Running zkl Unit Tests in Parallel

In zkl, a set of unit tests are typically collected into a file and, since these tests are self contained, it makes sense to run them in parallel to minimize the time it takes to get the test results. This note will take a look at one way to do this and also examine zkl code.

Overview

The zkl system test suites consists of several layers: A testing tool that is part of the core language, the tests themselves, and a wrapper that runs all the tests in parallel. Here is a brief overview of these components:

- The Unit Tester. The unit tester is a class that accepts zkl source code, compiles the code, runs it, compares the result against an expected value and logs the outcome. The UT can also test compiled code.
  Example:
  ```
  tester = Test.UnitTester();   // create a testing instance
  ```
- Tests. A set of tests that test a component (for example, functions) are usually collected in a file and that file is run.
  Example:
  ```
  tester.testSrc("var R = 1;",Void,Void,1); // expected result is 1
  tester.testSrc("this is a error;",
      "SyntaxError");     // Expect a SyntaxError exception
  ```
- The results:
  2 tests completed.
  Passed test(s): 2 (of 2)
  Failed test(s): 0, tests L()
  Flawed test(s): 0, tests L()
- Running tests in parallel. Since zkl supports threads and each test suite is a self contained unit, it makes sense to run them all at the same time. This saves a lot of time when there are hundreds of tests to run and makes regression testing a little less painful.
  The following code snippet shows the basic idea:
  ```
  class DoTest {
     fcn liftoff(fileName){ // running in thread
        test = Compiler.compileFile(fileName);
        test.__constructor();
     }
  }
  foreach fileName in (testFiles) { DoTest(fileName); }
  // wait for all the tests to complete
  ```
  For each test file, create a new thread class (DoTest) and have that thread compile and run the test suite.
- The results are then tabulated and look like this:
  …
  test/class.zkl
  77 tests completed.
  Passed test(s): 77 (of 77)

```
Failed test(s): 0, tests L()
Flawed test(s): 0, tests L()

test/thread.zkl
23 tests completed.
Passed test(s): 23 (of 23)
Failed test(s): 0, tests L()
Flawed test(s): 0, tests L()

Executive summary:
  456 tests completed (12 files)
  Passed test(s): 452 (of 456)
  Failed test(s):  4
  Flawed test(s):  0
  Failed files(s): 0
That took 24 seconds.
```

Details, Details (or, time for a second cup of coffee)

Now let's take an in depth look at the components, from the top down. A lot of details will be skipped, take a look at the source code for the full monty.

testThemAll.zkl

```
testFiles = File.glob("Test/[^Xx]*.zkl");
```

The above code finds all the test files. Glob() takes a Unix shell file name expression and uses that to match file names. In this case, all files in the Test directory that have the extension ".zkl" and don't start with "X" or "x" are returned in a list. For example:

```
L("Test/class.zkl","Test/dataref.zkl")
```

We need a way to for the test threads to talk to us. We'll use a pipe to pass the test results back to testThemAll and a counter to keep track of the tests.

```
var results = Thread.Pipe();
var N       = Atomic.Int();
```

Note that we use an atomic counter. Since more than one thread will be trying to change the counter at the same time, we have to ensure it can only be changed "atomically"[1].

A pipe is a "data hose" for threads. With a pipe, data can easily flow between processes without the programmer having to worry about details like thread synchronization, data locking, etc. It is just another Stream object (like a File or List) and can hold any object.

1 With variables, it is possible for one thread to change the variable value, and, before the thread can store the new value, another thread could have changed the stored value. Then, when the first thread finishes storing it's version of the new value, the second thread's changes will be lost. An example would be two threads running N += 1 at the same time and having the result being 1 (instead of 2).

Now, let's create a thread to run a test file:

```
class DoTest {
   fcn init(fileName) { N.inc(); self.launch(fileName); }
   fcn liftoff(fileName) {
      try {
         test = Compiler.compileFile(fileName);
         results.write(L( fileName, test.test() ));
      } catch { results.write(L( fileName, self.exception )); }
      N.dec();
   }
}
```

When an instance of the DoTest class is created, all it does is increment the counter (to say that another test thread is running), create a thread and return to the caller. The thread runs its tests while other threads are running their tests. self.launch creates a class thread and starts it running. liftoff is the entry point for the thread. The thread then tells the compiler to compile the file which contains a test suite. The compiler returns a class, which is then run (`test.__constructor()`) and the test file name and the results of running the test suite are written to the pipe. It is important to note that the each file is a UnitTester suite and returns a UnitTester class instance. If, for some reason, the file fails to compile, that is written to the pipe (the file name and result are written as a pair). Then the counter is decremented to tell testThemAll that we are done.

Meanwhile, back in testThemAll:

```
    // run each test suite (file) in a thread
testFiles.apply2(DoTest);
```

we take each test file, create a test thread with it and start the test. The apply method does the same thing as:

```
foreach fileName in (testFiles) { DoTest(fileName); }
```

and saves some typing.

Now that all the test suites are running (at the same time), we need to wait for them to finish so we can look at all the results:

```
Atomic.waitFor(fcn { (N == 0); });
results.close();
```

This waits for something to happen in an "efficient" manner, that is, consuming the least amount of system resources reasonable. In this case, we wait for the counter to count down to zero. Then close the pipe (we'll see why in a moment).
We could tabulate the results as each test finishes and that would be faster (because we could be doing work instead of waiting) but the speed up would be minimal and would make the code a little bit more complicated. Not worth it.

Now that all the tests have finished running, we can compile the results:

```
n = passed = failed = flawed = numFailedFiles = 0;
foreach file,result in (results){
   if (result.isInstanceOf(UnitTester.UnitTester)) {
      n      += result.N_;
      passed += result.passed_;
      failed += result.failed_;
      flawed += result.flawed_;
   }
   else numFailedFiles += 1;
}
```

Remember that the name of the test file and a class were written to the pipe as a pair (a two element list). Thus, we can read these two items from the pipe in one go. Now we check to see if the test was successful, if it was, the result is a UnitTester class, otherwise it was the exception that caused failure. The results are tabulated and we repeat until there is nothing left in the pipe (which is why we needed to close it, so we know when it is empty).

A test suite

```
tester = Test.UnitTester();
tester.testSrc("return(Void);","SyntaxError");
tester.testSrc("var R = 1;",Void,Void,1);
tester.stats();
returnClass(tester);
```

First, a UnitTester class instance is created, then some zkl source code is tested.

The testSrc method takes four parameters:

- The source code, as a string.
- The name of a compile time exception, if one is expected, else Void. Testing to make sure errors are caught is almost as important as verifying the code is correct.
- The name of a runtime exception, if one is expected, else Void.
- The expected result. The UnitTester looks for the variable "R" to compare the expected result to.

Note that last three parameters default to Void, so you can sometimes save some typing.

Now for something strange: A zkl code file is treated as one big class, thus, in this case, the tests are part of the class constructor[2]. Now, normally, class constructors return their class instance. But, in this case, that would break testThemAll, since it wants the UnitTester, not the test, so returnClass is used override this (and, in fact, is the only way to get out of a constructor without flowing off the end or throwing an exception).

While this may look a contrived way of writing [OO] code, it has the huge advantage of making it really easy to write tests: copy three lines (the first and last two) from an existing test file into a new file and starting cranking out test code in the same way you would write a script. And it is equally easy to run, just feed it to zkl:

```
zkl aTest.zkl
2 tests completed.
Passed test(s): 2 (of 2)
Failed test(s): 0, tests L()
Flawed test(s): 0, tests L()
```

If the test file is added to the test directory, then testThemAll will automagically include it in the next test run. And it can even be broken and not cause any problems. Thus, it is often handy to include broken code in a test suite as a note fix things (see below).
This "ease of use" is critical to encouraging tests to be written during "time critical" times, such as when bugs are found, while code is being developed or in the process of documenting classes.

2 Basically, a test file is a script

The stats() method prints out the test suite results:

```
test/dataref.zkl
92 tests completed.
Passed test(s): 90 (of 92)
Failed test(s): 2, tests L(4,92)
Flawed test(s): 0, tests L()
```

Source code

- testThemAll.zkl
- A test suite: Test/fcn.zkl

Notes on unit tests

Unit tests are great for writing small, targeted tests. But once the test source gets longer than about five lines, they start to suck: not much fun to maintain and all the other reasons we hate testing.