Recitation 16 – *Fun with PASL*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2015)

April 28th, 2015

1 Announcements

- *PASLLab* is due Friday May 1st.
- The final exam is coming up on Tuesday, May 5th, from 5:30-8:30pm in CUC McConomy. We will be having a review session over the weekend– details to come!

2 Setting up PASL

Today, we will be working on implementations of a couple Sequence library functions in PASL.

To get started, download and set up *PASLLab* from Autolab, following the instructions in the handout. Next, copy the accompanying starter code into the `examples.cpp` file.

As in *PASLLab*, you can run the command

```
make example.dbg
```

to build the executable, and run the tests with

```
./example.dbg -example map_flatten -proc 20
./example.dbg -example inject -proc 20
```

3 Parallel Map-Flatten

Fusioning is the process of composing multiple operations to avoid creating unnecessary intermediate data structures. For example, it is common in data-parallel algorithms to generate a number of arrays and flatten them into one. Instead of calling `map` then `flatten`, one could write a single "map flatten" that skips the intermediate array.

Implement the function

```
sparray map_flatten(const Map_func& f, const Size_func& g, const sparray& xs)
```

where f is a function that maps values to an sparray, g maps values to the corresponding sparray size, and xs is the input array of data to process.

Hint: scan might be useful.

4 Parallel Inject

Recall the function `inject` which takes a sequence of indices and values, and injects each value into an existing sequence, as seen in hash tables and sequence-based Boruvka's algorithm. In our Sequence library, we provide the invariant that if multiple values appear in the same index, only the last value is injected. As we will see, this invariant can be slightly tricky to implement in parallel.

Implement the function

```
sparray inject_random(const long* indices, const sparray& updates, const sparray& xs)
```

which injects each of `updates[i]` into index `indices[i]` of the sparray *xs*. When multiples of the same index appears, you may take any value.

Next, implement the function

```
sparray inject(const long* indices, const sparray& updates, const sparray& xs)
```

such that the ordering invariant described above is satisfied.

Hint: recall `std::atomic` and `compare_exchange_strong` from lecture.
You can create an array of *n* `atomics` of type *T* using `my_malloc<std::atomic<T>>(n)`.