Scala Collections Performance

GS.com/Engineering
March, 2015
Craig Motlin
Who am I?

- Engineer at Goldman Sachs
- Technical lead for GS Collections
  - A replacement for the Java Collections Framework
Goals

• Scala programs ought to perform as well as Java programs, but Scala is a little slower
Goals

From RedBlackTree.scala

/*
 * Forcing direct fields access using the @inline annotation helps speed up various operations (especially smallest/greatest and update/delete).
 * 
 * Unfortunately the direct field access is not guaranteed to work (but works on the current implementation of the Scala compiler).
 * 
 * An alternative is to implement the these classes using plain old Java code...
 */
Goals

• Scala programs ought to perform as well as Java programs, but Scala is a little slower
• Highlight a few performance problems that matter to me
• Suggest fixes, borrowing ideas and technology from GS Collections
GS Collections and Scala’s Collections are similar

- Mutable and immutable interfaces with common parent
- Similar iteration patterns at hierarchy root Traversable and RichIterable
- Lazy evaluation (view, asLazy)
- Parallel-lazy evaluation (par, asParallel)
GS Collections and Scala’s Collections are different

- Persistent data structures
- Hash tables
- Primitive specialization
- Fork-join
Persistent Data Structures
• Mutable collections are similar in GS Collections and Scala – though we’ll look at some differences
• Immutable collections are quite different
  • Scala’s immutable collections are persistent
  • GS Collections’ immutable collections are not
In computing, a **persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.
Wikipedia:

In computing, a **persistent data structure** is a data structure that always preserves the previous version of itself when it is **modified**.
Persistent Data Structures

• Typical examples: List and Sorted Set
Persistent Data Structures

• Typical examples: List and **Sorted Set**
• “Add” by constructing $O(\log n)$ new nodes from the leaf to a new root.

• **Scala sorted set** $\rightarrow$ binary tree
• **GSC immutable sorted set** $\rightarrow$ covered later
Wikipedia:
Persistent Data Structures

• Typical examples: **List** and Sorted Set
• “Prepend” by constructing a new head in $O(1)$ time. LIFO structure.

• **Scala immutable list** $\rightarrow$ linked list or stack
• **GSC immutable list** $\rightarrow$ array backed
Persistent Data Structures

• Extremely important in purely functional languages
• All collections are immutable
• Must have good runtime complexity
• No one seems to miss them in GS Collections
Persistent Data Structures

• Proposal: Mutable, Persistent Immutable, and Plain Immutable

• Mutable: Same as always
• Persistent: Use when you want structural sharing
• (Plain) Immutable: Use when you’re done mutating or when the data never mutates
Plain old immutable data structures

• Not persistent (no structural sharing)
• “Adding” is much slower: $O(n)$
• Speed benefits for everything else
• Huge memory benefits
Persistent Data Structures

- Immutable array-backed list
- Immutable $\rightarrow$ trimmed array
- No nodes $\rightarrow$ 1/6 memory
- Array backed $\rightarrow$ cache locality, should parallelize well
Lists: Size in MB by number of elements

- com.gs.collections.api.list.ImmutableList
- scala.collection.immutable.List
- scala.collection.mutable.ArrayBuffer

Measured with Java 8 64-bit and compressed oops
Persistent Data Structures

Performance assumptions:

• Iterating through an array should be faster than iterating through a linked list
• Linked lists won’t parallelize well with .par
• No surprises in the results – so we’ll skip

github.com/goldmansachs/gs-collections/tree/master/jmh-tests
github.com/goldmansachs/gs-collections/tree/master/memory-tests
Persistent Data Structures

- Immutable array-backed sorted set
- Immutable $\rightarrow$ trimmed, sorted array
- No nodes $\rightarrow$ $\sim1/8$ memory
- Array backed $\rightarrow$ cache locality
Persistent Data Structures

Sorted Sets: Size in MB by number of elements

- com.gs.collections.api.set.sorted.ImmutableSortedSet
- com.gs.collections.impl.set.sorted.mutable.TreeSortedSet
- scala.collection.immutable.TreeSet
- scala.collection.mutable.TreeSet
Persistent Data Structures

• Assumptions about contains:
• May be faster when it’s a binary search in an array (good cache locality)
• Will be about the same in mutable / immutable tree (all nodes have same structure)
Persistent Data Structures

Sorted Set contains() throughput (higher is better)

Million ops/s

Immutable
Mutable

GSC
Scala
Iteration
val scalaImmutable: immutable.TreeSet[Int] =
  immutable.TreeSet.empty[Int] ++ (0 to SIZE)

def serial Immutable_Scala(): Unit =
{
  val count: Int = this.scalaImmutable
    .view
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .count(each => (each + 1) % 10000 != 0)
if (count != 999800)
{
  throw new AssertionError
}
}
Persistent Data Structures

val scalaImmutable: immutable.TreeSet[Int] =
  immutable.TreeSet.empty[Int] ++ (0 to SIZE)

def serialImmutable.scala(): Unit =
{
  val count: Int = this.scalaImmutable
    .view
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .count(each => (each + 1) % 10000 != 0)
  if (count != 999800)
  {
    throw new AssertionError
  }
}
val scalaImmutable: immutable.TreeSet[Int] =
  immutable.TreeSet.empty[Int] ++ (0 to SIZE)

def serial Immutable Scala(): Unit =
{
  val count: Int =
    this.scalaImmutable.
    .view
    .filter(each => each % 10000 != 0)
    .map(String.valueOf)
    .map(Integer.valueOf)
    .count(each => (each + 1) % 10000 != 0)
  
  if (count != 999800)
  {
    throw new AssertionError
  }
}
private final ImmutableSortedSet<Integer> gscImmutable =
    SortedSets.immutable.withAll(Interval.zeroTo(SIZE));

@Benchmark
public void serial Immutable_gsc()
{
    int count = this.gscImmutable
        .asLazy()
        .select(each -> each % 10_000 != 0)
        .collect(String::valueOf)
        .collect(Integer::valueOf)
        .count(each -> (each + 1) % 10_000 != 0);

    if (count != 999_800)
    {
        throw new AssertionError();
    }
}
• Assumption: Iterating over an array is somewhat faster
Persistent Data Structures

Sorted Set Iteration throughput (higher is better)

Million ops/s

Serial / Immutable

Serial / Mutable

GSC
Scala
Persistent Data Structures

Next up, parallel-lazy evaluation

\[
\begin{align*}
& \text{this.scalaImmutable.view} \rightarrow \\
& \text{this.scalaImmutable.par} \rightarrow \\
& \text{this.gscImmutable.asLazy()} \rightarrow \\
& \text{this.gscImmutable.asParallel(this.executorService, BATCH_SIZE)}
\end{align*}
\]
• Assumption: Scala’s tree should parallelize moderately well

• Assumption: GSC’s array should parallelize very well
Persistent Data Structures

Sort Set Iteration throughput (higher is better)

<table>
<thead>
<tr>
<th></th>
<th>GSC</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial / Immutable</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Serial / Mutable</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>Parallel / Immutable</td>
<td>90</td>
<td>0</td>
</tr>
</tbody>
</table>

Measured on an 8 core Linux VM Intel Xeon E5-2697 v2
Persistent Data Structures

- Scala’s `immutable.TreeSet` doesn’t override `.par`, so parallel is slower than serial.
- Some tree operations (like `filter`) are hard to parallelize.
- `TreeSet.count()` should be easy to parallelize using fork/join with some work.
- New assumption: Mutable trees won’t parallelize well either.
Persistent Data Structures

Measured on an 8 core Linux VM
Intel Xeon E5-2697 v2

Sorted Set Iteration throughput (higher is better)

- Serial / Immutable
- Serial / Mutable
- Parallel / Immutable
- Parallel / Mutable

Million ops/s

GSC  Scala

8x
Persistent Data Structures

- **GS Collections** follow up:
  - `TreeSortedSet` delegates to `java.util.TreeSet`. Write our own tree and override `.asParallel()`

- **Scala** follow up:
  - Override `.par`
Persistent Data Structures

• Proposal: Mutable, Persistent, and **Immutable**
• All three are useful in different cases
• How common is it to share structure, really?
Hash Tables
Hash Tables

- Scala’s `immutable.HashMap` is a hash array mapped trie (persistent structure)
- Wikipedia: “achieves almost hash table-like speed while using memory much more economically”
- Let’s see
Hash Tables

- Scala’s `mutable.HashMap` is backed by an array of Entry pairs
- `java.util.HashMap.Entry` caches the hashcode
- `UnifiedMap<K, V>` is backed by `Object[]`, flattened, alternating K and V
- `ImmutableUnifiedMap` is backed by `UnifiedMap`
Maps: Size in MB by number of elements

- com.gs.collections.api.map.ImmutableMap
- com.gs.collections.impl.map.mutable.UnifiedMap
- java.util.HashMap
- scala.collection.immutable.HashMap
- scala.collection.mutable.HashMap
@Benchmark
public void get()
{
    int localSize = this.size;
    String[] localElements = this.elements;
    Map<String, String> localScalaMap = this.scalaMap;
    for (int i = 0; i < localSize; i++)
    {
        if (!localScalaMap.get(localElements[i]).isDefined())
        {
            throw new AssertionError(i);
        }
    }
}
Hash Tables

Map get() throughput by map size (higher is better)
Hash Tables

Map put() throughput by map size (higher is better)
@Benchmark
public mutable.HashMap<String, String> mutableScalaPut()
{
    int localSize = this.size;
    String[] localElements = this.elements;

    mutable.HashMap<String, String> map =
        new PresizableHashMap<>(localSize);

    for (int i = 0; i < localSize; i++)
    {
        map.put(localElements[i], "dummy");
    }

    return map;
}
HashMap ought to have a constructor for pre-sizing

```scala
class PresizableHashMap[K, V](val _initialSize: Int)
  extends scala.collection.mutable.HashMap[K, V]
{
  private def initialCapacity =
    if (_initialSize == 0) 1
    else smallestPowerOfTwoGreaterThan(_initialSize)

  private def smallestPowerOfTwoGreaterThan(n: Int): Int =
    if (n > 1) Integer.highestOneBit(n - 1) << 1 else 1

  table = new Array(initialCapacity)
  threshold = ((initialCapacity * _loadFactor) / 1000).toInt
}
```
Hash Tables

- `scala.collection.mutable.HashSet` is backed by an array using open-addressing
- `java.util.HashSet` is implemented by delegating to a `HashMap`
- `UnifiedSet` is backed by `Object[]`, either elements or arrays of collisions
- `ImmutableUnifiedSet` is backed by `UnifiedSet`
Sets: Size in MB by number of elements

- `com.gs.collections.api.set.ImmutableSet`
- `com.gs.collections.impl.set.mutable.UnifiedSet`
- `java.util.HashSet`
- `scala.collection.immutable.HashSet`
- `scala.collection.mutable.HashSet`
Primitive Specialization
Primitive Specialization

- Boxing is expensive
  - Reference + Header + alignment
- Scala has specialization, but most of the collections are not specialized
- If you cannot afford wrappers you can:
  - Use primitive arrays (only for lists)
  - Use a Java Collections library
Sets: Size in MB by number of elements

- com.gs.collections.impl.set.mutable.primitive.IntHashSet
- com.gs.collections.impl.set.mutable.UnifiedSet
- scala.collection.mutable.HashSet
Proposal: Primitive lists, sets, and maps in Scala

Not traversable – outside the collections hierarchy

Fix specialization on functions (lambdas)
  – Want for-comprehensions to work well
Fork/Join
val list = this.integers.par
  .filter(each => each % 10000 != 0)
  .map(String.valueOf)
  .map(Integer.valueOf)
  .filter(each => (each + 1) % 10000 != 0)
  .toBuffer

Assert.assertEquals(999800, list.size)
MutableList<Integer> list = this.integersGSC.asParallel(this.executorService, BATCH_SIZE) .select(each -> each % 10_000 != 0) .collect(String::valueOf) .collect(Integer::valueOf) .select(each -> (each + 1) % 10_000 != 0) .toList();

Verify.assertSize(999_800, list);
List<Integer> list = this.integersJDK
    .parallelStream()
    .filter(each -> each % 10_000 != 0)
    .map(String::valueOf)
    .map(Integer::valueOf)
    .filter(each -> (each + 1) % 10_000 != 0)
    .collect(Collectors.toList());

Verify.assertSize(999_800, list);
Stacked computation ops/s  (higher is better)

Serial Lazy

Parallel Lazy

Java 8  GS Collections  Scala

8x
Fork-Join Merge

- Intermediate results are merged in a tree
- Merging is $O(n \log n)$ work and garbage
Fork-Join Merge

- Amount of work done by last thread is $O(n)$
MutableList<Integer> list = this.integersGSC.asParallel(this.executorService).
.select(each -> each % 10_000 != 0)
.collect(String::valueOf).
.collect(Integer::valueOf)
.select(each -> (each + 1) % 10_000 != 0)
.toList();

Verify.assertEqualsSize(999_800, list);
public final class CompositeFastList<E> {
    private final FastList<FastList<E>> lists = FastList.newList();

    public boolean addAll(Collection<? extends E> collection) {
        FastList<E> collectionToAdd = collection instanceof FastList ? (FastList<E>) collection :
            new FastList<E>(collection);
        this.lists.add(collectionToAdd);
        return true;
    }
    ...
}
• Merging is $O(1)$ work per batch
Fork/Join

• Fork-join is general purpose but always requires merge work

• We can get better performance through specialized data structures meant for combining
GS Collections and Scala’s Collections are different
• Persistent data structures
• Hash tables
• Primitive specialization
• Fork-join
@motlin
@GoldmanSachs

stackoverflow.com/questions/tagged/gs-collections

github.com/goldmansachs/gs-collections/tree/master/jmh-tests
github.com/goldmansachs/gs-collections/tree/master/memory-tests
github.com/goldmansachs/gs-collections-kata

infoq.com/presentations/java-streams-scala-parallel-collections
Q&A