15-210 C++ Library Cheat Sheet

We will use a custom library for parallel algorithms in C++. Although the C++ standard library does include parallel algorithms, it is less comprehensive than our own collection. Furthermore, our library is designed to more closely match the API of the 15-210 Parallel SML library by adopting a more functional-programming style compared to the C++ standard library's imperative style.

We will add more functions to this document as the semester progresses and as we need them on future homework assignments.

1 Parallel Primitives

parallel_do

```
auto parallel_do(auto&& left, auto&& right)
auto parallel_do(auto&& left, auto&& right) -> std::pair<LeftResult, RightResult>
```

The `par::parallel_do` primitive takes two callables (i.e., function objects) and invokes them in parallel.

1. If `left` or `right` does not return anything, then `parallel_do` does not return anything
2. If both `left` and `right` return something, then `parallel_do` returns a pair of the return values.

In the second overload, `LeftResult` and `RightResult` are the return types of `left` and `right`.

2 Parallel Sequence

If you've programming in C++ before, you should be familiar with `std::vector`, which is C++'s dynamic array (a.k.a., list) type, equivalent to Python's `list` or Java's `ArrayList`. In C++, `std::vector` does not perform any of its operations in parallel, so we will use our own replacement called `par::sequence`. This type supports pretty much the same exact operations as `std::vector`, plus some more, and implements them all with parallelism in mind.

sequence type

`par::sequence<T>` denotes a sequence whose elements are of type `T`, for example, `par::sequence<int>` is a sequence of integers.

constructors

The `par::sequence` type can be constructed in many ways:

```
sequence()
sequence(const sequence& s)
sequence(sequence&& rv)
sequence(std::size_t)
sequence(std::size_t, const T&)
sequence(std::initializer_list<T>)
```

1. **default constructor** `par::sequence<T>()` constructs an empty sequence of element type `T`.

2-3. **copy/move constructor** `par::sequence<T>(seq)` creates a **copy** of the given sequence `seq`. Note that this is a *deep copy*, not an alias of `seq`.

   `par::sequence<T>(std::move(seq))` creates a sequence that takes ownership of the underlying elements of `seq`. Afterwards, the given sequence `seq` is empty.

4-5. **sized constructor** `par::sequence<T>(n)` where `n` is an integer creates a sequence consisting of `n` copies of the default value for the type `T`. For integer types, for example, the default value is zero. `par::sequence<T>(n, v)` creates a sequence consisting of `n` copies of the value `v`.

6. **initializer list constructor** `par::sequence<T>{v1, v2, v3, ...}` creates a sequence consisting of the elements `v1, v2, v3, ...` and so on.

3 Range Helpers

subrange

```
auto subrange(auto&& r, std::size_t i, std::size_t j) -> std::ranges::subrange<...>
```

The `subrange` function takes a range and two indices i and j and returns a view over the elements between indices i (inclusive) and j (exclusive). Note that a view refers to the elements of the original range, so no copies are made.

split_mid

```
auto split_mid(auto&& r) -> std::pair<std::ranges::subrange<>, std::ranges::subrange<>>
```

The `split_mid` function takes a range and returns a pair of views, the first referring to the first half of the elements and the other referring to the second half of the elements. i.e., it is equivalent to the pair `subrange(r, 0, mid)` and `subrange(r, mid, n)` where n is the size of `r` and `mid` is $n/2$.

4 Parallel Algorithms

tabulate

```
tabulate(std::size_t n, auto&& f) -> par::sequence<ValueType>
```

The expression `par::tabulate(n, f)` evaluates to a `par::sequence` of length n where the i^{th} element has value $f(i)$. The value type, `ValueType`, of the sequence is the decayed type of the return type of f.

map

```
map(auto&& r, auto&& f) -> par::sequence<ValueType>
```

The expression `par::map(r, f)` over a range of elements `r` evaluates to a `par::sequence` with the same length as `r` where the i^{th} element has value $f(r[i])$. That is, it applies the function f to every element of `r` and returns a sequence of the results. The value type, `ValueType`, of the sequence is the decayed type of the return type of f.

reduce

```
reduce(auto&& r) -> ValueType
reduce(auto&& r, auto I, auto&& f) -> ValueType
```

The `par::reduce` function computes a reduction over a range of elements with respect to a given identity element and associative function.

1. If no identity and operator are given, the default is 0 and plus
2. Computes the reduction with respect to the identity `I` and operator f

The return type, `ValueType` is deduced to be the same as the value type of the input range `r`.

scan

```
scan(auto&& r) -> std::pair<par::sequence<ValueType>, ValueType>
scan(auto&& r, ValueType I, auto&& f) -> std::pair<par::sequence<ValueType>, ValueType>
```

The `par::scan` function computes the generalized prefix sum of r with respect to a given identity element and associative function. That is, it computes the sequence

$$I, r[0], f(r[0], r[1]), f(f(r[0], r[1]), r[2]), \ldots$$

and the total generalized sum. `par::scan` is *exclusive*, meaning the first element of the result sequence is always I and the final element of the result sequence does not include the final element of `r`. The return value is a pair consisting of the sequence described above and the total generalized sum.

1. If no identity and operator are given, the default is 0 and plus
2. Computes the generalized sum with respect to the identity `I` and operator f

The type of the result and the identity element `I`, denoted by `ValueType` is deduced to be the same as the value type of the input range `r`.

scan_inclusive

```
scan_inclusive(auto&& r) -> par::sequence<ValueType>
scan_inclusive(auto&& r, ValueType I, auto&& f) -> par::sequence<ValueType>
```

The `par::scan_inclusive` function computes the generalized prefix sum of r with respect to a given identity element and associative function. That is, it returns the sequence

$$r[0], f(r[0], r[1]), f(f(r[0], r[1]), r[2]), \ldots$$

That is, it similar to `par::scan` except it is *inclusive*. The first element is always $r[0]$ and the final element includes the final element of `r`. The return value is the sequence of prefix sums. Unlike `par::scan`, the return value is not a pair with the total, since the total is the final element of the prefix sums.

1. If no identity and operator are given, the default is 0 and plus
2. Computes the generalized sum with respect to the identity `I` and operator f

The type of the result and the identity element `I`, denoted by `ValueType` is deduced to be the same as the value type of the input range `r`.

flatten

```
flatten(auto&& r) -> par::sequence<ValueType>
```

The `par::flatten` function takes a sequence of sequences (or more generally, any range of ranges) and concatenates the inner sequences into a single sequence.

The type `ValueType` is the value type of the inner sequences.