Implications of Parallel Implementations of List Filters

York College Of Pennsylvania

CS 365

Daniel Mashuda

**Introduction:**

Performing operations on a list (mapping) and getting a subset of a list (filtering) are something that developers do quiet often when building software. Till Java 8, there was no native, language feature, way for simple maps and filters to be written without iterating over a list. These features are mainstream in other languages, and offer a concise way of retrieving a subset or mutating a collection. In Java, a loop would be necessary for Collection operations.

In Java 8, Java gained the Streams Api, which allows for filtering and mapping of collections. Along with Streams, Java 8 brought Parallel Streams, which would execute whatever a stream was doing, either mapping or filtering, would execute in parallel to attempt to reduce the runtime of the filtering or mapping. Parallel stream incurs the runtime penalty of starting threads and shutting down threads every time Collection.parallelStream is called.

The objective is to implement filtering of a collection in Java in parallel without incurring the cost of creating and destroying threads for each filter operation. In order to circumvent this, the object to do parallel filtering will have to have an explicit shutdown function, which will be called when no more filtering is needed. As a secondary objective, the created parallel filter will be able to accept lambda expressions in the same manner that the Streams API accepts lambda expressions.

This implementation will be benchmarked against several different ways that a sub list can be created. The different ways being of creating a sub list are: For loop, Java Steams, Java Parallel Streams, Parallel filter (new implementation).

**Implementation:**

Because quantitative analysis is critical to comparing the different runtimes of these methods of creating sub lists, a benchmarking procedure was developed. The abstract class Benchmark strictly defines how a benchmark will run. The subclasses of Benchmark are only responsible for setup, teardown, and doWork functions. Setup and teardown runtimes are not counted toward the overall runtime of the benchmark, because only the doWork function’s runtime counts toward the benchmark score. Also, Benchmark runs the operation 10 times and calculates an average runtime in an attempt to smooth over any anomalies that may occur during the benchmarking.

For the benchmark, each method for creating a sub list returns a sub list of the items that contains a substring; this is done by the Java String.contains() method. By having all of the benchmarks use the same code to determine if an item from the super list should be in the sub list, it removes any chance of discrepancies. The specific implementation for the benchmarks is essentially the same, with the obvious syntactic differences due to the differences methods used. The benchmarks are located in the Benchmarks.contains package.

The parallel list filter is not an overly complicated implementation. This is due to the easily exploitable parallelism in filtering items from a list. The parallelism is easily exploitable because each item in a collection is checked against a condition, if it passes, it is added to a list to be returned. It can be easily inferred that there is not any interaction between elements in the list therefore the solution does not require any complex strategies to parallelize.

Notable classes in the parallel filter implementation are FilterWorker, ParallelFilter, and BaseFilter(interface). FilterWorker, as the name implies, does the processing of subsets of the larger collection. It manages interaction with the main thread through locking queues; the java.util.concurrent.LinkedBlockingQueue was used for both of the queues. The FilterWorker will shut down if it is passed a work range whose start and end points are the same value. The ParallelFilter is the exposed API class that manages the interactions between the main thread and the FilterWorkers. When a ParallelFilter is created, it creates FilterWorkers to be prepared to filter lists. The filter method does the dividing of work, passes the work to the FilterWorker’s, receives the result from the FilterWorker’s and then returns the combined result. The BaseFilter interface allows for the filter method to accept a Java 8 lambda expression which simplifies the use of the publicly exposed methods.

**Results:**

**Conclusions:**

**Appendix:**