Erroneous Techniques in Parallel Algorithms

This sections discusses zip iterators and erroneous technique in detail.

WE'LL COVER THE FOLLOWING

Erroneous Technique

It's also important to mention one aspect:

As the standard says in algorithms.parallel.exec:

> Unless otherwise stated, implementations may make arbitrary copies of elements (with type `T`) from sequences where `is_trivially_copy_-constructible_v<T>` and `is_trivially_destructible_v<T>` are true. [**Note**: This implies that user-supplied function objects should not rely on object identity of arguments for such input sequences. Users for whom the object identity of the arguments to these function objects is important should consider using a wrapping iterator that returns a non-copied implementation object such as reference_wrapper or some equivalent solution.]

Thus you cannot write:

```
vector<int> vec;
vector <int> other;
vector <int> external;
int* beg = vec.data();
std::transform(std::execution::par,
               vec.begin(), vec.end(), other.begin(),
               [&beg, &external](const int& elem) {
                   // use pointer arithmetic
                   auto index = &elem - beg;
                   return elem * externalVec[index];
               }
```

```
);
```

The code above uses pointer arithmetic to find the current index of the element. Then we can use this index to access other containers.

The technique, however, uses the assumption that `elem` is the exact element from the container and not its copy! Since the implementations might copy elements, the addresses might be completely unrelated! This faulty technique also assumes that the container is storing the items in a contiguous chunk of memory.

Only `for_each` and `for_each_n` have a guarantee that the elements are not being copied during the execution [alg.foreach]:

> Implementations do not have the freedom granted under [algorithms.parallel.exec] to make arbitrary copies of elements from the input sequence.

In the next lesson, we will implement a counting algorithm using parallel algorithms.