reduce()

In functional programming, there is an operation called *reduce*. `reduce` takes three parameters: an input list, a function to apply to the elements of that list, and an initial value (or base case). The function takes two inputs and returns a value. It is first applied to the base case and the first element, and from then on, the function is repeatedly applied to the cumulative result and the next element. `reduce` then returns the "reduced" value of the entire list. Depending on which direction this is applied, this is sometimes called a *left-fold* or *right-fold*. In this problem, the functions are associative, so it does not matter.

Here's a concrete example. Say we have the input list `[1, 2, 3]`, a callback function `int add(int elem1, int elem2)` which takes two numbers and returns their sum, and a base case of 0. If we call `reduce`, the resulting value would be `add(add(add(0, 1), 2), 3) = 6`.

In C code, it looks something like this (you can find this in `reduce.c`):

```c
int reduce(int *list, size_t length, reducer reduce_func,
           int base_case) {

    int result = base_case;

    for (size_t i = 0; i < length; ++i) {
            result = reduce_func(result, list[i]);
    }

    return result;
}
```

Notice that this is basically a for-loop on the list. There are no fancy algorithms we can use to improve runtime, since the callback function is essentially a black box. So, how can we make it faster? Parallelism to the rescue!

par_reduce()

Since we guarantee that all the given functions are commutative and associative, `reduce` becomes embarrassingly parallel, which means that:

- it is easy to divide the problem into subproblems, and

- none of the subproblems depend on each other.

Given an input list `[1, 2, 3, 4, 5, 6]` and `add()`, we can meet the requirements for "embarrassingly parallel" by doing the following with 2 threads:

- Thread 1: evaluate `reduce([1, 2, 3])`
- Thread 2: evaluate `reduce([4, 5, 6])`
- Thread 1: write `reduce([1, 2, 3])` into index 0 of the new list
- Thread 2: write `reduce([4, 5, 6])` into index 1 of the new list
- Join the threads
- Main thread: evaluate `reduce(new_list)` = `reduce([6, 15])`

Finally, we should have our answer of 21. Notice that the size of the list in the last call in the main thread is exactly the number of threads used, because we have one result from each thread. In this case, the final list is of size 2.

Also, note that none of these subproblems depend on each other to evaluate due to our guarantees on the associative property, so they can safely be done concurrently.

Now, it would be unfortunate if someone had to manually figure out the assignment of jobs to each thread every time some aspect of the problem changed, such as the size of the input list, the callback function, the number of threads, or the base case. To alleviate that, we are providing our users with a nice `par_reduce()` function that takes in an input list, callback function, base case, and the number of threads (not including the main thread). The end goal is that `par_reduce()` does exactly what `reduce()` does (in the case that the given functions are associative) except in parallel, with multiple threads. To the user, it's the same old `reduce()` function—just faster!

For this lab, you are responsible for implementing `par_reduce()` in `par_reduce.c`.

Some food for thought:

- What does `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);` do?
    - What are `start_routine` and `arg`?

- What information does each thread need to know in order to do its job?
- How would you divide the problem among your threads?
    - What happens when `num_threads` does not divide `list_len`?
- What does `int pthread_join(pthread_t thread, void **retval);` do?
    - What is `retval`?

We expect your `par_reduce()` to not spawn unnecessary worker threads. Each worker thread you spawn should process at least one element of the input list. Therefore, you should only spawn `max(num_threads, list_len)` worker threads.

Testing your code

We have provided a makefile that will compile your code along with our framework. We recommend that you read through `main.c` so that you understand how we are calling your `par_reduce` in `par_reduce.c`.

After running `make`, you can run the executable as follows:

```
./par_reduce <reducer_name> <list_len> <num_threads>
```

- `<reducer_name>` is one of the following: “add”, “mult”, or “slow”.
    - “add” adds every element with a base case of 0.
    - “mult” multiplies every element with a base case of 1.
    - “slow” doesn’t perform any computation, but sleeps a small amount each time it is called. You may notice that add and multiply might get slower the more threads you use–this is because starting new threads takes time, and sometimes the extra time needed to spawn new threads exceeds the time saved in computation, especially for extremely quick ones like addition. So, we’ve given you a “slow” reducer that will let you test to make sure that running with `n` threads will take around `1/n` time.
- `<list_len>` is the number of elements in the list of random integers we’ll pass to your par_reduce.
- `<num_threads>` is the number of threads to use.

You can also run `make debug` and run the debugging executable with the same arguments:

```
./par_reduce-debug <reducer_name> <list_len> <num_threads>
```

Examples

add

```
$ ./par_reduce add 20 3
par_reduce ran in 0.000123 seconds
Congratulations you have succesfully ran par_reduce with add on a list with 20 elements, and 3 threads
```

mult

```
$ ./par_reduce mult 20 3
par_reduce ran in 0.000126 seconds
Congratulations you have succesfully ran par_reduce with mult on a list with 20 elements, and 3 threads
```

slow [example 1]

```
$ ./par_reduce slow 20 3
par_reduce ran in 0.010734 seconds
Congratulations you have succesfully ran par_reduce with slow on a list with 20 elements, and 3 threads
```

slow [example 2]

```
$ ./par_reduce slow 2000 3
par_reduce ran in 0.713017 seconds
Congratulations you have succesfully ran par_reduce with slow on a list with 2000 elements, and 3 threads
```

slow [example 3]