Abstract

Range partition adaptors, a new type of adaptor for the C++ Standard Template Library (STL), can be the basis for a parallel version of the STL.

1 Introduction

The Standard Template Library, or STL [1, 2], is a large body of software components written in C++ [3]. It provides many of the basic algorithms and data structures of computer science, and has been accepted as part of the ANSI/ISO C++ standard.

The STL is a generic library [4]: its components are heavily parameterized. Its components are designed so that they may be combined together so long as certain specified requirements are satisfied; the inplace_merge algorithm, for example may be used with a linked list of strings, a vector of floating-point numbers, or a list of vectors of integers. Users may provide their own data types, algorithms, containers, and methods of iterating through containers.

We have chosen to parallelize the STL rather than some other library, for three reasons. First, the STL exists, is standard, and is in common use. Second, the STL is designed to be highly efficient. Third, the STL’s orthogonality—in particular, its decoupling of algorithms from containers—makes it possible to add parallel components without redesigning the entire library.

2 The Standard Template Library

The STL, like many other class libraries, includes a selection of container classes. Specifically, the STL containers are vector, list, deque, set, multiset, map, and multimap. The classes vector, list, and deque are sequential containers, and the classes set, multiset, map, and multimap are associative containers. The STL also includes a large collection of algorithms to manipulate the data stored in containers.

Decoupling algorithms from containers is possible because of iterators. Iterators are essentially a generalization of pointers: an iterator can be dereferenced using the unary 
* operator to obtain the value it refers to, it can be incremented to obtain the iterator that refers to the following element, and so on. Consider, for example, the following function.

```cpp
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
{
  while (first != last) f(*first++);
  return f;
}
```

This is the STL’s for_each algorithm: it applies a function object to every object in a range. Both the type of f and the type of first and last are generic types, or, in C++ terminology, template parameters. The iterators passed as arguments to for_each may thus be of any type that satisfies a set of requirements: it must be possible to compare two iterators for equality using operator== and operator!=, to dereference an iterator using operator*, to apply the function object f to a dereferenced iterator, and to increment an iterator using operator++.

Formally, the arguments first and last must satisfy the axioms of input iterators, which are part of the STL’s specification [1]. The STL also specifies axioms for forward iterators, bidirectional iterators, and random access iterators. Forward iterators satisfy all of the input iterator axioms as well as some additional axioms; similarly, bidirectional iterators satisfy a superset of the forward iterator axioms. Random access iterators satisfy the most stringent set of axioms: operations on random access iterators include arbitrary jumps (it + n and it - n), subscribing (it[n]), comparison (it1 < it2), and finding the distance between two iterators (it1 - it2). Note that “random access iterator” is not a type or a class: it is an abstract concept that refers to any type satisfying a set of axioms. Pointers, for example, are random access iterators, as are iterators for the STL’s deque class.

The algorithm for_each is typical in that its arguments form a range [first, last]1. For any STL container X, (X.begin(), X.end()) is a range that represents the entire container.

In addition to iterators, algorithms, and containers, the STL includes two other categories of components: allocators, which parameterize allocation and management of memory, and adaptors, which transform one interface into another.

1Note the asymmetry of this notation: first is the beginning of the range, and last is one past the end. This asymmetry is essential for many purposes, such as the representation of an empty range.
One example of an adaptor is `reverse_iterator`, which uses an underlying iterator to traverse a range in reverse order.

3 Parallel STL

One obvious strategy for performing an operation, such as `for_each`, given \( n \) parallel threads, is to divide the range into \( n \) pieces and then, in each thread, perform a sequential `for_each`. Within the context of the STL, this strategy has several immediate implications. First, the arguments to `for_each_par` should be random access iterators: dividing a range into pieces requires operations like `last - first` and `first + N/n`. Second, this strategy has a natural decomposition into two parts: dividing the range into \( n \) pieces, and performing `for_each` on each piece. The second part depends on the specific algorithm under consideration, but the first does not. Third, the division should be by means of an adaptor: performing a sequential `for_each` on the \( i^{th} \) piece requires iterators that refer to the \( i^{th} \) piece.

The essential insight of this strategy [5] is that parallelism is related to multidimensionality: it involves the conversion of a one-dimensional structure, a range, into a two-dimensional structure, a collection of subranges. We call the adaptor that performs this conversion a `range partition adaptor`. Using partition adaptors, `for_each_par` can be written as follows.

```cpp
template<class RandBasicIter, class Function, class PartitionAdaptor>
void for_each_par(RandBasicIter first, RandBasicIter last, Function f, PartitionAdaptor adpt) {
    adpt.partition(first, last, f, PartitionAdaptor adpt) {
        #pragma parallel if(n > 1)
        #pragma shared(adpt, f) local(n)
        #pragma pfor iterate(n;0; adpt.size(); 1)
        #pragma schedule(PartitionAdaptor::scheduling_tag)
        for(int n=0; n < adpt.size(); ++n)
            for_each(adpt.begin(n), adpt.end(n), f);
    }
}
```

The `#pragma` directives in this code are taken from the IRIS Power C compiler [6], and are based on the PCF extensions to Fortran [7].

Note that this algorithm is orthogonal in the same sense as existing STL components: it can be used with any kind of random access iterator, including pointers, and the user can provide any range partition adaptor that satisfies the partition adaptor requirements. By examining `for_each_par`, and other algorithms written using range partition adaptors, it is possible to deduce what those requirements must be.

- A default constructor. Other constructors are optional.
- Typedefs `base_iterator` and `subrange_iterator`. A `base_iterator` is used to iterate through the range being partitioned, and a `subrange_iterator` is used to iterate through each subrange.
- Typedef `scheduling_tag`. It is used to determine how the iterations of a parallel loop are scheduled, and it must be one of the following types: `simple_scheduling_tag`, `gss_scheduling_tag`, `interleave_scheduling_tag`, or `dynamic_scheduling_tag`.
- A method `partition` that takes two arguments, `first` and `last`, of type `base_iterator`; `(first, last)` must be a valid range.
- A method `size()` that returns the number of subranges produced by the partitioning.
- Methods `begin()` and `end()`. Each takes an integral argument \( n \) such that \( 0 \leq n < \text{size()} \), and each has the return type `subrange_iterator`. `(begin(n), end(n))` is the \( n^{th} \) subrange.
- A method `base()`, whose argument is of type `subrange_iterator` and return type is of type `base_iterator`. This method returns the iterator within the original range that corresponds to a particular subrange iterator. ("Corresponds to" means that dereferencing the two iterators yields the same element.)
- A method `iterations_par_chunk()` that returns an integer: the requested chunk size of a parallel loop. The return value is used only if `scheduling_tag` is either `interleave_scheduling_tag` or `dynamic_scheduling_tag`.

4 Conclusion

We have used range partition adaptors to write parallel versions of several simple STL algorithms, including `for_each`, `count`, `copy`, and `reverse`. Work on applying partition adaptors to more complicated algorithms, such as sort, is in progress.

References


