Macros for straightforward parallelism in Clojure

Zmick, David
zmick2@illinois.edu

1 Introduction

Clojure is a lisp–like language running on the Java Virtual Machine[7]. The language's default immutable data structures and sophisticated Software Transactional Memory (STM) system make it well suited for parallel programming[9]. 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 construct for concurrent programming[8], but these constructs are often too low level to be of much use to developers whose central concerns are not parallelism[6].

As a result, there are a variety of libraries designed to allow developers to take advantage of the parallelism potential in Clojure. Clojure built ins such as `pmap`[1] and `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[11], 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[1, 12]).

Using Clojure's macro system, I have implemented a set of macros which allow developers to take advantage of Clojure's task parallelism potential. I have shown that it is possible to attain reasonable degrees of parallelism with minimal modification to existing serial 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

Clojure has access to all of the JVM, so it has access to the Java `ForkJoinPool` library[10]. The `ForkJoinPool` library allows Java programmers to create lightweight, recursive tasks which are executed by a thread pool. Each thread maintains a queue of tasks to work on. When a thread runs out of tasks, it steals tasks from other threads. Work stealing has been used a scheduling mechanism is a variety of modern threading libraries and has been very successful [10, 4, 5] Users of a `ForkJoinPool` create a subclass of `RecursiveTask` to compute the value of some function. Each of these tasks may create more `RecursiveTask`s, submit them to the `ForkJoinPool`, then wait for their subtasks to complete, without blocking the pool. Clojure programmers can use the `ForkJoinPool` libraries from Clojure, but the interface isn't exactly programmer friendly (or idiomatic).

Clojure also has built in support for task parallelism via `future`[1]. Each call to `future` spawns a thread to result of some function. When a user is ready to access the computed value, they `deref` the future. `deref` will block until the value is computed. For a user, `future` seem like a natural way to parallelize recursive functions. Unfortunately, `future` does not work well when a large number of tasks are created. Because thread creation overhead is high, users must be careful not to create an excessive number of threads or create threads which do too little work.

Claypoole's implementation of `future` offers many improvements to the built in `future`. The Claypoole implementation executes tasks in a fixed sized thread pool, but, the executing thread will block if the task creates a new task, then immediately `deref`s it. This means that a programmer must be very careful not to create tasks in a Claypoole thread pool which depend on other tasks that will also be submitted to the pool, otherwise their code may deadlock.

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

The work in this paper is conceptually similar, but the interface I have used differs dramatically from the `future`s interface. My code also creates many tasks, but these tasks are created on a Java `ForkJoinPool`, so the tasks are much more lightweight, and they can be composed recursively.

In Common Lisp, the `lparallel`[2] library and macros are available. The macros I have implemented are very similar to the macros `lparallel` provides, but, Common Lisp programming conventions are not as ideal for these kinds of macros, and Common Lisp programmers can't use the battle tested `ForkJoinPool` implementation.

2.1 Other languages

In other languages, frameworks like Cilk++[3], OpenMP[4], Threading Building Blocks[5] have been implemented. Some of these require compiler support and advanced dependence analysis to guarantee correctness. The macros I've implemented are an attempt to bring some of the niceties of these libraries and compiler extensions to Clojure, without the need to compiler support or complicated dependence analysis.

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 thread—safe function with no side effects directly impacting the values of 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, a programmer writing a web scraper may not care what order (`download file1`) and (`download file2`) execute in, but the order 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, or interleave, calls to mostly pure functions, 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 5.1

```
;; cannot move the call outside
(if (pred? a b c ...)
    (do
        ;; 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

```
(let [a (bar 10)]
    ;; code that uses a
    (let [b (bar a)]
        ;; 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. In this paper, we rely on the judgment of the programmer to tell us when a function can be considered mostly pure.

[2] `https://lparallel.org/overview/`
[3] `https://www.cilkplus.org/`
[4] `http://openmp.org/wp/`
[5] `https://www.threadingbuildingblocks.org/`

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

(a) Example of a parlet form

```
(let
  [a (fork (new-task (fn [] (foo 1))))
   b (fork (new-task (fn [] (foo 2))))]
  ;; some other code here
  (+ (join a) (join b)))
```

(b) An expanded parlet form

Figure 3

4 parlet

The first of the parallel macros is called `parlet`. The `parlet` macro has exactly the same behavior 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.

In this example, the expressions `(foo value1)` and `(foo value2)` are both evaluated as `RecursiveTasks` in a `ForkJoinPool`. The calls to `foo` are both forked immediately, then we attempt to evaluate the body of the `parlet`. Each use of `a` and `b` is replaced with a call to `join`, to get the computed value. 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. Additionally, since the `ForkJoinPool` is designed for recursive workloads, tasks which are currently executing can create new tasks, submit them to the pool, then wait for the task to complete, without blocking tasks created by other `parlet` calls. This means that a programmer does not have to worry if functions called in the bindings of a `parlet` form also use `parlet`.

4.1 Dependencies

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

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

Figure 4: A parlet containing a dependency

Without the `parlet`, the let form in Figure 4 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 will halt the compiler and report an error to the user if any dependencies are found in the `parlet` form.

4.2 Correctness

This transformation is only safe when `foo` (and more generally, any function call in the bindings) is a mostly pure function. If the programmer chooses to use a `parlet` form, we assume that the functions called in the bindings 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 defparfun

`defparfun` allows a programmer to parallelize calls to a recursive function. The `defparfun` macro supports a granularity argument, allowing the programmer to specify when they would like to stop creating additional parallel tasks. The macro emits the expression provided for granularity inside of an `if` statement at the top of the function, so the programmer can use any arbitrary condition, including a condition dependent on the function's arguments, to decide when to stop spawning new tasks. Figure 7b defines a parallel Fibonacci function, which will only execute in parallel when the value of it's argument is greater than 35.

5.1 Implementation

In Clojure, any form which introduces control flow will eventually expand to an `if` for. Any form which introduces new bindings (including Clojure's destructuring mechanisms) will eventually expand to a `let` form. Because the constraints on movement of mostly pure functions only depend on control flow and variable bindings, we only need to make decisions about mostly pure function calls near `if` forms and `let` forms, in the full expanded versions of Clojure functions.

When provided a function to manipulate, this function first expands all of the other macros in the function body, to get the nice property described above. Then, the macro recursively crawls the function body,

```
(defn f [a]
  (if (pred? ....)
    (do ...)
    (do ... (f (+ 1 a))))))
```
(a)

```
(defn f [a]
  (if (pred? ....)
    (do ...)
    (let [e1 (f (+ 1 a))]
      (do ... e1))))
```
(b)

```
(defn f [...]
  (let [a ...]
    (let [b ...]
      (f (+ 1 a)))))
```
(c)

```
(defn f [...]
  (let [a ...]
    (let [e1 (f (+ 1 a))]
      (let [b ...] e1)))))
```
(d)

Figure 5: Moving a recursive function call

looking for recursive call sites. If a call site is found, the recursive call (in the function body) is replaced by a newly introduced variable, and we hold onto the original recursive call. When all of the subexpressions for a given expression have been evaluated and transformed, we check if the expression is a `if` form or a `let` form. If we are sitting on a `if` form, all of the bindings introduced by the `true` branch and by the `false` branch are emitted in a new `let` expression (Figures 5a and 5b). This guarantees that condition 1 holds (Section 3). If we are sitting on a `let` form, any of the bindings which depend on the expressions introduced by the `let` form are emitted. The remaining bindings continue to trickle upwards (see Figures 5c and 5d). This guarantees that condition 2 holds.

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. To complete the Fibonacci example, the `defparfun` expansion is shown in Figure 6.

Because the transformation does not violate any of the properties defined for mostly pure functions, this transformation is safe when the function being declared as a `parfun` is mostly pure.

```
(defn fib [n]
 ;; granularity check
 (if (< n 35)
  (if (or (= 0 n) (= 1 n))
   1
   (+ (fib (- n 1))
      (fib (- n 2))))

  ;; recursive case
  (if (or (= 0 n) (= 1 n))
   1
   (parlet [expr17300 (fib (- n 1))
            expr17301 (fib (- n 2))]
    (+ expr17300 expr17301)))))
```

Figure 6: Transformed Fibonacci function

6 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[6] library was used for Clojure code and the Caliper[7] library was used for Java code. 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 (including threads used for garbage collection)

6.1 Fibonacci

First we will look at the classical recursive Fibonacci example. Figure 7a shows the serial benchmark code;

[6] https://github.com/hugoduncan/criterium
[7] https://github.com/google/caliper

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

(a) Serial Fibonacci function

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

(b) Fibonacci function with `defparfun` added

Figure 7

7b shows the code parallel benchmark code. In Figure 8 the results from many trials of this code running 1, 2, 4, and 6 cores. Each benchmark computes `(fib 39)`. 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 `ForkJoinPool` implementation gets about the same speedup with 6 cores on these virtual machines. Previous `ForkJoinPool` benchmarks have shown much better speedups for similar code[10], so I suspect that the virtual machine configuration is somewhat responsible for the discrepancy in results.

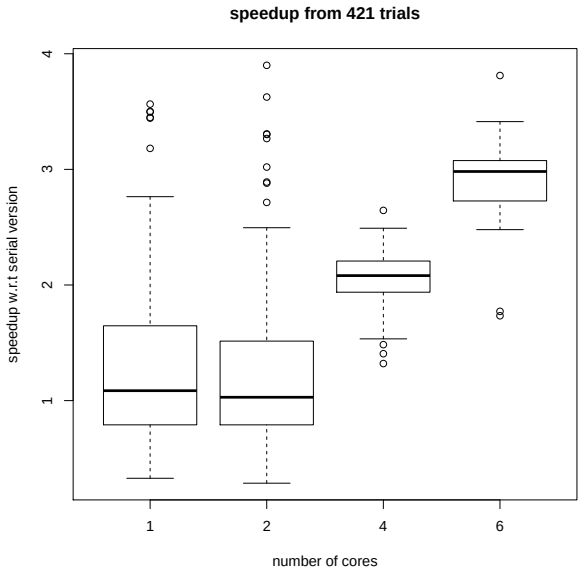

Figure 8: Fibonacci Clojure (defparfun) Performance

The large variance we see in the Clojure benchmarks is somewhat disturbing, especially since it does not show up in the Java results. Since it does not appear in the Java benchmarks, it is 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

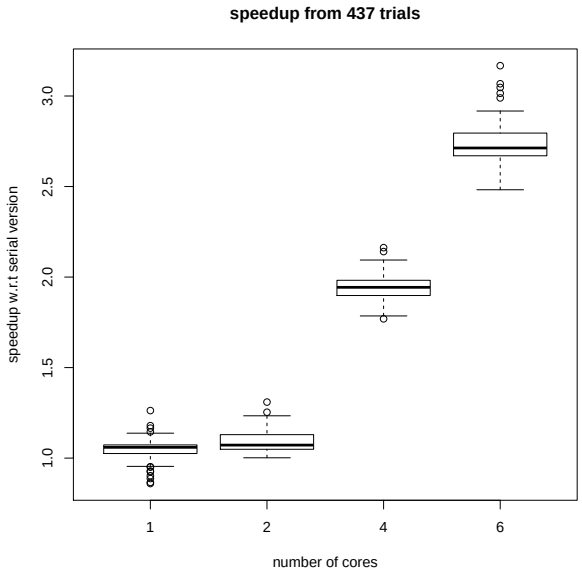

Figure 9: Fibonacci Java Performance

counterparts. Notice that the Java results do not have nearly as high deviations from mean runtime. While I cannot completely explain the variability, it seems to be caused by the increased pressure Clojure's function implementation and the `ForkJoinPool` wrapper tasks places on the JVM garbage collector. Every Clojure function is an `Object` created following the Clojure `IFn` interface[8]. When running on the `ForkJoinPool`, each function is further wrapped in a `RecursiveTask` object, causing additional allocations. This effectively moves the stack for the recursive function to the heap (eliminating Clojure's stack depth limit[9]). 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. We can avoid excessive task creation by controlling the granularity

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

[9] `http://clojure.org/about/functional_programming#_recursive_looping`

of parallelism, to an extent. 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 6.2.

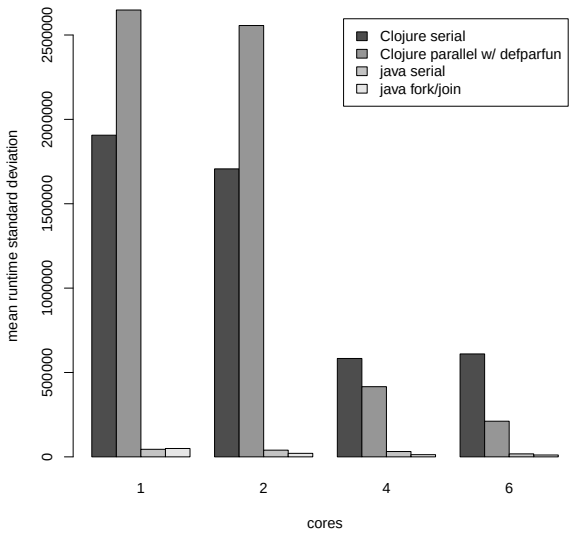

Figure 10: Fibonacci standard deviations

6.2 ID3

I also implemented a simple ID3[10] 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[11]. 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 overhead introduced by the `ForkJoinPool` wrapper does not 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.

7 Conclusions and Future Work

The Clojure macros I've implemented perform transformation which can speedup Clojure code to a degree

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

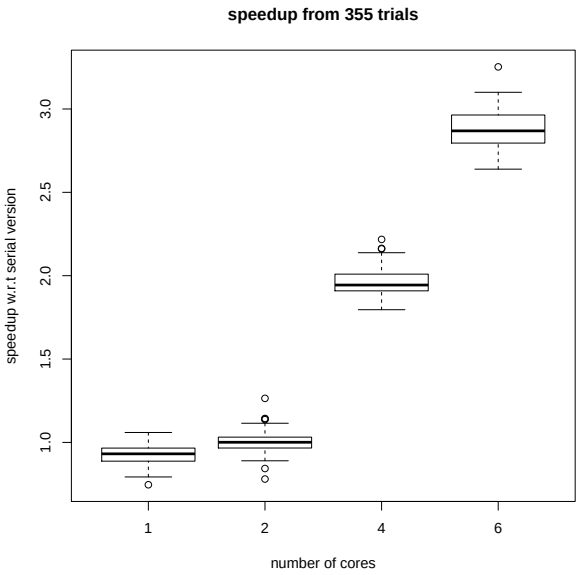

Figure 11: id3 Clojure (defparfun) Performance

which matches the speedups attained using hand-written Java code, running on the same hardware. Parallelism is difficult, and automatic parallelism is possible[3], but these techniques are complicated and often do not get the desired results and the research community has begun to feel the need for explicit parallelism in programs.[2] Languages like Clojure are well suited for this parallelism and techniques like mine can easily be implemented. In a language with a strong STM system and immutable structures, such transformations are easy to reason about, making it much simpler for programmers to implement explicitly parallel programs.

Macros of this style do not inhibit the programmers ability to use the other mechanisms implemented in the language, although interoperability with them could be improved. For example, one of the tests which was not discussed in this paper involved using the STM system from within a function declared with `defparfun`. Benchmarks on this code behaved correctly and performance improved as expected. However, if a programmer attempted to use a `pmap` or `future` inside of a `defparfun` or `parlet`, the two systems would create separate thread pools and the number of created threads would be large, possibly causing poor performance. There are also a variety of other useful macros in lparallel[12] that may be useful to implement in Clojure and would complement the macros I've implemented in this project.

[12] `https://lparallel.org/`

8 References

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

[2] Arvind, David August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J Yi. Programming multicores: Do applications programmers need to write explicitly parallel programs? In *IEEE Micro*, volume 30, pages 19–32, 2010.

[3] Utpal Banerjee, Rudolf Eigenmann, N. Nicolau, David A. Padua, and A. Alexandru. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[4] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 1–29, 1994.

[5] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk : An Efficient Multithreaded Runtime System. *Computer*, 30(8):207–216, 1995.

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

[7] Rich Hickey. The Clojure Programming Language. In Johan Brichau, editor, *Proceedings of the 2008 symposium on Dynamic languages*, page 1. University of Bern, ACM New York, NY, USA, 2008.

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

[9] Johann M. Kraus and Hans a. Kestler. Multicore parallelization in Clojure: a case study. *ELW '09: Proceedings of the 6th European Lisp Workshop*, (August):8—-17, 2009.

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

[11] 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.

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