentLinkedQueue: unbounded
  - LinkedBlockingQueue: bounded (thread may block)
  - Very important to read the documentation as there are subtleties about what operations are atomic or not
    - Let's look at https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/ConcurrentLinkedQueue.html

- High performance code written by experts
  - Using "lock free" magic we'll come back to
- Let's look at performance gains for a HashMap…

ConcurrentHashMap Performance



on my laptop

Conclusion

- If java.util.concurrent implements what you need, re-use it!
  - You have little hope of implementing something faster
- But it’s very important to master lower-level concurrency abstractions as well
  - Understanding them is often necessary for using higher-level abstractions well (and understand bugs)
  - And other languages don’t provide as much as Java!
- Hence our Homework Assignments so far
- Let’s now look at Homework Assignment #8…