Java Streams, Part 5: Parallel stream performance

Optimizing stream pipelines for parallel processing

Brian Goetz

July 18, 2016

This fifth and final installment of the Java Streams series continues the previous installment's discussion of factors that influence the effectiveness of parallel processing, and applies them to the Streams library. Find out why some stream pipelines parallelize better than others, and see how to analyze your own streams code for parallel performance.

View more content in this series

Part 4 of *Java Streams* discussed factors that can influence the effectiveness of parallelization. These factors include characteristics of the problem, the algorithm used to implement the solution, the runtime framework used to schedule tasks for parallel execution, and the size and memory layout of the data set. This installment applies these concepts to the Streams library and examines why some stream pipelines parallelize better than others.

Parallel streams

About this series
With the `java.util.stream` package, you can concisely and declaratively express possibly-parallel bulk operations on collections, arrays, and other data sources. In this series by Java Language Architect Brian Goetz, get a comprehensive understanding of the Streams library and learn how to use it to best advantage.

As you saw in Part 3, a stream pipeline consists of a stream source, zero or more intermediate operations, and a terminal operation. To execute a stream pipeline, we construct a "machine" that implements the intermediate and terminal operations, into which we feed the elements from the source. To execute a stream pipeline in parallel, we partition the source data into segments via recursive decomposition, using the `Spliterator` method `trySplit()`. The effect is to create a binary computation tree whose leaf nodes each correspond to a segment of the source data, and whose internal nodes each correspond to a point where the problem has been split into subtasks, and the results of two subtasks need to be combined. A sequential execution constructs one machine for the entire data set; a parallel execution constructs one machine for each segment of the source data, producing a partial result for each leaf. We then proceed up the tree, merging the partial results into bigger results, according to a merge function that is specific to the terminal operation. For example, for terminal operation `reduce()`, the binary operator used for reduction is

also the merge function; for `collect()`, the `Collector` has a merge function used to merge one result container into another.

The previous installment identified several factors that might cause a parallel execution to lose efficiency:

- The source is expensive to split, or splits unevenly.
- Merging partial results is expensive.
- The problem doesn't admit sufficient exploitable parallelism.
- The layout of the data results in poor access locality.
- There's not enough data to overcome the startup costs of parallelism.

We now examine each of these considerations, with an eye toward how they manifest in parallel stream pipelines.

Source splitting

With parallel streams, we use the `Spliterator` method `trySplit()` to split a segment of the source data in two. Each node in the computation tree corresponds to a binary split, forming a binary tree. Ideally, this tree would be balanced — with each leaf node representing exactly the same amount of work — and the cost of splitting would be zero.

This ideal is not achievable in practice, but some sources come much closer than others. Arrays are the best case. We can describe a segment of an array by using a reference to the array base and integral offsets of the start and end of the segment. The cost of splitting this segment into two equal segments is cheap: We compute the midpoint of the segment, create a new descriptor for the first half, and move the starting index for the current segment to the first element in the in the second half.

Listing 1 shows the code for `trySplit()` in `ArraySpliterator`. Arrays have low split costs — a few arithmetic operations and an object creation; they also split evenly (leading to balanced computation trees). The `Spliterator` for `ArrayList` has the same desirable characteristics. (As a bonus, when splitting arrays we also know the exact size of all splits, which allows us to optimize away copies in some stream pipelines.)

Listing 1. Implementation of ArraySpliterator.trySplit().

```
public Spliterator<T> trySplit() {
    int lo = index, mid = (lo + fence) >>> 1;
    return (lo >= mid)
           ? null
           : new ArraySpliterator<>(array,
                                    lo, index = mid,
                                    characteristics);
}
```

On the other hand, a linked list splits terribly. The cost of splitting is poor — to find the midpoint, we must traverse half the list, one node at a time. To lower splitting costs, we might accept more unbalanced splits — but this still doesn't help much. In the extreme case, we end up with a pathologically unbalanced (right-heavy) tree, where each split consists only of `(first element, rest of list)`. So, instead of *O(lg n)* splits, we have *O(n)* splits, which requires *O(n)* combining steps. We're left with the bad choice between a computation tree that's very expensive to create (limiting parallelism by contributing to the serial fraction of Amdahl's law), and a computation tree that admits relatively little parallelism and has high combination costs (because it's so unbalanced.) That said, it's not **impossible** to get parallelism out of a linked list — if the operations being performed for each node are sufficiently expensive. (See "The *NQ* model.")

Binary trees (such as `TreeMap`) and hash-based collections (such as `HashSet`) split better than linked lists, but not as well as arrays. Binary trees are fairly cheap to split in two, and if the tree is relatively balanced, the resulting computation tree will be as well. We implement `HashMap` as an array of buckets, where each bucket is a linked list. If the hash function spreads the elements well over the buckets, the collection should split relatively well (until you get down to a single bucket, and then you're back to a linked list, which ideally is small). However, both tree-based and hash-based collections generally don't split as predictably as arrays — we can't predict the size of the resulting splits, so we lose out on the ability to optimize away copies in some cases.

Generators as sources

Not all streams use a collection as their source; some use a generator function, such as `IntStream.range()`. The considerations that apply to collection sources can be applied directly to generators as well.

The next two examples show two ways to generate a stream consisting of integers 0 to 99. This code uses `Stream.iterate()` (the three-argument version of `iterate()` was added in Java 9):

```
IntStream stream1 = IntStream.iterate(0, n -> n < 100,
                                      n -> n + 1);
```

And this code uses `IntStream.range()`:

```
IntStream stream2 = IntStream.range(0, 100);
```

The two examples produce the same results, but have dramatically different splitting characteristics — and therefore will have dramatically different parallel performance.

`Stream.iterate()` takes an initial value and two functions — one to produce the next value, and one to determine whether to stop producing elements — just like a `for`-loop. It's intuitively clear that this generator is fundamentally sequential: It cannot produce element *n* until it has produced element *n-1*. As a result, splitting a sequential generator function has the same characteristics as splitting a linked list (the bad choice between high split costs and highly uneven splitting) and results in similarly poor parallelism.

On the other hand, the `range()` generator splits more like an array — it's easy and cheap to compute the midpoint of the range, without having computed the intervening elements.

Although the design of the Streams library was heavily influenced by the principles of functional programming, the difference in parallel performance characteristics between the two generator functions in the preceding examples illustrates a potential trap for those with a functional programming background. In functional programming, it's common and natural to generate a range by lazily consuming from an infinite stream constructed by iterative function application — but this idiom arose in a time when data-parallelism was an entirely theoretical concept. Functional programmers might easily reach for the familiar `iterate` idiom without immediately realizing the inherent sequentiality of this approach.

Result combination

Splitting the source — efficiently and evenly, or not — is a necessary cost to enable parallel computation. If we're lucky, the cost of splitting is modest enough that we can start forking off work early, avoiding running afoul of Amdahl's law.

Every time we split the source, we accrue an obligation to combine the intermediate results of that split. After the leaf nodes have completed the work on their segment of the input, we proceed back up the tree, combining results as we go.

Some combining operations — such as reduction with addition — are cheap. But others — such as merging two sets — are much more expensive. The amount of time spent in the combination step is proportional to the depth of the computation tree; a balanced tree will have depth of *O(lg n)*, whereas a pathologically unbalanced tree (such as that we get from splitting a linked list or an iterative generating function) will have depth of *O(n)*.

Another problem with expensive merges is that the last merge — in which two half-results are being merged — will be performed sequentially (because there is no other work left to do). Stream pipelines whose merge steps are *O(n)*— such as those using the `sorted()` or `collect(Collectors.joining())` terminal operations — might see their parallelism limited by this effect.

> *“ You might be surprised by the time and space costs of parallel execution where operations are tied to encounter order. With sequential execution, the obvious implementation is usually one in which we traverse the input in encounter order, so a dependence on encounter order is rarely either visible or costly. In parallel, such dependencies can be very costly ”*

Operation semantics

Just as some sources — such as linked list or iterative generator functions — are inherently sequential, some stream operations also have an inherently sequential aspect, which can serve as

an impediment to parallelism. These usually are operations whose semantics are defined in terms of *encounter order*.

For example, the `findFirst()` terminal operation yields the first element in the stream. (This operation is typically combined with filtering, so it usually ends up meaning "find the first element that satisfies some condition.") Implementing `findFirst()` sequentially is extremely cheap: Push data through the pipeline until some result is produced, and then stop. In parallel, we can easily parallelize the upstream operations, but when a result is produced by some subtask, we're not done. We still have to wait for all the subtasks that come earlier in the encounter order to finish. (At least we can cancel any subtasks that appear later in the encounter order.) A parallel execution pays all the costs of decomposition and task management, but is less likely to reap the benefits. On the other hand, the terminal operation `findAny()` is much more likely to reap a parallel speedup, because it keeps all the cores busy searching for a match, and can terminate immediately when it finds one.

Another terminal operation whose semantics are tied to encounter order is `forEachOrdered()`. Again, while it's often possible to fully parallelize the execution of the intermediate operations, the final applicative step is sequentialized. On the other hand, the `forEach()` terminal operation is unconstrained by encounter order. The applicative step can be executed for each element at whatever time and in whatever thread the element is made available.

Intermediate operations, such as `limit()` and `skip()`, can be constrained by encounter order as well. The `limit(n)` operation truncates the input stream after the *first n* elements. Like `findFirst()`, when elements are produced by some task, `limit(n)` must wait for all the tasks that precede it in the encounter order to finish before it knows whether to push those elements to the remainder of the pipeline — and it must buffer the produced elements until it knows whether or not they are needed. (For an *unordered* stream, `limit(n)` is allowed to select *any n* elements — and like `findAny()`, is much more parallel-friendly.)

You might be surprised by the time and space costs of parallel execution where operations are tied to encounter order. The obvious sequential implementations of `findFirst()` and `limit()` are simple, efficient, and require almost no space overhead, but the parallel implementations are complex and often involve considerable waiting and buffering. With sequential execution, the obvious implementation is usually one in which we traverse the input in encounter order, so a dependence on encounter order is rarely either visible or costly. In parallel, such dependencies can be very costly.

Fortunately, these dependencies on encounter order often can be eliminated through small changes to the pipeline. Frequently, we can replace `findFirst()` with `findAny()` without any loss of correctness. Similarly, as we saw in Part 3, by making the stream unordered via the `unordered()` operation, we can often remove the encounter-order dependence inherent in `limit()`, `distinct()`, `sorted()`, and `collect()` with no loss of correctness.

The various hazards to parallel speedup that we've looked at so far can be cumulative. Just as the three-argument `iterate()` source was far worse than the range constructor, combining the two-argument `iterate()` source with `limit()` is even worse, as it combines a sequential generation

step with an encounter-order-sensitive operation. For example, here's the least parallel-friendly way to generate a range of integers:

```
IntStream stream3 = IntStream.iterate(0, n -> n+1).limit(100);
```

Memory locality

Modern computer systems employ sophisticated multilevel caches to keep frequently used data as close (literally — speed of light is a limiting factor!) to the CPU as possible. Fetching data from L1 cache can easily be 100 times faster than fetching data from main memory. The more efficiently the CPU can predict which data will be needed next, the more cycles the CPU spends doing computation, and the fewer it spends waiting for data.

Data is paged into cache at the granularity of *cache lines*; today's x86 chips use a cache line size of 64 bytes. This rewards programs that have good *memory locality*— the propensity to access memory locations that are close to locations that have been recently accessed. Proceeding linearly through an array not only has excellent locality, but is also further rewarded by *prefetch*— when a linear pattern of memory access is detected, the hardware begins prefetching the next cache-line's worth of memory on the assumption that it will probably be needed soon.

Mainstream Java implementations lay out the fields of an object, and the elements of an array, contiguously in memory (though fields are not necessarily laid out in the order declared in the source file). Accesses of fields or array elements that are "near" the most recently accessed field or element have a good chance of hitting data that is already in cache. On the other hand, references to other objects are represented as pointers, so dereferencing an object reference is more likely to hit data that is not already in cache, causing a delay.

An array of primitives offers the best locality possible. After the initial dereference of the array reference, the data is stored consecutively in memory, so we can maximize the amount of computation per data fetch. An array of object references will get good locality when fetching the next reference in the array, but risks a cache miss when dereferencing those object references. Similarly, a class containing multiple primitive fields will likely lay the fields out near one another in memory, whereas a class containing many object references will require many dereferences to access its state. Ultimately, the more pointer-rich a data structure is, the more pressure traversing such a data structure places on the memory fetch units, which can have a negative effect both on computation time (while the CPUs spend time waiting for data) and parallelism (as many cores fetching simultaneously from memory put pressure on the bandwidth available to transfer data from memory to cache).

The *NQ* model

To determine whether parallelism will offer a speedup, the final factors to consider are the amount of data available and the amount of computation performed per data element.

In our initial description of parallel decomposition, we appealed to the notion of splitting the source until the segments are so small that a sequential approach for solving the problem on that segment

would be more efficient. How small the segments must be depends on the problem being solved, and specifically, how much work is done per element. For example, computing the length of a string involves far less work than computing the string's SHA-1 hash. The more work done per element, the lower the threshold for "large enough to extract parallelism." Similarly, the more data we have, the more segments into which we can divide it without running afoul of the "too small" threshold.

A simple but useful model for parallel performance is the *NQ* model, where *N* is the number of data elements, and *Q* is the amount of work performed per element. The larger the product *N*Q*, the more likely we are to get a parallel speedup. For problems with trivially small *Q*, such as adding up numbers, you generally want to see *N* > 10,000 to get a speedup; as *Q* increases, the data size required to get a speedup decreases.

Many of the impediments to parallelism — such as splitting cost, combining cost, or encounter order sensitivity — are moderated by higher *Q* operations. While the splitting characteristics of a `LinkedList` might be awful, it's still possible to get a parallel speedup given a large enough *Q*.

Conclusion

Because parallelism offers only the potential for faster runtime, you should use it only when it produces an actual speedup. Develop and test your code using sequential streams; then, if your performance requirements suggest that further improvement is needed, consider parallelism as a possible optimization strategy. Although measurement is critical to ensuring that your optimization efforts aren't counterproductive, for many stream pipelines, you can determine via inspection that they aren't good candidates for parallelization. Factors that chip away at potential parallel speedup include poorly or unevenly splitting sources, high combination costs, dependence on encounter order, poor locality, or not enough data. On the other hand, a large amount of computation (*Q*) per element can make up for some of these deficiencies.

Related topics

- The JMH benchmarking harness
- Moore's law
- Amdahl's law
- Java Concurrency in Practice (Brian Goetz, et al., Addison-Wesley Professional)
- Package documentation for java.util.stream
- The Java fork-join library
- IBM Code: Java journeys