Parallel Stream Operations

If operations on individual data items are entirely independent of one another, these operations can be performed by concurrent threads, provided that the infrastructure handles data-visibility (*happens-before*) requirements.

Java's Streams API supports this directly. Operations on the stream elements should generally be *non-interfering* and *stateless*.

Non-interfering means an operation does not affect the source of data.

Thread safety for parallel streams

Generally, operations should

- Not mutate existing data, but create new data for results
- Avoid side-effects
- Use only their own arguments as input

Parallel collections

Java's `collect` operation is a variation of a reduce operation, modified to mutate data, rather than continually create new objects to represent intermediate results. The effect is to reduce object allocation and initialization load on the CPUs and also reduce load on the garbage collector.

For such an approach to be safe, the objects being mutated must either be thread-safe (which is likely to create thread contention in the general case) or thread-confined.

The behavior of Java's three-argument `collect` method is built around thread-confined mutable objects.

A mutating version of reduce

Each thread must have its own work-in-progress mutable result

The system decides how many threads, so we must provide a means of building those result objects, hence a `Supplier<R>` for a result type R.

The second operation merges data one at a time into the current thread's result object.

The third operation merges one thread's result into that of another. This will be used only for parallel streams and will be called enough times to get all the accumulated results into a single result object.

Constraints on reduce/collect

Reduce/collect operations generally should be associative.

If the operation is also commutative, then order need not be maintained in the stream.

Order is generally maintained if the source and collector are ordered, but this can be disabled using `mystream.unordered()`.

Architectural considerations for parallel operation

Concurrency always involves some additional overhead. Throughput gains might depend on ratio of real computation work to that overhead.

If your process completes faster, parallel might be an option, but would you be taking those CPUs away from other equally important work? If so it's unlikely to be a smart choice to run parallel. In a multi-client server system, those CPUs usually do have something else important to do.

Maintaining ordered in a streams can be hugely expensive if many items arrive substantially out of order.

If the arrival rate of items from the streams is low relative to the time taken to update results, a single threadsafe / synchronized result object might be preferred.