Exploration of parallelization efficiency in the Clojure programming language

Midwest Instruction and Computing Symposium

April 25, 2014

Henry Fellows, Joe Einertson, and Elena Machkasova

Introduction

Our project is a comparison of parallelism methods in the Clojure programming language.

- Relatively new language.
- Designed for efficent parallel operations.
- Recently added new parallel library.

Motivations.

- Interest in using Clojure as an educational tool.
- Using concurrency in functional language.
- Developing parallel algorithms.

Table of contents

Intro to Clojure

- Clojure is a dialect of Lisp.
- Runs on the Java Virtual Machine (JVM).
- First introduced in 2007 by Rich Hickey.
- Immutable data structures.
- Built-in support for parallelism.

Functional Languages and Lisps

Functional Languages

- Clojure is a functional language.
- Treat computation as the evaluation of functions.
- Functional languages avoid direct memory manipulation.

Lisp is a family of programming languages

- Lisp-1 (1958)
- Common Lisp (1984)
- Racket (1994)
- Clojure (2007)

Prefix Notation

Can be generalized to `(function arg1 ... argN)`.

```
(+ 2 3)
=> 5
```

Basic function syntax: `(defn name [args] expr)`

```
(defn add1 [num] (+ num 1))
(add1 3)
=> 4
```

Vectors

A type of collection in Clojure. Accessing items by index is $O(\log n)$.

```
(get [2 7 4 9 5] 3)
=> 9
```

High Order Functions

Functions can take functions as arguments.

```
(map add1 [0 1 2 3 4])
 => [1 2 3 4 5]
```

Another high order function, `reduce`.

```
(reduce + [1 2 3])
 => 6
```

The combination of `reduce` and `map`.

```
(reduce + (map sqrt [1 4 25]))
 => 8
```

Concurrency

- Most processors are now being built with multiple cores.
- Concurrency is the execution of multiple computations simultaneously.
- Programming concurrent programs is *considered hard*.
- Deadlocking: two tasks are waiting for resources that the other task holds.
- Immutable data structures make concurrency easier.

Parallel Computation in Clojure

Clojure has several methods of parallelism.

- `pmap` is one of the early methods of parallelism in Clojure.
- Reducers is a new library introduced in 2012.

Pmap

- A parallel version of `map`.
- Has the same syntax as `map`.
- On a sufficiently large collection, it will create additional threads.

```
(pmap add1 [0 1 2 3 4])
=> [1 2 3 4 5]
```

Reducers

- Released by Rich Hickey in May 2012.
- Built on Java's fork/join framework.
- Reducers provides parallel higher-order functions, with the same names as their serial counterparts.
- `r/fold` is used in place of reduce.

Implementation of Reducers

- All collections come with a traversal mechanism.
- All reducers functions (`r/map`, `r/filter`) except `r/fold` provide a recipe.
- `r/fold` causes the evaluation of all recipes attached to a collection in parallel.
- Fork/Join framework creates one thread per core (as reported by OS).

```
(r/fold + (r/map sqrt [1 4 25]))
 => 8
```

Test Structure

- Computationally expensive operations on large sets of integers

Three tests:

- Count-primes

```
(reduce + (map (one-if-prime-else-zero [...])))
```

- Sum-primes

```
(reduce + (map (zero-if-composite-else-n [...]))
```

- Sum-sqrt

```
(reduce + (map (sqrt [...])))
```

Test Structure, Continued

Standard version:

```
(reduce + (map (sqrt [...])))
```

Version with pmap:

```
(reduce + (pmap (sqrt [...])))
```

Version with r/fold:

```
(r/fold + (map (sqrt [...])))
```

Version with r/fold and r/map:

```
(r/fold + (r/map (sqrt [...])))
```

Test sub-Structure

Name	Description
map + reduce	serial map, serial reduce
pmap + reduce	parallel map, serial reduce
map + r/fold	serial map, parallel reduce
pmap + r/fold	parallel map, parallel reduce
r/map + r/fold	reducers parallel map, parallel reduce
r/fold	parallel reduce

Table : Configurations for our tests

The `r/fold` configuration does not have a mapping phase: the test code was rewritten to make it work with a single reduce.

Data Sets

Count-primes

- Collection is 100,000 random integers between 0 and 1 billion.
- repeated 100 times, with new data each time.

Sum-primes

- Collection is 10,000 random integers between 0 and 1 billion.
- repeated 1000 times, with new data each time.

Sum-sqrt

- Collection is 10,000 random integers between 0 and 1 billion.
- repeated 1000 times, with new data each time.

Test Enviroments

- an Intel i7 CPU, with 4 cores.
- an Intel i5 CPU, with 2 cores.
- an AMD FX-8350 CPU, with 8 cores.

Sum-Primes Results

Run	reduce, map	reduce, pmap	r/fold, pmap	r/fold, map	r/fold	r/fold, r/map
i7	208.0	66.4	61.7	207.0	57.2	54.6
i5	279.3	250.6	284.3	280.8	132.0	131.0
AMD	266.9	225.1	248.4	275.5	59.2	63.6

Table : Sum-Primes averages (ms).

Count-Primes Results

Run	reduce, map	reduce, pmap	r/fold, pmap	r/fold, map	r/fold
i7	2084.6	604.5	597.1	2065.7	535.8
i5	2802.8	2567.7	2585.6	2774.0	1269
AMD	2662.2	2411.3	2426.6	2647.9	557.6

Table : Count-Primes averages (ms).

Sum-Sqrt Results

Run	reduce, map	reduce, pmap	r/fold, pmap	r/fold, map	r/fold
i7	115.4	128.7	109.7	28.6	30.5
i5	120.1	401.3	414.0	60.0	58.0
AMD	115.9	359.5	367.6	32.8	32.4

Table : Sum-Sqrt averages (ms).

Pmap and Thread Thrashing

Pmap is unreliable.

- Running times ranging from close to the best parallel runs, to worse than serial.
- Close to 2.5 times slower than serial methods.

Pmap creates too many threads.

- This causes *thread thrashing*.
- The number of treads leads to excessive context switching.
- Causing the process to choke on its own overhead.

Reducers

- Reducers is *fast*, running 15% faster than pmap, when pmap was working well.
- `r/fold` + `r/map`, runs as fast as the one step `r/fold`.
- Relatively reliable.

Environments

Intel i7

- Resistant to thread thrashing.
- Caused by hyper-threading?

Intel i5

- Slowest machine tested
- Not resistant to thread thrashing.

AMD Fx-8350

- Slightly resistant to thread thrashing.
- Does not scale as well.
- Due to micro-architecture?

Conclusion

There's a lot to look into;

- Thread balancing in reducers.
- Optimal thread management.
- The effects of CPU architecture on thread thrashing.

We still want to continue on our main interest, parallel algorithm development in functional languages.

The authors thank Jon Anthony for helpful discussions and methodology suggestions.