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.[5] The language's default immutable data structures and sophisticated Software Transactional Memory (STM) system make it well suited for parallel programming.[7] 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[6], but these constructs are often too low level to be of much use to developers whose central concerns are not parallelism[4].

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 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[9], 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, 10]). 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

Clojure has access to all of the JVM, so it has access to the Java `ForkJoinPool` library[8]. The `ForkJoinPool` library allows Java programmers to create lightweight, recursive tasks, then execute them on an efficient, work stealing, `ExecutionEngine`. 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. The execution engine for the `ForkJoinPool` is a thread pool. Each thread maintains it's own queue of tasks to work on. Whenever a thread runs out of tasks to work on, 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 [8, 2, 3] Clojure programmers can use the `ForkJoinPool` from Clojure, but the interface isn't exactly programmer friendly.

Clojure also has some built in support for task parallelism via `future`[1]. Each call to future creates a thread to compute the result of a function call in

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

the background. When a user is ready to access a value, they can `deref` the future to get the computed value (`deref` will block until the value is computed). For a user, `future`s initially seem like a fairly natural way to parallelize recursive functions. Unfortunately, Futures only work well when a small number of tasks are created or when the tasks do a lot of blocking I/O. Thread creation overhead is high, and people users must be careful not to create an excessive number of threads or create threads to do too little work. Claypoole offers many improvements to the builtin Clojure `future`s, but it still creates a thread per task. 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 tasks for every thread, but these tasks are created on a Java `ForkJoinPool`, so the tasks are much more lightweight.

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.

2.1 Other languages

In other languages, frameworks like Cilk++[3], OpenMP[4], Threading Building Blocks[5] all exist. Some of these require compiler support and advanced dependence analysis to guarantee correctness. These libraries all have great performance, but they can't be leveraged from Clojure. 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, 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, or interleave, 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 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

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 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 behavior as

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

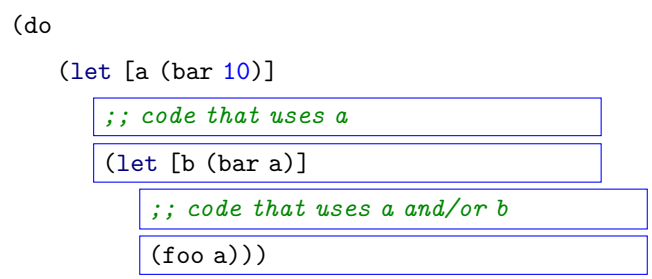

Figure 2: The call to the mostly pure function `foo` can only be moved to the boxed nodes

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`[8]. The calls to `foo` are both forked immediately, then we attempt to evaluate the body of the `parlet`. 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 execution. This means that nested `parlet` forms (Figure 4) will not block each other.

```
(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.

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 conditions dependent on the functions arguments, to decide when to stop spawning new tasks.

For an example see Figure 6. This 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

To do implement this macro, I first expand all of the `let` expressions in the function so that every `let`

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

Figure 6: Fibonacci function with `defparfun` added

only has a single binding. Then, the syntax tree is crawled, finding every recursive function call. Every recursive call is replaced by a newly introduced variable. At the termination of each recursive call, the bindings introduced in the subexpression are returned. Once all of the subexpressions are evaluated, the algorithms decides if it must emit a `let` form for any of the bindings the subexpressions introduced. If the expression in question is an `if` statement, a let statement is introduced for the `true` branch and on the `false` branch, each containing all of the bindings introduced by the subexpressions for each branch. If the statement is a let statement, only the bindings which depend on the binding introduced by the let statement are emitted (following the let of course). This operation effectively pushed the evaluation of the function all as far "up" in the function as it can, following the constraints described in Section 3.

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.

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.

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

```
(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 7: Transformed function

introduces, the Criterium[6] library was used for Clojure code and the Caliper[7] 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.

6.1 Fibonacci

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

Figure 8: Serial Fibonacci function

First we will look at the classical recursive Fibonacci example. Figure 8 shows the serial benchmark code; 6 shows the code parallel benchmark code. In Figure 9 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 10, 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[8], so I suspect that the virtual machine configuration is somewhat responsible for the discrepancy in results.

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

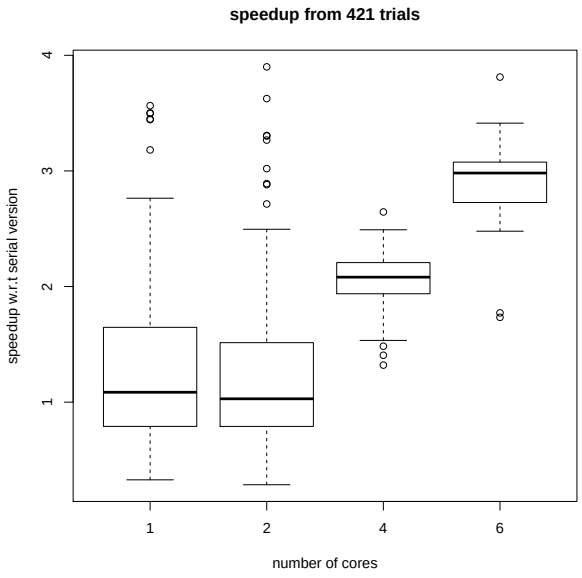

Figure 9: Fibonacci Clojure (defparfun) Performance

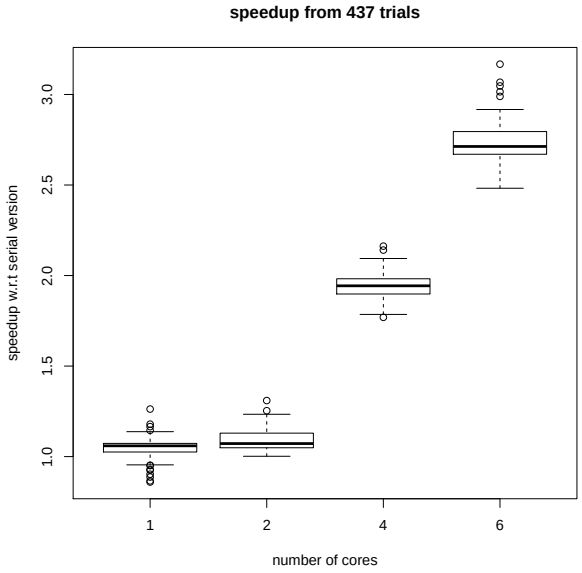

Figure 10: Fibonacci Java Performance

The large variance we see in the Clojure benchmarks is somewhat disturbing, especially since it does not show up in the Java results. Since 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 11 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 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. 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. We can avoid excessive task creation by controlling the granularity 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.

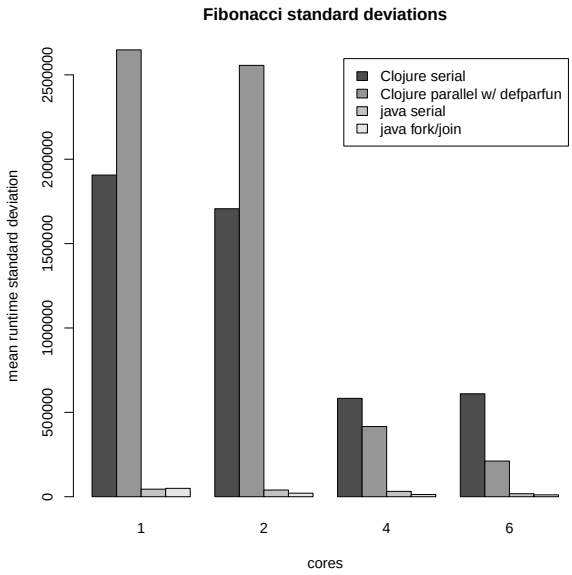

Figure 11:

6.2 ID3

I also implemented a simple ID3[9] 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[10].

[8] `http://clojure.org/reference/special_forms#fn`
[9] `https://en.wikipedia.org/wiki/ID3_algorithm`
[10] `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 12 shows that we get the 3x speedup we expect on these virtual machines with the `ID3` algorithm.

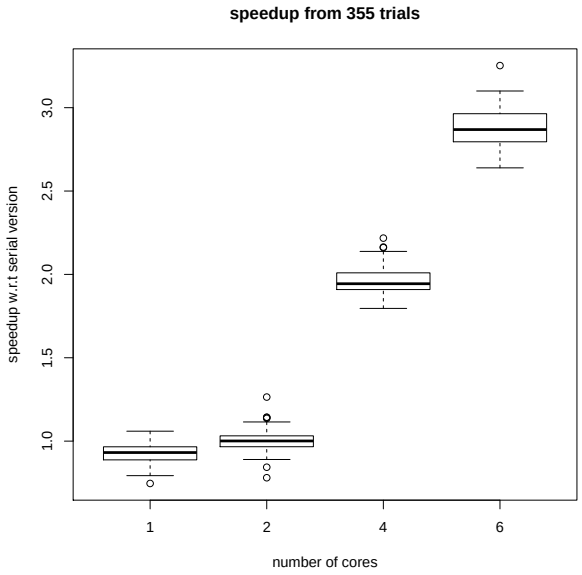

Figure 12: id3 Clojure (defparfun) Performance

7 Conclusions and Future Work

Clojure's conventions make transformations like this possible, and simple.

8 References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

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