JVM concurrency: Java and Scala concurrency basics

Understand concurrency in the Java language and the added options that Scala provides

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

25 March 2014

The Java™ platform provides excellent support for concurrent programming in all JVM-based languages. Scala extends the concurrency support in the Java language with even more ways to share work among processors and coordinate the results. This first article in a new series on JVM concurrency covers the state of the art in concurrent programming in Java 7 and introduces some of the Scala enhancements. The article also helps prepare the way for understanding concurrency features in Java 8.

View more content in this series

Decades of continual rapid advances in processor speeds came to an end around the turn of the century. Since then, processor manufacturers have improved chip performance more by adding cores than by increasing clock speeds. Multi-core systems are now the norm for everything from cell phones to enterprise servers, and this trend is likely to continue and accelerate. Increasingly, developers must address multiple cores in their application code to fulfill performance requirements.

In this series of articles, you'll get a look at some newer approaches to concurrent programming for both the Java and Scala languages, including how Java is incorporating ideas already explored in Scala and other JVM-based languages. This first installment gives you a backdrop for understanding the broader picture of concurrent programming on the JVM by introducing some of the current state-of-the-art techniques for both Java 7 and Scala. You'll learn how you can use the Java `ExecutorService` and `ForkJoinPool` classes to simplify concurrent programming. You'll also get an introduction to some of the basic Scala features that extend concurrent-programming options beyond what's available in plain Java. Along the way, you'll see how different approaches affect concurrent-programming performance. Subsequent installments will cover the concurrency improvements in Java 8, along with extensions including the Akka toolkit for scalable Java and Scala programming.

Java concurrency support

Concurrency support has been a Java feature since the platform's earliest days, with clean implementation of threads and synchronization giving it an edge over competing languages. Scala is based on Java and runs on the JVM, with direct access to all of the Java runtime (including all the concurrency support). So before looking into the Scala features, I'll start with a quick review of what the Java language already supplies.

About this series

Now that multi-core systems are ubiquitous, concurrent programming must be applied more widely than ever before. But concurrency can be difficult to implement correctly, and you need new tools to help you use it. Many of the JVM-based languages are developing tools of this type, and Scala has been particularly active in this area. This series gives you a look at some of the newer approaches to concurrent programming for the Java and Scala languages.

Basic Java threading

Threads are easy to create and use in Java programming. They're represented by the `java.lang.Thread` class, and the code to be executed by a thread is in the form of a `java.lang.Runnable` instance. You can create large numbers of threads if needed in your application — well into the thousands. When multiple cores are available, the JVM uses them to execute multiple threads concurrently; threads in excess of the number of cores share the cores.

Java 5: A concurrency watershed

Java included support for threads and synchronization from the beginning. But the less-than-bulletproof initial specifications for sharing data among threads occasioned major changes in the Java language update for Java 5 (JSR-133). The Java Language Specification for Java 5 corrected and formalized the operation of `synchronized` and `volatile`. The specification also spelled out how immutable objects work with multithreading. (Basically, immutable objects are always thread-safe provided that references aren't allowed to "escape" while the constructor is being executed.) Previously, interactions among threads generally required use of the blocking `synchronized` operation. The changes enabled non-blocking coordination among threads through the use of `volatile`. As a result, new concurrent collection classes were added in Java 5 that support non-blocking operations — a major improvement over the earlier blocking-only approach to thread safety.

Coordinating the actions of threads gets confusing. One complication occurs because the Java compiler and the JVM are free to reorder operations in your code, as long as everything stays consistent from your program's perspective. For example: If two addition operations use different variables, the compiler or JVM can execute the operations in the reverse order to what you specified, provided the program doesn't use either sum before both operations are complete. This flexibility to reorder operations helps improve Java performance, but the consistency guarantee applies only within a single thread. Hardware can also create problems with threads. Modern systems use multiple levels of cache memory, and the caches are generally not seen identically by all cores in the system. When one core modifies a value in memory, the change might not be immediately visible to the other cores.

Because of these issues, when one thread is working with data modified by another thread, you must explicitly control how the threads interact. Java uses special operations to provide this control, establishing orderings in the views of data seen by different threads. The basic

operation is for threads to use the `synchronized` keyword to access an object. When a thread synchronizes on an object, the thread obtains exclusive access to a lock that's unique to that object. If another thread already holds that lock, the thread that wants to acquire it must wait, or *block*, until the lock is released. When the thread resumes execution inside a `synchronized` block of code, Java guarantees that the thread "sees" everything written by other threads that previously held that same lock — but only data written by those threads up to the time they released the lock by leaving their own `synchronized` block. This guarantee applies both to the reordering of operations performed by the compiler or JVM and to hardware memory cache. The interior of a `synchronized` block, then, is an island of stability in your code where threads can take turns executing, interacting, and sharing information safely.

Use of the `volatile` keyword on a variable provides a slightly weaker form of safe interaction among threads. The `synchronized` keyword guarantees that when you obtain the lock, you see other thread's stores, and that other threads obtaining the lock after you will see your stores. The `volatile` keyword breaks this guarantee into two separate pieces. If a thread writes to a `volatile` variable, all its prior writes to that point are first flushed. If a thread reads the variable, it sees not only the value written to that variable but also all other values written by the writing thread. So reading a `volatile` variable provides the same sort of memory guarantee as *entering* a `synchronized` block, and writing a `volatile` variable gives the same sort of memory guarantee as *leaving* a `synchronized` block. But there's one big difference: Reading or writing a `volatile` variable never blocks.

Abstracting Java concurrency

Synchronization is useful, and many multithreaded applications are developed in Java using only the basic `synchronized` block. But coordinating threads can be messy, especially when you're dealing with many threads and many locks. Ensuring that threads interact only in safe ways *and* that you're avoiding potential deadlocks (two or more threads waiting for each other to release locks before they can continue execution) gets difficult. Abstractions that support concurrency without directly dealing with threads and locks give developers better ways to handle common use cases.

The `java.util.concurrent` hierarchy includes collections variations that support concurrent access, wrapper classes for atomic operations, and synchronization primitives. Many of these classes were designed to support non-blocking access, which avoids issues with deadlock and enables more-efficient threading. The classes make it easier to define and regulate interactions among threads, but they still suffer from some of the complexity of the basic threading model.

A pair of abstractions in the `java.util.concurrent` package support a more decoupled approach to handling concurrency: the `Future<T>` interface, and the `Executor` and `ExecutorService` interfaces. These related interfaces are in turn the basis for many Scala and Akka extensions to Java concurrency support, so it's worth looking at the interfaces and their implementations in more detail.

`Future<T>` is a holder for a value of type `T`, with the twist that the value is generally not available until sometime after the `Future` is created. The value is the result of an asynchronous operation that might execute concurrently. The thread receiving the `Future` can call methods to:

- See if the value is available
- Wait for the value to become available
- Retrieve the value when it is available
- Cancel the operation if the value is no longer needed

Specific implementations of `Future` are structured to support different ways of handling the asynchronous operation.

`Executor` is an abstraction around something that executes tasks. This "something" will ultimately be a thread, but the details of how the thread handles the execution are hidden by the interface. `Executor` is of limited usefulness on its own, with the `ExecutorService` subinterface providing extension methods to manage termination and to produce `Future`s for the results of tasks. All standard implementations of `Executor` also implement `ExecutorService`, so in practice, you can ignore the root interface.

Threads are relatively heavyweight resources, and it makes sense to reuse rather than allocate and discard them. `ExecutorService` simplifies the sharing of work among threads while also enabling automatic reuse of threads, leading to easier programming and better performance. The `ThreadPoolExecutor` implementation of `ExecutorService` manages a pool of threads to perform tasks.

Applying Java concurrency

Practical applications of concurrency often involve tasks that require external interactions (with a user, with storage, or with other systems) that are independent of your main processing logic. That type of application is hard to condense down to a simple example, so for concurrency demonstrations, people often use simple computationally intensive tasks such as math calculations or sorting. I'll use a similar example.

The task is to find the nearest known word to an unknown input, where *nearest* is defined in terms of *Levenshtein distance*: the minimum number of character additions, deletions, or changes required to transform the input into the known word. I use code based on a sample in the Levenshtein distance article on Wikipedia to compute the Levenshtein distance for each known word and return the best match (or an indeterminate result, if multiple known words have the same distance).

Listing 1 shows Java code for calculating the Levenshtein distance. The calculation generates a matrix with rows and columns matching the sizes of the two texts being compared, plus one in each dimension. For efficiency, this implementation uses a pair of arrays sized to the target text to represent successive rows of the matrix, swapping the arrays in each pass because I need only the values from the immediately prior row to compute the next row.

Listing 1. Levenshtein distance calculation in Java

```
/**
 * Calculate edit distance from targetText to known word.
 *
 * @param word known word
 * @param v0 int array of length targetText.length() + 1
 * @param v1 int array of length targetText.length() + 1
 * @return distance
 */
private int editDistance(String word, int[] v0, int[] v1) {

    // initialize v0 (prior row of distances) as edit distance for empty 'word'
    for (int i = 0; i < v0.length; i++) {
        v0[i] = i;
    }

    // calculate updated v0 (current row distances) from the previous row v0
    for (int i = 0; i < word.length(); i++) {

        // first element of v1 = delete (i+1) chars from target to match empty 'word'
        v1[0] = i + 1;

        // use formula to fill in the rest of the row
        for (int j = 0; j < targetText.length(); j++) {
            int cost = (word.charAt(i) == targetText.charAt(j)) ? 0 : 1;
            v1[j + 1] = minimum(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost);
        }

        // swap v1 (current row) and v0 (previous row) for next iteration
        int[] hold = v0;
        v0 = v1;
        v1 = hold;
    }

    // return final value representing best edit distance
    return v0[targetText.length()];
}
```

If you have a large number of known words to compare against the unknown input, and you're running on a multi-core system, you can use concurrency to speed up the processing: Break up the set of known words into multiple chunks and process each chunk as a separate task. By changing the number of words in each chunk, you can easily change the granularity of the task breakdown to see the effect on overall performance. Listing 2 shows Java code for the chunked calculation, taken from the `ThreadPoolDistance` class in the sample code. Listing 2 uses a standard `ExecutorService` with the thread count set to the number of available processors.

Listing 2. Chunked distance calculation in Java with multiple threads

```
private final ExecutorService threadPool;
private final String[] knownWords;
private final int blockSize;

public ThreadPoolDistance(String[] words, int block) {
    threadPool = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    knownWords = words;
    blockSize = block;
}

public DistancePair bestMatch(String target) {

    // build a list of tasks for matching to ranges of known words
    List<DistanceTask> tasks = new ArrayList<DistanceTask>();
```

```
    int size = 0;
    for (int base = 0; base < knownWords.length; base += size) {
        size = Math.min(blockSize, knownWords.length - base);
        tasks.add(new DistanceTask(target, base, size));
    }
    DistancePair best;
    try {

        // pass the list of tasks to the executor, getting back list of futures
        List<Future<DistancePair>> results = threadPool.invokeAll(tasks);

        // find the best result, waiting for each future to complete
        best = DistancePair.WORST_CASE;
        for (Future<DistancePair> future: results) {
            DistancePair result = future.get();
            best = DistancePair.best(best, result);
        }

    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
    return best;
}

/**
 * Shortest distance task implementation using Callable.
 */
public class DistanceTask implements Callable<DistancePair>
{
    private final String targetText;
    private final int startOffset;
    private final int compareCount;

    public DistanceTask(String target, int offset, int count) {
        targetText = target;
        startOffset = offset;
        compareCount = count;
    }

    private int editDistance(String word, int[] v0, int[] v1) {
        ...
    }

    /* (non-Javadoc)
     * @see java.util.concurrent.Callable#call()
     */
    @Override
    public DistancePair call() throws Exception {

        // directly compare distances for comparison words in range
        int[] v0 = new int[targetText.length() + 1];
        int[] v1 = new int[targetText.length() + 1];
        int bestIndex = -1;
        int bestDistance = Integer.MAX_VALUE;
        boolean single = false;
        for (int i = 0; i < compareCount; i++) {
            int distance = editDistance(knownWords[i + startOffset], v0, v1);
            if (bestDistance > distance) {
                bestDistance = distance;
                bestIndex = i + startOffset;
                single = true;
            } else if (bestDistance == distance) {
                single = false;
            }
```

```
        }
        return single ? new DistancePair(bestDistance, knownWords[bestIndex]) :
             new DistancePair(bestDistance);
    }
}
```

The `bestMatch()` method in Listing 2 constructs a list of `DistanceTask` instances, then passes the list to the `ExecutorService`. This form of call to the `ExecutorService` takes a parameter of type `Collection<? extends Callable<T>>` representing the tasks to be executed. The call returns a list of `Future<T>` representing the results of the executions. The `ExecutorService` asynchronously fills in these results with the values returned by invoking the `call()` method on each task. In this case, the `T` type is `DistancePair`— a simple value object for a distance and the matched word, or only a distance when no unique match occurs.

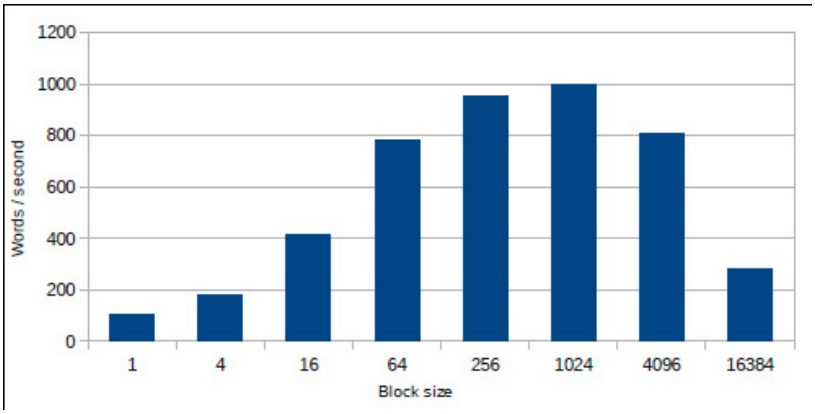
The original thread of execution in the `bestMatch()` method waits in turn for each `Future` to complete, accumulating the best result and returning it when finished. With multiple threads handling the execution of the `DistanceTask`s, the original thread waits for only a fraction of the results. The rest of the results are completed concurrently with the ones the original thread waits for.

Concurrency performance

To take full advantage of the number of processors available on a system, you must configure the `ExecutorService` with at least as many threads as there are processors. You also must pass at least as many tasks as there are processors to the `ExecutorService` for execution. In practice, you probably want to have significantly more tasks than processors for best performance. That way, processors keep busy with one task after another and fall idle only at the end. But because overhead is involved — in creating the tasks and futures, switching threads among the tasks, and finally returning the results of the tasks — you must keep the tasks large enough so that the overhead is proportionately small.

Figure 1 shows how measured performance varies with different numbers of tasks when the test code is run on my four-core AMD system using Oracle's Java 7 for 64-bit Linux®. Each input word is compared in turn with 12,564 known words, and each task finds the best match within a range of the known words. The full set of 933 misspelled input words is run repeatedly, with pauses between passes for the JVM to settle, and the best time after 10 passes is used in the graph. As you can see from Figure 1, the performance in input words per second looks stable over a reasonable range of block sizes (basically, from 256 to >1,024), dropping only near the extremes where the tasks become either very small or very large. The final value, for block size 16,384, creates only one task, so shows single-threaded performance.

Figure 1. `ThreadPoolDistance` performance



Fork-Join

Java 7 introduced another implementation of `ExecutorService`: the `ForkJoinPool` class. `ForkJoinPool` is designed for handling tasks efficiently that can be repeatedly broken down into subtasks, using the `RecursiveAction` class (when the task produces no result) or the `RecursiveTask<T>` class (when the task has a result of type `T`) for tasks. `RecursiveTask<T>` provides a convenient way to consolidate results from subtasks, as shown in Listing 3.

Listing 3. `RecursiveTask<DistancePair>` example

```
private ForkJoinPool threadPool = new ForkJoinPool();

private final String[] knownWords;

private final int blockSize;

public ForkJoinDistance(String[] words, int block) {
    knownWords = words;
    blockSize = block;
}

public DistancePair bestMatch(String target) {
    return threadPool.invoke(new DistanceTask(target, 0, knownWords.length, knownWords));
}

/**
 * Shortest distance task implementation using RecursiveTask.
 */
public class DistanceTask extends RecursiveTask<DistancePair>
{
    private final String compareText;
    private final int startOffset;
    private final int compareCount;
    private final String[] matchWords;

    public DistanceTask(String from, int offset, int count, String[] words) {
        compareText = from;
        startOffset = offset;
        compareCount = count;
        matchWords = words;
    }

    private int editDistance(int index, int[] v0, int[] v1) {
        ...
    }
```

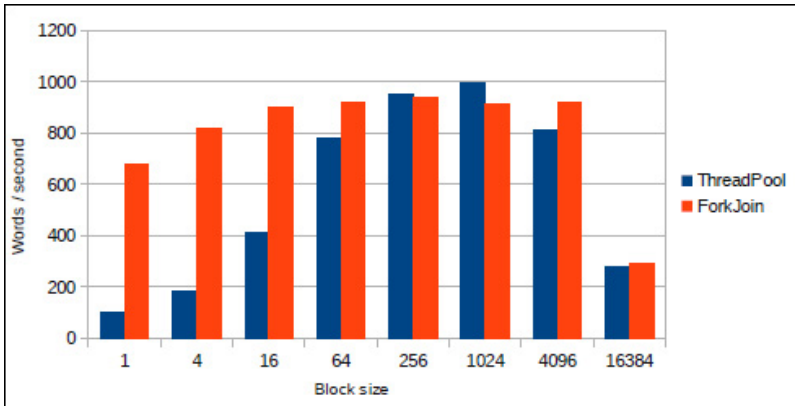
```
    /* (non-Javadoc)
     * @see java.util.concurrent.RecursiveTask#compute()
     */
    @Override
    protected DistancePair compute() {
        if (compareCount > blockSize) {

            // split range in half and find best result from bests in each half of range
            int half = compareCount / 2;
            DistanceTask t1 = new DistanceTask(compareText, startOffset, half, matchWords);
            t1.fork();
            DistanceTask t2 = new DistanceTask(compareText, startOffset + half,
                compareCount - half, matchWords);
            DistancePair p2 = t2.compute();
            return DistancePair.best(p2, t1.join());
        }

        // directly compare distances for comparison words in range
        int[] v0 = new int[compareText.length() + 1];
        int[] v1 = new int[compareText.length() + 1];
        int bestIndex = -1;
        int bestDistance = Integer.MAX_VALUE;
        boolean single = false;
        for (int i = 0; i < compareCount; i++) {
            int distance = editDistance(i + startOffset, v0, v1);
            if (bestDistance > distance) {
                bestDistance = distance;
                bestIndex = i + startOffset;
                single = true;
            } else if (bestDistance == distance) {
                single = false;
            }
        }
        return single ? new DistancePair(bestDistance, knownWords[bestIndex]) :
            new DistancePair(bestDistance);
    }
}
```

Figure 2 shows how the performance of the `ForkJoin` code from Listing 3 compares with that of the `ThreadPool` code from Listing 2. The `ForkJoin` code is much more stable across the full range of block sizes, dropping significantly only when you get to a single block (meaning the execution is single-threaded). The standard `ThreadPool` code shows better performance only at the 256 and 1,024 block sizes.

Figure 2. `ThreadPoolDistance` performance compared to `ForkJoinDistance` performance



These results show that if you can tune the task size in an application for best performance, you might do a little better with the standard `ThreadPool` than with `ForkJoin`. But understand that the "sweet spot" for `ThreadPool` depends on the task, the number of processors available, and potentially other aspects of your system. In general, `ForkJoin` gives you excellent performance with minimal need for tuning, so you're best off using it whenever possible.

Scala concurrency basics

Scala extends the Java programming language and runtime in many ways, including adding more and easier ways to handle concurrency. For starters, the Scala version of `Future<T>` is much more flexible than the Java version. You can create futures directly from blocks of code, and you can attach callbacks to futures for handling completions. Listing 4 shows some examples of Scala futures in use. The code first defines the `futureInt()` method to supply `Future<Int>`s on demand, then uses the futures in three different ways.

Listing 4. Scala `Future<T>` sample code

```
import ExecutionContext.Implicits.global

val lastInteger = new AtomicInteger
def futureInt() = future {
  Thread sleep 2000
  lastInteger incrementAndGet
}

// use callbacks for completion of futures
val a1 = futureInt
val a2 = futureInt
a1.onSuccess {
    case i1 => {
      a2.onSuccess {
        case i2 => println("Sum of values is " + (i1 + i2))
      }
    }
}
Thread sleep 3000

// use for construct to extract values when futures complete
val b1 = futureInt
val b2 = futureInt
for (i1 <- b1; i2 <- b2) yield println("Sum of values is " + (i1 + i2))
Thread sleep 3000

// wait directly for completion of futures
val c1 = futureInt
val c2 = futureInt
println("Sum of values is " + (Await.result(c1, Duration.Inf) +
  Await.result(c2, Duration.Inf)))
```

The first example in Listing 4 attaches callback closures to a pair of futures, so that when both have completed, the sum of the two result values is printed to the console. The callbacks are nested directly on the futures in the order they were created, but they work the same if you change the order. If the future has already been completed when you attach the callback, the callback still runs, though with no guarantee that it will run immediately. The original execution thread pauses at the `Thread sleep 3000` line to enable the futures to complete before moving on to the next example.

The second example demonstrates the use of the Scala *`for` comprehension* to extract values from the futures asynchronously and use them directly in an expression. The `for` comprehension is a Scala construct that you can use to express complex combinations of operations (`map`, `filter`, `flatMap`, and `foreach`) concisely. It's generally used with various forms of collections, but Scala futures implement the same monadic methods used for accessing collection values. So you can use a future as a special sort of collection — one that contains at most one value (and might not even contain that one value until some point in the future). In this case, the `for` statement is saying to take the results of the futures and use those result values in an expression. Behind the scenes, this technique generates pretty much the same code as the first example, but writing it in the form of linear code yields a simpler expression that's easier to understand. As with the first example, the original execution thread pauses so that the futures can finish before moving on to the next example.

The third example uses blocking waits to obtain the results of the futures. This is equivalent to how Java futures work, though in the Scala case, a special `Await.result()` method call taking a maximum wait-time parameter makes the blocking wait explicit.

The code in Listing 4 doesn't obviously pass futures off to an `ExecutorService` or equivalent, so if you haven't worked with Scala, you might wonder how the code behind a future is executed. The answer lies in the top line in Listing 4: `import ExecutionContext.Implicits.global`. Scala APIs often use `implicit` values for parameters that will be frequently reused across a block of code. The `future { }` construct requires that an `ExecutionContext` be available as an implicit parameter. This `ExecutionContext` is a Scala wrapper for a Java `ExecutorService` and is used in the same way to execute tasks using one or more managed threads.

Beyond these basic operations with futures, Scala also provides a way to convert any collection into one that uses parallel programming. After you convert your collection to parallel form, any standard Scala collection operation (such as `map`, `filter`, or `fold`) that you perform on the collection will automatically be done in parallel where possible. (You'll see an example of this a little further on in this article, as part of the Listing 7 code that finds the best match for a word using Scala.)

Error handling

Futures in both Java and Scala must deal with the issue of error handling. In the Java case, as of Java 7, futures can throw an `ExecutionException` as an alternative to returning a result. Applications can define their own subclasses of `ExecutionException` for specific types of failures, or they can chain exceptions to pass on details, but that's the limit of the flexibility.

Scala futures provide more flexible handling of errors. You have two ways to complete a Scala future: as a success with a result value (assuming one is expected), or as a failure, with an associated `Throwable`. You can also handle the completion of a future in multiple ways. In Listing 4 the `onSuccess` method is used to attach callbacks for handling the successful completion of a future. You can also use `onComplete` to handle any form of completion (which wraps the result or throwable in a `Try` to accommodate both cases), or `onFailure` to handle specifically an error result.

This flexibility of Scala futures extends to all operations that you can perform using futures, so you can integrate your error handling directly into the code.

The Scala `Future<T>` also has a closely related `Promise<T>` class. A future is a holder for an outcome that might (or might not — there is no inherent guarantee that a future will ever complete) become available at some point. After the future is completed, the outcome is fixed and unchangable. A promise is the other side of this same contract: a one-time assignable holder for an outcome, in the form of either a result value or a throwable. You can get a future from a promise, and when the outcome is set on the promise, it is also set on that future.

Applying Scala concurrency

Now that you're familiar with some of the basic Scala concurrency concepts, it's time to look at code for the Levenshtein distance problem. Listing 5 shows a more-or-less idiomatic Scala implementation of the Levenshtein distance calculation, basically matching the Java code in Listing 1 but in functional style.

Listing 5. Levenshtein distance calculation in Scala

```
val limit = targetText.length
/** Calculate edit distance from targetText to known word.
  *
  * @param word known word
  * @param v0 int array of length targetText.length + 1
  * @param v1 int array of length targetText.length + 1
  * @return distance
  */
def editDistance(word: String, v0: Array[Int], v1: Array[Int]) = {

  val length = word.length

  @tailrec
  def distanceByRow(rnum: Int, r0: Array[Int], r1: Array[Int]): Int = {
    if (rnum >= length) r0(limit)
    else {

      // first element of r1 = delete (i+1) chars from target to match empty 'word'
      r1(0) = rnum + 1

      // use formula to fill in the rest of the row
      for (j <- 0 until limit) {
        val cost = if (word(rnum) == targetText(j)) 0 else 1
        r1(j + 1) = min(r1(j) + 1, r0(j + 1) + 1, r0(j) + cost);
      }

      // recurse with arrays swapped for next row
      distanceByRow(rnum + 1, r1, r0)
    }
  }

  // initialize v0 (prior row of distances) as edit distance for empty 'word'
  for (i <- 0 to limit) v0(i) = i

  // recursively process rows matching characters in word being compared to find best
  distanceByRow(0, v0, v1)
}
```

The Listing 5 code uses the tail-recursive `distanceByRow()` method for each row-value calculation. This method first checks how many rows have been calculated, and if that number matches

the number of characters in the word being checked it returns the result distance. Otherwise, it calculates the new row values, then finishes by calling itself recursively to calculate the next row (swapping the two row arrays in the process, so that the new current row values are passed in correctly). Scala converts tail-recursive methods to the equivalent of Java `while` loops, so the similarity to the Java code is preserved.

There's one major difference between this code and the Java code, though. The `for` comprehensions in the Listing 5 code use closures. Closures aren't always handled efficiently by current JVMs (see Why is using for/foreach on a Range slow? for details), so they add considerable overhead to the innermost loop of the calculation. As written, then, the Listing 5 code doesn't run as fast as the Java version. Listing 6 shows a rewrite that replaces the `for` comprehensions with added tail-recursive methods. This version is much more verbose but performs on a par with the Java version.

Listing 6. Calculation code restructured for performance

```
val limit = targetText.length

/** Calculate edit distance from targetText to known word.
  *
  * @param word known word
  * @param v0 int array of length targetText.length + 1
  * @param v1 int array of length targetText.length + 1
  * @return distance
  */
def editDistance(word: String, v0: Array[Int], v1: Array[Int]) = {

  val length = word.length

  @tailrec
  def distanceByRow(row: Int, r0: Array[Int], r1: Array[Int]): Int = {
    if (row >= length) r0(limit)
    else {

      // first element of v1 = delete (i+1) chars from target to match empty 'word'
      r1(0) = row + 1

      // use formula recursively to fill in the rest of the row
      @tailrec
      def distanceByColumn(col: Int): Unit = {
        if (col < limit) {
          val cost = if (word(row) == targetText(col)) 0 else 1
          r1(col + 1) = min(r1(col) + 1, r0(col + 1) + 1, r0(col) + cost)
          distanceByColumn(col + 1)
        }
      }
      distanceByColumn(0)

      // recurse with arrays swapped for next row
      distanceByRow(row + 1, r1, r0)
    }
  }

  // initialize v0 (prior row of distances) as edit distance for empty 'word'
  @tailrec
  def initArray(index: Int): Unit = {
    if (index <= limit) {
      v0(index) = index
      initArray(index + 1)
    }
  }
```

```
  initArray(0)

  // recursively process rows matching characters in word being compared to find best
  distanceByRow(0, v0, v1)
}
```

Listing 7 shows Scala code to perform the same sort of blocked distance calculation as in the Listing 2 Java code. The `bestMatch()` method finds the best match for the target text within a particular block of words handled by the `Matcher` class instance, using the tail-recursive `best()` method to scan through the words. The `*Distance` classes create multiple `Matcher` instances, one for each block of words, then coordinate the execution and combination of the matcher results.

Listing 7. Block-at-a-time distance calculation in Scala with multiple threads

```
class Matcher(words: Array[String]) {

  def bestMatch(targetText: String) = {

    val limit = targetText.length
    val v0 = new Array[Int](limit + 1)
    val v1 = new Array[Int](limit + 1)

    def editDistance(word: String, v0: Array[Int], v1: Array[Int]) = {
      ...
    }

    @tailrec
    /** Scan all known words in range to find best match.
      *
      * @param index next word index
      * @param bestDist minimum distance found so far
      * @param bestMatch unique word at minimum distance, or None if not unique
      * @return best match
      */
    def best(index: Int, bestDist: Int, bestMatch: Option[String]): DistancePair =
      if (index < words.length) {
        val newDist = editDistance(words(index), v0, v1)
        val next = index + 1
        if (newDist < bestDist) best(next, newDist, Some(words(index)))
        else if (newDist == bestDist) best(next, bestDist, None)
        else best(next, bestDist, bestMatch)
      } else DistancePair(bestDist, bestMatch)

    best(0, Int.MaxValue, None)
  }
}

class ParallelCollectionDistance(words: Array[String], size: Int) extends TimingTestBase {

  val matchers = words.grouped(size).map(l => new Matcher(l)).toList

  def shutdown = {}

  def blockSize = size

  /** Find best result across all matchers, using parallel collection. */
  def bestMatch(target: String) = {
    matchers.par.map(m => m.bestMatch(target)).
      foldLeft(DistancePair.worstMatch)((a, m) => DistancePair.best(a, m))
  }
}

class DirectBlockingDistance(words: Array[String], size: Int) extends TimingTestBase {
```

```
  val matchers = words.grouped(size).map(l => new Matcher(l)).toList

  def shutdown = {}

  def blockSize = size

  /** Find best result across all matchers, using direct blocking waits. */
  def bestMatch(target: String) = {
    import ExecutionContext.Implicits.global
    val futures = matchers.map(m => future { m.bestMatch(target) })
    futures.foldLeft(DistancePair.worstMatch)((a, v) =>
      DistancePair.best(a, Await.result(v, Duration.Inf)))
  }
}
```

The two `*Distance` classes in Listing 7 show different ways of coordinating the execution and combination of `Matcher` results. `ParallelCollectionDistance` uses the previously mentioned parallel-collections feature of Scala to hide the details of parallel computations, needing only a simple `foldLeft` to combine the results.

`DirectBlockingDistance` is a little more explicit, creating a list of futures and then using a `foldLeft` on that list with a nested blocking wait for each individual result.

Performance, one more time

Both of the Listing 7`*Distance` implementations are reasonable approaches to handling the `Matcher` results. (And they're far from the only reasonable approaches. The sample code includes a couple of other implementations I tried in my experimentations but don't include in the article.) In this case, performance is a main concern, so Figure 3 shows how these two implementations perform relative to the Java `ForkJoin` code.

Figure 3. `ForkJoinDistance` performance compared to performance of Scala alternatives

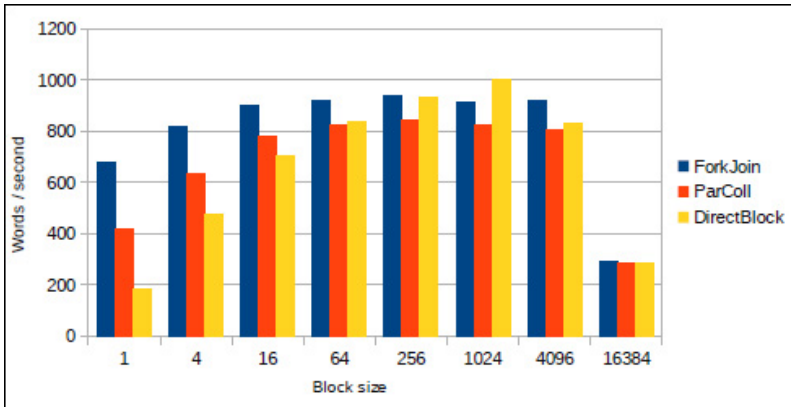


Figure 3 shows that the Java `ForkJoin` code performs better than either of the Scala implementations overall, though the `DirectBlockingDistance` provides better performance at the 1,024 block size. Both Scala implementations deliver better performance than the Listing 1 `ThreadPool` code across most of the range of block sizes.

These performance results are only illustrative, not definitive. If you run the timing tests on your own system, you'll likely see differences in the relative performance, especially if you have a

different number of cores at work. And if I wanted to get the best performance for the distance task, I'd implement optimizations: I could sort the known words by length and start comparisons with words of the same length as the input (because the edit distance is always at least as large as the difference in word length). Or I could use early outs from the distance calculation when it exceeds the best prior value. But as a relatively simple algorithm, this experiment does a fair job of showing both how concurrent operations can improve performance and the impact of different approaches to sharing the work.

Performance aside, it's interesting to compare the two versions of the Scala control code in Listing 7 with the Java code Listing 2 and Listing 3. The Scala code is significantly shorter and (assuming you understand Scala!) clearer than the Java code. Scala and Java interoperate well, as you can see in the full sample code for this article: Scala code runs the timing tests of both the Scala and Java code, and the Java code in turn works directly with portions of the Scala code. Thanks to this easy interoperability, you can introduce Scala into an existing Java code base without a wholesale changeover. It often makes sense to use Scala initially for the high-level control of Java code, so that you can take full advantage of Scala's powerful expressive features without any significant performance impact from closures or conversions.

The simplicity of the Listing 7 `ParallelCollectionDistance` Scala code is especially appealing. By using this approach, you can abstract concurrency completely out of the code so that you're writing what looks like a single-threaded application while still gaining the benefits of multiple processors. Fortunately for those who like the simplicity of this approach but might be unwilling or unable to jump into Scala development, Java 8 is bringing a similar feature to straight Java programming.

Future<series>

Now that you've seen the basics of both Java and Scala concurrency operation, the next article in this series will look at how Java 8 is improving concurrency support for Java (and potentially for Scala, too, in the longer term). Many of the Java 8 changes will look familiar — many of the same concepts used in Scala concurrency features are being included in Java 8 — so you'll soon be able to use some of the Scala techniques in your ordinary Java code. Read the next installment to learn how.

Resources

Learn

- Scalable Scala: Series author Dennis Sosnoski shares insights and behind-the-scenes information on the content in this series and Scala development in general.
- Sample code for this article: Get this article's full sample code from the author's repository on GitHub.
- "*Java theory and practice*: Introduction to nonblocking algorithms" (Brian Goetz, developerWorks, April 2006): Concurrency guru Brian Goetz discusses and demonstrates nonblocking algorithms made possible by the language changes in Java 5.
- "*Java theory and practice*: Going atomic" (Brian Goetz, developerWorks, November 2004): Find out how atomic variable classes in `java.util.concurrent` enable the development of highly scalable non-blocking algorithms.
- "*Java theory and practice*: Stick a fork in it, Part 1 " (Brian Goetz, developerWorks, November 2007): Learn how the fork-join abstraction provides a natural mechanism for decomposing many algorithms to exploit hardware parallelism effectively.
- "*Java theory and practice*: Stick a fork in it, Part 2 " (Brian Goetz, developerWorks, March 2008): See how you can use `ParallelArray` classes in Java, which simplify parallel sorting and searching operations on in-memory data structures
- Parallel Programming made easier with Java 7 ForkJoin (video | 56:05): Watch as Kavitha Varadarajan of the IBM Java Technology Center explains Fork-Join concepts and demonstrates the capabilities through code examples.
- Scala: Scala is a modern, functional language on the JVM.
- "*The busy Java developer's guide to Scala*: Explore Scala concurrency" (Ted Neward, developerWorks, February 2009): Learn more about the concurrency features and libraries that the Scala language and environment provide.
- "*The busy Java developer's guide to Scala*: Dive deeper into Scala concurrency" (Ted Neward, developerWorks, April 2009): Learn about Scala actors.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- IBM Java developer kits: Discover the IBM Java SDK and runtime environment for your platform.
- IBM SDK, Java Technology Edition Version 8: Participate in the IBM SDK for Java 8.0 Beta Program.
- Evaluate IBM products.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Dennis Sosnoski

Dennis Sosnoski is a Java and Scala developer with extensive experience developing scalable systems. He's well-known in the XML and web services areas, where his background includes the development of JiBX XML data binding and work on several open source web services frameworks (most recently, Apache CXF). Dennis is a frequent presenter at Java user groups and conferences and has written many articles for developerWorks, including the popular *Java web services* series. Learn more about his web services training and consulting work at his Sosnoski Software Associates Ltd site, and follow his ongoing explorations of concurrent programming on the JVM at his Scalable Scala site.