Parallelism in Functional Programming

Denis Miginsky

State: functional and imperative view

	Imperative programming	Functional programming
State representation	Mutable variables	Immutable variables
State change	Same object with new state	New object with modified state
Timeline	Just here and now	Stream of states

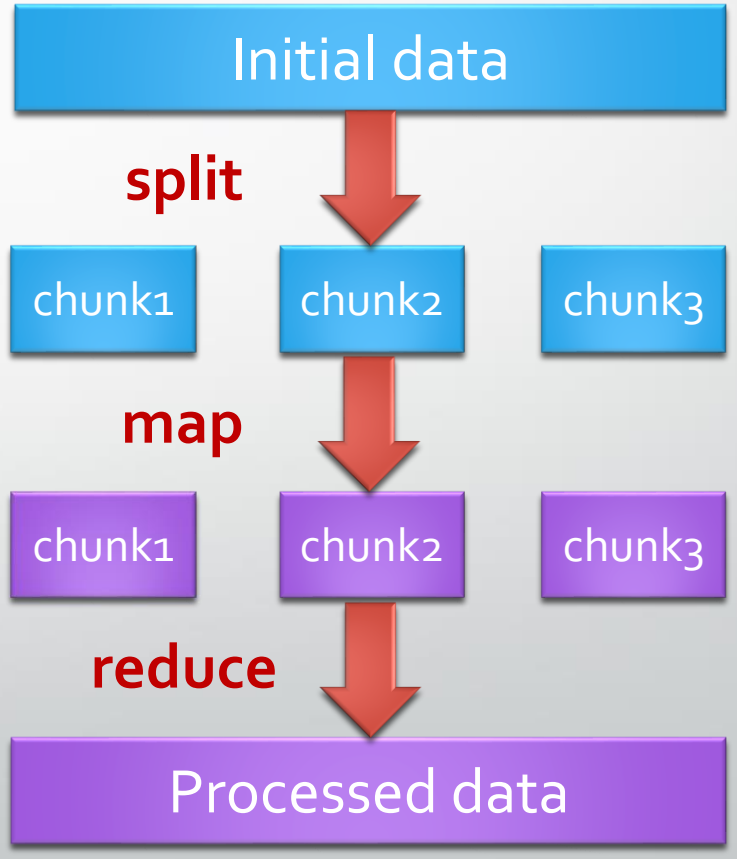
Parallelism: functional and imperative

	Standard imperative programming	Parallel imperative programming	Functional programming
Canonical model of computation	Turing machine	???	λ-calculus
Evaluation order	Total order	Total order for each thread	Partial order
Parallelism support	No	Yes, explicit	Yes, implicit
Synchronization	Not applicable	Semaphore, critical section, barrier, etc.	Not needed
Possible sync errors	Not applicable	Race conditions, dead locks, live licks, etc.	None
Concurrent mutable state	Not applicable	Shared state (memory) with synchronization	Virtually impossible
Parallelism efficiency	Not applicable	**Depends on actual restrictions on evaluation order**	

Data parallelism: split-map-reduce

When an input data could be split into pieces where each one could be processed independently, such pieces could be processed in parallel without any synchronization.

It is applicable both for imperative and functional programming. However, mechanisms could be different.



Parallel stream

Stream can be viewed both as a sequence of data or as a data collection (of unknown size in general case). It is similar wave-particle dualism in physics.

Many stream operations such as map of filter don't care about the particular order of elements' processing. Parallelism could be applied just by replacing their implementations and without any modifications in outer code.

Final efficiency will depend mostly on split and reduce phases and overheads (thread initialization, traffic, etc.)

Parallel map

```
(time (->> (range 1 100001)
           (map #(Math/sqrt %))
           (reduce +)))
;;>>"Elapsed time: 11.783792 msecs"
;;>>2.1082008973917928E7

(time (->> (range 1 100001)
           (pmap #(Math/sqrt %))
           (reduce +)))
;;>>"Elapsed time: 212.571818 msecs"
;;>>2.1082008973917928E7
```

```
(defn heavy-sqrt [x]
  (Thread/sleep 10)
  (Math/sqrt x))

(time (->> (range 1 101)
           (map heavy-sqrt)
           (reduce +)))
;;>>"Elapsed time: 1004.909001 msecs"
;;>>671.4629471031477

(time (->> (range 1 101)
           (pmap heavy-sqrt
           (reduce +)))
;;>>"Elapsed time: 47.131062 msecs"
;;>> 671.4629471031477
```

In Java the same can be done by converting the initial collection into stream using **parallel_stream()** call instead of regular **stream()**

Semi-parallel (cascade) reduce

```
(defn p-reduce-1
  ([value-f init-val coll]
    (p-reduce-1 value-f value-f init-val coll))
  ([value-f merge-f init-val coll]
    (let [chunk-size (int (Math/ceil (Math/sqrt (count coll)))),
          parts (partition-all chunk-size coll)]
      (reduce merge-f init-val
              (pmap (partial reduce value-f init-val)
                    parts)))))

(reduce + 0 (range 1 16))        ;>> 120
(p-reduce-1 + 0 (range 1 16))    ;>> 120
;;will it always be equivalent to reduce?
```

p-reduce: profiling

```
(defn heavy+ [& args]
  (Thread/sleep 10)
  (apply + args))

(time (reduce heavy+ 0 (range 1 16)))
;;>>"Elapsed time: 149.303206 msecs"

(time (p-reduce-1 heavy+ 0 (range 1 16)))
;;>>"Elapsed time: 79.559131 msecs"
```

How to implement pmap?

Easy way:

1. split an input collection into chunks
2. run a separate thread and the regular map for each chunk
3. join to each thread and concatenate results

Cons:

- How to split a collection of unknown size?
- Threads are expensive. One chunk per thread could cost too much.

Future

A future is and object with a provided algorithm (e.g. in form of λ-expression) how to evaluate it into its value.

A future could be:

- run (presumably in separate thread)
- asked for value (will wait until the value will be evaluated)
- monitored for its state (running, completed, failed)

A typical future usage scenario:

1. Initialize multiple futures
2. Run them all (thread pool is advisable)
3. Wait for results and collect them

Thread pool

A thread pool is an object that accepts computational tasks (e.g. futures) and spread them among a set of threads (of either fixed or variable size).

A thread pool runs tasks in the order they provided to it and tries to utilize as many threads as possible.

Number of tasks could be (and usually is) much larger than the pool size. Some tasks can wait for free thread. This is also managed by thread pool.

A thread pool only manages threads and tasks. Synchronization and results acquisitions are out of its scope.

Future & Thread pool: Java

```
ExecutorService executor = Executors.newFixedThreadPool(4);
//create 10 futures first
List<FutureTask<Integer>> futures =
        IntStream.range(0, 10)
                .mapToObj(i->new FutureTask<Integer>(
                        ()->IntStream.range(i*100, (i+1)*100)
                                .reduce(Integer::sum)
                                .getAsInt()))
                .collect(Collectors.toList());
//and run all of them
futures.forEach(x->executor.submit(x));
//now we can collect the result
int result = 0;
for (FutureTask<Integer> x : futures) {
    try {
        result += x.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
System.out.println("Result: " + result);
executor.shutdown();
```

Futures incorrect order

```
public static int FUTURES = 10;
…
ExecutorService executor = Executors.newFixedThreadPool(4);

List<FutureTask<Integer>> futures =
          Stream.iterate(  new FutureTask<Integer>(()->1),
                           prev->new FutureTask<Integer>(
                                    ()->prev.get()+1))
                .limit(FUTURES)
                .collect(Collectors.toList());
Collections.reverse(futures);
futures.forEach(x->executor.submit(x));
try {
   //this will run forever (will wait, actually)
    System.out.println("Last value: " + futures.get(0).get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
executor.shutdown();

//always run futures in the order they were created!
```

Future: Clojure

```clojure
;;create and run a future with provided code
(future (inc 1))
;;do the same but with 0-arity function as argument
(future-call (fn [] (inc 1)))
;;get the result (wait for it if necessary)
(deref f)
@f
;;cancel the given future
(future-cancel f)
;;monitor the state
(future? f)
(future-done? f)
(future-cancelled? f)
;;unlike futures in Java, Clojure's ones will be run immediately
;;using implicit thread pool
;;it minimizes possible problems with dependent futures
;;but restricts thread pool fine tuning

;;futures don't violate the purity, they could be considered as
;;functional primitives
;;unless their dependencies forms a cycle
```

Future safety: deadlock

```
(declare f2)

(def f1 (future
          (Thread/sleep 100)
          (inc @f2)))

(def f2 (future
          (Thread/sleep 100)
          (inc @f1)))

;;it will run forever
@f1
```

Future-based p-reduce

```
(defn p-reduce-2
  ([value-f init-val coll]
    (p-reduce-2 value-f value-f init-val coll))
  ([value-f merge-f init-val coll]
    (let [chunk-size (int (Math/ceil (Math/sqrt (count coll)))),
          parts (partitiona-all chunk-size coll)]
      (->> parts
        ;;common pattern: all the futures must be run first
        (map (fn [coll1]
               (future (reduce value-f init-val coll1))))
        ;;and after that all the results must be collected
        (map deref)
        (reduce merge-f init-val)))))

(time (p-reduce-2 heavy+ 0 (range 1 16)))
;;>>"Elapsed time: 190.200817 msecs"
;;The result is even worse than with regular reduce
;;Why?
```

Future-based p-reduce (fixed)

```
(defn p-reduce-3
  ([value-f init-val coll]
    (p-reduce-3 value-f value-f init-val coll))
  ([value-f merge-f init-val coll]
    (let [chunk-size (int (Math/ceil (Math/sqrt (count coll)))),
          parts (partition-all chunk-size coll)]
      (->> parts
        (map (fn [coll1]
               (future (reduce value-f init-val coll1))))
        ;;do not forget to cancel map's laziness!
        (doall)
        (map deref)
        (reduce merge-f init-val)))))

(time (p-reduce-3 heavy+ 0 (range 1 16)))
;;>>"Elapsed time: 79.757561 msecs"
;;Finally it works!
```

Task C3

Implement a parallel variant of **filter** using futures. Each future must process a block of elements, not just a single element. The input sequence could be either finite or infinite. Thus, the implemented filter must possess both laziness and performance improvement by utilizing parallelism.

Cover code with unit tests.

Demonstrate the efficiency using **time**.

Parallel adaptation example: matrices

Q: Is it possible to implement a parallel version of the square matrices' addition and multiplication?

Q: How to split a data?

Q: Is this parallelism efficient?

Q: Now the system is distributed, each chunk is processed on a separate node and traffic costs. How to minimize traffic and achieve better efficiency.