# Exploration of parallelization efficiency in the Clojure programming language

Henry Fellows, Joe Einertson, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
fello056@umn.edu, eine0017@umn.edu, elenam@umn.edu

## Abstract

In modern processing environments, concurrency - simultaneous execution of computations - is an important tool. Despite the importance of parallelism, it is often poorly supported, or awkward to use effectively. Clojure is a Lisp dialect designed for concurrency and portability, released in late 2007. Clojure features immutable data structures and built-in support for concurrency. In 2012 `clojure.core.reducers` was added to the language: a novel library that provides a set of high-level functions for even more convenient and efficient parallel processing of data collections.

In this study, we focus on testing the various methods of parallel processing in Clojure and explore the functionality of the reducers library. Timing the execution of computationally expensive and highly parallelizable data processing allows us to directly observe the differences between different methods of parallelism provided in Clojure. We present the results, analysis, and conclusions.

# 1 Introduction

Clojure is a dialect of Lisp, developed and first introduced in 2007 by Rich Hickey [2], with a focus on concurrency and portability. By compiling into Java bytecode, Clojure offers programmers ultra-portable code that leverages the widespread nature of the Java virtual machine. Recently, Rich Hickey released a new library, reducers, for the Clojure programming language that introduces major changes to the concurrency tools offered in Clojure's core library. A subject of research we were interested in pursuing was the development of concurrent algorithms. A run-time comparison of parallel methods is a good way of exploring that topic. Since Clojure provides high-level libraries that allow writing correct and efficient parallel code without an extensive OS background, exploring Clojure reducers library can open a way into writing parallel algorithms to students early in their academic career and to others with a similar degree of preparation. To that end, we sought to compare some of the built in methods for adding concurrency to Clojure code. We provide a introduction 2 to Clojure, reducers, and another concurrency tool, pmap. Our methodology is presented in section 3, leading to our results, discussion, and future work in sections 4 and 5.

# 2 Background

This section is an introduction to Clojure, the Reducers library, and the `pmap` function.

## 2.1 Clojure

The Lisp family of languages has several features in common, the most visible of which is the polish-prefix notation. Cojure, as a Lisp, uses the polish-prefix notation, which can be generalized to `(function arg1 arg2 ... argN)`. For example, after interpretation,

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

Here => denotes the Clojure interpreter response, so `(+ 2 3)`, when typed in the Clojure interpreter, evaluates to 5. Note that + in Clojure is a function, and not an operation, as in C or Java.

Functions in Clojure are created using the `defn` macro, which has the basic syntax `(defn name [args] expr)`, where `name` is the function's name, `args` are function arguments listed with spaces in-between, and `expr` is the expression defining the behavior of the function.

```
(defn add1 [num] (+ num 1))
(add1 4)
=> 5
```

The first line in the above example defines the function `add1`, and the second line applies it to 4. In the first line `add1` marks the name of the function, `num` is the argument, and `(+ num 1)` is the expression.

Below we introduce vectors, an important Clojure data type for our discussion – a type of collection in Clojure. Vectors are a collection of items, indexed by continuous integers: accessing items by index has a $O(log_{32}n)$ complexity. They are denoted by `[brackets]`.

```
(get [1 2 3 4 5] 3)
=> 4
```

Here, `get` retrieves an element from a vector at index 3 (starting at zero), with the syntax `(get collection index)`. Lisps have a philosophy of treating code as data, meaning that all functions can take functions (or other code) as arguments. Because of that philosophy, most Lisps, including Clojure, provide a rich set of higher-order functions that allow for powerful manipulation of data. Clojure includes `map`, a function that takes a function and a collection and then applies the function to every item in the collection, returning the resulting collection. The type of the collection that gets returned is not a vector, but a more general collection type, so it is included in parentheses, and not brackets. However, for the most part this distinction is not important for our discussion.

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

Pre-defined functions in Clojure include `reduce`, also known as fold in other Lisps, which recursively applies the function to two arguments: an element of a collection and the result of applying a reduce to the rest of the collection. This process is repeated until the end of the collection is reached.

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

The next of the major high-order functions is `filter`, which takes a predicate (a function that returns a boolean and has no side-effects) and a collection and returns a new collection of the items in the original collection for which the predicate (when applied to the item) returns true.

```
(filter even? [1 2 3 4 5])
=> (2 4)
```

Another common feature in Clojure is *laziness*. Laziness is the concept of delaying evaluation of an expression until the value it returns is needed. For example, `if` statements in many languages are lazy. In pseudocode, `if a then b else c` is an example of lazy evaluation. Normally, `b` and `c` are only evaluated depending on the value of `a`, and the `if` wouldn't work if both `b` and `c` were evaluated. In Clojure, the most common example of laziness, other than `if`, is sequences. For example, if you have a sequence that has a hundred expressions but only retrieve the first 50, only the first 50 are evaluated. More interestingly, you can have infinite sequences: the function `range`, called with no arguments, returns the infinite sequence of non-negative integers, starting at 0:

```
(take 10 (range))
=> (0 1 2 3 4 5 6 7 8 9)
```

Of course, these values only exist after they are requested by `take`, which is a function that takes the first $n$ elements of a collection. Most of higher-order functions that produce collections are lazy. However, functions that compute a single value when given a collection, such as `reduce`, perform what is referred to as *eager* evaluation, i.e. they fully evaluate their parameter collection.

## 2.2 Introduction to concurrency

Lately, processor clock speed has plateaued: simply running the CPU faster is becoming increasingly more difficult from the engineering standpoint. In order to maintain throughput growth, processors are now being built with multiple cores. In order to be able to effectively use modern processors, programmers must begin to use concurrency. Concurrency is simply the execution of multiple computations simultaneously. Most programming languages support concurrent programming, but many provide poor support, or have awkward interfaces. Programming concurrent programs is hard, especially when it comes to shared resources (commonly memory). Simply speaking, having two parts of a program access one resource at the same time can cause problems. Modern concurrency systems use some sort of access management to prevent this, by giving one thread exclusive access to a resource while the thread reads or writes to it. Unfortunately, this model can cause problems such as deadlocking, where two tasks are waiting for resources that the other task holds. If not resolved, both threads will wait forever.

Clojure uses immutable datatypes by default; the value of an immutable datatype cannot be changed after it has been created. New copies of an object are created every time the object is modified, making it impossible for threads to modify objects as they are read, therefore making it impossible to accidentally leave an object in an inconsistent state. However, variables that contain references to objects often need to be updated to point to the newest copy of the object. When a need for a direct manipulation arises, Clojure provides several built-in models of accessing shared resources, depending on the synchronization requirements. For instance, if a resource needs to be updated eventually, but the order of updates by multiple threads and the exact timing of each update do not matter, one can use agents. Software Transactional Memory (STM) provides a more fine-grained control over synchronization of updates by bundling related updates to multiple resources into a transaction that operates in an all-or-nothing manner. When a thread accesses a resource, it records its actions in a log; near the end of the transaction, is checks to see if another thread has modified the resource. If the resource has been accessed, it restarts, and attempts to successfully modify the resource again.

Additionally, Clojure provides drop-in replacements for the built in higher order functions (discussed in Section 2.1) by their concurrent counterparts that allow programmers not familiar with concurrency to use concurrency in their programs.

## 2.3 Introduction to pmap

A parallel version of `map` called `pmap` was an early implementation of a parallel of a higher-order function. With the same arguments as `map`, `pmap` is a drop-in replacement. For example, one can replace `map` in the example in Section 2.1 by `pmap`, and (with sufficiently large input data) it will utilize multiple cores on a computer, likely resulting in increase in speed:

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

Importantly, `pmap` is semi-lazy; it tries to use as little processing as possible, only returning results as needed; that is, it can occasionally appear to evaluate as a serial function. If the value is used (by a function, for example), pmap will evaluate the entire expression immediately (i.e. eager evaluation), and always in parallel [4]. A function `doall` in Clojure forces eager evaluation on lazy expressions. If we imagine a long running function, `expensive-function`, that runs for 3000ms, we can time it and see the difference `doall` makes.

```
(time (pmap expensive-function [1 2 3 4]))
=> 12000 ms
(time (doall (pmap  expensive-function  [1 2 3 4])))
=> 3000 ms
```

When `doall` forces eager evaluation, it runs faster - roughly by a factor of the number of cores (in this case, four). Of course, this is an hypothetical example; non-parallel operations and overhead make practical examples messier. `pmap` utilizes multi-threading, creating a thread for every chunk of the collection it is trying to process. Too large number of threads may result in too frequent context switching attempts, which may stall the program (the effect known as *thread thrashing*).

## 2.4 Introduction to reducers

Reducers library was released by Rich Hickey in May 2012 [3]. Reducers provides higher-order functions for parallel manipulation of collections. Most functions provided by reducers are drop-in replacements for their serial counterparts, requiring minimal changes to a program when changing it from serial to parallel, although they may impose stricter restrictions on functions that they take as parameters. The names of the reducer functions for the most part directly match the corresponding serial functions. For instance, `r/map` is a reducer version of `map`, `r/filter` is a reducer version of `filter`, and so on. A notable exception is `r/fold` which is a reducer version of `reduce`.

Reducers are implemented using Java Fork/Join framework that manages distribution of tasks between multiple threads of control. The task balancing algorithm allows idle threads to "steal" work from threads that are busy, thus providing efficient work distribution between processors.

In addition to providing efficient work distribution via Fork/Join, reducers optimize the number of collection traversals, while utilizing parallel processing. Consider, for example, a serial `reduce` applied to the result of a serial `map`. Note that `map` is lazy and `reduce` is eager, i.e. it fully evaluates a sequence it is applied to. Given that `map` returns a lazy collection, only evaluation of `reduce` will force `map` to be evaluated. However, if one were to attempt to perform this computation in parallel, they would have a problem since by the time the lazy result of `map` is returned to `reduce` and needs to be evaluated, the execution is back to a single-threaded model, thus making it impossible to take advantage of a hypothetical parallel execution of `map`.

A straightforward alternative to this scenario would be to force `map` to fully execute by wrapping it into a `doall`. This would force parallel execution of `map`, assuming that it is capable of such execution. However, once its execution is done, `reduce` (regardless of whether parallel or not) will traverse the collection again, to perform its own operation on each element. This means that we would have a separate collection traversal for each function applied to it.

Reducers allow us to avoid these issues: functions applied to a collection via `r/map`, `r/filter`, etc., are accumulated, but not applied until `r/fold` is applied. Once `r/fold` is applied, the collection is divided into chunks that are distributed among threads, and all accumulated functions are applied to each element as a part of the reducing process. Thus the collection is traversed exactly once (when `r/fold` is applied), and in parallel.

For instance, if we apply a square function via `r/map`, then filter the result by selecting elements larger than a certain threshold via `r/filter`, and finally sum up the resulting elements via `r/fold`, the collection (assuming that it is sufficiently large) would be divided among the CPU cores, and all three operations (squaring, checking against the threshold, and the summation, if needed) would be applied to each element at the same time. The details of how this work is allocated to different threads and different cores lie in the implementation of the Fork/Join framework, and may not be immediately obvious.

# 3 Methodology

## 3.1 Test structure

To determine the performance implications of the new reducers library, we have created several test examples that perform somewhat computationally expensive operations on large sets of integers (10,000 or 100,000, depending on the example). For each example and for each randomly generated data set we ran the test in six different configurations, one of which uses `r/fold` without a separate map-like traversal, whereas the other five configurations use a combination of a version of map and a version of reduce, some single-threaded, and some multi-threaded.

The test configurations are summarized in table 1. Here `map` and `reduce` are the standard serial functions, `pmap` is a parallel map as described in Section 2.3, and `r/fold` and

| 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 1: Configurations for our tests

`r/map` are reducers functions, as described in Section 2.4. We used the default settings for all of these functions. Note that because the r/fold configuration does not have a mapping phase, the test code for it needed to be rewritten slightly to move the functionality that was done by map in the other tests into the reducing stage. Section 3.2 provides details of our testing code. Also important is to note that our tests were run with new data each time.

Since the requirements for parallelizing reduce are more strict than for parallelizing map, this many-part methodology allows us to analyze not only running time, but also to determine whether parallel reduction offers such a speedup over single-threaded or parallel map-based solutions so as to warrant the additional overhead of coding it.

## 3.2 Functions used in testing

Two of the tests described are centered on determining which numbers in the collection are prime. Specifically, we use a probabilistic algorithm called the *Fermat primality test*, which takes two parameters: a number to test for primality, and an integer that determines the number of trials to be run. The higher the number of trials, the more precise the result and the more processing power used. We set the maximum number of trials per number to 5 trials, which gives a high degree of accuracy for a moderate amount of processing power. Fermat primality test is used for very large numbers since faster algorithms exist for smaller numbers. We use numbers on the order of one billion.

The Fermat primality test relies heavily on exponentiation and modular arithmetic, operations which are relatively expensive for large numbers. The specific mathematics behind the Fermat primality test are tangential to and beyond the scope of this paper. The reasons this test was chosen is because it is simple to implement, computationally expensive, and an example of a real-world operation which may benefit from Clojure's reducers.

### 3.2.1 Compare-count-primes

The first test we describe is an algorithm to determine the number of prime integers in a collection of integers. In the r/fold implementation, we use a simple reduce operation to count the number of primes:

```
(defn reduce-num-primes [n count]
   (if (p/fermat-test n 5)
         (inc count)
         count))
```

Here `p/fermat-test` is our function for performing the primality test; it returns `true` if the number is prime, `false` otherwise. `n` is the number being tested, `count` is the counter for the prime numbers in a collection, `inc` is a function that increments its parameter by 1, and 5 is the number of tries that the primality test is going to attempt trying to disprove that `n` is prime. Simply, for each element which is found to be prime via the Fermat primality test, we increase the running sum of primes by one (starting from 0). This implementation leverages r/fold, meaning it is a parallel reduce operation.

As described previously, we tested the pure r/fold implementation against configurations that used varying combinations of reduction and mapping functions. In these cases, we used this mapping function:

```
(defn count-primes-pre-reducer [n]
    (if (p/fermat-test n 5) 1 0))
```

Here the work of determining the primality of the numbers is moved into a mapping operation, which returns 1 for each element if prime, or 0 if not. Then we use r/fold to sum the result in parallel. This is simpler from an implementation standpoint, as our mapping function does not have to conform to the restrictions of r/fold, and summation is trivial to use with r/fold. Note that this functionality could have also been implemented using `filter`, but we chose to test it with map-like functions only, for uniformity with other tests.

The data used for this test is a collection of 100,000 random integers uniformly distributed between 0 and 1 billion. This test was run 100 times, with new data each time.

### 3.2.2 Compare-sum-primes

The second test we perfomred is similar to compare-count-primes, but sums the values of all primes (ignoring composite numbers) rather than simply counting them. This is a more computationally expensive test, since the numbers being *summed* are very large, rather than 0s and 1s. The purpose of this test is to add computational complexity to the reducing phase to determine what effect the distribution of work has on running time.

First, there is our basic, all-in-one reduce function:

```
(defn reduce-sum-primes [sum n]
  (if (p/fermat-test n 5)
         (+' sum n)
         sum))
```

Here `sum` is the running sum of primes. `+'` is addition on large numbers in Clojure since regular + has a restriction on the size of numbers it can be applied to.

The mapping function used during the second stage of testing is similarly simple:

```
(defn sum-primes-pre-reducer [n]
    (if (p/fermat-test n 5) n 0))
```

This test also used a collection of 10,000 random integers uniformly distributed between 0 and 1 billion, but was run 1,000 times.

### 3.2.3 Compare-sum-sqrt

The final test we describe utilizes another computationally-expensive algorithm: integer square root. This method of computing a square root returns the floor of the exact square root of a number. Like computing primality, calculating integer square roots is a computationally difficult task, which represents a good opportunity to utilize and measure parallelism.

To calculate the integer square root, we use a Clojure math library, Clojure Numeric Tower. In our code, we refer to this library as "cljmath."

This test is a straightforward "calculate-and-sum" algorithm, and as such its one-pass reduce function is very basic:

```
(defn reduce-sum-sqrt [a b]
    (+' a (cljmath/sqrt b)))
```

For the multi-step configurations, our map function is simply the `cljmath/sqrt` function: we calculate the square root as the map step, and sum the square roots as the reduce step.

This test used a collection of 100,000 random integers uniformly distributed between 0 and 1 billion, and was run 1,000 times.

## 3.3 Execution methodology

Following the methods shown above, we ran the tests on several different machines referred to as box1, box2, and box3.

- Box 1 has an Intel i7 (4700MQ) CPU, with 4 hyper-threaded cores, running Windows 7.
- Box 2 has an Intel i5 (650) CPU, with 4 cores, running Fedora 18.
- Box 3 has an AMD FX-8350 CPU, with 8 cores, running Fedora 18.

The full set of tests was run three times on each machine. The first run is discarded due to Java virtual machine warm-up, a period where the program runs significantly slower than it would normally [1] . This period is caused by the JVM preforming dynamic optimization and lazy loading relevant portions of code. In order to provide data that is unambiguous

and easy to interpret, we omit that portion of the data from the results. It is interesting to note that the ratio between each configuration in a test remained constant, despite warm-up.

# 4 Results and discussion

## 4.1 Results

In this section we show the results of running our tests across the machines mentioned above: in our tables, we sometimes abbreviate "reduce" to "red". We use `reduce + map` to provide a baseline which is just a serial run on one core; tests running at similar times are noted as running at *serial time*.

| Machine | red + map | red + pmap | r/fold + pmap | r/fold + map | r/fold | r/fold + r/map |
|---|---|---|---|---|---|---|
| Box 1 | 208.0 | 66.4 | 61.7 | 207.0 | 57.2 | 54.6 |
| Box 2 | 279.3 | 250.6 | 284.3 | 280.8 | 132.0 | 131.0 |
| Box 3 | 266.9 | 225.1 | 248.4 | 275.5 | 59.2 | 63.6 |

Table 2: Sum-Primes averages (ms).

For the Sum-Primes set of tests (see table 2), in the first four tests, `reduce+map`, `reduce+pmap`, `r/fold+map`, `r/fold+pmap`, almost all the combinations run in serial time, except for Box 1. Box 1 has `reduce + pmap` and `r/fold + pmap` running slightly slower than `r/fold` and `r/fold + r/map`. `r/fold` and `r/fold + r/map` run between 280% and 110% faster than serial time.

| Machine | reduce + map | reduce + pmap | r/fold + pmap | r/fold + map | r/fold |
|---|---|---|---|---|---|
| Box 1 | 2084.6 | 604.5 | 597.1 | 2065.7 | 535.8 |
| Box 2 | 2802.8 | 2567.7 | 2585.6 | 2774.0 | 1269 |
| Box 3 | 2662.2 | 2411.3 | 2426.6 | 2647.9 | 557.6 |

Table 3: Count-Primes averages (ms).

For the Count-Primes set of tests (see table 3), in the first four tests, `reduce + map`, `reduce + pmap`, `r/fold + map`, `r/fold + pmap`, almost all the combinations run in serial time, except for Box 1. Box 1 has `reduce + pmap` and `r/fold + pmap`somewhat slower than `r/fold`. `r/fold` runs between 290% and 100% faster than serial time.

| Machine | reduce + map | reduce + pmap | r/fold + pmap | r/fold + r/map | r/fold |
|---|---|---|---|---|---|
| Box 1 | 115.4 | 128.7 | 109.7 | 28.6 | 30.5 |
| Box 2 | 120.1 | 401.3 | 414.0 | 60.0 | 58.0 |
| Box 3 | 115.9 | 359.5 | 367.6 | 32.8 | 32.4 |

Table 4: Sum-Sqrt averages (ms).

For the Sum-Sqrt set of tests (see table 4), The first three tests, excepting Box 2, run in serial time. Box 2 runs much longer than serial time in `reduce + pmap` and `r/fold + pmap` with `reduce + map` running in serial time. `r/fold` and `r/fold + r/map` runs on all boxes between 270% and 110% faster than serial time

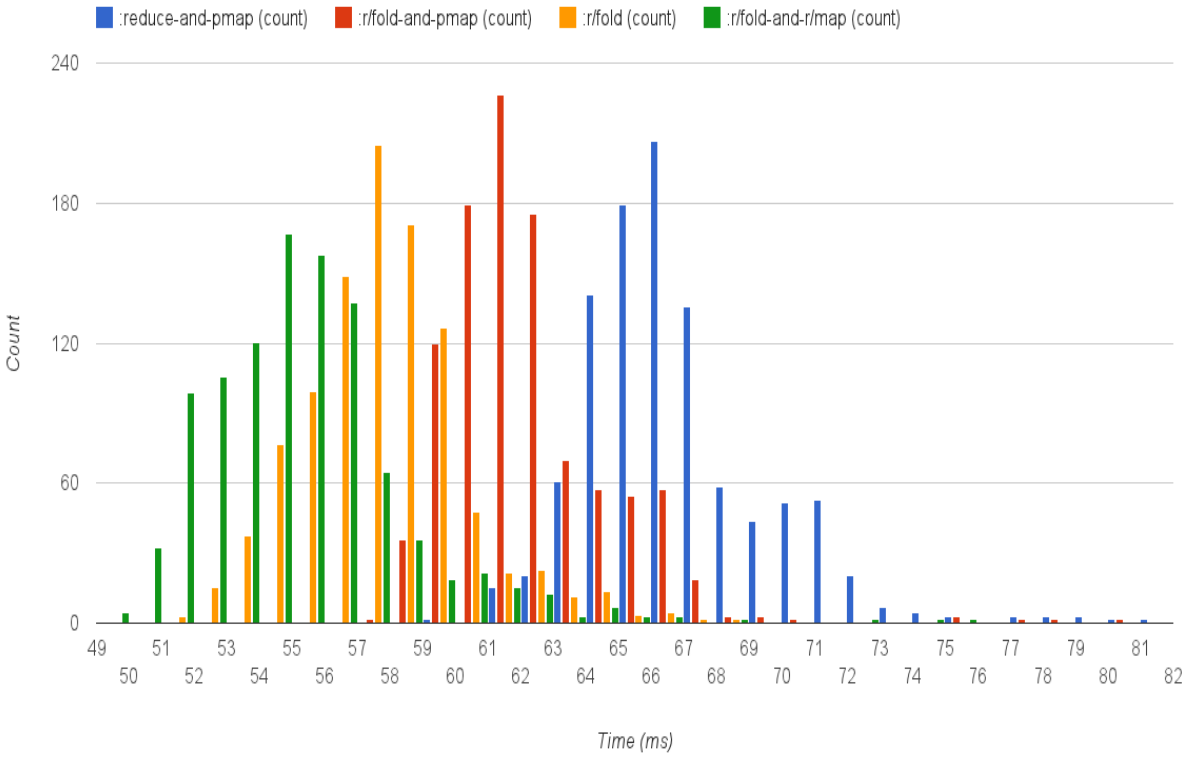


Figure 1: Parallel combinations in Compare-Sum-Primes, Box 1, Run 2 (ms)

Figure 1 shows the distribution of function run times for configurations that run faster than serial time; because this is Box 1, `reduce + pmap` and `r/fold + pmap` are running properly, in parallel. Figure 2 shows the distribution of function run times for the configurations that run in serial time. Only `reduce + map` and `r/fold + map` run in serial time.

A typical mode for a run of all 6 configurations was to find that the distribution was in two peaks: one for the functions running in the serial time and and the other one for those taking advantage of parallelization. Figure 1 is slightly atypical since `reduce` + `pmap` was grouped with the functions running in parallel time rather than serial time.

## 4.2 Possible influence of machine differences

### 4.2.1 Box 1

Box 1, with an Intel I7-4700MQ CPU, was the fastest machine tested by a fair margin. It was also the only box where `pmap` ran faster than `reduce + map`. We suspect that

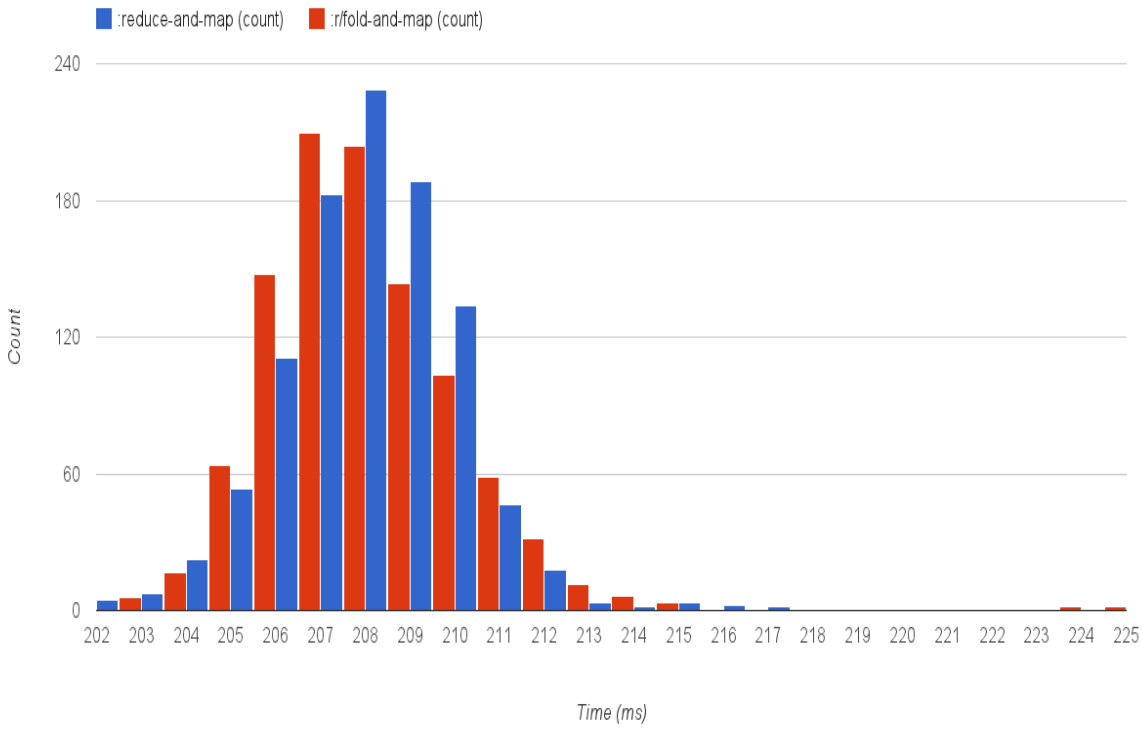


Figure 2: Serial combinations in Compare-Sum-Primes, Box 1, Run 2 (ms)

`pmap` is causing thread thrashing, and given that this machine is the only one with hyper-threading, it seems likely that hyperthreading has some effect on the CPU's ability to manage large amounts of threads without thrashing. Reducers ran close to the expected maximum speedup given the number of cores - 300%.

### 4.2.2 Box 2

Box 2, with an Intel I5-650 CPU, was the slowest machine tested. Notably, `pmap` ran much slower than serial time on compare-sum-sqrt. This machine does not have hyper threading, and `pmap`'s performance in that test is what lead us on to thread thrashing in the first place. Reducers ran close to the expected maximum speedup given the number of cores - 300%.

### 4.2.3 Box 3

Box 3 was our only AMD machine, running a fx-8350 CPU. This machine, while also without hyper-threading, still handled `pmap` better than Box 2. It did not, however, reach the expected speedup - as an 8 core CPU, using reducers should show a 700% improvement, but it actually ran near a 300% improvement. The fx-8350 CPU is technically an 8 core CPU; it's architecture, where each pair of cores is packaged into a module that shares some components between cores [5], may have something to do with this.

### 4.3 Discussion

We have found that the reducers library, in all tested cases, reduces the execution time significantly. In both Intel machines, the reduction was close to a factor of the number of cores. The AMD fx-8350 machine tested did not show this same pattern. We suspect that the somewhat novel Bulldozer architecture, where each pair of cores is packaged into modules that share components [5], was a factor.

Combining reducers functions, such as `r/fold` + `r/map`, runs within a standard error of the one-step `r/fold` method. It seems likely that they have the same, or approximately the same runtime. This was expected since, as we discussed in Section 2.4, reducers perform all their operations in a single parallel traversal of a collection.

There are a few interesting observations about `pmap` based on our results. Firstly, configurations with `pmap` perform the same, regardless of whether a serial `reduce` or a parallel `r/fold` was used. This implies that the collection returned by `pmap` was not suited for optimizations performed by `r/fold`. There is a variety of collection types, with various degree of laziness, that can be returned from a mapping function. Further studies are needed to determine the exact collection type and how it affects the subsequent application of reduction. One should note that `pmap` and reducers were never designed to work together, so it is not surprising that the benefits of `pmap` do not prompt benefits of `r/fold`.

Another curious observation about `pmap` is that it can produce running times ranging from close to the best parallel runs (such as just r/fold configuration) to times worse than serial. In its best results `reduce` + `pmap` ran much faster than serial time, and was still slower than reducers only by 15%. However, in the worst cases `pmap` was up to 244% slower than serial methods. We observed that `pmap` generates a large thread poll, so thread thrashing (see Section 2.3) is a likely explanation for this slow-down. Unless one has fine-grained control over the number of threads `pmap` generates, it is difficult to predict whether using `pmap` would be beneficial for a given problem on a given machine.

## 5 Conclusions and future work

We conclude that the Reducers library is a significant improvement over `pmap` in efficiency, providing a faster, more stable, and predictable platform for concurrency. `pmap` is highly dependent on environmental conditions, especially if used simply as replacement for `map` without any specific adjustments for multi-core load balancing. Both reducers and `pmap` show machine-specific behaviors, likely due to the differing micro-architectures of the respective CPUs. `r/fold` works the same as `r/fold + r/map`, confirming that using r/map and other functions in proper modular style does not lead to slower behavior.

### 5.1 Future Work

Over the course of the study, we found several new questions, largely stemming from reducers. Firstly, we would like to find the source of the variation in observed runtime. Secondly, we would like to look more into the specifics of threads behavior and load balancing for reducers. Is there an optimal number of threads? What effect does CPU architecture have on Clojure's concurrency? We also would like to understand the behavior of `pmap`better: finding that a parallel function is running slower than a serial function is unusual. While we do have a hypothesis as to why we see this behavior, we do not have enough information to conclude our hypothesis is correct. Then, there is still the overarching question of our original goal: what are good approaches to developing more complex concurrent programs in Clojure, and would these approaches be accessible to programmers without a background in operating systems.

## 6 Acknowledgments

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

## References

[1] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[2] R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[3] R. Hickey. Reducers - a library and model for collection processing, 2012.

[4] Z. Kim. Clojuredocs - pmap, 2010.

[5] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh. Design of the two-core x86-64 amd bulldozer module in 32 nm soi cmos. *Solid-State Circuits, IEEE Journal of*, 47(1):164–176, Jan 2012.