Recitation 1

PASL

1.1 Announcements

- *DPLab* is due **Wednesday afternoon**.
- *PASLLab* will be released on Wednesday also and will be due at the end of the semester.

1.2 map_flatten

If you would like to see the code run on your computer, begin by downloading the files `rec14.hpp` and `rec14-bench.cpp`. You can put these in the top directory of PASLLab once it is released. Then, edit PASLLab's Makefile to add: `rec14-bench.cpp` to the list of programs, i.e.

```
PROGRAMS=\
  sandbox.cpp \
  check.cpp \
  bench.cpp \
  rec14-bench.cpp # add me here.
                  # don't forget the slash on the previous line.
```

Task 1.1. *Using PASL primitives, implement the function*

```
template <class Map_func, class Size_func>
sparray map_flatten(const Map_func& f,
                    const Size_func& g,
                    const sparray& xs);
```

where, at a high-level, the goal is to compute

$$\texttt{flatten}\,\langle f(x) : x \in \texttt{xs}\rangle.$$

Begin by thinking of a sequential implementation and then parallelizing it. You should assume that the function arguments are typed as follows, where `f(xs[i])` *is a pointer to the front of an array of length* `g(xs[i])`.

$$f : \texttt{value\_type} \to \texttt{value\_type*}$$
$$g : \texttt{value\_type} \to \texttt{long}$$

If we were to implement map_flatten sequentially, we could create a new array of the sum of the sizes of the inner arrays in `xs` and use two for-loops, one looping through the inner arrays and the other through the elements of each inner array, to map every element and add it to the new array.

To parallelize this procedure, we would want to perform both for-loops in parallel. However, such an approach requires that every inner array knows the location of its elements in the new, large array independently of other inner arrays.

The first step is then to determine the offsets of the subarrays in the output. We can compute this by mapping `g` across the input followed by a plus-scan. Note that we're using the fusioned form of `scan_excl` here, which performs a `map` for us.

```
auto plus = [] (value_type a, value_type b) { return a + b; };
auto offsets = scan_excl(plus, g, 0l, xs);
```

The output of a `scan_excl` is a struct containing two fields, `partials` and `total`. The former is an `sparray` the same length as the input which contains each exclusive prefix sum, while the latter is the sum of the entire input. Therefore we can go ahead and allocate the result array, since we know its length.

```
sparray result = sparray(offsets.total);
```

Next, we'd like to map `f` across the input to discover each subarray, then write these subarrays to `result`. This can be accomplished with two nested `parallel_for` loops. So, we'll need to declare two granularity controllers (for now, lets just call these `C1` and `C2`). After this step, we simply return the `result` array.

```
par::parallel_for(C1, 0L, xs.size(), [&] (long i) {
  value_type* elems = f(xs[i]);
  par::parallel_for(C2, 0L, g(xs[i]), [&] (long j) {
    result[offsets.partials[i] + j] = elems[j];
  });
});
```

Note that `parallel_for` assumes that the code body given to it is constant-time, which is not true for the outer loop. So, we need to write a complexity function. The complexity function given to a `parallel_for` is assumed to take two parameters which describe a range of iterations of the for-loop, and return the complexity of that entire range. Note that any particular iteration i of our loop has a complexity of `g(xs[`i`])`, but in general, a range of iterations $[\ell, h)$ has complexity

$$\sum_{i=\ell}^{h-1} \texttt{g(xs[}i\texttt{])}.$$

These ranges can be easily calculated using the output of the scan we computed earlier. Our complexity function therefore looks like the following:

```
auto complexity = [&] (long lo, long hi) {
  long upper = (hi == xs.size()) ?
               offsets.total :
               offsets.partials[hi];
  return upper - offsets.partials[lo];
};
```

The completed code is given below.

Algorithm 1.2. *`map_flatten` in PASL*

```
loop_controller_type C1("map_flatten_1");
loop_controller_type C2("map_flatten_2");
template <class Map_func, class Size_func>
sparray map_flatten(const Map_func& f, const Size_func& g,
                    const sparray& xs) {
  long n = xs.size();

  auto plus = [] (value_type a, value_type b) { return a + b; };
  auto offsets = scan_excl(plus, g, 0L, xs);

  sparray result = sparray(offsets.total);
  auto complexity = [&] (long lo, long hi) {
    long upper = (hi == n) ? offsets.total : offsets.partials[hi];
    return upper - offsets.partials[lo];
  };
  par::parallel_for (C1, complexity, 0L, n, [&] (long i) {
    value_type* elems = f(xs[i]);
    par::parallel_for (C2, 0L, g(xs[i]), [&] (long j) {
      result[offsets.partials[i] + j] = elems[j];
    });
  });

  return result;
}
```

Remark 1.3. *These controller declarations are technically not correct. We should really templatize the controllers over the classes `Map_func` and `Size_func`, just as `map_flatten` is. You can find examples of these kinds of declarations in the `sparray.hpp` source file.*

Remark 1.4. *Some of the techniques used here may also be useful when implementing BFS in PASLLab. Feel free to reuse any code from this recitation, although you may want to make some modifications...*

1.3 inject

Throughout the semester, we've largely kept the sequence function `inject` shrouded in mystery. Let's see how the magic works!

Task 1.5. *Using PASL, implement the function*

```
sparray inject(const sparray& xs,
               const sparray& indices,
               const sparray& updates);
```

which returns the result of injecting into `xs`*. We require that* `indices` *and* `updates` *be the same length, such that for each* i*, we attempt to write* `updates[i]` *at position* `indices[i]` *in* `xs`*. Note that you should not destructively modify* `xs`*.*
If there are multiple updates specified at the same position, then all except the last should be ignored. (We want to match the behavior of `inject` *as specified in the 15210 Library.)*

Let's step back for a moment and review the *compare-and-swap* (`CAS`) operation. Given a memory location ℓ and two values x and y, this operation atomically performs the following:

1. Compare x against the contents of the memory location ℓ.
2. If they are equal, write y at ℓ and return `true`.
3. Otherwise, return `false`.

A simple extension of `CAS` is called a *priority update*[1]. This operation takes a memory location ℓ and a value y and attempts to write y at ℓ, but only if y is "greater than" the current value stored at ℓ (we write "greater than" in quotes because we could really use any comparison function). We can implement a priority update as follows:

1. Load the contents of ℓ into x.
2. While $y > x$:
   (a) If $\texttt{CAS}(\ell, x, y)$ then return.
   (b) Otherwise, load the contents of ℓ into x.

Priority updates allow multiple threads to converge upon some "maximum" value stored at a shared memory location. We can use this for `inject`. If m is the number of updates, the general idea is this: for each $0 \leq i < m$, perform a priority update at a location `temp[indices[i]]`

[1] See http://www.eecs.berkeley.edu/~jshun/contention.pdf

where we attempt to write i. Notice that the largest i will be the last thing written at this location. For each position in the output, this effectively chooses which update will be written at that position.

The full code is shown below. Note that we allocate and initialize the `temp` array by filling it with invalid indices, to detect which positions in the output will not change from the input. We implement compare-and-swap using the builtin `compare_exchange_strong` operation provided by the C++ `std::atomic` class. This function is slightly different than the pseudocode given above. Specifically,

$$\ell.\texttt{compare\_exchange\_strong}(x, y)$$

requires that x is a reference. If the CAS fails, then the contents of ℓ will be written into x.

Algorithm 1.6. *`inject` in PASL.*

```
loop_controller_type C3("inject_contr_1");
loop_controller_type C4("inject_contr_2");
sparray inject(const sparray& xs,
               const sparray& indices,
               const sparray& updates) {
  long n = xs.size();
  long m = updates.size(); // must be equal to indices.size()

  const long NO_UPDATE = -1L;
  auto temp = my_malloc<std::atomic<long>>(n);

  par::parallel_for (C3, 0L, n, [&] (long i) {
    temp[i].store(NO_UPDATE);
  });

  par::parallel_for (C4, 0L, m, [&] (long i) {
    std::atomic<long>& cell = temp[indices[i]];
    long curr = cell.load();
    // below, curr is updated if the CAS fails
    while (i > curr && !cell.compare_exchange_strong(curr, i))
  });

  sparray result = tabulate([&] (long i) {
    long idx = temp[i].load();
    return idx == NO_UPDATE ? xs[i] : updates[idx];
  }, n);

  free(temp);
  return result;
}
```

1.4 Benchmarking

Try running some speedup experiments! The two bench arguments are `map_flatten` and `inject`, respectively. For example, the following injects m randomly placed updates into an array length n. In the `map_flatten` benchmark, n is the initial array size, and m is the size of each subarray (so the output is length nm).

```
make rec14-bench.opt rec14-bench.baseline

./prun speedup -baseline "./rec14-bench.baseline" \
-parallel "./rec14-bench.opt -proc 1,5,10,15,20" \
-bench inject -n 100000,1000000 -m 100000000,200000000

./pplot speedup -series n,m
```