Optimizing Performance in Real-World Problems

Nikola Peric
nikola.peric03@gmail.com
February 2016

About me

- Full-stack developer at market research firm, Synqrinus http://www.synqrinus.com/

- Synqrinus conducts online surveys and focus groups so a lot of my work has to do with automating the data analysis from those sources

0

The Basics

Wax on, wax off

0

When and why to optimize for performance

- Performance, for most scenarios, is a secondary need
- It arises after the initial application is built

- Optimization allows for your users to be more efficient, effective, or have a better experience

0 First step

- Is to not use any of the functions and techniques I'm going to talk about

- It's about reducing redundant calls in your code
- It's about cleaning up and optimizing your initial code to begin with
- For many every day cases, this along will be enough

0 Tools of the trade

- Benchmark, benchmark, benchmark
- Any change made with performance in mind should be measured
- A more advanced alternative to simply running time across multiple iterations is the Criterium library
- https://github.com/hugoduncan/criterium

1

Memoization

Think memory, but without the r

1 What is memoization?

- `memoize` wraps a function with a basic cache in the form of an atom
- Think of it as "**remembering**" the output to a given input

- The **parameters** passed through to a given function are treated as the **keys** to the map stored in the atom
- When the function is called with the same parameters, there is **no recalculation necessary** and the result is simply **looked up**

1

When should I use memoization?

Do use

- if you are sending the **same parameters** as inputs to **computationally intensive** functions
- if the function calls are *referentially transparent* (i.e. the output alone is sufficient)

Do not use

- if you expect the **output to change** over time
- if there are **side effects** you expect to run within the function
- if your **outputs/inputs are sufficiently large** that they would cost a sizable amount of memory

1 **Problem time!**

Background

- With some of the data we work with there is a map that requires retrieval and formatting from the database before we can work with it
- Often times when one project is being analyzed, the same map of data has to get formatted repeatedly
- This seemed like a perfect opportunity to use memoization

1 Problem time!

Before

```
(defn format-syn-datamap
  [datamap]
  (->> datamap
       (map #(into {}
                   {(keyword (:id %))
                    (:map %)}))
       (apply merge)))

(defn formatted-datamap
  [datamap-id]
  (format-syn-datamap (db/get-datamap datamap-id)))
```

Criterium bench execution time means

12.7 ms

1 Problem time!

Before

```
(defn format-syn-datamap
  [datamap]
  (->> datamap
       (map #(into {}
                   {(keyword (:id %))
                    (:map %)}))
       (apply merge)))

(defn formatted-datamap
  [datamap-id]
  (format-syn-datamap (db/get-datamap datamap-id)))
```

After

```
(def formatted-datamap
  (memoize
    (fn [datamap-id]
      (format-syn-datamap
        (db/get-datamap datamap-id)))))
```

Criterium bench execution time means

12.7 ms → **95.9 ns**

>100,000x faster
(differs based on actual scenario)

fun fact: 1ms=1,000,000ns

1 core.memoize

- If you find yourself using memoize you'll notice that there are some features that would be nice to have, such as...
- ... clearing the cache
- ... limiting the size of the cache (e.g. to speed up access for commonly accessed results, or recently accessed)

- For this, and more, there's core.memoize
- https://github.com/clojure/core.memoize

2

Parallelization (with `pmap`)

This is one the things Clojure's good for right?

2 What is parallelization and `pmap`?

- Parallelization is running multiple calculations at the same time (across multiple threads)
- `pmap` is basically a “parallelized map”
- Note: `pmap` is lazy! Simply calling pmap won’t cause any work to begin

- What pmap tries to do is wrap each element of the coll(s) you are mapping as a future, and then attempt to deref and synchronize based on the number of threads available
- Sounds confusing right? A simpler way to imagine it would be:

```
(doall (map #(future (f %)) coll)
```

2 When should I use `pmap`?

Do use

- if the function that is being mapped is a computationally heavy function
- if we're talking about CPU intensive tasks

Do not use

- if the time saved from running the function in parallel will be lost from coordination of the items in the collection
- if you don't want to max the CPU

Also note

- There are so many other ways to apply parallel processing in Clojure! We'll talk about one more later, but if performance is important to you, you will want to read more about it
- Useful functions: `future`, `delay`, `promise`

2 Problem time!

Background

- We have a collection of maps as the raw data (thousands of items in the coll)
- We want to run a computationally intensive function and use the outputs to generate a new map (`calc-fn`)
- We also want to map this process multiple times, once for each variable we wish to calculate

- Note: for sake of this example, some elements of the following fn have been simplified

2 Problem time!

Before

```
(defn row-calc [data weight-var vars calc-fn conditions]
  (map
    (fn [v]
      (map #(let [[value size] (calc-fn v %1 nil weight-var)]
              {:value       value
               :size        size
               :conditions  %2})
           data conditions)) vars))
```

Note: depending on complexity of arguments, `calc-fn` may be very computationally intensive, or not much at all. I choose a very basic set of arguments for this benchmark

Criterium bench execution time means

27.4 ns

2 Problem time!

Before

```
(defn row-calc [data weight-var vars calc-fn conditions]
  (map
    (fn [v]
      (map #(let [[value size] (calc-fn v %1 nil weight-var)]
              {:value       value
               :size        size
               :conditions  %2})
           data conditions)) vars))
```

After

```
(defn row-calc [data weight-var vars calc-fn conditions]
  (map
    (fn [v]
      (pmap #(let [[value size] (calc-fn v %1 nil weight-var)]
               {:value       value
                :size        size
                :conditions  %2})
            data conditions)) vars))
```

Criterium bench execution time means

27.4 ns → **22.1 ns**

>1.2x faster (~20%)

(differs based on actual scenario)

3

Reducers

More parallelization... and more!

3 What are reducers?

- While we were looking for `pmap` if you wanted a parallel reduce, there's reducers!
- **core.reducer** offers parallelization for common functions such as `map`, `filter`, `mapcat`, `flatten`*

- Imagine a scenario where you are apply a map over a filter
- What if you could compute these not sequentially, but in parallel, i.e. reduce through your collection(s) only once?
- That's the power of reducers

**caveat – some functions in core.reducer do not support parallelization (e.g. `take`, `take-while`, `drop`)*

3 How do I use core.reducers?

- Reference clojure.core.reducers namespace (we will be aliasing the namespace as “r” from here on)
- Create a reducer from one of the following: `r/map`, `r/mapcat`, `r/filter`, `r/remove`, `r/flatten`, `r/take-while`, `r/take`, `r/drop`
- Apply the reduction with one of the following functions: `r/reduce`, `r/fold`, `r/foldcat`, `into`, `reduce`

3 What is `fold`?

- `fold` is a parallalized reduce/combine form of `reduce`
- It is used in the form

```
(r/fold reducing-fn reducer)
```

- `reducing-fn` must be associative
- `reducing-fn` must be a monoid (i.e. give its identity even when 0 arguments are passed)

- `fold` does all this by chunking your collection into smaller parts, and then reducing and combining them back together all while maintaining order
- Essentially it's reduce on steroids

3 When should I use reducers?

Do use

- if you want easy to use parallelism for commonly used functions such a `map` or `filter`
- if you have a large amount of data to apply computations to (see `fold`)
- if you want a parallel reduce

Do not use

- if you don't care for parallelism and really just wanted composed functions that iterate through all items once (in which case, see *transducers* http://clojure.org/reference/transducers)
- if you don't want to max the CPU (for most core.reducer features)

3 Problem time!

Background

- We want to map through a large collection of maps and select a single value from each map
- Then from the result sequence we sum up the values

- This is an excellent test of fold's parallel partinioning/reducing, and r/map's parallelism

3 Problem time!

Before

```
(defn weighted-total [data weight-var]
  (reduce + (map weight-var data)))
```

Criterium bench execution time means

136.3 µs

3 Problem time!

Before

```
(defn weighted-total [data weight-var]
  (reduce + (map weight-var data)))
```

After

```
(defn weighted-total [data weight-var]
  (r/fold + (r/map weight-var data)))
```

Criterium bench execution time means

136.3 µs → **29.4 µs**

>4.6x faster

4 Closing Thoughts

4 Stop

- Does the business value created from pursuing additional optimization outweigh the investment?
- If no, stop
- If yes, continue

4 Finding areas for optimization

- Often times there are multiple areas that can require attention

- Possible elements to look for include...
- map/filter/any manipulation of collections
- Calculations that are known to be computationally expensive (parallelize or memoize if reasonable)

4 Summary

- Benchmark, benchmark, benchmark
- Sometimes a perceived optimization can lose you time under certain scenarios

- Optimize only when reasonable to do so
- There are trade offs to optimization

- Happy efficiency hunting!

Thank you!

Any questions?

You can reach me at

- nikola.peric03@gmail.com