1. INTRODUCTION

Clojure is a lisp–like language running on the Java Virtual Machine. The language's default immutable data structures and it's sophisticated STM system make it well suited for parallel and concurrent programming. Additionally, 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.

Taking advantage of Clojure's parallel potential is not entirely straightforward, however. Interfacing with Java libraries directly is simple, but the interface code often feels slightly out of place when surrounded by other Clojure code. The STM system is easy to use, but the STM constructs are often too low level to be immediately useful.

As a result, there are a variety of libraries designed to allow developers to take advantage of the parallelism and concurrency potential in the language. Many of these library functions and builtins are data-parallel; they are designed to apply some sort of operation to a set of data. This is often the kind of parallelism that developers desire, or are accustomed to identifying, but Clojure's nature as a functional language with immutable structures also makes it possible to exploit parallelism in the control flow of the code.

Using Clojure's macro system, we have implemented a few macros which allow developers to take advantage of Clojure's parallelism potential when their parallelism is more directly expressed in the control flow of their application. We have shown that it is possible to attain noticeable degrees of parallelism with minimal code changes (with respect to serial code) through these macros, without the need for sophisticated dependency analysis (find OpenMP paper) often required in other parallelism systems.

This paper is divided into n sections. The first discusses the macros we've implemented, along with the appropriate use cases for each. In the second section, we present benchmarks demonstrating the cases in which the parallelism macros can (and cannot) improve performance of existing code.

2. MACROS

2.1 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 it's 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 1, 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 1: Example of a parlet form

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

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.

This transformation is obviously only safe when `foo` does not have side effects which may impact the other code in the function, or the values of terms used in subsequent let terms. In Clojure, it is very common (although not guaranteed by the compiler) for function calls to be pure.

Nested parlet macros will also behave quite well.

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

Figure 2: Nested parlet forms

In Figure 2, 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.

The parlet macro also supports 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 3: A parlet containing a dependency

Without the parlet, the let form in Figure 3 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 simple dependency check, as well as the commonplace purity in Clojure code, allow us to ensure correct parallelism with this simple macro. Because the Clojure compiler does not enforce purity, we still need to trust the programmer to validate the purity assumption before inserting the macro.

2.2 parexpr

Parexpr breaks down expressions and (aggressively) evaluates them in parallel. parexpr uses parlet to parallelize forms. Suppose again that I had a long running function foo, I wanted to call it twice, and add the results. The code in Figure 4 will make both of the long running calls to foo in parallel.

The macro crawls the expression and expands every subexpression into a parlet. This macro has limited applicability,

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

Figure 4: Example using parexpr to evaluate long running functions

but, we can also convince ourselves of its correctness easily.

2.3 defparfun

The defparun 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)))))
```

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

The macro first identifies any recursive call sites, then moves the function call as far "up" in the function as it can, without breaking any dependencies. Again, this dependency check is relatively simple. 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).

3. BENCHMARKING

3.1 Setup

4. RESULTS

5. CONCLUSIONS