Macros for straightforward parallelism in Clojure

Zmick, David
zmick2@illinois.edu

April 11, 2016

1 Introduction

Clojure is a lisp–like language running on the Java Virtual Machine. The language's default immutable data structures and sophisticated Software Transactional Memory (STM) system make it well suited for parallel programming. Because Clojure runs on the Java Virtual Machine, Clojure developers can take advantage of existing cross–platform parallelism libraries, such as Java's excellent `ExecutorService` framework, to write parallel code.

However, taking advantage of Clojure's parallel potential is not entirely straightforward. STM has proven to be very successful as a clear construct for concurrent programming[5], but these constructs are often too low level to be of much use to developers whose central concerns are not parallelism[3].

As a result, there are a variety of libraries designed to allow developers to take advantage of the parallelism potential in Clojure. Clojure builtins such as pmap[1] and the new reducers[2] library provide data parallel sequence manipulation and transformation functions. Third—party libraries like Tesser[3] and Claypoole[4] provide more data parallel APIs with slightly different goals than the builtin functions. Developers have a good relationship with data parallel problems[7], but Clojure's nature as a functional language with immutable structures also makes it possible to easily exploit control parallelism (also known as task parallelism[2, 8]). An astute reader may observe that both Claypoole and the Clojure standard library include task parallel functions. These will be discussed in the Section 2

Using Clojure's macro system, I have implemented a set of macros which allow developers to take advantage of Clojure's parallelism potential when their existing code is written such that parallelism is exposed through control flow. I have shown that it is possible to attain reasonable degrees of parallelism with minimal code changes, with respect to serial code, using these macros. The intended users of these macros are developers whose primary concern is not performance, but may benefit from a simple mechanism with which they can take advantage of the many cores in their machines. Developers who are extremely concerned with performance and want a high degree of control should turn elsewhere (perhaps even to Java) to write their highly tuned code.

[1] `https://clojuredocs.org/clojure.core/pmap`

[2] `http://clojure.org/reference/reducers`

[3] `https://github.com/aphyr/tesser`

[4] `https://github.com/TheClimateCorporation/claypoole`

2 Related Work

2.1 Clojure and other lisps

Makes lots of threads, unatural blah. Note sure what's bad about lparallel other than it isn't implemented in Clojure. Clojure also has access to Java's libraries, like ForkJoin.... but hard to use and not really a part of the Clojure language.

Briefly, a `ForkJoinPool` is an executor for lightweight (often recursive) tasks. The execution engine uses a work stealing scheduler to schedule the lightweight tasks across a thread pool.

2.2 Other languages

Compiler support is bad. Performance is quite good. Only real downside is that these are not implemented in Clojure. ForkJoin pools are essentially Cilk [1, 4]. Both provide the support and runtime needed to easily parallelize code, but neither give programmers the ability to mark a function as "parallel" then let the macro system deal with granularity and the (small) syntactic overhead (cilk has very little extra syntax).

3 Mostly Pure Functions

Before discussing the macros I've implemented, I need to loosely define a "mostly pure" function. A mostly pure function is a function that has no side effects which directly impact the values of other user-defined values, at the current level of abstraction. Mostly pure functions can be reordered (or interleaved) without impacting the values of user variables, although the change may impact I/O behavior and output order of a program. In some cases, the order of certain side effects may not matter to a programmer. For example, it may not matter if we reorder `(download file1)` and `(download file2)` for a programmer writing a web scraper, but it may matter to a programmer writing a I/O constrained server. When a programmer feels that it may be acceptable to change the order of execution of calls to mostly pure functions (the call is "portable"), we can reorder them subject to these constraints:

1. A call to a mostly pure function f in a block B in a function's control flow graph can safely be moved to a block P for which all paths in the

graph though P also go through B. Figure 1 provides and example of this constraint.

2. All of the arguments to the function are available at any block P which is a candidate location for the function call. See Figure 2 for an example.

The first constraint is introduced so that we avoid network requests or print statements which never would have originally occurred along any given path of execution. We do not want to allow reordering which introduces new computations or would result in unpredictable performance. The second constraint ensures that we don't ever violate the most basic of correctness properties. A more detailed algorithm for finding safe locations for portable mostly pure functions in Clojure code is discussed in Section 6.1

```
;; we cannot move the call outside of the if.
;; There is a path on which the call will never occur
(if (pred? a b c ...)
   (do
      ;; some code that uses a, b, and/or c
      (foo a b c))
   (bar a b c))
```

Figure 1: The call to the mostly pure function `foo` can only be moved to the boxed nodes

```
(do
   (let [a (bar 10)]
      ;; some code that uses a
      (let [b (bar a)]
         ;; some code that uses a and/or b
         (foo a)))
```

Figure 2: The call to the mostly pure function `foo` can only be moved to the boxed nodes

Many Clojure functions fit this definition due to Clojure programming conventions and default immutable data structures. This concept is not rigorously defined as there is not a good definition for the semantics of such functions. Instead, in this paper, we rely on the judgment of the programmer to tell us when a mostly pure function is portable.

4 parlet

The first of the parallel macros is called `parlet`. `parlet` is a modified version of the Clojure `let` form. The `parlet` macro has exactly the same semantics as Clojure's `let`, but it evaluates all of its bindings in parallel. For example, suppose I had some long running function `foo`. I need to add the result of two calls to this function. In Figure 3, we use `parlet` to make two calls to `foo`, then add the results.

```
(parlet
  [a (foo value1)
   b (foo value2)]
   ... ; some other code here
  (+ a b))
```

Figure 3: Example of a parlet form

In this example, the expressions `(foo value1)` and `(foo value2)` are both evaluated as `ForkJoinTasks` in a `ForkJoinPool`[6].

The calls to `foo` are both `forked` immediately, then we attempt to evaluate the body of the `let`. This means that the code in the body of the `let` which does not depend on the computations of `a` and `b` can execute without delay.

Since the `ForkJoinPool` is designed for recursive workloads, it allows tasks which are currently executing to create new tasks, submit them to the pool, then wait for the task to complete. None of the pool's threads will block when this occurs. This means that nested `parlet` forms (Figure 4) will execute without blocking.

In Figure 4, calls to the function `foo` will be processed in the background until we reach the line using the variables that the results are bound to. This means that parlets can be nested with little concern (again as long as functions are pure).

```
(parlet
  [a (foo 100)]
  ;; some other code not using a
  (parlet
    [b (foo 200)]
    (+ a b)))
```

Figure 4: Nested parlet forms

Code like this may not arise when the code is human generated, but it may arise when the code is generated by another macro. We will see some examples of this later.

4.1 Dependencies

The `parlet` macro also supports simple dependency detection. Clojure `let` forms can use names defined previously in the let, and the bindings are evaluated from first to last.

```
(parlet
  [a 1
   b (+ 1 a)]
  a)
```

Figure 5: A parlet containing a dependency

Without the `parlet`, the let form in Figure 5 would evaluate to 2. If we plan on evaluating each binding in parallel, we can't allow the bindings to have dependencies. So, the `parlet` macro looks through the bindings in the macro to determine if any of them depend on any of the names defined previously in the macro. If there are any, the macro will halt the compiler and report an error to the user.

This transformation is only safe when `foo` (and more generally, and function call in the bindings) is a mostly pure function. Here, we punt the definition of mostly pure to the programmer. If the programmer chooses to use a `parlet` form, we assume that the functions are mostly pure. This simple dependency check, as well as the programmers promise that all function called are mostly pure Clojure code, allow us to ensure correct parallelism with this macro.

5 parexpr

The `parexpr` macro breaks down expressions and (aggressively) evaluates them in parallel. Suppose again that I had a long running function `foo` which I wanted to call twice, and add the results. The code in Figure 6 will make both of the long running calls to `foo` in parallel. The `parexpr` macro crawls the expression and expands it into multiple nested `parlet` forms, filling each `parlet` with as many evaluations as it can.

```
(parexp (+ (foo value1) (foo value2)))
```

Figure 6: Example using parexpr to evaluate long running functions

6 defparfun

The `defparfun` macro is the most interesting of the 3 macros. `defparfun` allows a programmer to parallelize calls to a recursive function. For an example see Figure 7. This defines a parallel Fibonacci function, which will only execute in parallel when the value of it's argument is greater than 30.

6.1 Imposed Constraints

The objective of the `defparfun` macro is to enable simple, predictable parallelism for recursive functions which are mostly pure. We also aim to support flexible functions, but we cannot promise correctness when the macro is used with flexible functions.

A mostly pure function call may be moved to a node P in the control flow graph for the function iff every path through P in the original function would execute the function call, and if the arguments to the function call are all available in P. We do not allow function calls to be introduced on a path which they would not have existed on before to preserve

6.2 Moving portable mostly pure functions

The macro first identifies any recursive call sites. It then introduces a name for the call, then pushes the evaluation of the function all as far "up" in the function as it can. We stop pushing a function call higher in a function

when we encounter an `if` form, or we encounter any existing `let` expression which introduces a value that the function call depends on. We can think of this operation as an operation on a single "basic block" in the function. The function call can be freely moved inside of the function as long as the function call does not introduce any new behavior to the function.

After this transformation, it is possible to replace the let forms which bind the function results to their values with `parlet` forms providing the same bindings. The introduction of the parlet form introduces parallelism, so each recursive call will execute in the `ForkJoin` pool, in parallel. Each of these subsequent recursive calls may create more tasks, which will also be executed in the pool. `ForkJoin` pools are designed to handle this type of computation, so the computation will proceed without blocking (as long as function is pure and performs no thread–blocking I/O).

7 Benchmarking

To run benchmarks, I used Google's Cloud Compute virtual machines. For each trial, a virtual machine was created. Each virtual machine had either 1, 2, 4, or 6 cores and 6 gigabytes of RAM. First, the serial version of the code was run, then, on the same machine, the parallel version of the code was run. After both trials finished running, the data was copied back to my local machine and the virtual machine was destroyed. For every pair of serial/parallel executions, the speedup was computed. These per–machine speedups are used to generate the plots shown.

To workaround the difficulties JVM benchmarking introduces, the Criterium[1] library was used for Clojure code and the Caliper[2] library was used with Java code. The JVM does not expose a mechanism to control the total number of threads it creates (including garbage collection threads). Because each benchmark was run on it's own virtual machine with a constrained number of cores, the number of threads the JVM could create was controlled.

7.1 Fibonacci

First we will look at the classical recursive Fibonacci example. Figure 7 shows the code used for the serial and the parallel benchmarks. In Figure 8

[1] `https://github.com/hugoduncan/criterium`
[2] `https://github.com/google/caliper`

```
(defn fib [n]
  (if (or (= 0 n) (= 1 n))
    1
    (+
     (fib (- n 1))
     (fib (- n 2)))))
```

(a) Serial recursive Fibonacci

```
(defparfun fibparfun [n] (< n 35)
  (if (or (= 0 n) (= 1 n))
    1
    (+
     (fibparfun (- n 1))
     (fibparfun (- n 2)))))
```

(b) Fibonacci with `defparfun`

Figure 7

the results from many trials of this code running 1, 2, 4, and 6 cores. Each benchmark computes `(fib 35)`. We see that we get about a 3x speedup with 6 cores. This speedup isn't quite what we would hope to see, but, as can be seen in Figure 9, the handwritten Java Fork/Join implementation gets about the same speedup with 6 cores on these virtual machines.

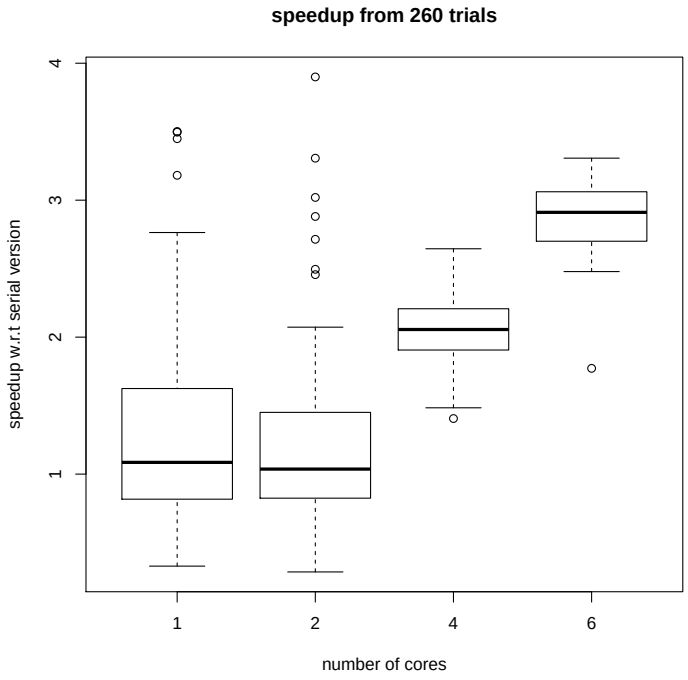

Figure 8: Fibonacci Clojure (defparfun) Performance

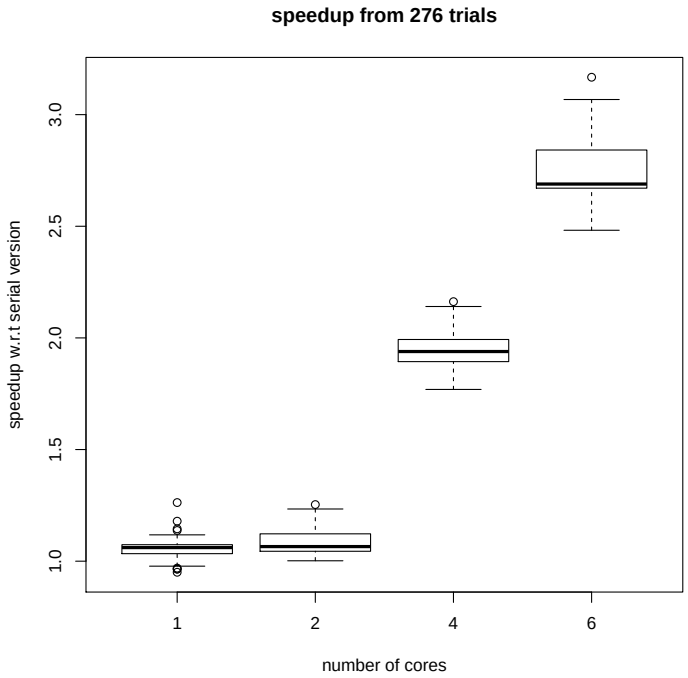

Figure 9: Fibonacci Java Performance

The large variance we see in the Clojure benchmarks is somewhat disturbing, especially since it does not show up in the Java results. Since the large variance does not appear in the Java benchmarks, it is probably not a result of variable performance in the Google Cloud Platform virtual machines. In Figure 10 I show the standard deviation of the mean runtime for the serial and parallel Clojure Fibonacci functions, along with their Java counterparts. Notice that the Java results do not suffer from the same extremely large variability problem. While I cannot completely explain the variability in the results, it seems to be caused by Clojure's function implementation, and the increased pressure that the Fork/Join wrapper tasks places on the JVM garbage collector. Every Clojure function is an `Object` created following the Clojure `IFn` interface[3]. When running on the `ForkJoinPool`, each function is further wrapped in a `RecursiveTask` object, causing additional allocations. We can avoid excessive task creation by controlling the granularity of parallelism, but, these tasks will always move functions into `RecursiveTask`s

[3] `http://clojure.org/reference/special_forms#fn`

and introduce some amount of additional overhead. The large number of allocations causes additional garbage collector work. The garbage collector behaves somewhat non—deterministically, so I believe this is the explanation for the large variation in runtime for the serial and recursive Fibonacci code. The Fibonacci example highlights this problem because the function call overhead greatly exceeds the amount of work each call is doing. We will see an example for which this is not the case in Section 7.2.

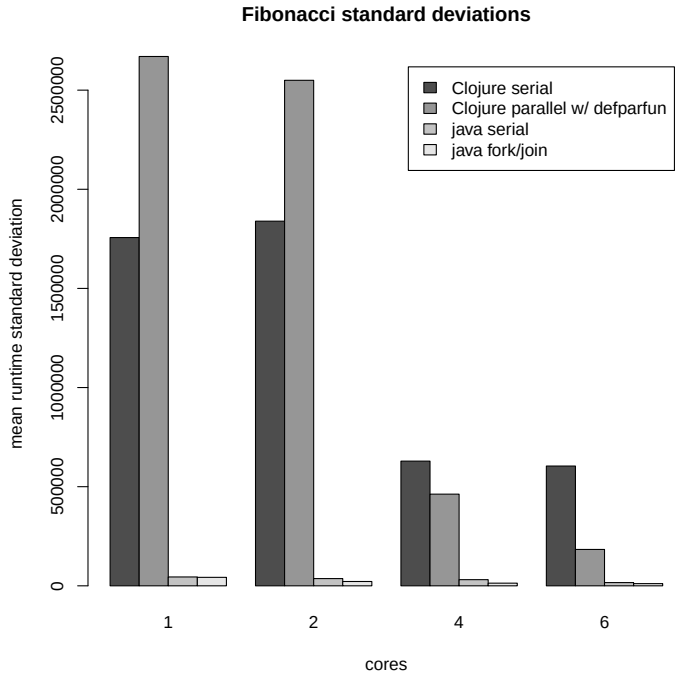

Figure 10

7.2 ID3

I also implemented a simple ID3[4] classifier in Clojure. The code is bit longer, so it is not included in this paper, but it can be found on this project's Github page[5].

[4] `https://en.wikipedia.org/wiki/ID3_algorithm`
[5] `https://github.com/dpzmick/auto_parallel`

For each benchmark, a random 1,000 element dataset was created. Each element of the dataset was given 100 random features. The `ID3` algorithm implementation ran until it was out of attributes to pivot on.

The `ID3` code does much more work in each function call, so the additional overhead created by Clojure's function implementation does not seem to impact the results as much as it does in the Fibonacci benchmark.

Figure 11 shows that we get the 3x speedup we expect on these virtual machines with the `ID3` algorithm.

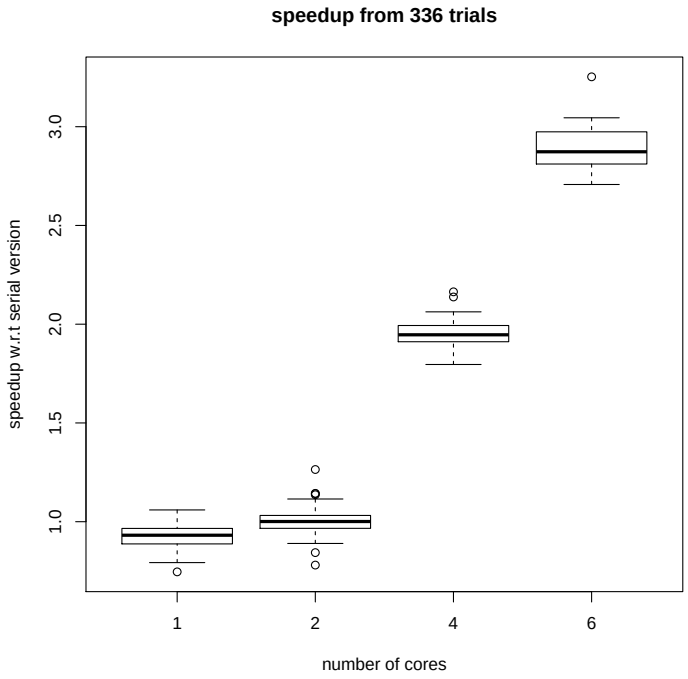

Figure 11: id3 Clojure (defparfun) Performance

8 Conclusions and Future Work

9 References

[1] Cilk 5.4.6 Reference Manual, 2001.

[2] Diego Andrade, Basilio B Fraguela, James Brodman, and David Padua. Programming in Multicore Systems. *Library*.

[3] Hans-J Boehm. Transactional memory should be an implementation technique, not a programming interface. *Practice*, pages 1–6, 2009.

[4] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[5] S.P. Jones. Beautiful concurrency. *Beautiful Code*, (1):385–406, 2007.

[6] Doug Lea. A Java Fork/Join framework. *Java Grande*, pages 36–43, 2000.

[7] Semih Okur and Danny Dig. How Do Developers Use Parallel Libraries? *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 54:1–54:11, 2012.

[8] Jose Rodr, Antonio J Dorta, and Casiano Rodr. Exploiting task and data parallelism.