I've been thinking a lot about "parallel blocks" recently and I am beginning to think they can be made to work very simply. The main thing that is needed is a type qualifier `const` that means "read-only". This would be a type prefix with very low precedence, just like `immutable` and `shared` in D. The type `const T` would refer to an instance of `T` that cannot be modified. This is a deep property, so, given some record types defined like:

```
type R1 = {...};
type R2 = { f1: @R1 };
```

If I had a variable `r2` of type `const @R2`, then the expression `r2.f1` has type `const @R1`.

Given this tool, the basic idea is that a parallel block (written `{ par || }`) can access its surrounding environment, but all of the types of the variables become `const`. This means that a block can read the heap via its environment but it cannot make any changes to it. This also means that any number of parallel blocks can execute in parallel without affecting each other.

However, this does not mean that parallel blocks cannot create and mutate data at all! They are free to allocate new structures in the task-local heap and mutate those, for example. They can even return those newly created structures to their caller. If regions come in, they could use their own stack frame as well. Finally, as described below, we can use unique pointers to give parallel blocks access to some data which they are free to modify if desired.

Also: here I am assuming fixed-length arrays, not the "always unique" vectors we have today, but everything still applies.

The critical invariant

In all the examples that follow, the critical invariant which is maintained is that only parallel blocks are executed in parallel: the "master thread" is effectively suspended while the parallel execution occurs. This ensures that there are no races between the parallel child threads and the common parent.

Parallel map etc using parallel blocks

We can define a parallel map function over arrays (this is clearly pseudocode):

```
fn par_map<S,T>(v: [S], b: par block(const S) -> T) -> [T] unsafe {
    let result = create uninitialized vector with length vec::len(v);
    spawn parallel tasks for each i = 0..vec::len(v) {
        result[i] = b(v[i]);
    }
    ret result;
}
```

This would be used like:

```
fn do_something(v: [T]) {
    par_map(v) { par |e: const T|
        let ctx = @{...};
        do_some_complex_computation(ctx, e)
    };
}
```

Note that the parallel block is allowed to create a (mutable!) variable `ctx` that is (presumably) used during the complex computation. But all data that the parent could reach, which was inherited either through upvars or through the parameter, is `const` and hence (temporarily) immutable. The execution of the parallel blocks themselves would be done using work-stealing or similar techniques.

It's not hard to imagine a number of other similar primitives that implement more complex patterns than `par_map`. The work by Michael McCool on Structured Parallel Programming would be a good starting point. In fact, probably the next thing to do is to look at those patterns and see how well they can be fit into this framework (not sure if there is a more recent publication).

In-place map using parallel blocks

But what if wanted to map over an array in place? To keep things simple, let's imagine that we want to overwrite the input array. That can be done in basically the same way with a function whose signature is:

```
fn par_map_in_place<S:send>(-v: ~[S], b: par block(&x: S)) -> ~[S] {
    spawn parallel tasks for each i = 0..vec::len(v) {
        b(v[i]);
    }
    ret v;
}
```

The difference from `par_map()` above is that this version does not create a new array but rather modifies the existing one in place. There are two subtle points:

- The type variable `S` is declared as sendable. This means that it does not access shared state (i.e., no contained pointers except for unique pointers). This ensures that it can be safely modified in place.
- This function both takes and returns the same vector. This is kind of a hack. What would be needed is a way to say a borrowed unique vector that the block can't access. This doesn't seem too hard but it's not something that's in the type system today (well, maybe using a mode like `&&` in combination with a type `~[S]` would achieve this, I'm not sure). Anyway this is a minor point.

Merging par and pure

A recent e-mail to the Rust mailing list got me thinking that perhaps the right way to think about parallel blocks is to call them `pure` blocks. They are kind of a pure function, after all, from their upvars and arguments to their output. The only thing that I don't quite like is that a so-called `pure` block, if given a non-const argument, would be free to modify that argument in place. This is not very pure, but things like `par_map_in_place()` rely on it.