The Parallel-Container Template Library User's Guide

Deepsea Project

10 February 2015

Introduction

The purpose of this document is to serve as a working draft of the design and implementation of the Parallel Container Template Library (PCTL).

Essential terminology.

A *function call operator* (or, just "call operator") is a member function of a C++ class that is specified by the name `operator()`.

A *functor* is a C++ class which defines a call operator.

A *right-open range* is . . .

work span

Containers

Sequence containers

Table 1: Sequence containers that are provided by pctl.

Class name	Description
`parray`	Array class
`pchunkedseq`	Parallel chunked sequence class

Associative containers

Table 2: Associative containers that are provided by pctl.

Class name	Description
set	Set class
map	Associative map class

Parallel array

Table 3: Template parameters for the `parray` class.

Template parameter	Description
`Item`	Type of the objects to be stored in the container
`Alloc`	Allocator to be used by the container to construct and destruct objects of type `Item`

```
namespace pasl {
namespace data {

template <class Item, class Alloc = std::allocator<Item>>
class parray;

} }
```

Table 4: Parallel-array type definitions.

Type	Description
`value_type`	Alias for template parameter `Item`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`
`pointer`	Alias for `value_type*`
`const_pointer`	Alias for `const value_type*`
`iterator`	Iterator
`const_iterator`	Const iterator

Table 5: Parallel-array constructors and destructors.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item
populate constructor	constructs a container with a specified number of values that are computed by a specified function
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given parallel array
destructor	destructs a container

Template parameters

Item type

```
class Item;
```

Type of the items to be stored in the container.

Objects of type `Item` should be default constructable.

Allocator

```
class Alloc;
```

Allocator class.

Iterator

The type `iterator` and `const_iterator` are instances of the random-access iterator concept.

Constructors and destructors

Empty container constructor

```
parray();
```

Complexity. Constant time.

Constructs an empty container with no items;

Fill container

```
parray(long n, const value_type& val);
```

Constructs a container with `n` copies of `val`.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Populate constructor

```
// (1) Constant-time body
parray(long n, std::function<Item(long)> body);
// (2) Non-constant-time body
parray(long n,
       std::function<long(long)> body_comp,
       std::function<Item(long)> body);
// (3) Non-constant-time body along with range-based complexity function
parray(long n,
       std::function<long(long,long)> body_comp_rng,
       std::function<Item(long)> body);
```

Constructs a container with `n` cells, populating those cells with values returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

In the second version, the value returned by `body_comp(i)` is used by the constructor as the complexity estimate for the call `body(i)`.

In the third version, the value returned by `body_comp(lo, hi)` is used by the constructor as the complexity estimate for the calls `body(lo)`, `body(lo+1)`, ... `body(hi-1)`.

Complexity. TODO

Copy constructor

```
parray(const parray& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Initializer-list constructor

```
parray(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Work and span are linear in the size of the resulting container.

Move constructor

```
parray(parray&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

Destructor

```
~parray();
```

Destructs the container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Operations

Table 6: Parallel-array member functions.

Operation	Description
`operator[]`	Access member item
`size`	Return size
`resize`	Change size

Operation	Description
`swap`	Exchange contents
`begin cbegin`	Returns an iterator to the beginning
`end cend`	Returns an iterator to the end

Indexing operator

```
reference operator[](long i);
const_reference operator[](long i) const;
```

Returns a reference at the specified location `i`. No bounds check is performed.

Complexity. Constant time.

Size operator

```
long size() const;
```

Returns the size of the container.

Complexity. Constant time.

Resize

```
void resize(long n, const value_type& val); // (1)
void resize(long n) {                        // (2)
  value_type val;
  resize(n, val);
}
```

Resizes the container to contain `n` items.

If the current size is greater than `n`, the container is reduced to its first `n` elements.

If the current size is less than `n`,

1. additional copies of `val` are appended
2. additional default-inserted elements are appended

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the work and span are linear and logarithmic in $\max(m, n)$, respectively.

Exchange operation

```
void swap(parray& other);
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

Iterator begin

```
iterator begin() const;
const_iterator cbegin() const;
```

Returns an iterator to the first item of the container.

If the container is empty, the returned iterator will be equal to end().

Complexity. Constant time.

Iterator end

```
iterator end() const;
const_iterator cend() const;
```

Returns an iterator to the element following the last item of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.

Complexity. Constant time.

Parallel chunked sequence

Table 7: Template parameters for the `pchunkedseq` class.

Template parameter	Description
`Item`	Type of the objects to be stored in the container
`Alloc`	Allocator to be used by the container to construct and destruct objects of type `Item`

```
namespace pasl {
namespace data {

template <class Item, class Alloc = std::allocator<Item>>
class pchunkedseq;

} }
```

Table 8: Parallel chunked sequence type definitions.

Type	Description
`value_type`	Alias for template parameter `Item`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`

Table 9: Parallel chunked sequence constructors and destructors.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item
populate constructor	constructs a container with a specified number of values that are computed by a specified function
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given parallel array
destructor	destructs a container

Template parameters

Item type

```
class Item;
```

Type of the items to be stored in the container.

Objects of type `Item` should be default constructable.

Allocator

```
class Alloc;
```

Allocator class.

Iterator

TODO

Constructors and destructors

Empty container constructor

```
pchunkedseq();
```

Complexity. Constant time.

Constructs an empty container with no items;

Fill container

```
pchunkedseq(long n, const value_type& val);
```

Constructs a container with `n` copies of `val`.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Populate constructor

```
// (1) Constant-time body
pchunkedseq(long n, std::function<Item(long)> body);
// (2) Non-constant-time body
pchunkedseq(long n,
            std::function<long(long)> body_comp,
            std::function<Item(long)> body);
// (3) Non-constant-time body along with range-based complexity function
pchunkedseq(long n,
            std::function<long(long,long)> body_comp_rng,
            std::function<Item(long)> body);
```

Constructs a container with `n` cells, populating those cells with values returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

In the second version, the value returned by `body_comp(i)` is used by the constructor as the complexity estimate for the call `body(i)`.

In the third version, the value returned by `body_comp(lo, hi)` is used by the constructor as the complexity estimate for the calls `body(lo)`, `body(lo+1)`, ... `body(hi-1)`.

Complexity. TODO

Copy constructor

```
pchunkedseq(const pchunkedseq& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. Work and span are linear and logarithmic in the size of the resulting container, respectively.

Initializer-list constructor

```
pchunkedseq(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Work and span are linear in the size of the resulting container.

Move constructor

```
pchunkedseq(pchunkedseq&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

Destructor

```
~pchunkedseq();
```

Destructs the container.

Complexity. Work and span are linear and logarithmic in the size of the container, respectively.

Sequential operations

Table 10: Sequential operations of the parallel chunked sequence.

Operation	Description
`seq.operator[]`	Access member item
`seq.size`	Return size
`seq.swap`	Exchange contents

Indexing operator

```
reference operator[](long i);
const_reference operator[](long i) const;
```

Returns a reference at the specified location `i`. No bounds check is performed.

Complexity. Constant time.

Size operator

```
long size() const;
```

Returns the size of the container.

Complexity. Constant time.

Exchange operation

```
void swap(pchunkedseq& other);
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

Parallel operations

Table 11: Parallel operations of the parallel chunked sequence.

Operation	Description
`tabulate`	Repopulate container changing size
`resize`	Change size

Tabulate

```
void tabulate(long n, std::function<value_type(long)> body);
void tabulate(long n,
              std::function<long(long)> body_comp_rng,
              std::function<value_type(long)> body);
```

Resizes the container so that it contains `n` items.

The contents of the current container are removed and replaced by the `n` items returned by the `n` calls, `body(0)`, `body(1)`, ..., `body(n-1)`, in that order.

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the work and span are linear and logarithmic in $\max(m, n)$, respectively.

Resize

```
void resize(long n, const value_type& val);  // (1)
void resize(long n) {                        // (2)
  value_type val;
  resize(n, val);
}
```

Resizes the container to contain `n` items.

If the current size is greater than `n`, the container is reduced to its first `n` elements.

If the current size is less than `n`,

1. additional copies of `val` are appended
2. additional default-inserted elements are appended

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the work and span are linear and logarithmic in $\max(m, n)$, respectively.

Data-parallel operations

Indexed-based for loop

```
parray<long> xs = { 0, 0, 0, 0 };
parallel_for(0, xs.size(), [&] (long i) {
  xs[i] = i+1;
});

std::cout << "xs = " << xs << std::endl;
```

```
xs = { 1, 2, 3, 4 }
```

Template parameters

The following table describes the template parameters that are used by the different version of our parallel-for function.

Table 12: All template parameters used by various instance of the parallel-for loop.

Template parameter	Description
`Iter`	Type of the iterator to be used by the loop
`Body`	Loop body
`Seq_body_rng`	Sequentialized version of the body
`Comp`	Complexity function for a specified iteration
`Comp_rng`	Complexity function for a specified range of iterations

Loop iterator

```
class Iter;
```

At a minimum, any value of type `Iter` must support the following operations. Let `a` and `b` denote values of type `Iter` and `n` a value of type `long`. Then, we need the subtraction operation `a-b`, the comparison operation `a!=b`, the addition-by-a-number-operation `a+n`, and the increment operation `a++`.

As such, the concept of the `Iter` class bears resemblance to the concept of the random-access iterator. The main difference between the two is that, with the random-access iterator, an iterable value necessarily has the ability to dereference, whereas with our `Iter` class this feature is not used by the parallel-for loop and therefore not required.

Loop body

```
class Body;
```

```
void operator()(Iter i);
```

Sequentialized loop body

```
class Seq_body_rng;
```

```
void operator()(Iter lo, Iter hi);
```

Complexity function

```
class Comp;
```

```
long operator()(Iter i);
```

Range-based compelxity function

```
class Comp_rng;
```

```
long operator()(Iter lo, Iter hi);
```

Instances

```
namespace pasl {
namespace pctl {

template <class Iter, class Body>
void parallel_for(Iter lo, Iter hi, Body body);

template <class Iter, class Body, class Comp>
void parallel_for(Iter lo, Iter hi, Comp comp, Body body);

} }
```

```
namespace pasl {
namespace pctl {
namespace range {

template <
  class Iter,
  class Body,
  class Comp_rng
>
void parallel_for(Iter lo,
                  Iter hi, Comp_rng comp_rng,
                  Body body);

template <
  class Iter,
  class Body,
  class Comp_rng,
  class Seq_body_rng
>
void parallel_for(Iter lo,
                  Iter hi,
                  Comp_rng comp_rng,
                  Body body,
                  Seq_body_rng seq_body_rng);

} } }
```

Reduction

Table 13: Abstraction layers used by pctl for reduction operators.

Abstraction layer	Description
Level 0	Apply a specified monoid to combine the items of a range in memory that is specified by a pair of iterator pointer values
Level 1	Introduces to the above a lift operator that allows the client to perform along with a reduction a specified tabulation, where the tabulation is injected into the leaves of the reduction tree
Level 2	Introduces to the above an operator that provides a sequentialized alternative for the lift operator
Level 3	Introduces to the above a “mergeable output” type that enables destination-passing style reduction
Level 4	Introduces to the above a “splittlable input” type that replaces the iterator pointer values

```
namespace pasl {
namespace pctl {
```

```
using scan_type = enum {
  forward_inclusive_scan,
  forward_exclusive_scan,
  backward_inclusive_scan,
  backward_exclusive_scan
};

} }
```

Level 0

Table 14: Shared template parameters for all level-0 reduce operations.

Template parameter	Description
`Iter`	Type of the iterator to be used to access items in the input container
`Item`	Type of the items in the input container
`Combine`	Associative combining operator
`Weight`	Weight function (optional)

At this level, we have two types of reduction for parallel arrays. The first one assumes that the combining operator takes constant time and the second does not.

```
namespace pasl {
namespace pctl {

template <class Iter, class Item, class Combine>
Item reduce(Iter lo, Iter hi, Item id, Combine combine);

template <
  class Iter,
  class Item,
  class Weight,
  class Combine
>
Item reduce(Iter lo,
            Iter hi,
            Item id,
            Weight weight,
           Combine combine);

} }

namespace pasl {
namespace pctl {

template <
  class Iter,
  class Item,
  class Combine
>
parray<Item> scan(Iter lo,
                  Iter hi,
                  Item id,
                  Combine combine,
                  scan_type st);

template <
  class Iter,
  class Item,
  class Weight,
  class Combine
>
parray<Item> scan(Iter lo,
                  Iter hi,
                  Item id,
                  Weight weight,
                  Combine combine,
                  scan_type st)

} }
```

Item iterator

```
class Iter;
```

An instance of this class must be an implementation of the random-access iterator.

An iterator value of this type points to a value from the input stream (i.e., a value of type `Item`).

Item

```
class Item;
```

Type of the items to be processed by the reduction.

Associative combining operator

```
class Combine;
```

The combining operator is a C++ functor that takes two items and returns a single item. The call operator for the `Combine` class should have the following type.

```
Item operator()(const Item& x, const Item& y);
```

The behavior of the reduction is well defined only if the combining operator is *associative.*

Associativity. Let `f` be an object of type `Combine`. The operator `f` is associative if, for any `x`, `y`, and `z` that are values of type `Item`, the following equality holds:

```
f(x, f(y, z)) == f(f(x, y), z)
```

Example: the "max" combining operator. The following functor is associative because the `std::max` function is itself associative.

```
class Max_combine {
public:
  long operator()(long x, long y) {
    return std::max(x, y);
  }
};
```

Weight function

```
class Weight;
```

The weight function is a C++ functor that takes a single item and returns a non-negative "weight value" describing the size of the item. The call operator for the weight function should have the following type.

```
long operator()(const Item& x);
```

Example: the array-weight function. Let `Item` be `parray<long>`. Then, one valid weight function is the weight function that returns the size of the given array.

```
class PArray_weight {
public:
  long operator()(const parray<long>& xs) {
    return xs.size();
  }
};
```

Examples ***Example: taking the maximum value of an array of numbers.*** The following code takes the maximum value of `xs` using our `Max_combine` functor.

```
long max(const parray<long>& xs) {
  return reduce(xs.cbegin(), xs.cend(), LONG_MIN, Max_combine());
}
```

Alternatively, one can use C++ lambda expressions to implement the same algorithm.

```
long max(const parray<long>& xs) {
  return reduce(xs.cbegin(), xs.cend(), LONG_MIN, [&] (long x, long y) {
    return std::max(x, y);
  });
}
```

Complexity There are two cases to consider for any reduction `reduce`(lo, hi, id, f): (1) the associative combining operator f takes constant time and (2) f does not.

(1) Constant-time associative combining operator. The amount of work performed by the reduction is $O(hi - lo)$ and the span is $O(\log(hi - lo))$.

(2) Non-constant-time associative combining operator. We define $\mathcal{R}$ to be the set of all function applications $f(x, y)$ that are performed in the reduction tree. Then,

- The work performed by the reduction is $O(n + \sum_{f(x,y) \in \mathcal{R}(f,id,lo,hi)} W(f(x, y)))$.
- The span of the reduction is $O(\log n \max_{f(x,y) \in \mathcal{R}(f,id,lo,hi)} S(f(x, y)))$.

Under certain conditions, we can use the following lemma to deduce a more precise bound on the amount of work performed by the reduction.

Lemma (Work efficiency). For any associative combining operator f and weight function w, if for any x, y,

1. $w(f(x, y)) \leq w(x) + w(y)$, and

2. $W \leq c(w(x) + w(y))$, for some constant c,

where W denotes the amount of work performed by the call $f(x, y)$, then the amount of work performed by the reduction is $O(\log(hi - lo) \sum_{lo \leq it < hi}(1 + w(*it)))$.

Example: using a non-constant time combining operator. Now, let us consider a case where the associative combining operator takes linear time in proportion with the combined size of its two arguments. For this example, we will consider the following max function, which examines a given array of arrays.

```
long max0(const parray<parray<long>>& xss) {
  parray<long> id = { LONG_MIN };
  auto weight = [&] (const parray<long>& xs) {
    return xs.size();
  };
  auto combine = [&] (const parray<long>& xs1,
                      const parray<long>& xs2) {
    parray<long> r = { std::max(max(xs1), max(xs2)) };
    return r;
  };
  parray<long> a =
    reduce(xss.cbegin(), xcc.cend(), id, weight, combine);
  return a[0];
}
```

Example source code in max.hpp and max.cpp.

Let us now analyze the efficiency of this algorithm. We will begin by analyzing the work. To start, we need to determine whether the combining operator of the reduction over `xss` is constant-time or not. This combining operator is not because the combining operator calls the `max` function twice. The first call is applied to the array `xs` and the second to `ys`. The total work performed by these two calls is linear in $|\texttt{xs}| + |\texttt{ys}|$. Therefore, by applying the work-lemma shown above, we get that the total work performed by this reduction is $O(\log |\texttt{xss}| \max_{\texttt{xs} \in \texttt{xss}} |xs||)$. The span is simpler to analyze. By applying our span rule for reduce, we get that the span for the reduction is $O(\log |xss| \max_{\texttt{xs} \in \texttt{xss}} \log |xs|)$.

When the `max` function returns, the result is just one number that is our maximum value. It is therefore unfortunate that our combining operator has to pay to package the current maximum value in the array `r`. The abstraction boundaries, in particular, the type of the `reduce` function here leaves us no choice, however. In the next level of abstraction, we are going to see that, by generalizing our `reduce` function a little, we can sidestep this issue.

Level 1

Index passing. TODO: explain

Table 15: Template parameters that are introduced in level 1.

Template parameter	Description
`Result`	Type of the result value to be returned by the reduction
`Lift`	Lifting operator
`Lift_idx`	Index-passing lifting operator
`Combine`	Associative combining operator
`Lift_comp`	Complexity function associated with the lift funciton
`Lift_comp_idx`	Index-passing lift complexity function

```
namespace pasl {
namespace pctl {
namespace level1 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift
>
Result reduce(Iter lo,
              Iter hi,
              Result id,
              Combine combine,
              Lift lift);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp,
  class Lift
>
Result reduce(Iter lo,
              Iter hi,
              Result id,
              Combine combine,
              Lift_comp lift_comp,
              Lift lift);

} } }

namespace pasl {
namespace pctl {
namespace level1 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift
>
parray<Result> scan(Iter lo,
                    Iter hi,
                    Result id,
                    Combine combine,
                    Lift lift,
                    scan_type st);


template <
  class Iter,
  class Result,
  class Combine,
  class Lift_idx
>
parray<Result> scani(Iter lo,
                     Iter hi,
                     Result id,
                     Combine combine,
                     Lift_idx lift_idx,
                     scan_type st);

} } }

namespace pasl {
namespace data {
namespace level1 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_idx
>
Result reducei(Iter lo,
               Iter hi,
               Result id,
               Combine combine,
               Lift_idx lift_idx);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_idx,
  class Lift_idx
>
Result reducei(Iter lo,
               Iter hi,
               Result id,
               Combine combine,
               Lift_comp_idx lift_comp_idx,
               Lift_idx lift_idx);

} } }

namespace pasl {
namespace data {
namespace level1 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp,
  class Lift
>
parray<Result> scan(Iter lo,
                    Iter hi,
                    Result id,
                    Combine combine,
                    Lift_comp lift_comp,
                    Lift lift,
                    scan_type st);

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_idx,
  class Lift_idx
>
parray<Result> scani(Iter lo,
                     Iter hi,
                     Result id,
                     Combine combine,
                     Lift_comp_idx lift_comp_idx,
                     Lift_idx lift_idx,
                     scan_type st);

} } }
```

Result

```
class Result
```

Type of the result value to be returned by the reduction.

This class must provide a default (i.e., zero-arity) constructor.

Lift

```
class Lift;
```

The lift operator is a C++ functor that takes an iterator and returns a value of type `Result`. The call operator for the `Lift` class should have the following type.

```
Result operator()(Iter it);
```

Index-passing lift

```
class Lift_idx;
```

The lift operator is a C++ functor that takes an index and a corresponding iterator and returns a value of type `Result`. The call operator for the `Lift` class should have the following type.

```
Result operator()(long pos, Iter it);
```

The value passed in the `pos` parameter is the index corresponding to the position of iterator `it`.

Associative combining operator

```
class Combine;
```

Now, the type of our associative combining operator has changed from what it is in level 0. In particular, the values that are being passed and returned are values of type `Result`.

```
Result operator()(const Result& x, const Result& y);
```

Complexity function for lift

```
class Lift_comp;
```

The lift-complexity function is a C++ functor that takes an iterator and returns a non-negative number of type `long`. The `Lift_comp` class should provide a call operator of the following type.

```
long operator()(Iter it);
```

Index-passing lift-complexity function

```
class Lift_comp_idx;
```

The lift-complexity function is a C++ functor that takes an index and an iterator and returns a non-negative number of type `long`. The `Lift_comp_idx` class should provide a call operator of the following type.

```
long operator()(long pos, Iter it);
```

Examples

```
long max1(const parray<parray<long>>& xss) {
  using iterator = typename parray<parray<long>>::const_iterator;
  auto combine = [&] (long x, long y) {
    return std::max(x, y);
  };
  auto lift_comp = [&] (iterator it_xs) {
    return it_xs->size();
  };
  auto lift = [&] (iterator it_xs) {
    return max(*it_xs);
  };
  auto lo = xss.cbegin();
  auto hi = xss.cend();
  return level1::reduce(lo, hi, 0, combine, lift_comp, lift);
}
```

Example source code in max.hpp and max.cpp.

Level 2

Table 16: Template parameters that are introduced in level 2.

Template parameter	Description
`Seq_reduce_rng`	Sequential alternative body for the reduce operation
`Lift_comp_rng`	Range-based lift complexity function
`Seq_scan_rng_dst`	Sequential alternative body for the scan operation

```
namespace pasl {
namespace pctl {
namespace level2 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_rng,
  class Lift_idx,
  class Seq_reduce_rng
>
Result reduce(Iter lo,
              Iter hi,
              Result id,
              Combine combine,
              Lift_comp_rng lift_comp_rng,
              Lift_idx lift_idx,
              Seq_reduce_rng seq_reduce_rng);

} } }

namespace pasl {
namespace pctl {
```

```
namespace level2 {

template <
  class Iter,
  class Result,
  class Combine,
  class Lift_comp_rng,
  class Lift_idx,
  class Seq_reduce_rng
>
parray<Result> scan(Iter lo,
                    Iter hi,
                    Result id,
                    Combine combine,
                    Lift_comp_rng lift_comp_rng,
                    Lift_idx lift_idx,
                    Seq_reduce_rng seq_reduce_rng,
                    scan_type st);

} } }
```

Sequential alternative body for the lifting operator

```
class Seq_reduce_rng;
```

The sequential-reduce function is a C++ functor that takes a pair of iterators and returns a result value. The `Seq_reduce_rng` class should provide a call operator with the following type.

```
Result operator()(Iter lo, Iter hi);
```

Range-based lift-complexity function

```
class Lift_comp_rng;
```

The range-based lift-complexity function is a C++ functor that takes a pair of iterators and returns a non-negative number. The value returned is a value to account for the amount of work to be performed to apply the lift function to the items in the right-open range `[lo, hi)` of the input sequence.

```
long operator()(Iter lo, Iter hi);
```

Sequential alternative body for the scan operation

```
class Seq_scan_rng_dst;
```

The sequential-scan function is a C++ functor that takes a pair of iterators, namely `lo` and `hi`, and writes its result to a range in memory that is pointed to by `dst_lo`. The `Seq_scan_rng_dst` class should provide a call operator with the following type.

```
Result operator()(Iter lo, Iter hi, typename parray<Result>::iterator dst_lo);
```

Examples The following function is useful for building a table that is to summarize the cost of processing any given range in a specified sequence of items. The function takes as input a length value `n` and a weight function `weight` and returns the corresponding weight table.

```
template <class Weight>
parray<long> weights(long n, Weight weight);
```

The result returned by the call `weights(n, w)` is the sequence `[0, w(0), w(0)+w(1), w(0)+w(1)+w(2), ..., w(0)+...+w(n-1)]`. Notice that the size of the value returned by the `weights` function is always `n+1`. As an example, let us consider an application of the `weights` function where the given weight function is one that returns the value of its current position.

```
parray<long> w = weights(4, [&] (long i) {
  return i;
});
std::cout << "w = " << w << std::endl;
```

The output is the following.

```
w = { 0, 0, 1, 3, 6 }
```

```
long max2(const parray<parray<long>>& xss) {
  using iterator = typename parray<parray<long>>::const_iterator;
  parray<long> w = weights(xss.size(), [&] (const parray<long>& xs) {
    return xs.size();
  });
  auto lift_comp_rng = [&] (iterator lo_xs, iterator hi_xs) {
    long lo = lo_xs - xss.cbegin();
    long hi = hi_xs - xss.cbegin();
    return w[hi] - w[lo];
  };
  auto combine = [&] (long x, long y) {
    return std::max(x, y);
  };
  auto lift = [&] (long, iterator it_xs) {
    return max(*it_xs);
  };
  auto seq_reduce_rng = [&] (iterator lo_xs, iterator hi_xs) {
    return max_seq(lo_xs, hi_xs);
  };
  iterator lo_xs = xss.cbegin();
  iterator hi_xs = xss.cend();
  return level2::reduce(lo_xs, hi_xs, 0, combine, lift_comp_rng, lift, seq_reduce_rng);
}

template <class Iter>
long max_seq(Iter lo_xs, Iter hi_xs) {
  long m = LONG_MIN;
  for (Iter it_xs = lo_xs; it_xs != hi_xs; it_xs++) {
    const parray<long>& xs = *it_xs;
    for (auto it_x = xs.cbegin(); it_x != xs.cend(); it_x++) {
      m = std::max(m, *it_x);
    }
  }
  return m;
}
```

Example source code in max.hpp and max.cpp.

Level 3

Table 17: Template parameters that are introduced in level 3.

Template parameter	Description
`Input_iter`	Type of an iterator for input values
`Output_iter`	Type of an iterator for output values
`Output`	Type of the object to manage the output of the reduction
`Lift_idx_dst`	Lift function in destination-passing style
`Seq_reduce_rng_dst`	Sequential reduce function in destination-passing style

```
namespace pasl {
namespace pctl {
namespace level3 {

template <
  class Input_iter,
  class Output,
  class Result,
  class Lift_comp_rng,
  class Lift_idx_dst,
  class Seq_reduce_rng_dst
>
void reduce(Input_iter lo,
            Input_iter hi,
            Output out,
            Result id,
            Result& dst,
            Lift_comp_rng lift_comp_rng,
            Lift_idx_dst lift_idx_dst,
            Seq_reduce_rng_dst seq_reduce_rng_dst);

} } }

namespace pasl {
namespace pctl {
namespace level3 {

template <
  class Input_iter,
  class Output,
  class Result,
  class Output_iter,
  class Lift_comp_rng,
  class Lift_idx_dst,
  class Seq_scan_rng_dst
>
void scan(Input_iter lo,
          Input_iter hi,
          Output out,
          Result& id,
          Output_iter outs_lo,
          Lift_comp_rng lift_comp_rng,
          Lift_idx_dst lift_idx_dst,
          Seq_scan_rng_dst seq_scan_rng_dst,
          scan_type st);
} } }
```

Input iterator

```
class Input_iter;
```

An instance of this class must be an implementation of the random-access iterator.

An iterator value of this type points to a value from the input stream.

Output iterator

```
class Output_iter;
```

An instance of this class must be an implementation of the random-access iterator.

An iterator value of this type points to a value from the output stream.

Output

```
class Output;
```

Type of the object to receive the output of the reduction.

Table 18: Constructors that are required for the `Output` class.

Constructor	Description
copy constructor	Copy constructor

Table 19: Required constructors for the `Output` class.

Public method	Description
`init`	Initialize given result object
`copy`	Copy the contents of a specified object to a specified cell
`merge`	Merge result objects

Table: Public methods that are required for the `Output` class.

Copy constructor

```
Output(const Output& other);
```

Copy constructor.

Result initializer

```
void init(Result& dst) const;
```

Initialize the contents of the result object referenced by `dst`.

Copy

```
void copy(const Result& src, Result& dst) const;
```

Copy the contents of `src` to `dst`.

Merge

```
void merge(Result& src, Result& dst) const;                   // (1)
void merge(Output_iter lo, Output_iter hi, Result& dst) const; // (2)
```

(1) Merge the contents of `src` and `dst`, leaving the result in `dst`.

(2) Merge the contents of the cells in the right-open range `[lo, hi)`, leaving the result in `dst`.

Example: cell output

```
namespace pasl {
namespace pctl {
namespace level3 {

template <class Result, class Combine>
class cell_output {
public:

  using result_type = Result;
  using array_type = parray<result_type>;
  using const_iterator = typename array_type::const_iterator;
  result_type id;
  Combine combine;

  cell_output(result_type id, Combine combine)
  : id(id), combine(combine) { }

  cell_output(const cell_output& other)
  : id(other.id), combine(other.combine) { }

  void init(result_type& dst) const {
    dst = id;
  }

  void copy(const result_type& src, result_type& dst) const {
    dst = src;
  }

  void merge(const result_type& src, result_type& dst) const {
    dst = combine(dst, src);
  }

  void merge(const_iterator lo, const_iterator hi, result_type& dst) const {
    dst = id;
    for (const_iterator it = lo; it != hi; it++) {
      dst = combine(*it, dst);
    }
  }

};

} } }
```

Destination-passing-style lift

```
class Lift_idx_dst;
```

The destination-passing-style lift function is a C++ functor that takes an index, an iterator, and a reference on result object. The call operator for the `Lift_idx_dst` class should have the following type.

```
void operator()(long pos, Input_iter it, Result& dst);
```

The value that is passed in for `pos` is the index in the input sequence of the item `x`. The object referenced by `dst` is the object to receive the result of the lift function.

Destination-passing-style sequential lift

```
class Seq_reduce_rng_dst;
```

The destination-passing-style sequential lift function is a C++ functor that takes a pair of iterators and a reference on an output object. The call operator for the `Seq_reduce_dst` class should have the following type.

```
void operator()(Input_iter lo, Input_iter hi, Result& dst);
```

The purpose of this function is provide an alternative sequential algorithm that is to be used to process ranges of items from the input. The range is specified by the right-open range `[lo, hi)`. The object referenced by `dst` is the object to receive the result of the sequential lift function.

Examples TODO

Level 4

Table 20: Template parameters that are introduced in level 4.

Template parameter	Description
`Input`	Type of input to the reduction
`Convert_reduce`	Function to convert the items of a given input and then produce a specified reduction on the converted items
`Convert_scan`	Function to convert the items of a given input and then produce a specified scan on the converted items
`Seq_convert_scan`	Alternative sequentialized version of the `Convert_scan` function
`Convert_reduce_comp`	Complexity function associated with a convert function
`Seq_convert_reduce`	Alternative sequentialized version of the `Convert_reduce` function

```
namespace pasl {
namespace pctl {
namespace level4 {

template <
```

```
  class Input,
  class Output,
  class Result,
  class Convert_reduce_comp,
  class Convert_reduce,
  class Seq_convert_reduce
>
void reduce(Input& in,
            Output out,
            Result id,
            Result& dst,
            Convert_reduce_comp convert_comp,
            Convert_reduce convert,
            Seq_convert_reduce seq_convert);

} } }

namespace pasl {
namespace pctl {
namespace level4 {

template <
  class Input,
  class Output,
  class Result,
  class Output_iter,
  class Merge_comp,
  class Convert_reduce_comp,
  class Convert_reduce,
  class Convert_scan,
  class Seq_convert_scan
>
void scan(Input& in,
          Output out,
          Result& id,
          Output_iter outs_lo,
          Merge_comp merge_comp,
          Convert_reduce_comp convert_reduce_comp,
          Convert_reduce convert_reduce,
          Convert_scan convert_scan,
          Seq_convert_scan seq_convert_scan,
          scan_type st);

} } }
```

Input

```
class Input;
```

Table 21: Constructors that are required for the `Input` class.

Constructor	Description
copy constructor	Copy constructor

Table 22: Public methods that are required for the `Input` class.

Public method	Description
`can_split`	Return value to indicate whether split is possible
`size`	Return the size of the input
`slice`	Return a specified slice of the input
`split`	Divide the input into two pieces

Copy constructor

```
Input(const Input& other);
```

Copy constructor.

Can split

```
bool can_split() const;
```

Return a boolean value to indicate whether a split is possible.

Size

```
long size() const;
```

Returns the size of the input.

Slice

```
Input slice(parray<Input>& ins, long lo, long hi);
```

Returns a slice of the input that occurs logically in the right-open range `[lo, hi)`, optionally using `ins`, the results of a precomputed application of the `split` function.

Split

```
void split(Input& dst);            // (1)
parray<Input> split(long n));      // (2)
```

(1) Transfer a fraction of the contents of the current input object to the input object referenced by `dst`.

The behavior of this method may be undefined when the `can_split` function would return `false`.

(2) Divide the contents of the current input object into at most `n` pieces, returning an array which stores the new pieces.

Example: random-access iterator input

```
namespace pasl {
namespace pctl {
namespace level4 {

template <class Input_iter>
class random_access_iterator_input {
public:

  using self_type = random_access_iterator_input;
  using array_type = parray<self_type>;

  Input_iter lo;
  Input_iter hi;

  random_access_iterator_input() { }

  random_access_iterator_input(Input_iter lo, Input_iter hi)
  : lo(lo), hi(hi) { }
  bool can_split() const {
    return size() >= 2;
  }

  long size() const {
    return hi - lo;
  }

  void split(random_access_iterator_input& dst) {
    dst = *this;
    long n = size();
    assert(n >= 2);
    Input_iter mid = lo + (n / 2);
    hi = mid;
    dst.lo = mid;
  }

  array_type split(long) {
    array_type tmp;
    return tmp;
  }

  self_type slice(const array_type&, long _lo, long _hi) {
    self_type tmp(lo + _lo, lo + _hi);
    return tmp;
  }

};

} } }
```

Convert-reduce complexity function

```
class Convert_reduce_comp;
```

The convert-complexity function is a C++ functor which returns a positive number that associates a weight value to a given input object. The `Convert_reduce_comp` class should provide the following call operator.

```
long operator()(const Input& in);
```

Convert-reduce

```
class Convert_reduce;
```

The convert function is a C++ functor which takes a reference on an input value and computes a result value, leaving the result value in an output cell. The `Convert_reduce` class should provide a call operator with the following type.

```
void operator()(Input& in, Result& dst);
```

Sequential convert-reduce

```
class Seq_convert_reduce;
```

The sequential convert function is a C++ functor whose purpose is to substitute for the ordinary convert function when input size is small enough to sequentialize. The `Seq_convert_reduce` class should provide a call operator with the following type.

```
void operator()(Input& in, Result& dst);
```

The sequential convert function should always compute the same result as the ordinary convert function given the same input.

Convert-scan

```
class Convert_scan;
```

The convert function is a C++ functor which takes a reference on an input value and computes a result value, leaving the result value in an output cell. The `Convert_scan` class should provide a call operator with the following type.

```
void operator()(Input& in, Output_iter outs_lo);
```

Sequential convert-scan

```
class Seq_convert_scan;
```

The sequential convert function is a C++ functor whose purpose is to substitute for the ordinary convert function when input size is small enough to sequentialize. The `Seq_convert_scan` class should provide a call operator with the following type.

```
void operator()(Input& in, Output_iter outs_lo);
```

The sequential convert function should always compute the same result as the ordinary convert function given the same input.

Examples TODO

Derived operations

Sorting

Merge sort

Quick sort

Sample sort

Radix sort