Macros for straightforward parallelism in Clojure

Zmick, David
zmick2@illinois.edu

April 7, 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[3], but these constructs are often too low level to be of much use to developers whose central concerns are not parallelism[2].

As a result, there are a variety of libraries designed to allow developers to take advantage of the parallelism potential in Clojure.[1][2][3][4][5] Many of these library functions and builtins are data-parallel; they are designed to apply some sort of operation to a set of data. Developers have a good relationship with data parallel problems[5], 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, 6]).

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. These transformation can be applied to idiomatic Clojure code without the need for sophisticated dependency analysis often required in other parallelism systems, and do not inhibit the programmer's ability to leverage other Clojure concurrency constructs (such as the STM system).

[1] `https://github.com/clojure/core.async`
[2] `https://github.com/TheClimateCorporation/claypoole`
[3] `https://clojuredocs.org/clojure.core/pmap`
[4] `http://clojure.org/reference/reducers`
[5] `https://github.com/aphyr/tesser`

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.

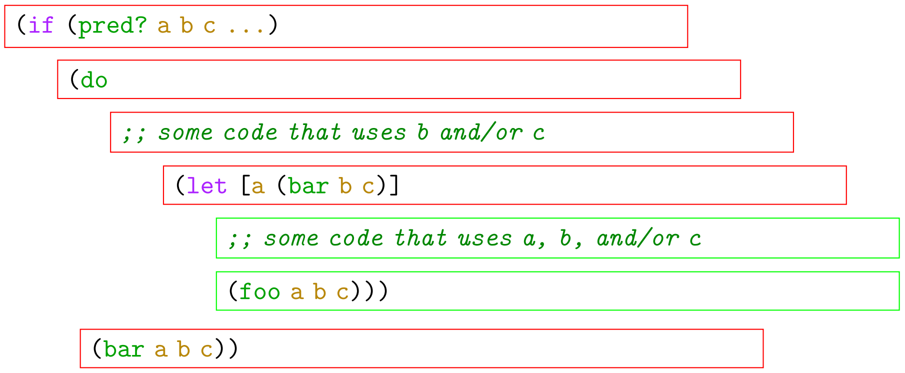
2 Mostly Pure Functions

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 without impacting the values of user variables, although a change in order may impact I/O behavior and output order of a program. When a programmer feels that it may be acceptable to change the order of 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.

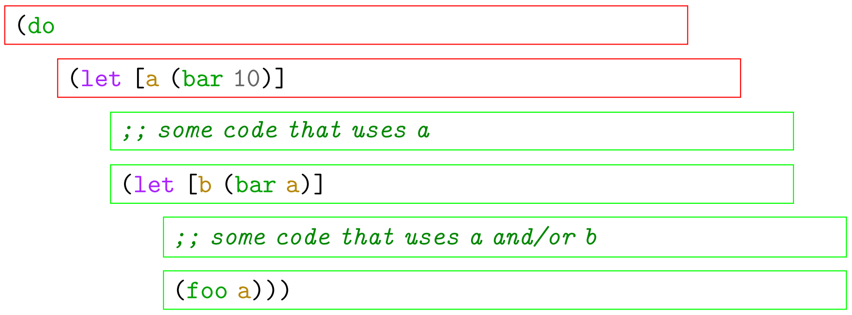
2. All of the arguments to the function are available at any block P which is a candidate location for the function call.

The first constraint is introduced so that we avoid making network requests or print statements which never would have originally occurred. We are only allowing these functions to be reordered within blocks where they would previously have been executed. Figures 1 and 2 demonstrate safe and unsafe positions a mostly pure function call can be moved to.



```
(if (pred? a b c ...)
    (do
        ;; some code that uses b and/or c
            (let [a (bar b c)]
                ;; 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 nodes which are green



```
(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 nodes which are green

Many functions fit this definition. For example, it usually does not matter when I call `println` as long as all of the of the arguments are available, and I don't call it in some totally unexpected location. 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 function should be considered mostly pure and when it should not be. Luckily, in Clojure, many functions are mostly pure, by convention.

3 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`[4]. 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.

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.

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

Figure 4: Nested parlet forms

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).

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.

3.1 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.

4 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

5 defparfun

The `defparfun` macro is the most interesting of the 3 macros. `defparfun` allows a programmer to parallelize calls to a recursive function. This is best explained with an example:

```
(defparfun fib [n] (> n 30)
  (if (or (= n 1) (= n 2))
      1
      (+
        (fib (- n 1))
        (fib (- n 2)))))
```

Figure 7: Example using defparfun to define a fibonacci function

This defines a parallel Fibonacci function, which will only execute in parallel when the value of it's argument is greater than 30.

5.1 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

5.1.1 how it works or something

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).

6 Benchmarking

To run benchmarks, we used Google's Cloud Compute product, because it allowed easy creation of machines with various configurations. Each benchmark was done on a virtual machine created to run the given benchmark. After the benchmark completed, the machines were destroyed. This allowed us to control the number of cores the JVM would use on every machine, since there is no good way to specify the number of cores the JVM will use for every thread it creates (including its garbage collection threads). A variety of small problems were implemented with serial recursive code, then invocations of the macros were added to the serial code. For any given test, the difference in the serial code and the parallel code is usually about 10 characters. I will present the results from a subset of the tests here. Each test was tested with a variety of different configurations and parallelism granularities where chosen to try and effectively use the resources available. For the tests with a fixed granularity (eg $n < 30$) the granularity was held constant for all tests. Some tests have dynamic granularity computed with respect to the number of cores available and the size of the input. For these tests, the granularity was not fixed.

6.1 Fibonacci

First we will look at the classical recursive fibonacci example. This benchmark was implemented in both Clojure and Java, to compare the speedups gotten with the macros vs the speedup gotten with handwritten Java Fork/Join code. Figure 8 displays the average speedup over many tests for the Java Fork/Join code. As we can see, we only get about a 3x speedup with 6 cores, in the simple Java implementation.

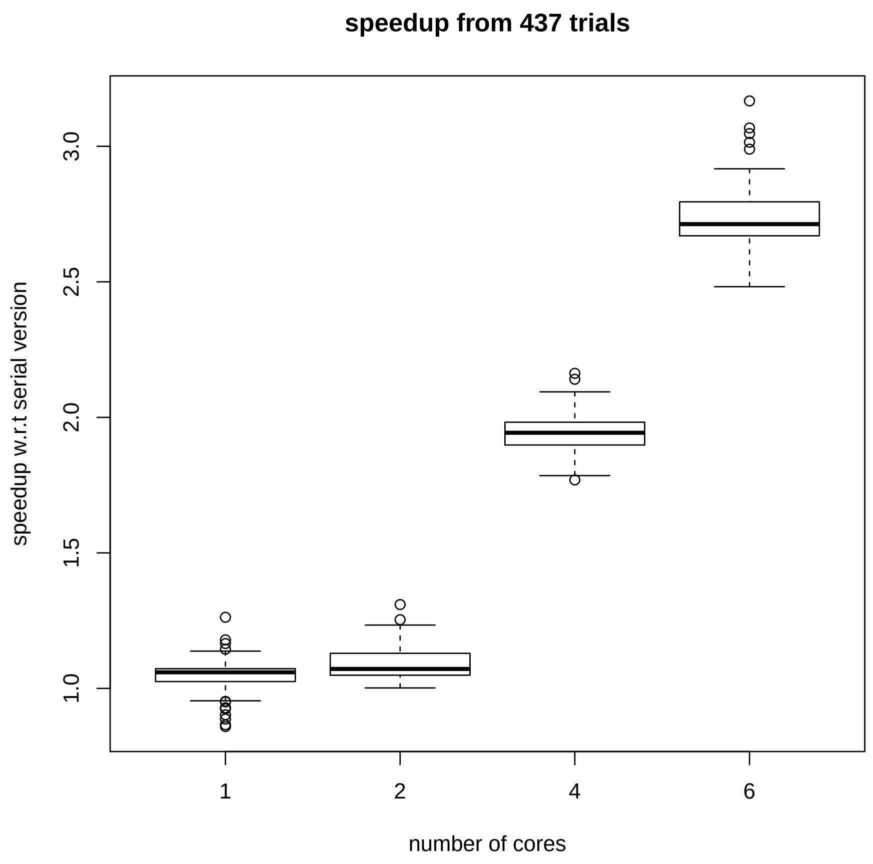


Figure 8: Fibonacci Java Performance

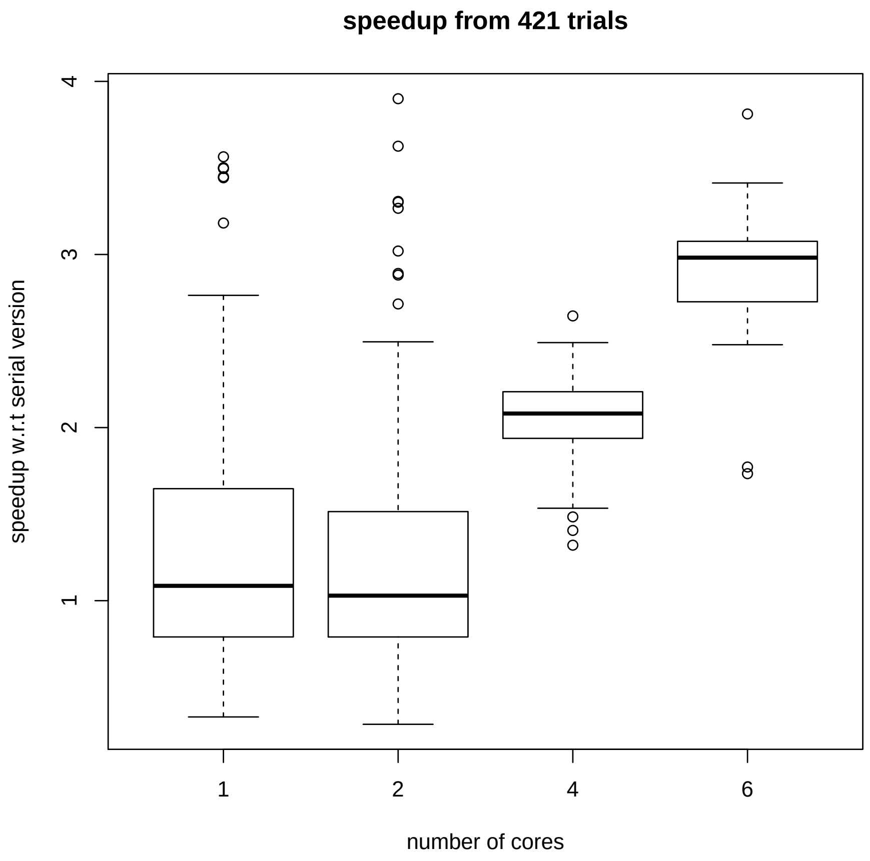


Figure 9: Fibonacci Clojure (defparfun) Performance

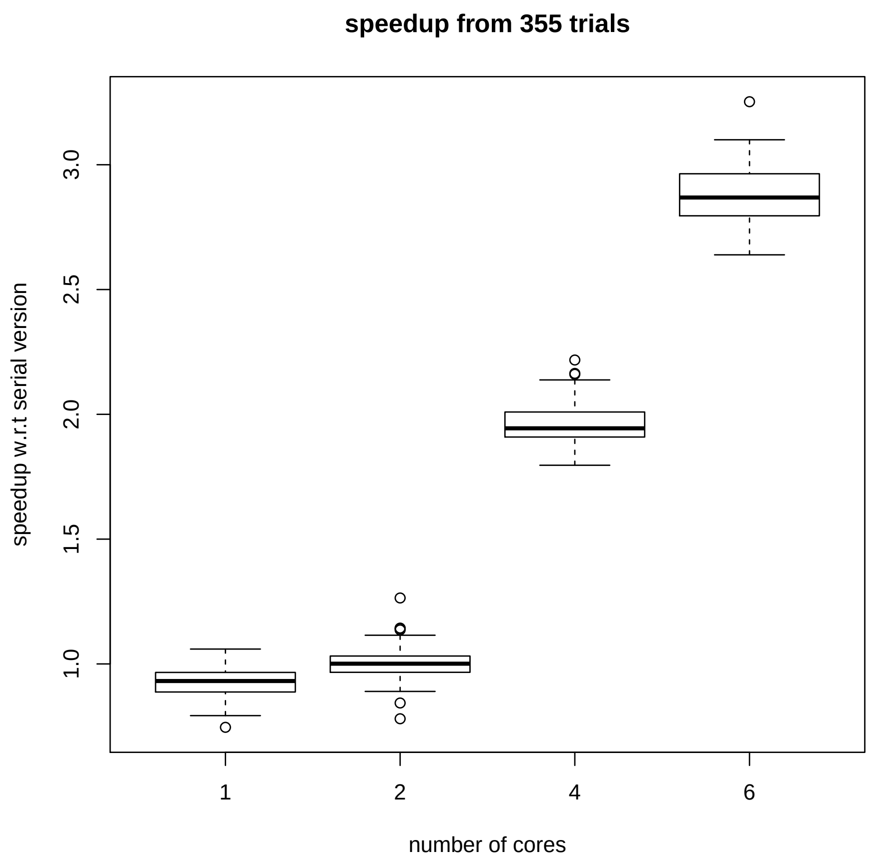


Figure 10: id3 Clojure (defparfun) Performance

7 Conclusions

References

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

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

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

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

[5] 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.

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